The Pennsylvania State University
The J. Jeffrey and Ann Marie Fox Graduate School

AN ATTENTION MODEL FOR REMOTE SENSING LANDCOVER CLASSIFICATION

A Thesis in

Spatial Data Science

By

Shelton O. Simmons

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2026

The thesis of Shelton O. Simmons was reviewed and approved by the following:

Bing Zhou
Assistant Teaching Professor of Geography
Thesis Advisor

Fritz Connor Kessler
Professor of Geography

Anthony Robinson
Professor of Geography
Director of Online Geospatial Education Programs

Abstract

Herein, the design of a transformer used to perform remote sensing landcover image classification is analyzed. The transformer was designed to use the traditional remote sensing indices to perform the classification. A custom dataset was created for this study using Landsat 8 and 9 OLI imagery. The dataset consists of 50 images of each landcover class; urban, water, vegetation, agricultural, forest, and bare earth with an additional 10 images per class for validation. Each image was classified using the remote sensing indices to produce square clips that are roughly 90 pixels per side.

The vision transformer is trained using this custom training set by giving a bonus value to pixels which fall into the target range for the various remote sensing indices and a penalty for those that are classified while outside of the target range. This transformer had an overall accuracy gain of 1% compared to the transformer that didn't use thresholds. Furthermore, the reward and penalty logic was shown to reduce the training loss of the transformer.

Table of Contents

List of Tables

List of Figures

Facilis descensus Averni;
Noctes atque dies patet atri janua Ditis;
Sed revocare gradum, superasque evadere ad auras,
Hoc opus, hic labor est.
--Virgil, The Aenid, Book VI

Chapter 1 Introduction

"Remote sensing is the practice of deriving information about the Earth's land and water surfaces using images acquired from an overhead perspective, using electromagnetic radiation in one or more regions of the electromagnetic spectrum, reflected or emitted from the Earth's surface" (Campbell & Wynne, 2011, p. 6). Thus, remote sensing can be defined as the use of radiation to study the Earth without coming into direct contact. We have been using our eyes to study the Earth since time immemorial. However, modern remote sensing requires the use of cameras, at a bare minimum, to record radiation from the Earth's surface. This radiation can be reflected or emitted by the Earth depending on the user's needs.

The advent of the camera allowed us to systematically produce images of the Earth's surface. The image taken by the camera preserves the spectral information of the subject. The reds, oranges, and pinks of a sunrise can be stored on film or digitally to allow others to experience the same sunrise. The only limitations in the image are those from the camera itself, ranging from the common camera vernacular of focal length and shutter speed to more esoteric terms such as CCD size and spectral resolution. While most common cameras operate in the visible spectrum, using RGB, red-green-blue, values for pixels, remote sensing calls for more spectra or colors than can be detected by the naked eye. A traditional CRT television would use different values of red, green, and blue to produce all of the images on the television. In 1956 Robert Colwell used color infrared film to study cereal crops. This calls for the replacement of the traditional RGB values with NIR-Red-Green values. In other words, near-infrared, NIR, gets mapped to the Red channel, Red gets mapped to the Green Channel, and Green gets mapped to the Blue channel (Campbell & Wynne, 2011, p. 84). This combination mapping results in color infrared, CIR, which emphasizes vegetation and water bodies. In CIR imagery vegetation appears bright red, while bodies of water appear black.

Figure 1 A pair of Landsat 9 images showing the same extent. The leftmost image shows colors in traditional RGB format while the rightmost image shows colors in CIR. Landsat Images courtesy of the USGS Geological Survey.

Figure 1 shows a pair of images created by the author using a Landsat 9 scene and ArcGIS Pro. The image on the left shows a traditional RGB image with colors that one would expect. The vegetation is in green, bare earth is brown, buildings appear white. In the rightmost image, vegetation appears red, with water appearing black. This sort of mapping highlights vegetation and water. The feature that is circled in the rightmost image appears like vegetation in the RGB image, the CIR band combination allows the user to identify it properly as water.

Figure 1 was created by merely shifting the band combinations around to incorporate the near-infrared band. Other band combinations can be produced by similar adjustments to emphasize different features in an image. However, some remote sensing sensors record a multiplicity of bands beyond the traditional RGB bands. The addition of multiple bands led to the discovery that the difference between bands can allow for better emphasis than either band alone (Curran, 1980). This led to the creation of band ratios. "Band ratios are quotients between measurements of reflectance in separate portions of the spectrum" (Campbell & Wynne, 2011, p. 483). When features have different spectral properties the band ratio can better illustrate the contrast between those features. One of the most commonly used band ratios is the normalized difference vegetation index, NDVI. It is given by

$$NDVI = \frac{NIR - R}{NIR + R} \quad (1)$$

Equation 1 shows the formula for the NDVI (Campbell & Wynne, 2011). The NDVI highlights the difference between near-infrared and red bands. Vegetation has a strong absorption in the red band and high reflectance in the near-infrared band. Other features such as water and man-made structures will not have the same spectral response.

There are a number of indices used in remote sensing, akin to the NDVI, which can be used to emphasize different features. These indices can also be used to help perform landcover classification of images. In terms of the NDVI, water typically has values less than zero while highly vegetated areas tend to have values near unity. Thus, various remote sensing indices can be used to perform the supervised classification of an image, image classification moderated by an analyst. Land cover classification is the process of taking an image, typically satellite or aerial, and assigning classes to the pixels in the image such as water or vegetation.

Remote sensing has advanced from the use of cameras in aircraft to the use of satellite mounted sensors. While, some data is still collected via aircraft, especially with the advent of unmanned aerial vehicles or drones, the volume of satellite imagery has far surpassed the amount produced by more traditional cameras in aircraft. As the number of earth-observation satellites increases, the number of analysts required to process the imagery will also increase. However, satellites are producing more data than can be analyzed manually without artificial intelligence. "Earth observation data will generate about 2 Exabytes (EB) by 2032, growing at a CAGR of 11%" (Oni, 2023). Thus, the data is expected to grow at a compound annual growth rate of 11%. As the amount of data increases, the methods for processing the data must be able to keep up with the influx of data. Artificial intelligence provides a means of keeping up with the data. "Deep learning utilizes hierarchical artificial neural networks to identify patterns within data and extract valuable features from large and complex datasets" (Janga, Asamani, Sun, & Cristea, 2023).

The most used deep learning algorithm in remote sensing is the convolutional neural network. Convolutional neural networks, CNNs, are one form of AI architecture. They are made of convolutional layers which are location invariant. This makes them good at locating objects when their position may change from image to image (Kalaitzis, Bayaraa, & Rossi). These convolutional neural networks can be used to classify images pixel by pixel.

The main drawback to using neural networks or other forms of AI in remote sensing is the number of images required to train a model. A popular trainer for images is ImageNet which has 14,197,122 images. It would be physically impossible for an analyst to manually produce that number of images to train a dataset. The main drawback to using ImageNet is that the images aren't made for remote sensing. Remote sensing images are typically top-down views which resemble maps. This type of image reduces the height or relief of features such as trees and buildings and makes an image that more closely resembles maps. Any model made using those images would have to be fine-tuned for remote sensing imagery. While other datasets for remote sensing exist, none have the sheer volume possessed by ImageNet.

Convolutional networks have another noteworthy problem. Large CNNs suffer from vanishing gradients. Neural networks are trained via a process called back-propagation. In other words, the output from the model is used in a feedback system to improve the results of the model. An image would be fed into the model and the output would be created based on the current weights and biases. The differences between the predicted and actual outputs are used to generate a loss via a loss function. Then a derivative is taken of the loss function. This derivative is used to update the weights that are used to generate the output. This is done by minimizing the loss function (Hardesty, 2017). The deeper the neural network, the more layers it has. If there are too many layers, then the derivative will go to 0 which nullifies the effects of additional layers (Janga, Asamani, Sun, & Cristea, 2023). Residual networks, ResNets, were created to address this problem by skipping layers by using shortcut projections. Thus, the ResNet is an improved version of the CNN.

This thesis seeks to determine if an AI model that uses the remote sensing indices produces valid results. The remote sensing indices will be used to help train the model. This will be done by using attention layers to use the remote sensing indices to classify images. Attention layers were initially created for natural language processing, NLP, algorithms to allow access, or attend, to all of the elements in a sentence at each time step opposed to single words or phrases. Regarding remote sensing, attention layers enable the model to look at various image patches and try to find relationships between them. The goal of this thesis is to determine if using remote sensing indices will teach the model to attend to those values when classifying images. This method is more akin to how analysts perform supervised classification using software like Trimble's eCognition.

Chapter 2 Remote Sensing

2.1 The Electromagnetic Spectrum

Radiation refers to any "process which carries energy through space" (Gribbin, 1998, p. 327). This space can be filled with various gases, such as an atmosphere, or as empty as the vacuum of outer space. This radiation can be generated by the motion of charged particles including electrons changing energy levels in atoms. In this sense, motion refers to acceleration because electrons only emit radiation when they speed up, slow down, change direction, or change energy level. Radiation could also come from changes in the nucleus of the atom as well as changes in the thermal motion of particles (Campbell & Wynne, 2011, p. 31).

Electromagnetic radiation is any "radiation, including light, which is produced by interacting electric fields and magnetic fields moving together through space (or a suitably transparent medium) at the speed of light" (Gribbin, 1998, p. 119). This radiation is carried by the electromagnetic field from one point to another. Feynman gives a good example of a cork floating in water as an analogy for a charged particle in an electromagnetic field. To move the cork, we can splash in the water to produce waves. The closer we are to the cork, the more influence the waves will have on the cork's motion. The cork can also vibrate in the water and produce waves which move away from the cork. The fact that the cork can influence things far from it indirectly by vibrating in place is a direct analogy for a source of radiation. The waves that it sends out would be the radiation. The water represents the electromagnetic field. The electromagnetic field carries radiation of various frequencies which can all be caused by taking a charge and shaking it back and forth. "If we shake a charge back and forth more and more rapidly, and look at the effects, we get a whole series of different kinds of effects, which are all unified by specifying but one number, the number of oscillations per second" (Feynman, 2011, p. 31). Thus, the white noise in CRT televisions, the light that we can detect with our eyes, and gamma rays are all aspects of the same phenomena with their respective frequencies being the only difference between them.

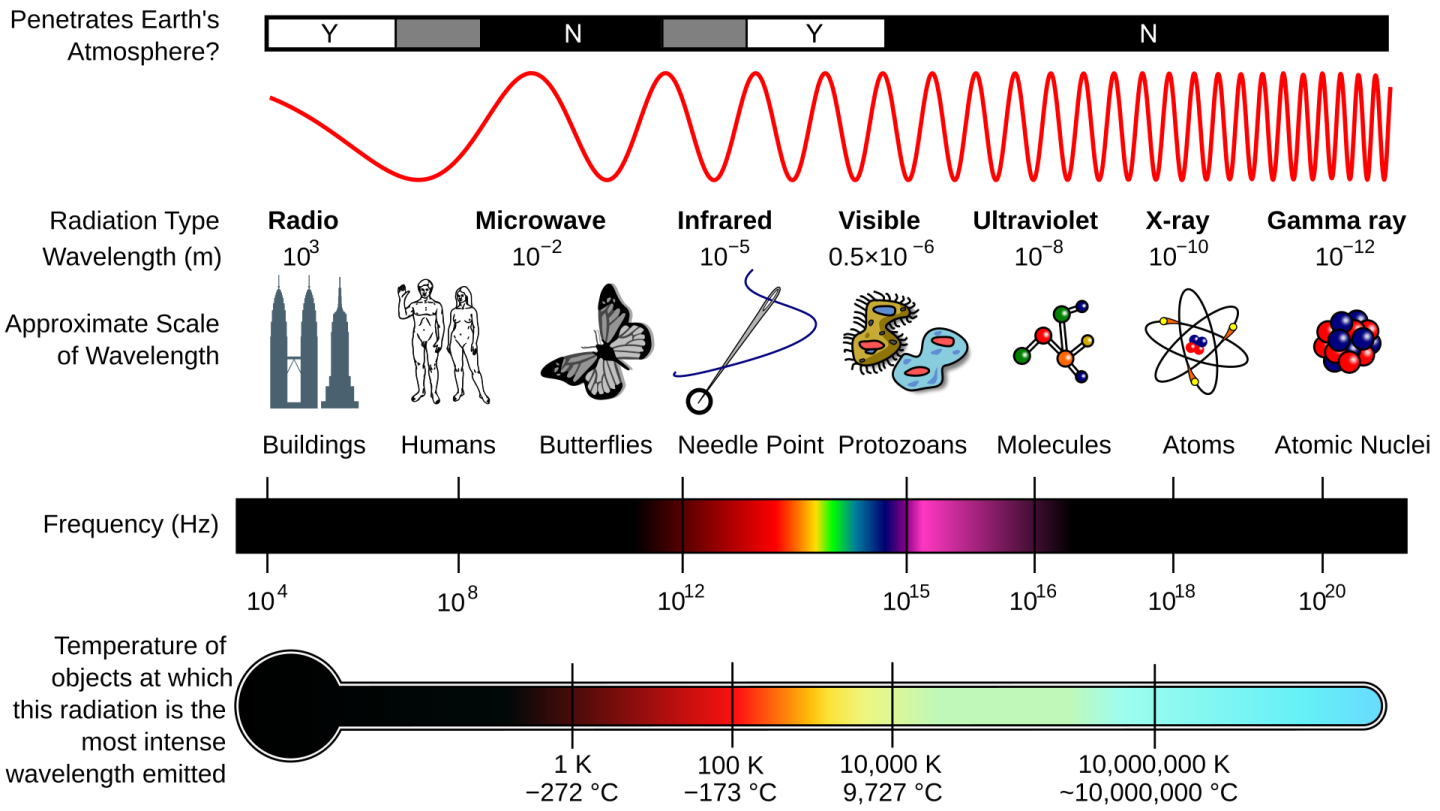


Figure 2 The electromagnetic spectrum. Adapted from an image by Inductiveload and NASA, licensed under CC BY-SA 3.0. (Inductiveload & NASA, 2025).

Figure 2 shows a picture of the electromagnetic spectrum. The frequency ranges from 10^4 to 10^{20} Hz. Feynman was stating that difference between various parts of the electromagnetic spectrum, such as radio waves and ultraviolet waves is their frequency. The only caveat is that gamma rays traditionally come from nuclei while X-rays come from electron transitions within atoms. Gamma rays and X-rays of the same frequency are physically indistinguishable. While Feynman described radiation in terms of frequency, another way of describing radiation is to talk about its wavelength.

$$\lambda = \frac{c}{\upsilon} \qquad (2)$$

Equation 2 shows the relationship between frequency, υ, and wavelength, λ. They have an inverse relationship. As the frequency increases the wavelength decreases and vice versa. In remote sensing, electromagnetic radiation is typically referred to using wavelength.

According to Figure 2 the visible spectrum corresponds to wavelengths on the order of 0.5 x 10^{-5} m or 0.5 μm. It ranges from 0.4 – 0.7 μm with blue corresponding to wavelengths of 0.4 – 0.5 μm,

green to wavelengths of 0.5 – 0.6 μm, and red to wavelengths of 0.6 – 0.7 μm. The colors that we see when we observe an object are based on the wavelengths reflected from the surface of said object. Thus, a blue pen appears blue because it reflects blue light, while absorbing the other wavelengths. All of the colors that we associate with various objects are based on different wavelengths of visible light being reflected and absorbed.

The ultraviolet, UV, spectrum refers to wavelengths that are beyond violet light. Violet light has the smallest wavelength that is visible to human eyes. UV light is easily scattered by the atmosphere which makes it unsuitable for more remote sensing applications.

The infrared spectrum refers to wavelengths that are larger than those of red light, the largest wavelengths that can be detected with the naked eye. Infrared, IR, radiation ranges from 0.72 to 15 μm making it far wider than the visible spectrum. IR radiation can be separated into two distinct categories, radiation that behaves similarly to optical sensors and radiation that is quite different from visible radiation. Near infrared, NIR, and midinfrared radiation fall into the former category. Therefore, it can be captured and manipulated using traditional means such as cameras, films, and filters. Far infrared radiation falls into the latter category and requires different tools to record their radiation. It requires "special cooled detectors containing crystals like germanium whose electrical resistance is very sensitive to heat" (Caltech Infrared Processing and Analysis Center, n.d.).

2.2 A History

Remote sensing can be defined as the study of the Earth's surface via indirect, noninvasive means. It began when the first images were taken of the Earth from aerial ballons by Gaspard-Felix Tournachon in 1858. Roughly 50 years later a plane piloted by Wilbur Wright captured the first aerial photographs from an airplane. World War I helped to prove the importance of aerial imagery for reconnaissance and surveillance. After World War I, cameras began to be designed especially for use in aircraft. Aerial images were used to perform measurements of the Earth's surface using photogrammetry, the methodology of using a photograph to reveal its inherent geometry. The use of aerial imagery, once a niche field, began to spread into the public sector. The Great Depression helped further remote sensing by leading to some of the first aerial surveys in the U.S.A. World War

II led to the use of nonvisible parts of the spectrum. While the electromagnetic spectrum was well understood, the war brought together people skilled in physics with the need to apply it to practical problems such as radar. When the war ended, those people were able to apply their skills in the civilian sector. The Cold War presented another watershed moment in the history of remote sensing in that it led to the creation of the CORONA strategic reconnaissance satellite. This satellite presented the ability to regularly collect imagery from space.

War led to many discoveries in the field of remote sensing. One of them is camouflage detection film. This film was discovered by Robert Colwell to delineate the difference between actual vegetation and camouflage. He used a color infrared film which shows a stark contrast between vegetation and manmade surfaces.

Figure 3 View of the Hollywood Hills, photographed using Kodak infrared color slide film with no filter, developed with the E-6 process. Image courtesy of Jeremya Greene, licensed under CC BY-SA 4.0 (Greene, 2014).

Figure 3 shows a color infrared image of the Hollywood Hills. In this image vegetation is shown in different shades of red. Dead vegetation appears in shades of grey. Bare earth and man-made structures are shown in shades of white. Figure 3 highlights the use of replacing a typical RGB

band with the infrared band. Camouflage was designed to resemble green vegetation, not the invisible infrared band which made it very useful during WWII. The application of this war technology to the identification and health of cereal crops helped bridge the gap between military and civilian fields.

Evelyn Pruitt coined the term remote sensing during the 1960s when she realized that aerial imagery was insufficient to describe the various forms of media used to study the Earth. This was due to the fact that "photography" refers to writing with visible light. Various satellites were launched into space during the 1950s and 1960s including Sputnik, Explorer I, TIROS-1 and CORONA. They were capable of detecting the nonvisible parts of the electromagnetic spectrum. Furthermore, she thought that the word "aerial" was too limited because she realized that the future of Earth observation was beyond the air, in outer space.

The field of remote sensing changed with the advent of Landsat 1 in 1972. It was the first satellite that was created explicitly to study the Earth. This satellite allowed for easy access to multispectral data for large regions of the Earth. This data was collected constantly and with the satellite orbiting the Earth 14 times a day. Use of Landsat 1 led to county lines being redrawn and the discovery of new islands such as Landsat Island. Landsat 1 produced 80 m resolution imagery that was digital. The routine availability of digital imagery led to the creation of digital analysis and specialized software to analyze the data. Furthermore, Landsat 1 served as an archetype for future land observation satellites.

It carried two different sensors, the Return Beam Vidicon (RBV) and the Multispectral Scanner System (MSS). The RBV collected data in 3 different bands, a blue-green band from 475 – 575 nm, an orange-red band from 580 – 680 nm, and a red-near-infrared band from 690 – 830 nm. The RBV wasn't as important historically as the MSS. The Multispectral Scanner System produced 80 m digital imagery with 4 spectral bands; a green band from 0.5 – 0.6 μm, a red band from 0.6 – 0.7 μm, and 2 near infrared bands, one from 0.7 – 0.8 μm and the other from 0.8 to 1.1 μm.

Future developments in sensors led to the satellite imagery being created at higher resolutions, up to 10 m during the 1980s and submeter resolution during the 1990s. Commercial satellites such as Geoeye and IKONOS were also developed.

The various iterations of the Landsat satellites as well as the other previously mentioned satellites are referred to as multispectral because their sensors operate in several well-defined spectral ranges. During the 1980s scientists at Caltech's Jet Propulsion Laboratory were developing sensors that collect 200 or more well-defined spectral ranges. Such sensors are referred to as hyperspectral sensors. "For example, the Airborne Visible and Infrared Imaging Spectrometer (AVIRIS) has 224 bands in the region from 400 – 2,500 nm spaced just 10 nm apart based on the FWHM criterion" (Jensen, 2014, p. 14). There are even ultraspectral sensors that are capable of operating with hundreds of bands. Meigs, Otten, & Cherezova describe how such a sensor can operate with 512 different spectral bands. These sensors are capable of detecting aerosols and hazardous materials. Furthermore they are capable of improved classification due to their more accurate spectral signatures (Meigs, Otten, & Cherezova, 1998).

2.3 Image Classification

Digital images are recorded on the CCD of a camera and given values based on the brightness of the image. The range of these brightness values is determined by the spectral resolution of the camera. Bits are the binary values used to store images. The higher the spectral resolution of the camera, the more bits its CCD would have. An 8-bit CCD would be able to store 2^8 or 256 unique values. These values correspond to the colors that we see in the image. Higher resolution CCDs can better distinguish the difference between various shades of a given color such as cerulean and teal.

Figure 4 An image showing side-by-side views of the same scene. The left image was captured with a bit depth of 24 while image has a bit depth of 8. Photograph taken by the author on 10/28/2021 at 6:20 am.

Figure 4 shows an image taken of the sunrise over Dillon High School, in Dillon SC. The image on the right has a higher bit depth at 24-bits per pixel, also known as true color. The image on the right has a lower bit depth of 8-bits per pixel. Looking at the region outlined in black, one can see that the shades are continuous in the 24-bit image while in the 8-bit image the differences in color are more pronounced.

Image classification is the process of taking an image, typically satellite or aerial, and assigning classes to the pixels in the image. This is done by analyzing the brightness values associated with the pixels and assigning classes to them based on those values. "By comparing pixels to one another and to pixels of known identity, it is possible to assemble groups of similar pixels into classes that are associated with the informational categories of interest to users of remotely sensed data (Campbell & Wynne, 2011, p. 335). One can classify a pixel based on nearby pixels or on a pixel that has already been classified.

Image classification can be explained in various ways, one of which is based on what is being classified. "The simplest form of digital image classification is to consider each pixel individually, assigning it to a class based on its several values measured in separate spectral bands" (Campbell & Wynne, 2011, p. 336). This type of classification is referred to as pixel-based image classification. During pixel-based image classification, each pixel is classified based on the brightness values stored in the various bands of an image. Traditionally, RGB or red-blue-green values are assigned to pixels. However multispectral imagery may have bands that can't be detected by the human eye such as near-infrared or thermal bands. These bands can be assigned arbitrary RGB values by the analyst to aid in image classification as per the use of color infrared film. After being processed, the spectral characteristics of the pixels are then used to classify the image.

Another form of image classification is called object-based image classification, or OBIA. OBIA is a process in which groups of pixels are classified simultaneously. It "applies a logic intended to mimic some of the higher order logic employed by human interpreters, who can use sizes, shapes, and textures of regions, as well as the spectral characteristics used for conventional pixel-based classification" (Campbell & Wynne, 2011, p. 371). Object-based image classification is a more human approach to image classification because our minds classify a pen based on all of the light received from the pen opposed to focusing on the individual photons refracted from the pen.

Object-based image classification has advantages over pixel-based classification. By considering objects, the topology of the objects can be used to aid in their classification. For example, when classifying a region with lakes, one would expect water pixels to be adjacent to one another opposed to vegetation pixels. Object-based classification was in vogue historically because of the challenges in creating user interfaces for practical use. Presently, DEFINIENS and eCognition are some of the best object-based classification programs available. However, implementing object-oriented classification can be time-consuming, as the analyst must devote considerable effort to trial and error, learning the most efficient approach to classification of a specific image for a specific purpose" (Campbell & Wynne, 2011, p. 372).

Object-based image analysis, OBIA, software allows the user to group sets of "pixels based on similar spectral signatures or different variables such as soils, buildings, plots, etc." (Rana & Kharel, 2019, p. 1). In general, one has to define the land cover classes for the image. Then the remote sensing indices must be manually typed in by the user. Care has to be taken when writing the calculation for the indices as it is easy to mistype the more complicated expressions. Next, a set of keys has to be made for each land cover class.

MSAVI2	NDVI	NDWI	Pan NIR-dev	Pan NIR	Green	NIR	Red	Green-dev	NIR-dev	Red-dev	Homogeneity	Contrast
-0.0281350	-0.0139009	0.00752935	6.9390783	249.7967480	249.9733925	246.2372506	2.5990158	7.2936021	10.9246171	2.5990158	0.1674279	4413.6022258
-0.0382553	-0.0188077	0.0146624	5.0438421	244.4109589	251.6849315	244.4109589	253.7808219	4.6075480	9.6923012	0.8316772	0.0468319	3398.0299824
-0.1235311	-0.0583088	0.0531255	5.9041690	243.4229391	250.9892473	225.6666667	253.6129032	4.9827544	11.5025321	1.2272205	0.0324700	1989.7982277
0.0651594	-0.0316203	0.0185851	7.1894603	245.8024691	246.6296296	237.6296296	253.1481481	7.4691868	12.1497369	2.0494571	0.0188542	5577.9765808
-0.0736265	-0.0355864	0.0207398	12.8144327	237.8666667	238.7000000	229	245.900000	13.0541181	17.3089572	8.0802228	0.0294304	5339.2375000
-0.0371770	-0.0182872	0.0127831	5.1516636	249.7037037	250.8888889	245.5555556	253.6666667	4.2803023	10.2318795	0.9428090	0.0240429	74384.2957746
-0.4560560	-0.1863620	0.1437159	8.0394955	205.8571429	217.4285714	162.7857143	237.3571429	9.4922095	6.1780553	8.4482216	0.0150664	6355.4553571
-4.007831	-0.1675201	0.1262285	10.7631952	205.9215686	215.7058824	167.3529412	234.7058824	14.0409682	9.4117647	8.8368527	0.00962431	6966.2132353
-0.1714044	-0.0791367	0.0663749	8.5591781	235.6666667	243.6666667	213.3333333	250	10.2089286	9.9610352	5.5075705	0.00143493	8538.6875000
-0.0168013	-0.00834749	0.00538434	4.6225891	251.9468599	252.3478261	249.6449275	253.8478261	4.2963621	8.7921249	0.7792804	0.3133463	4160.1394799
-0.0599872	-0.0291831	0.0224561	6.4312273	246.4791667	249	238.0625000	252.3750000	6.1339221	10.1209483	3.0388114	0.024 2116	5507.8893281

Table 1 A classification key used to classify pavement in a 1994 NAPP image of Florence, SC.

Table 1 was created to classify the pavement in a 1994 NAPP image of Florence, South Carolina. Such a table has to be created for each land cover class. These tables have to be created manually by copying the values for the indices such as NDVI, MSAVI2, and MTVI2. After each table is created, a key has to be created by trial and error for classification. For example, MSAVI ≤ 0, MSAVI ≥ -1.3, Green ≥100, and Red ≥ 70 was used as a key to classify grass.

I am proposing an AI that operates on the same principle as OBIA in that it creates image patches and classifies them based on the remote sensing indices.

Chapter 3 AI in Remote Sensing

3.1 History of Artificial Neural Networks

There are a number of different AI architectures used in remote sensing. They are based on artificial neural networks, ANNs. ANNs were first posited by McCulloch and Pitts in 1943. They developed a simplified computational model to explain how neurons might work in concert to perform complex computations (McCulloch & Pitts, 1943).

Artificial neurons are modeled on biological neurons.

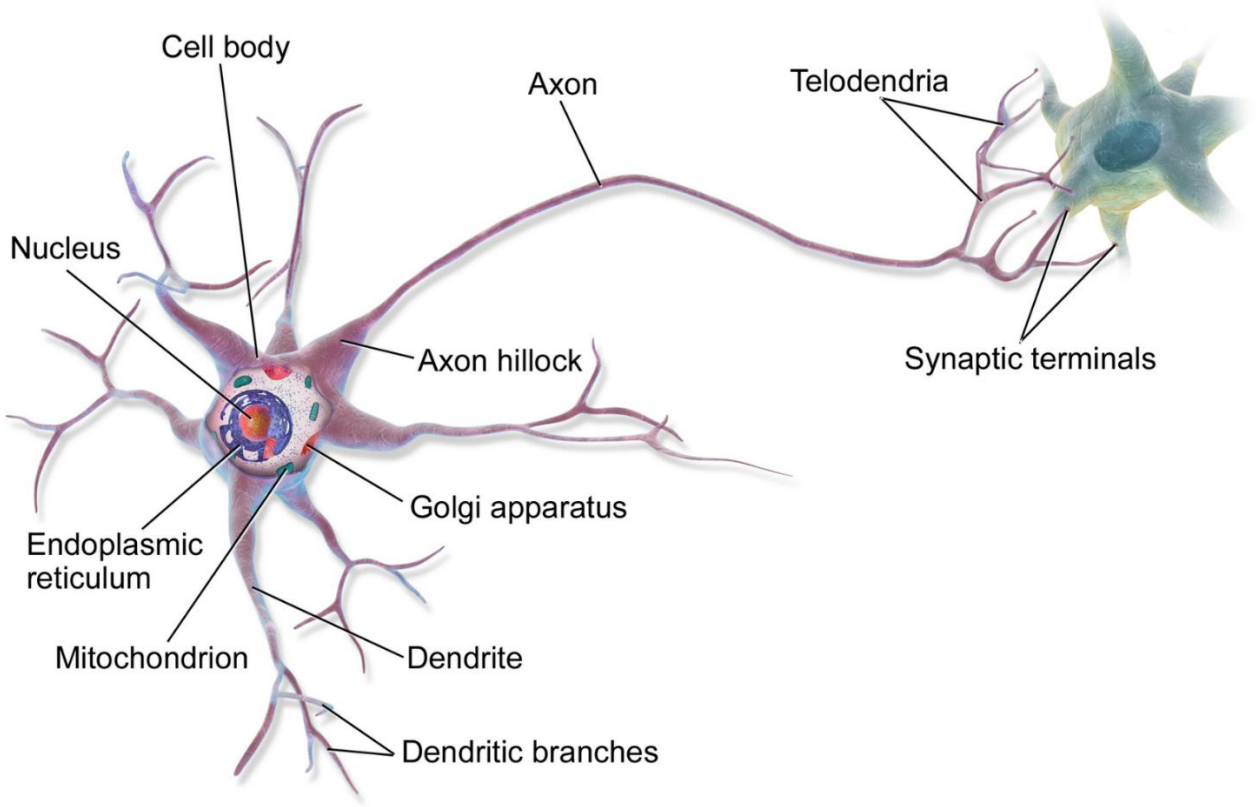


Figure 5 A multipolar biological neuron. Image courtesy of Bruce Blaus, licensed by CC By 3.0 (Blaus, 2013).

A biological neuron is shown in Figure 5. Biological neurons are cells of the nervous system that contain all of the structure common to animal cells, such as the nucleus, mitochondria, and endoplasmic reticulum, but they also contain specialized structures such as an axon and dendrites. The axon is a long filament that is used to conduct electrical impulses called action potentials away from the cell to other neurons, glands or muscles. Axons have branches near the end called telodendria. When an axon conducts a signal, the signal goes along a telodendron and releases

chemicals signals at the synapse. These chemical signals are collected by a nearby dendrite which relays that information to the cell. Dendrites are small branched filaments that fan out from the cell. Those chemical signals are also called neurotransmitters. When a neuron receives a threshold amount of neurotransmitters within a few milliseconds, it sends its own signal, thus repeating the pattern (Géron, 2023) (Wikipedia contributors, n.d.).

McCullough and Pitts showed that the use of simple artificial neurons could be used to compute any kind of logical proposition. Their artificial neuron used binary inputs to create a binary output. The artificial neuron operates when it reaches a certain threshold value from the inputs.

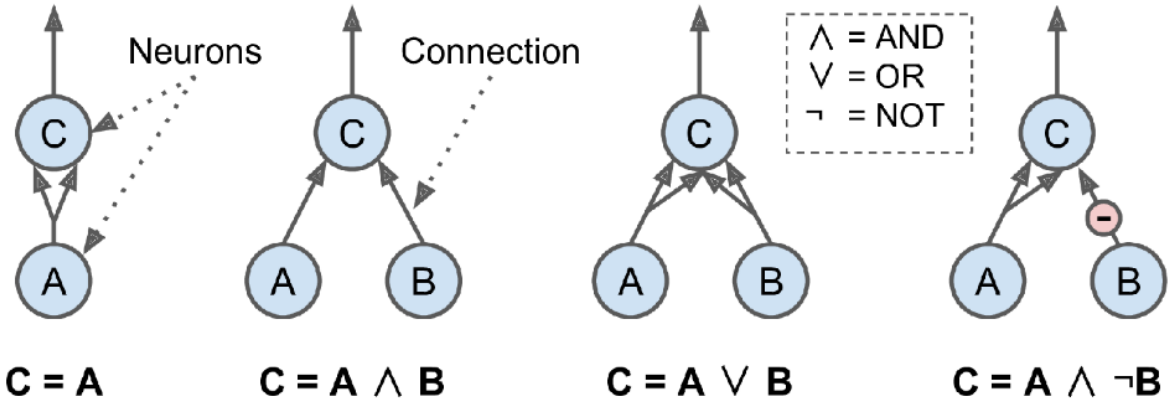


Figure 6 ANNs performing simple logical computations. Image courtesy of Aurélien Géron (Géron, 2023).

Figure 6 shows an image of some simple ANNs. In this case, the next neuron isn't active until it receives two signals from the input. The leftmost image serves as an identity function. When neuron A is activated, neuron C also becomes activated. The next image shows the results of the AND operator. Neuron C will only become active if it receives signals from neuron A and neuron B. The third image shows the result of the OR operator. In this case, neuron C will only become active when either neuron A or neuron B sends a signal. The final image demonstrates how inhibition can work. Neuron C will only become active if it receives a signal from neuron A and not from neuron B.

The perceptron is one of the earliest and simplest artificial neural networks. It was invented in 1957 by Frank Rosenblatt. Perceptrons are based on a different type of artificial neuron called a threshold logic unit.

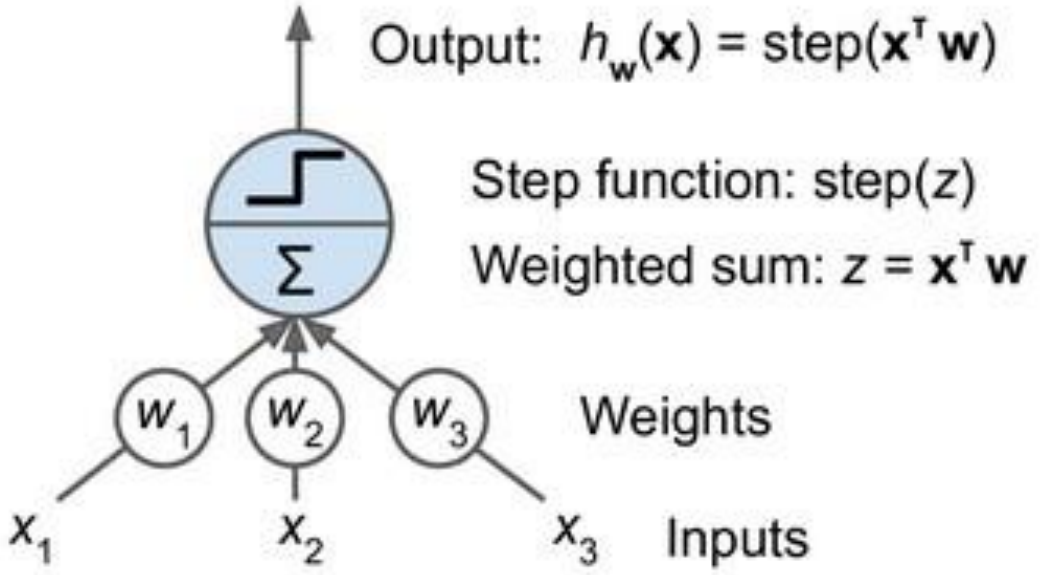


Figure 7 Threshold logic unit: an ANN that computes a weighted sum of its inputs plus a bias before the application of a Heaviside step function. This image courtesy of Trokas (Trokas, 2023).

Figure 7 shows a threshold logic unit or TLU. TLUs are based on numerical inputs instead of binary values. Furthermore, each input is associated with a weight. The TLU first takes the inputs and weights to compute the dot product. The bias is then added to the function. Those two steps are generally combined to form a linear function reminiscent of the slope-intercept equation of a line,

$$z = X^T W + b \tag{3}$$

Next, a step function is applied to the result, in this case the Heaviside function. The Heaviside function returns binary values.

$$heaviside(z) = \begin{cases} 0 \; if \; z < 0 \\ 1 \; if \; z \geq 0 \end{cases} \tag{4}$$

The Heaviside function is given by Equation 4. For inputs less than 0 the Heaviside function returns 0, otherwise it returns a value of unity. This step function allows the TLU to be used in binary classification. If the results of the linear function are within a certain threshold it assigns one class, otherwise the other class is assigned.

When multiple TLU are put in a single layer, a perceptron is formed.

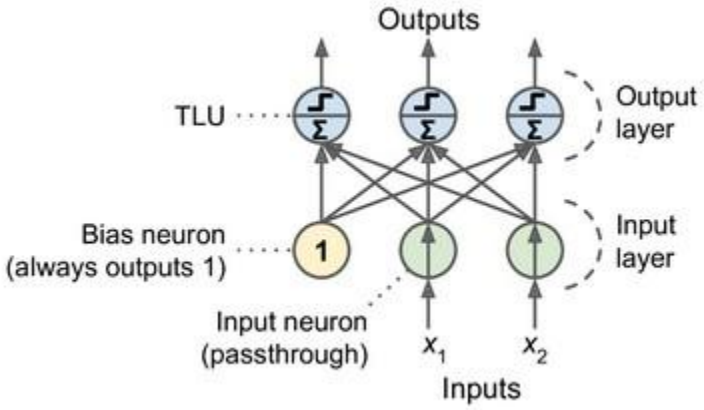


Figure 8 Architecture of a perceptron with two inputs, a bias neuron, and three outputs. Image courtesy of Trokas (Trokas, 2023).

Figure 8 shows a diagram of a perceptron. In this perceptron the bias neuron and all of the inputs are connected to each TLU forming a so-called fully connected layer. This perceptron can classify features into 6 different classes, 2 per TLU. In such a network Equation 3 takes the form of

$$h_{W,b}(\boldsymbol{X}) = \phi(\boldsymbol{XW} + \boldsymbol{b}) \tag{5}$$

where **X** is the matrix of input features, **W** is the matrix containing the weights, **b** is the bias vector, and φ is the activation function. The activation function replaces the step function required by a single TLU.

Perceptrons are trained using a version of Hebb's rule. Hebb's rule is named after Donald Hebb who posited that if a biological neuron triggers another neuron often, then the connection between these neurons will strengthen. Perceptrons also take into account the error made by the network when a prediction is made. Whenever an output neuron makes a wrong prediction, the weights for that lead to the correct input are strengthened. The perceptron learning rule is given by

$$w_{i,j}^{(next\ step)} = w_{i,j} + \eta\left(y_j - \hat{y}_j\right)x_i \tag{6}$$

where η is the learning rate, y_j is the target output of the j^{th} neuron for the current training epoch, $\hat{y}_j$ is the output of the j^{th} output neuron for the current training epoch, x_i is the input for the current epoch, and $w_{i,j}$ is the weight between the i^{th} input and the j^{th} neuron. The perceptron convergence theorem states that while perceptrons are incapable of learning complex patterns, the algorithm will converge

to a solution as long as the training instances are linearly separable. This means that it must be physically possible to draw a line between the separate binary classes.

In 1969 Minsky and Papert published a book which proved that simple feed-forward perceptrons couldn't solve the XOR problem. They studied the case in which perceptrons operated on nonseparable training instances. By doing so they developed the Perceptron Cycling Theory which states that single-layer perceptrons couldn't apply to nonlinear functions. The single-layer perceptron would continue to erroneously classify points, update weights, and fail to converge (Minsky & Papert, 1988). They were studying the Perceptron Convergence Theorem and the nonseparable case, in which training instances aren't linearly separable. They showed that for a single layer peceptron, it was impossible to solve the XOR problem.

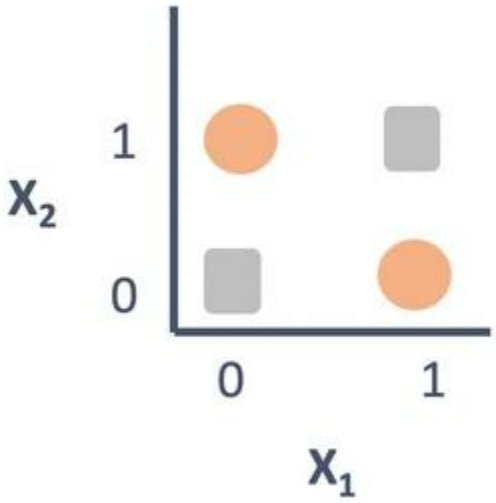


Figure 9 The XOR problem. This image courtesy of Sean Trott (Trott, 2024).

Figure 9 illustrates the XOR problem. It should be obvious to the reader that no solitary line can separate the distinct classes in the image. Rosenblatt proved that a single-layer perceptron would converge on a solution for linear instances. "Rosenblatt had investigated more complex networks, but he was never able to effectively extend the perceptron rule to such networks" (Hagan, Demuth, Beale, & De Jesús, 2014, pp. 4-19). As Rosenblatt died in 1971 and he may not have had time to respond to the work of Minsky and Papert.

When multiple layers of perceptrons are stacked together they are capable of solving the XOR problem. This form of artificial neural network is called a multilayer perceptron or MLP. A multilayer perceptron is made of one input layer, one or more layers of threshold logic units in a so-called hidden layer, and another layer of TLUs in the output layer.

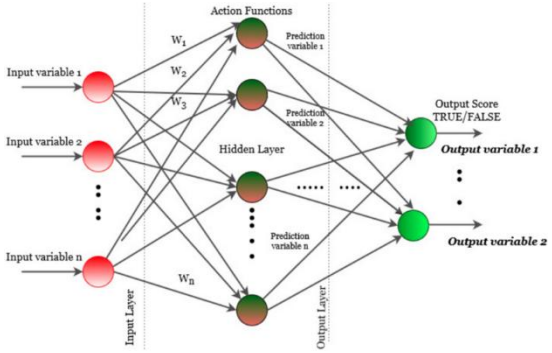


Figure 10 Architecture of a multilayer perceptron with three inputs, one hidden layer of four neurons, and two output neurons. This image courtesy of Naskath Jahangeer, Sivakamansundari Ganesan, and A. Alif Siddiqua Begum (Jahangeer, Ganesan, & Begum, 2022).

Figure 10 shows an MLP. The arrows show the direction of the connections between each element in the MLP, in other words, an input goes into a TLU in the hidden layer before proceeding to an output neuron. It is fully connected because each input is linked to each one of the TLUs in the hidden layer as well as the output neurons. This sort of network is called a feedforward neural network or FNN. A deep neural network is one that contains a deep stack of hidden layers. Care must be taken when discussing deep neural networks due to the rise in the computing ability of computers. During the 1990s an ANN with two hidden layers was considered deep. Currently, one can find neural networks with dozens to hundreds of layers (Géron, 2023).

Modern neural networks make use of a process called gradient descent, a fundamental step in their training. During gradient descent, the gradient or slope of the loss function is found. This can be used to determine the direction of the local minimum value of the loss function. Then the algorithm proceeds in the direction of the minimum to reduce it to zero. Seppo Linnainmaa discovered how to do this in multilayer perceptrons in 1970 by using a process called reverse-mode automatic differentiation. By going forwards and backwards through the network, he was able to observe how each parameter in the model affected the model's error. This enabled the model to adjust the weights and biases to reduce error. The combination of reverse mode-automatic differentiation with gradient descent is termed backpropagation. David Rumelhart showed how backpropagation can be used to

train artificial neural networks to learn useful internal representations by replacing the step function with an activation function called the sigmoid function or logistics function. During backpropagation, transfer functions are created to assign weights that highlight the most effective relationships between the input and output data. The hidden layers also find training errors and adjust the weights and biases to minimize those errors using gradients (Campbell & Wynne, 2011).

The logistics function is an irrational function that approaches "different horizontal asymptotes to the right and the left" (Larson, Hostetler, & Edwards, 2003, p. 234). A sigmoid function is any mathematical function that has a sigmoid or S-shaped curve. The sigmoid function is given by

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{7}$$

where z is given by Equation 3.

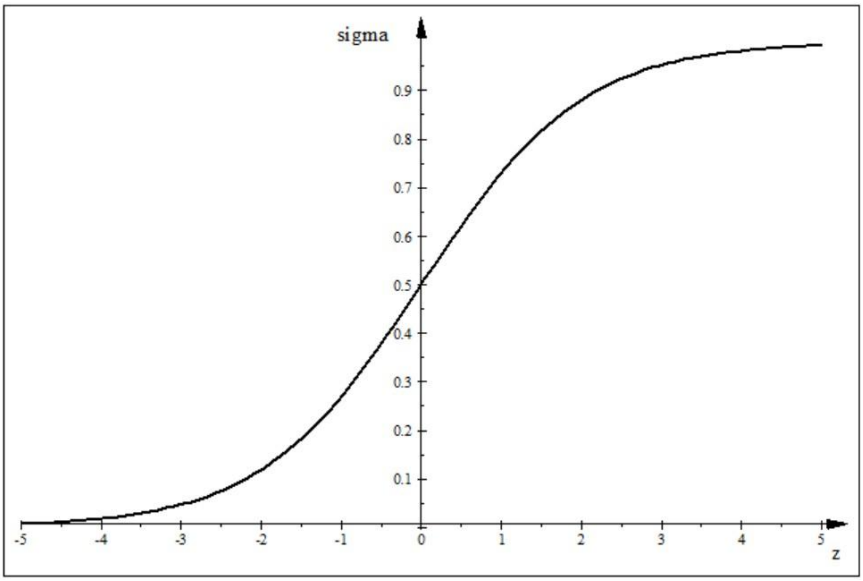


Figure 11 The logistic function. Image created by the author using Scientific Workplace.

As shown in Figure 11 the sigmoid function is so-named because of the S-shape. It is well behaved for all values of z.

Multilayer perceptrons can be used to perform classification. Each output neuron is capable of performing a binary classification using an activation function. With the sigmoid function it will yield results between zero and unity. For example, when performing landcover classification for a single

class, results greater than 0.5 would represent water while values lower than that would represent not water. To classify multiple classes, more output neurons would need to be added (Géron, 2023).

3.2 Artificial Neural Networks

There are a number of different AI architectures used in remote sensing. They are based on artificial neural networks, ANNs. ANNs are designed to mimic animal minds by mimicking neurons. They rely on a series of linkages between the input and the output. The input data can be anything from text to multispectral remote sensing imagery. The output consists of various classes dictated by the analyst. These networks rely heavily on training data which defines the relationship between the input and output data. When the artificial neural network is trained, it creates weights and biases within a set of hidden layers that link the input and output data. The more connections that the network makes between the input and the output, the better the weights and biases used in the hidden layers. During training these weights and biases are optimized such that the model can properly assign classes to features in the model.

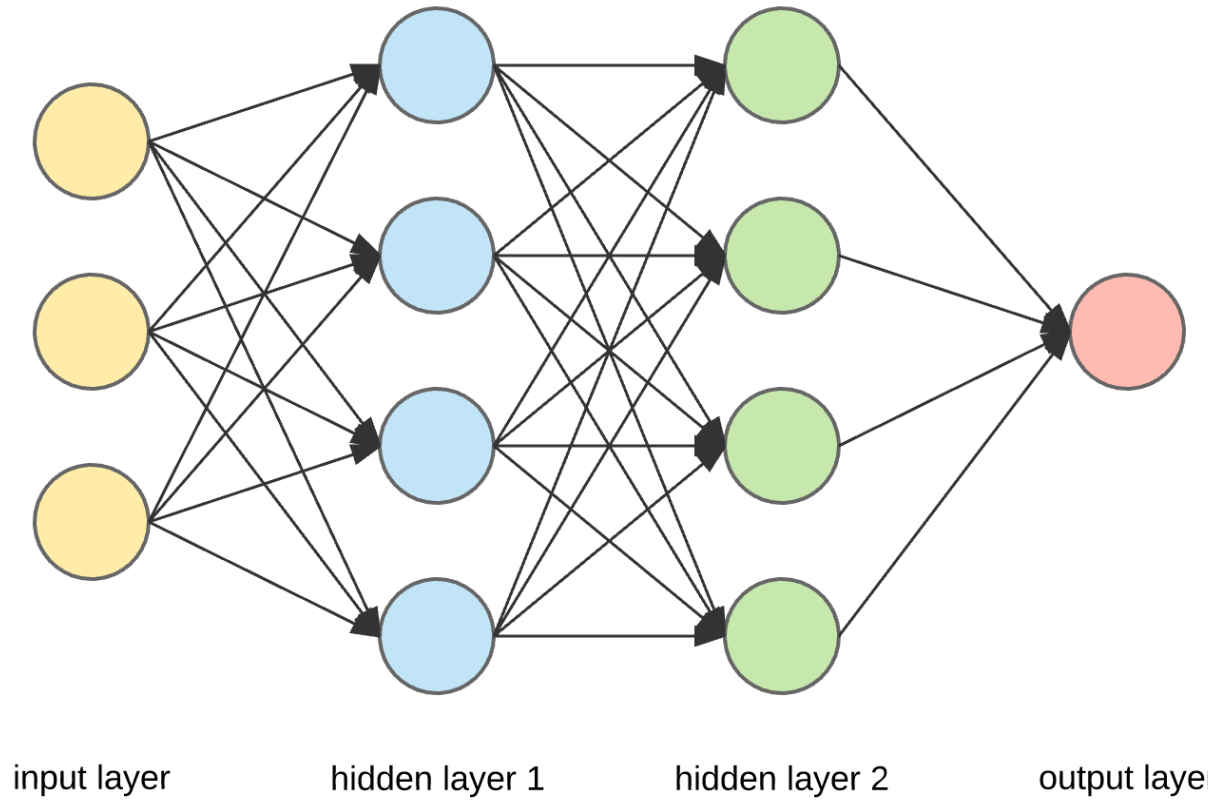


Figure 12 An image of an artificial neural network, ANN, with three inputs, two hidden layers composed of 4 neurons each, and a single output neuron. Image courtesy of Harsh (Harsh, n.d.).

Figure 12 shows an image of an ANN. The input layer consists of the input being fed into the model. The hidden layers and output layers are composed of neurons. The arrows represent the messages being passed from neuron to neuron. The direction of the arrows indicate that this is a feedforward neural network, FNN. Therefore, information from the input layer is passed through a number of hidden layers before going to the output layer where image classification occurs. The number of hidden layers, and the number of neurons in each hidden layer depends on the model in question. A general neuron can be defined by:

$$a = \sigma(\boldsymbol{w}\boldsymbol{x} + b) \tag{8}$$

where **w** denotes a weight vector, b is the bias, σ is the activation function, and a is the output. Each neuron takes a vector of inputs, **x**, and performs an operation to generate an output a (Maggiori, Tarabalka, Charpiat, & Alliez, 2017). Each arrow represents the weight from one neuron to the next. These weights are further optimized via backpropagation. During backpropagation the derivatives of the loss function are found with respective to weights. These derivatives are used to minimize the gradient. If the loss function is given by L then the weights are updated as

$$w_i \leftarrow w_i + \lambda \frac{\partial L}{\partial w_i} \tag{9}$$

where λ is the learning rate. Thus, backpropagation gives the model a way of updating the weights by the addition of a constant number multiplied by the slope of the loss function. A positive value of the slope means that increasing the weight will increase the loss and vice versa. A slope of zero implies that the weight has reached a minimum or maximum value.

3.3 Convolutional Neural Networks

The most popular neural network in remote sensing is the convolutional neural network, CNN. "CNNs consist of a stack of learned convolution filters that extract hierarchal contextual image features, and are a popular form of deep learning network" (Maggiori, Tarabalka, Charpiat, & Alliez, 2017, p. 645). As per ANNs they were designed to mimic the animal brain but in this instance they were designed to mimic the visual cortex.

David H. Hubel and Torsten Wiesel performed a series of animal experiments. They showed that neurons in the visual cortex can only react to a fraction of the total field of view called a local receptive field.

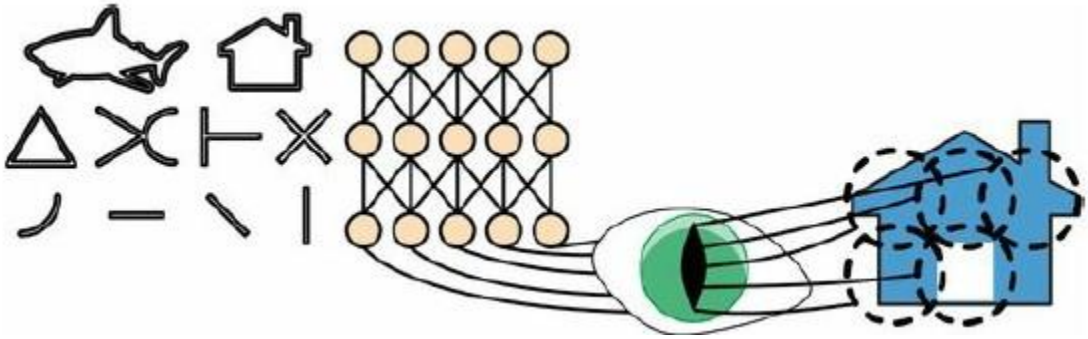


Figure 13 Biological neurons in the visual cortex respond to stimuli in specific patterns of the visual field called receptive fields. As the signal travels through the network of neurons it gets mapped to more complicated patterns via larger receptive fields until it becomes a recognizable pattern such as a house. This image courtesy of Aurélien Géron (Géron, 2023).

Figure 13 illustrates the concept of the receptive field. Each neuron has a field of view, or receptive field, illustrated by the dashed lines. These receptive fields of different neutrons combine to form the visual field of the organism. Hubel and Wiesel also showed that neurons with larger receptive fields react to the combination of patterns from the lower-level neurons. These neurons only interact locally, with nearby neurons, which led to the idea that higher-level-neurons rely on input from their lower-level neighbors. Furthermore, they discovered that the neurons acted like filters. Some neurons only responded to images of horizontal lines while others only reacted to images of vertical lines. These contributions led to the neocognitron, the predecessor of the convolutional neural network (Géron, 2023).

CNNs are a special form of neural network that associates every neuron with a spatial coordinate in the input image. The output $a_{i,j}$, can be computed as

$$a_{i,j} = \sigma\left(\left(\boldsymbol{W} * \boldsymbol{X}\right)_{\boldsymbol{i,j}} + b\right) \tag{10}$$

where $a_{i,j}$ is the output associated with the pixel located at point (i,j) in the image, $\mathbf{W}$ is a kernel with learned weights, $\mathbf{X}$ is the input to the layer and * is the convolution operation (Maggiori, Tarabalka, Charpiat, & Alliez, 2017). In this network, the size of the kernel determines the size of the

neighborhood in which connections to other neurons can be made and the same filter must be applied to each location in the image.

At the heart of a CNN is the convolutional layer. In a convolutional layer, each neuron is only linked to stimuli in their receptive field. In the first layer, a neuron is connected to pixels in its receptive field. In the second layer, a neuron is connected to other neurons in its receptive field and so on. This hierarchical construction not only mimics the visual cortex, and images, it also makes the network easier to train by reducing the number of parameters resulting in a regularized loss function.

The filters of a CNN also have a receptive field. These filters act as spatial feature detectors and produce a feature map. The simplest form of filter can be described in terms of the filters described by Hubel and Torsten, a vertical slit and a horizontal slit.

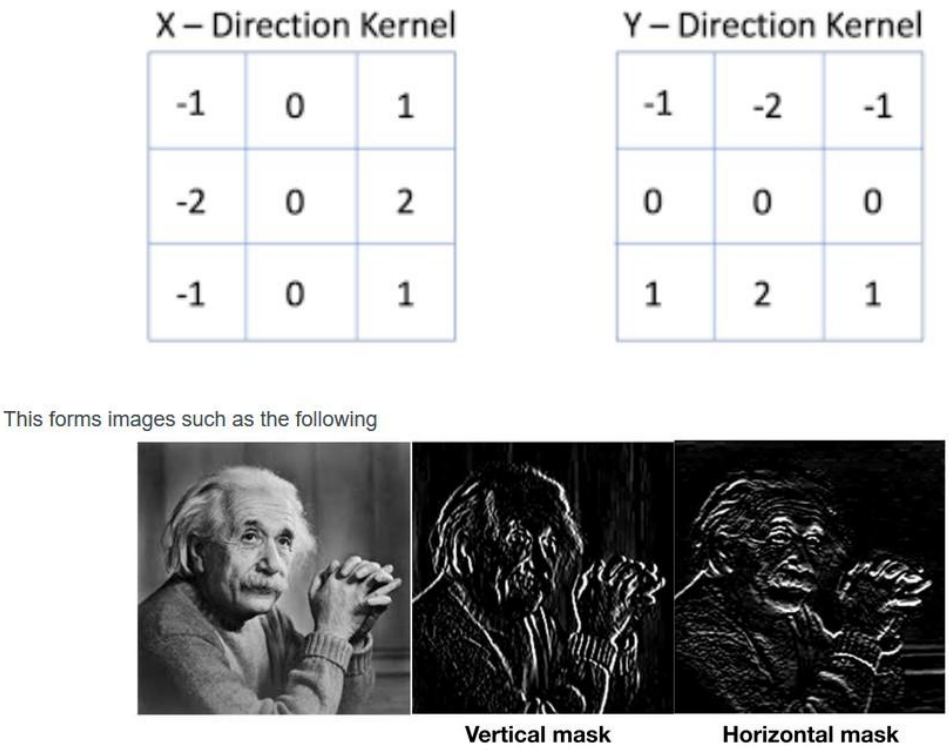


Figure 14 The top image shows a horizontal and vertical kernel. The bottom image shows an image and the results of each kernel on the image to produce a vertical mask and a horizontal mask. This image courtesy of Priyanshi Sharma (Sharma, n.d.).

Figure 14 shows the results two 3 x 3 kernels, a horizontal kernel and a vertical kernel. The X-Direction kernel multiplies pixel values to the left of the center by -1, -2, and -1, respectively. The values in the center don't contribute while the values to the right are multiplied by 1, 2, and 1,

respectively. After multiplication, the right side is subtracted from the left. This will produce a strong vertical edge if the pixels to the left and right are quite different. The image at the bottom labeled Vertical Mask is a feature map that shows the results of this kernel on the image. Similarly, the vertical kernel multiplies pixel values to the top of the center by -1, -2, and -1, respectively, while multiplying the bottom values by 1,2, and 1, respectively. The middle row doesn't contribute. Then the bottom is subtracted from the top to highlight horizontal edges when there is a large difference between the top and bottom rows. During training the model learns the best filters for each layer and learns to combine them into more complex patterns.

Convolutional kernels are 3-dimensional because they operate on all of the feature maps within a network. For example, if a RGB image is used, the kernel will need to cover all three channels in the image. A 90 x 90 RGB image will have a separate layer for each color so it can be represented by a tensor that is 90 x 90 x 3. If we apply a Sobel filter from Figure 14 then it would have the dimensions of 3 x 3 x 3, 3 x 3 for the filter and another x 3 for each channel in the image. This 3-dimensional kernel would produce a 2-dimensional feature map. Stacking those feature maps results in a 3-dimensional tensor. This in turn makes the convolutional layer itself 3-dimensional. Furthermore, a convolutional layer can have multiple kernels, each of which will generate a feature map. Each feature map is composed of one neuron per pixel, all of which share the same weights and biases. Every neuron in a given location (i,j) of a feature map will be connected to the outputs of the antecedent neurons in the same row or column in previous feature maps.

Another feature of modern CNNs is the use of downsampling. Downsampling increases the receptive field of neurons by reducing the resolution of feature maps. The goal of the convolutional neural network is for the upper layers to have a larger receptive field than the lower levels. There are two ways to accomplish this; by increasing the kernel size or downsampling the feature maps. Increasing the kernel size increases the computational cost of the model. Downsampling allows the model to minimize its computational cost while maximizing the receptive fields of the upper levels. They do this either by introducing stride or pooling layers. Pooling layers operate similarly to interpolation but in this case they are used to reduce the spatial size of feature maps, opposed to increasing the size. They accomplish this by either taking the maximum value in a region, maximum pooling, or taking the average value in a region, average pooling. Stride refers to skipping some

convolutions by applying the filter once to every four locations (Maggiori, Tarabalka, Charpiat, & Alliez, 2017).

3.4 Recurrent Neural Networks

In CNNs the output of one layer in the network is passed on to the next layer. This is the feedforward pattern that is inherent in all FNNs. Recurrent Neural Networks, RNNs, take the output of a layer and remap it back to the layer's input.

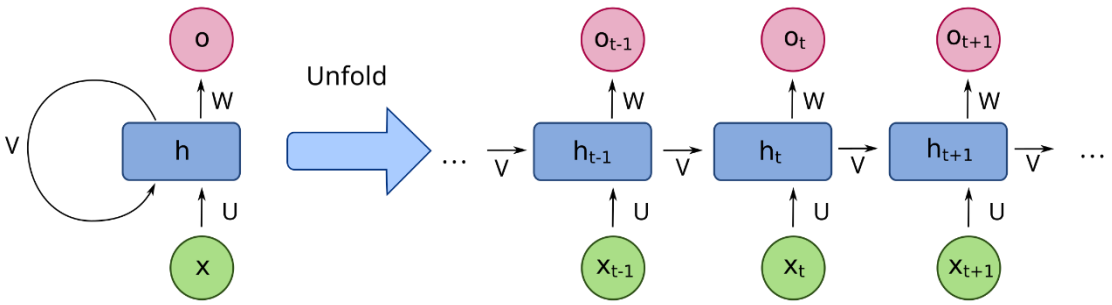


Figure 15 A recurrent neuron on the left. The right shows the same neuron unrolled through time. Image courtesy of fdeloche, licensed by CC BY-SA 4.0 (fdeloche, 2017).

This enables the network to remember observations. Figure 15 illustrates how the recurrent neuron sends information back to itself at the beginning of each timestep where x represents the input layer, h represents the recurrent neuron, and o represents the output layer. The weight matrices of the network are represented by U, V, and W respectively. U represents weight for the input to the hidden state. W represents the weight from the recurrent neuron from its past state to its present state. V represents the weight for the hidden state to the output state. To understand a recurrent neuron, we can look at an equation for its output at some time t, o_t.

$$o_t = O(V \bullet h_t + C) \tag{11}$$

Where O is an activation function such as the softmax function h_t is the state of the hidden layer, or in this case the recurrent neuron, at time t, and C is a bias. The hidden state is given by

$$h_t = \sigma(U \bullet X + W \bullet h_{t-1} + B) \tag{12}$$

where σ is an activation function such as the hyperbolic tangent, X is the input, and h_{t-1} is the state of the recurrent neuron in the last time step (Introduction to Recurrent Neural Networks, 2025). During each step, the recurrent neuron not only receives information from the input, it also receives information from its past self. At the first time step this is usually set to zero. Mapping the network against a time-axis is called unrolling the network through time because it shows the same RNN at different points in time.

RNNs have a form of memory because the output of any recurrent neuron is determined in part by the output of its previous time steps. The part of the RNN that preserves the memory across multiple time steps is termed a memory cell. Its state is given by Equation 12. However, when an RNN has a large sequence to process, it will forget some of the first inputs due to backpropagation. The goal of backpropagation is to minimize the loss function via gradient descent. In a recurrent neural network, as the gradient gets smaller and smaller, the inputs from previous time steps gets smaller and smaller. If one tries to retain that information then the gradients could blow up. This problem was solved by adding memory cells with long-term memory (Hochreiter & Schmidhuber, 1997).

The key to long-term memory is enabling the network to learn what information should be stored in the memory cell.

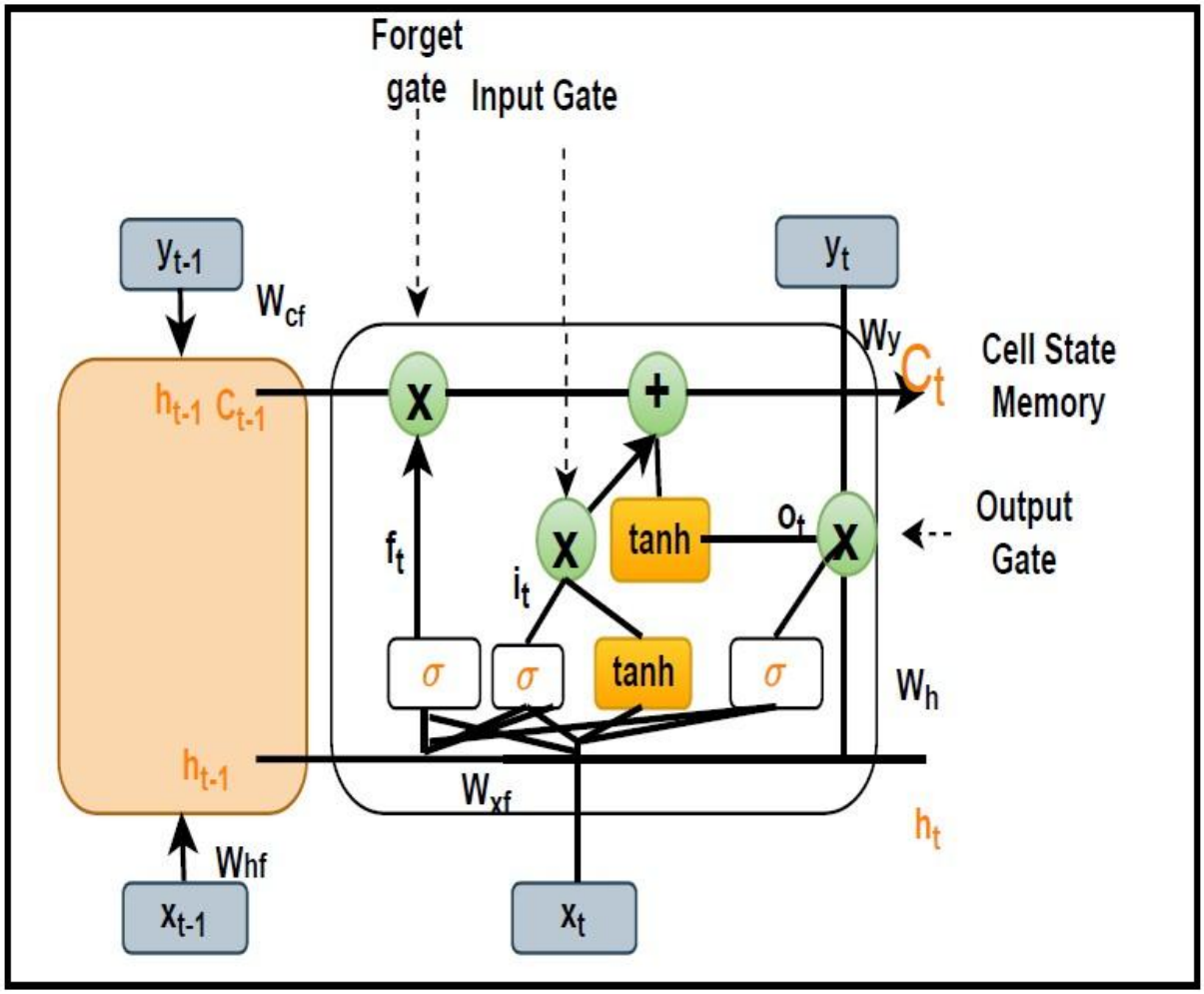


Figure 16 Complete representation of a Long Short-Term Memory cell. Image courtesy of Parul Madan, Vijay Singh, Devesh Singh, Manoj Diwakar, Bhaskar Pant, and Avadh Kishor (Madran, et al., 2022).

Figure 16 shows a Long Short-Term Memory cell or LSTM cell. The quintessential feature of the memory cell is that it has a forget gate. This gate enables the network to selectively determine which information isn't relevant to the task at hand. The plus operator allows the cell to store more information. In other words, during each time step some memories are forgotten while new ones are retained. The ability to selectively determine which information is forgotten and which is remembered led to advances in the field of natural language processing, NLP. Andej Karpathy showed how a RNN could be trained to generate Shakespearean text. His model generated the following sentence by imitating King Lear,

> O, if you were a feeble sight, the courtesy of your law,
>
> Your sight and several breath, will wear the gods

With his heads, and my hands are wonder'd at the deeds,

So drop upon your lordship's head, and your opinion

Shall be against your honour.

(Karpathy, 2015)

This quote is indistinguishable from Shakespeare. While they can generate meaningful text, one of their failings becomes transparent when they are tasked with translating from one language to another.

When tasking a RNN with translating a sentence such as “You ever think that maybe what you are is what’s trapping you inside whatever it is you’re trapped inside (Erikson, 2000, pp. 332-333)?” the RNN would run into difficulties. It might translate it as “Do you think that you’re stuck?”, which flattens the tone and nuance of Erikson’s prose. An RNN would focus on each word in the sentence separately rather than focusing on phrases within the sentence (Géron, 2023). The model is incapable of focusing on what is important in the sentence and transferring it over many time steps. Attention mechanisms enable the network to do just that.

3.5 Attention

Attention mechanisms were invented for natural language processing. The quintessential idea behind attention mechanisms is that input sequences have to be analyzed in context of the entire sequence rather than in a piece-wise fashion. Bahdanau, Cho, and Bengio proposed that the transformation of a sentence into a fixed length vector serves as a bottleneck. Their solution was to allow the model to automatically use context clues to predict a word. This process shortens the number of steps between an input and its translation thereby reducing the impacts of short-term memory on the model (Géron, 2023). The RNN is now able to focus on specific words in the sentence by applying different weights to different words in the sentence (Zhou). These weights come from an attention layer.

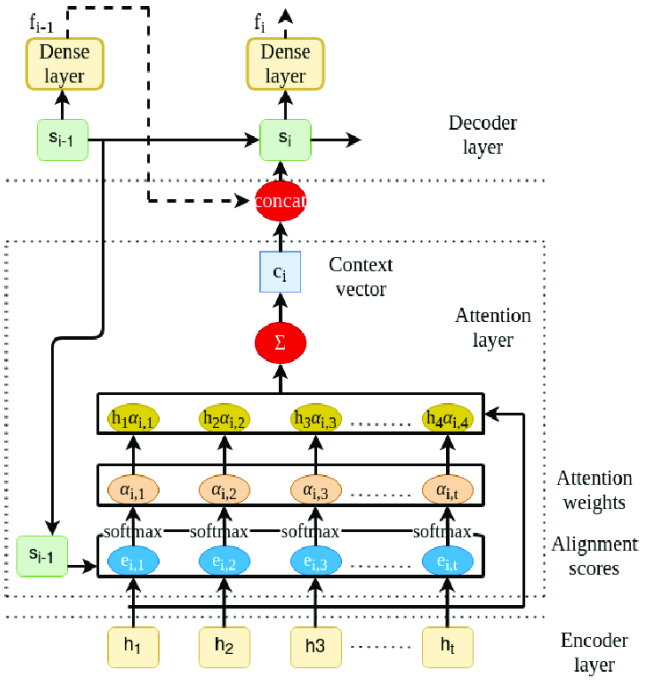


Figure 17 The architecture of an attention layer. This image courtesy of Chien-Ngyuen Nhu and Minho Park (Nhu & Park, 2022).

Figure 17 shows an attention layer in an RNN. It is composed of an encoder, an attention layer, and a decoder. The attention layer is comprised of a context vector, attention weights, and an alignment score layer. The encoder transforms the input into a feature representation. In the context of translation, it would take in a sentence and convert that to a sequence of vectors. The decoder takes the sequence of vectors and uses them to generate the translation. The purpose of the attention layer is to allow the decoder to choose which information in the encoder it needs. It does this by creating a new context vector, c, for each time step in the decoder. This context vector is a function of the hidden states of the encoder, h, and the hidden state of the previous time step. Attention weights, α, are assigned to all of those hidden states which allows the attention layer to give different weights to different parts of the input sentence. The attention layer enables the model to focus on those parts of the sentence that are most important (Nhu & Park, 2022).

In 2017 the transformer was introduced. It is an architecture based on a series of attention layers without any recurrence or convolution. Previously, the best networks employed an encoder and decoder layer connected via an attention layer as per Figure 17. Vaswani et al. were able to prove that transformers outperformed other models while allowing training to be parallelized across multiple GPUs.

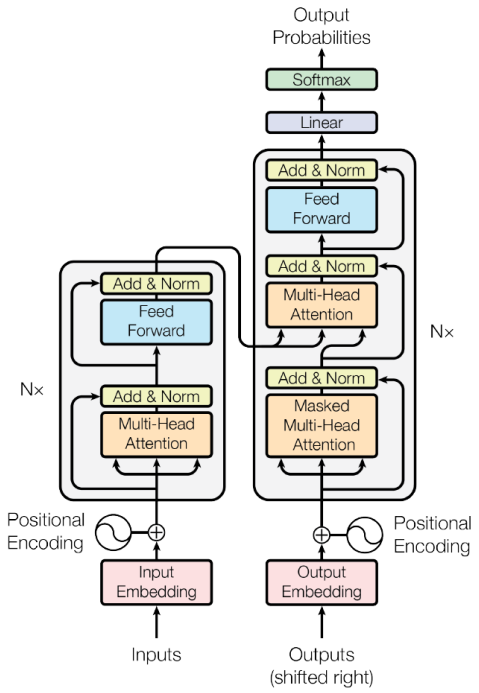


Figure 18 The original 2017 transformer architecture. This image courtesy of Vaswani et al. (Vaswani, et al., 2017).

Figure 18 shows the original transformer as per Vaswani et al. The left side of the image shows the encoder while the decoder is on the right. The encoder layer has two sub-layers, a multi-head attention layer and a feedforward network. The decoder layer is similar to the encoder layer but it has two multi-head attention layers instead of one. The extra multi-head attention layer acts as a mask to ensure that the model only has access to information from the past (Vaswani, et al., 2017).

The encoder creates feature representations of each part of the sequence, for each word a sentence that was used as input for translation. If we take Erikson's quote as an example, "You ever think that what you are is trapping you inside whatever it is that you're trapped inside", the encoder will process each word separately. "You" can refer to a single person or a multiplicity of people such as in "You made an A on your test" or "You guys". "You" can also refer to a specific person or a generic person such as in "You can see how easily the transformer model works". "You" can also be used archaically such as in "Get you gone" (you, n.d.). As the word "you" goes through the encoder, the feature representation will capture the exact meaning being used in the sentence.

The decoder takes the feature representation of each translated word into a feature representation of the word that follows it in the translated sentence. If we're translating Erikson's

quote to Bisaya “you” would translate to “ka” and “ever” to “labut pa sa”. The feature representation of “ka” would be transformed into the feature representation of “labut pa sa” and so on and so on until the entire sentence is translated.

After the decoder, the softmax function is a form of activation function. It will assign probabilities to the possible outputs of the translated sentence. The word or phrase that has the highest probability will be added to the translation.

The Add & Norm layer has two distinct functions. The “Add” refers to the addition of residual or skip connections. These residual connections add the original input of the sublayer to its output. This allows the model to train deeper while minimizing the vanishing gradient problem. The “Norm” refers to a layer normalization layer. This layer standardizes the feature representations of the inputs. This ensures that the model has a consistent scale and distribution across all feature representations.

The multi-head attention layer is so-called because it applies a number of parallel attention layers or heads. As per Figure 19 the multi-head attention layer consists of h separate layers. “Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions (Vaswani, et al., 2017, p. 6004).” The use of a single attention head would limit the receptive field of the layer.

Vaswani et al. used a form of attention termed Scaled Dot-Product Attention.

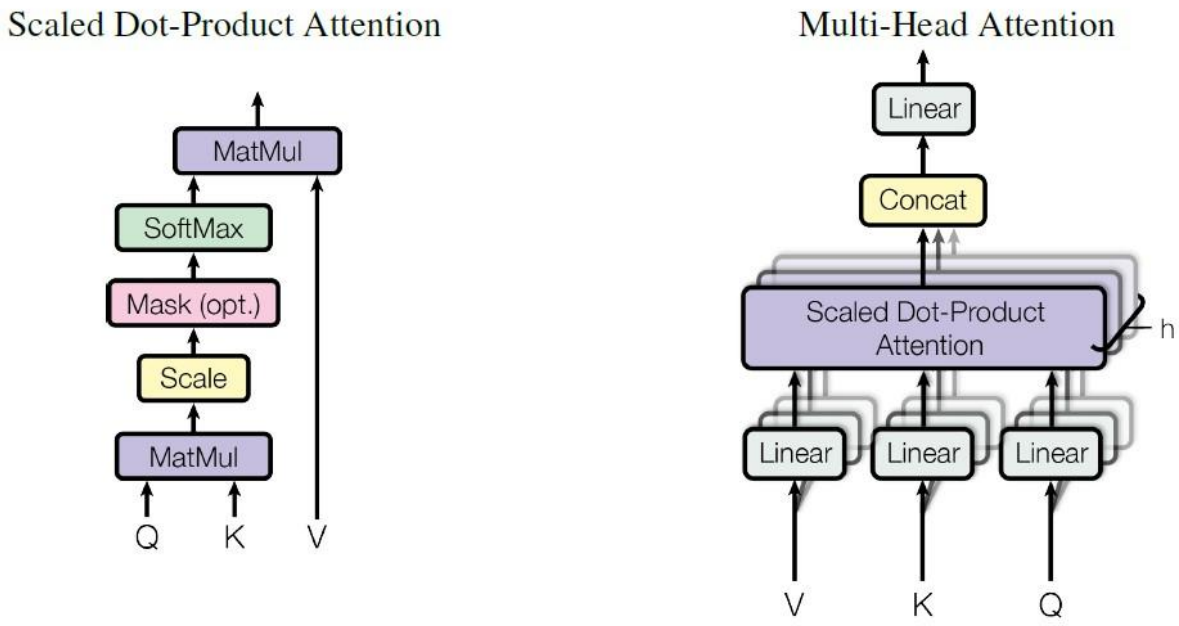


Figure 19 The left image shows Scaled Dot Product Attention. The right image shows a Multi-Head Attention Layer consisting of h layers. This image courtesy of Viswani et al. (Vaswani, et al., 2017).

This is illustrated by Figure 19. When an input sequence enters an attention layer, it gets transformed into three distinct feature representations, queries, keys, and values. Scaled Dot Product Attention takes an input of queries, Q, and keys, K, that are k-dimensional with values that are v-dimensional. The attention output is given by

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{k}}\right)V \tag{13}$$

where Q is a feature representation of the input, K is the feature representation of the keys and V is the feature representation of the values. As Q, K, and V are all matrices, K^T is the transpose of the K matrix (Vaswani, et al., 2017). $\sqrt{k}$ acts as a scale factor used to retain the gradients.

The encoder's multi-head attention layer modifies each feature representation by considering all of the words in the same sentence. The decoder's masked multi-head attention layer's function is similar to that of the encoder's but with a caveat. It doesn't attend to words that follow it. The decoder's multi-head attention layer enables the decoder to consider all of the words in the input

sentence. For example, when translating “ever” to “labut pa sa” it would pay lots of attention to both “you” and “think”. The positional encodings in Figure 18 store the location of each word in the sentence. Otherwise, the transformer would shuffle the individual words and thus lose the contextual relationships between the words.

3.6 Vision Transformers

The first use of transformers for vision related tasks involved using transformers for caption generation. Xu et al. used a CNN to process the image and generate feature maps. They then used a recurrent neural network with attention layers to generate the captions word by word. They were also able to understand their model.

Figure 20 The left image is an image that was input into the model. The right image shows where the model focused its attention when it generated the label “stop”. This image courtesy of Xu et al. (Xu, et al., 2015).

Since their model shows them what it is focusing on when it generates captions, every classification can show them how the model is behaving. They were able to “exploit those visualizations to get an intuition as to why those mistakes were made” (Xu, et al., 2015, p. 2056). This sort of explainability isn’t inherent in other models. This capability is illustrated by Figure 20 which shows a stop sign. The leftmost image is a normal stop sign and thee right image shows the stop sign has been highlighted because the transformer is focused on that part of the image.

Figure 21 The input image is on the left. The rightmost image shows the object of the model's attention. This image courtesy of Xu et al. (Xu, et al., 2015).

Figure 21 helps to illustrate this fact by showing us that the model thought that it was classifying a bird. As the pair of giraffes do resemble a bird in flight, it is obvious that the model needs more training to better differentiate birds from bird-like objects.

Dosovitsky et al. used a transformer to classify images. They did this by splitting an image into patches. The patches were then provided as a sequence to the transformer. The image patches were treated the same way as words in a natural language processing transformer. They found that the models underperformed when they were trained on medium size datasets such as the ImageNet. When they trained the model on larger datasets, their model was at its worst, on par with state of the art image recognition benchmarks.

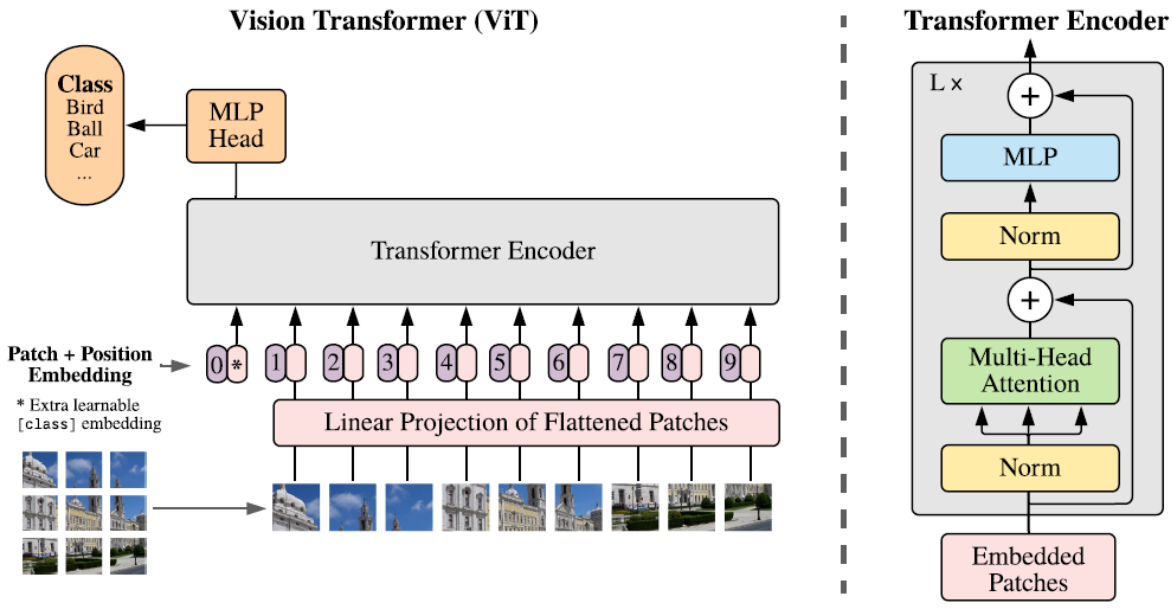


Figure 22 The original vision transformer. This image courtesy of Dosovitskiy et al. (Dosovitskiy, et al., 2020).

Figure 22 shows Dosovitskiy et al.'s vision transformer, ViT. They followed Vaswani et al.'s transformer as closely as possible. They cut each image into regular square patches termed token embeddings. The patches or embeddings are flattened from H x W x C to N x (P^2*C) where H, W, and C are the height, width, and number of channels in each input image. N is the number of patches given by

$$N = \frac{HW}{P^2} \tag{14}$$

where P is the resolution of the image patch. The number of patches is equivalent to the length of the input sequence for the transformer. They added a classification token to perform the image classification. The transformer has a depth of D. All of the image patches are flattened to this size to produce the so-called patch embeddings. A special token is created to the beginning of the patch embedding sequence in a process called prepending. This token becomes the feature representation that is used as a summary of the entire image. It gets a classification head in the form of a multi-layer perceptron, MLP, that has a hidden layer at pre-training and a single layer for fine-tuning. Standard one-dimensional position embeddings are then added to the patch embeddings.

Zhai et al. have found that weight decay has a large impact on model accuracy for models trained on small amounts of data. Weight decay is a form of regularization. Regularization refers to any changes in the training algorithm that reduce the generalization error in the model. Generalization error is also known as validation error; it is the error that the model makes when it is tested against new data (Wikipedia Contributors, n.d.). Weight decay is a form of regularization. It is given by

$$w = (1 - \lambda)\omega - \alpha\Delta C_0 \tag{15}$$

where λ is the decay factor, C_0 is the unregularized cost function, ω is the weight, and α is the learning rate (Mishra, 2020). The purpose of weight decay is to minimize the weights of the model by multiplying them by the decay factor. Zhai et al. discovered that decoupling the weight decay from both the final layer and weights improved the performance of the model.

I opted to use a vision transformer for my model because of the attention mechanism. They are able to focus on the spectral and contextual information in the image and find relationships between them. Thus, they should be able to use my remote sensing thresholds to perform landcover classification. In effect, they should be able to be able to identify water, for example, based on its spectral signature and then explain why this is water and not bare earth to the analyst.

Chapter 4 Data Preparation

The training dataset was made from Landsat 8 & 9 OLI scenes. The scenes were gathered from January of 2014 through December of 2024. All scenes were limited to the continental USA to make it easier to train the model. An initial search was performed on Wikipedia to find the largest lakes, national forests, deserts, grasslands, cities, and the snowiest places in the continental USA. From this list sites were selected based on their size and their optical clarity. Because the remote sensing indices are being used, I had to ensure that the sites were pure enough to be easily classified using remote sensing indices. While it is possible to classify any scene with the indices, the volume of images being analyzed limited the amount of time that could be used to classify a particular image.

I used USGS Earth Explorer to find all of the Landsat scenes. I tried to find scenes with minimal cloud cover over my selected regions.

4.1 Workflow

Figure 23 A flowchart illustrating the steps taken to process the images.

Figure 23 shows a brief summary of the steps involved in processing the images. The following paragraphs offer a more of an expanded view of the workflow.

All of the images were manually selected to be squares with about 90 pixels on a side. To clip them I first created a feature dataset to store all of the boundaries. Then I populated the feature

dataset with feature classes or shapefiles. These shapefiles will be the boundaries that I intend to use to clip the Landsat scenes. First, I created a point in the scene, then I used the Direction-Distance tool to create another point 90 pixels away. Because the tool doesn't accept pixel measurements, I had to convert the pixels to meters by multiplying by $\frac{30m}{1\ pixel}$ resulting in 2700 meters. This allowed me to get most of the sides to be 2700 meters exactly, however the final point wouldn't always align properly resulting in some clips that weren't exactly 90 square pixels.

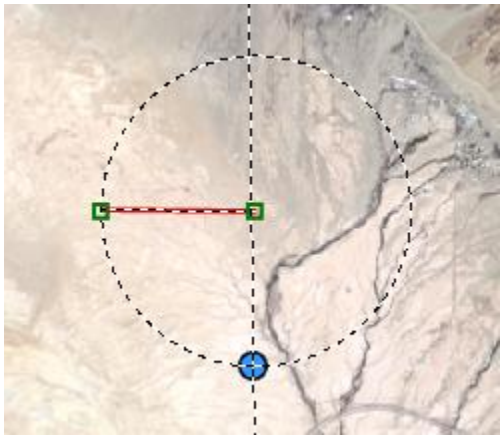


Figure 24 A clip from Landsat 9 Collection 1 Level-1 LC09_L1TP_039037_20200828_20240825_02_T1in ArcGIS Pro showing the use of the Direction-Distance tool. This image courtesy of (U.S. Geological Survey, 2024).

Figure 24 shows how I used the Direction-Distance tool to create my boundaries. I determined the direction by eyeballing the screen and trying to get the lines to line up evenly. I wasn't initially aware of this tool so some of the earlier images might be off due to my manually trying to create the boundaries point by point.

The images were all loaded into ArcGIS Pro to clip them prior to classifying them in eCognition. I used the Geoprocessing tool Composite Bands to load in the bands that I wanted for the image, namely, bands 2, 3, 4, 5, 6, 7, 10, and 11.

Band 2	Blue
Band 3	Green
Band 4	Red
Band 5	NIR
Band 6	SWIR1

Band 7	SWIR2
Band 10	TIRS1
Band 11	TIRS2

Table 2 A table showing the band designations for Landsat OLI 8 & 9 that were used in this thesis. The band designations were adopted from (USGS, 2025).

Table 2 shows the band designations of Landsat 8 & 9 OLI and TIRS. I used the same order to create composite images in ArcGIS Pro.

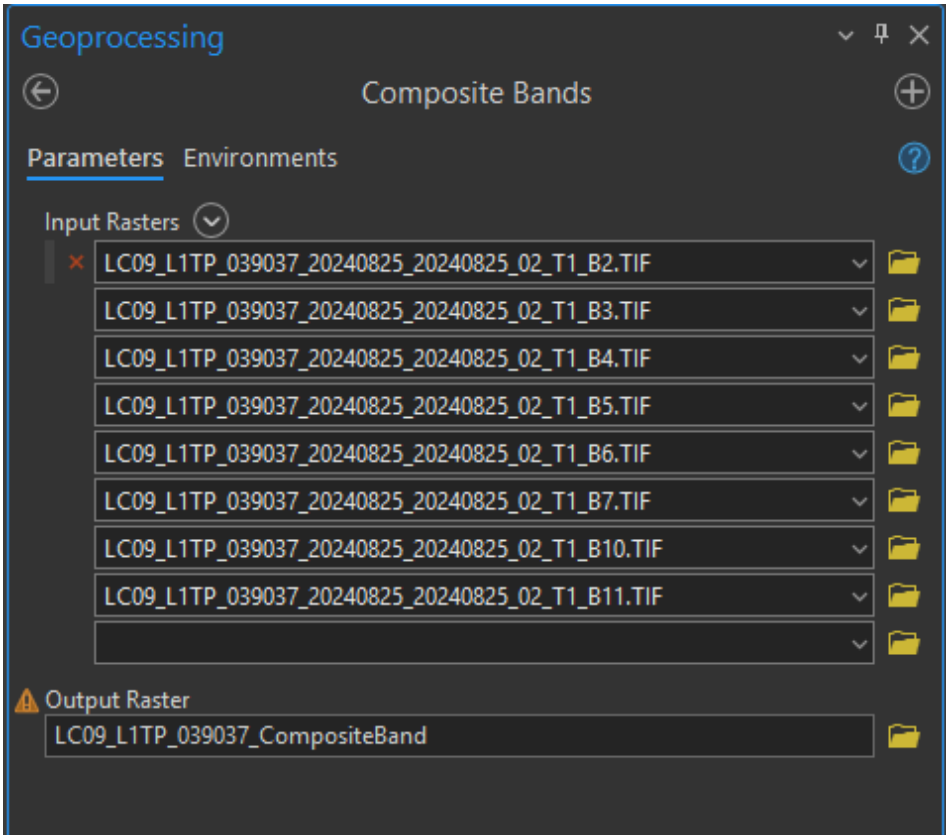


Figure 25 An image taken from ArcGIS Pro showing how the composite bands were created.

Figure 25 shows how I created composite images from the various Landsat bands. Care had to be taken to ensure that the order of the bands was identical for every composite image that I created. They were created in the same coordinate system as the original Landsat bands, however, since the coordinate system isn't required for the attention model I won't go into details about it here.

The next step is to select the sites and clip the rasters. Before I selected the sites, I used ArcGIS Pro to display the image using either RGB or CIR. I noticed that RGB was fine for the water images while CIR was good for identifying crop, forest, grassland, and bare earth.

Figure 26 A clip from Landsat 8 LC08_L1TP_025033_20240612_20240628_02_T1 in ArcGIS Pro in CIR. This image courtesy of (U.S. Geological Survey, 2024).

Figure 26 was displayed in CIR, color infrared, to easily detect the vegetation in the image. CIR emphasizes vegetation by displaying it in bright red. Bare earth appears gray or tan while water appears blue-black. CIR is created by mapping the NIR band to the red channel, red to the green channel, and green to the blue channel. Furthermore, displaying the image in this band combination ensures that I can use the remote sensing indices to classify the image.

The images were clipped using the Geoprocessing Tool Clip Raster.

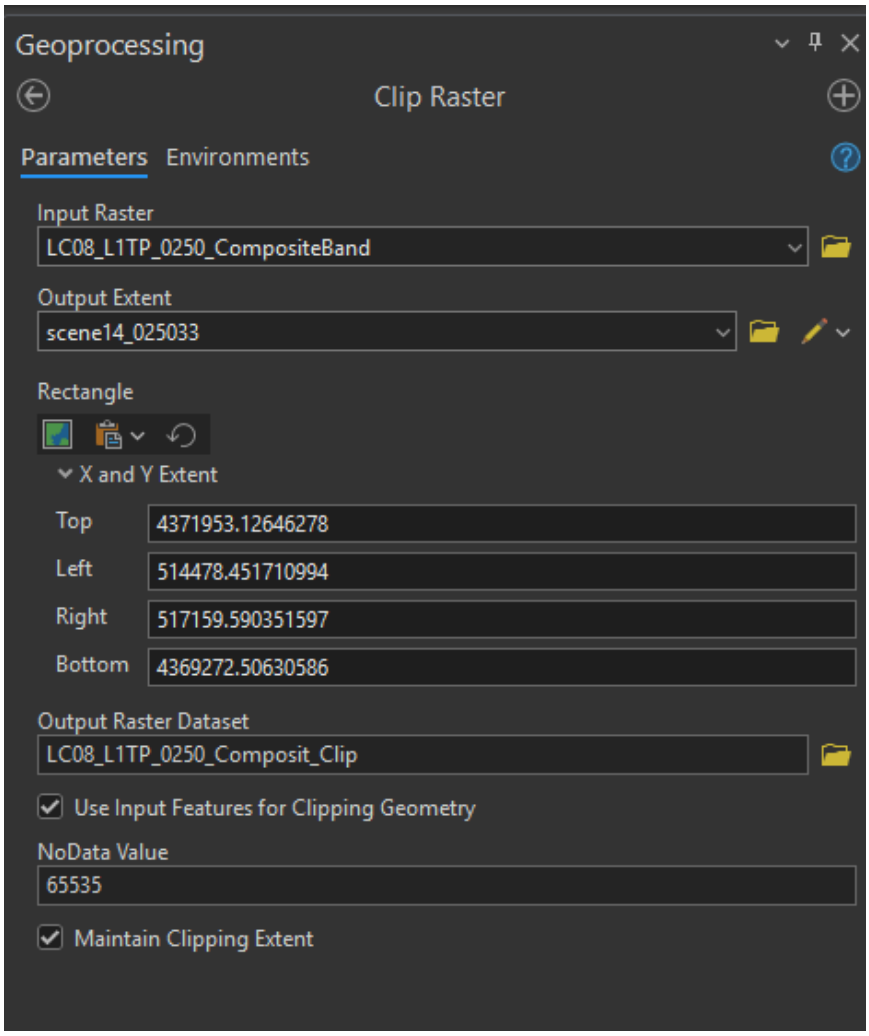


Figure 27 shows the Geoprocessing Tool Clip Raster in ArcGIS Pro.

Figure 27 shows the settings that I used in Clip Raster. The Input Raster is the composite raster created from the separate bands of the Landsat image. The Output Extent is the feature class boundary that was created around the areas of interest. The box, Use Input Features for Clipping Geometry, is used to ensure that the images are clipped exactly to the geometry of the boundary. The Maintain Clipping Extent is used to ensure that images are clipped to the exact extent of the boundary.

I adopted a naming convention in which the row and column numbers were used along with "scenex" where "x" was a number that I used to help me keep track of the number of scenes.

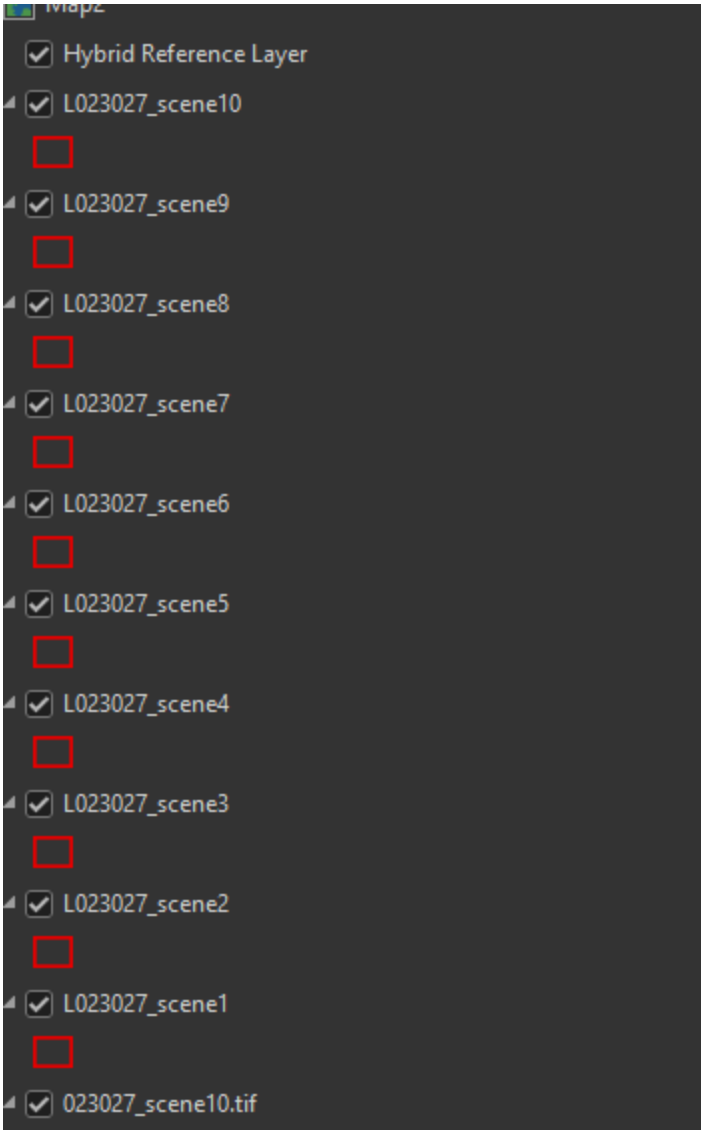


Figure 28 An image taken from ArcGIS Pro showing the contents pane. The naming convention that I used shows the row and column numbers of the Landsat scene followed by "_scenex".

Figure 28 shows how I used the naming convention for all of the boundaries that I created to clip the images. I used the same naming convention for all of the clipped images, however, I removed the "L" from the front of the names. I assumed that it would be easier to manipulate the clipped images in Python if all files were named using the same naming convention.

After the images were clipped, they were classified using eCognition. The images were loaded into eCognition using the same band pattern as shown in Figure 25. I then added all of the remote sensing indices to eCognition and the various landcover classes. Next the images were classified by using thresholds to select the various landcover classes. After the image was classified, it was saved as a shapefile.

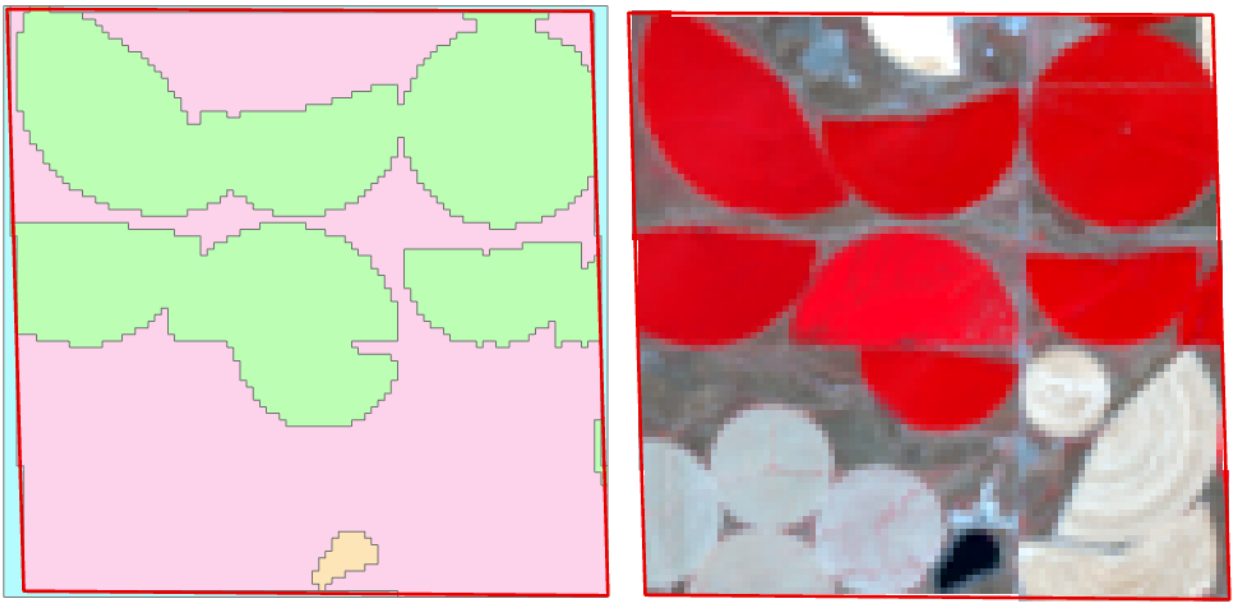


Figure 29 A side-by-side comparison of an image classified by eCognition and a CIR composite of the same image. In the classified image, green denotes crop, pink denotes bare earth, orange denotes water, and blue denotes no data. The CIR composite shows crop in red, bare earth in shades of gray and tan, and water is in black. The CIR composite was created from LC09_L1TP_042030_20230727_20230728_02_T1 (U.S. Geological Survey, 2023).

Figure 29 shows the results of classification from eCognition. The image on the right shows the original Landsat scene in CIR which emphasizes vegetation and water. The image on the left was classified using NDVI with thresholds of -1.0,0 for water, 0,0.31 for bare earth and 0.31,1.0 for crop. There are some regions of the CIR composite that have streaks of red which indicate some potential crop growth but in the interest of time those regions were classified as bare earth. Also, there are some structures in the CIR composite but they were also classified as bare earth in the interest of time. Other images were similarly classified.

4.2 Site Selection

Lake Okeechobee, Lake Michigan, Lake Superior, Lake Erie, Great Salt Lake, the Salton Sea and regions of the Atlantic Ocean near Florida were chosen for sources of the water sites. Some images were taken from smaller waterbodies that were in the same Landsat scene such as Pymatuning Reservoir in Pennsylvania or Gull Lake, and Hardy Dam Pond in Michigan. Some Landsat scenes had multiple waterbodies in them which allowed me to get more diverse scenes for the dataset from the same scene.

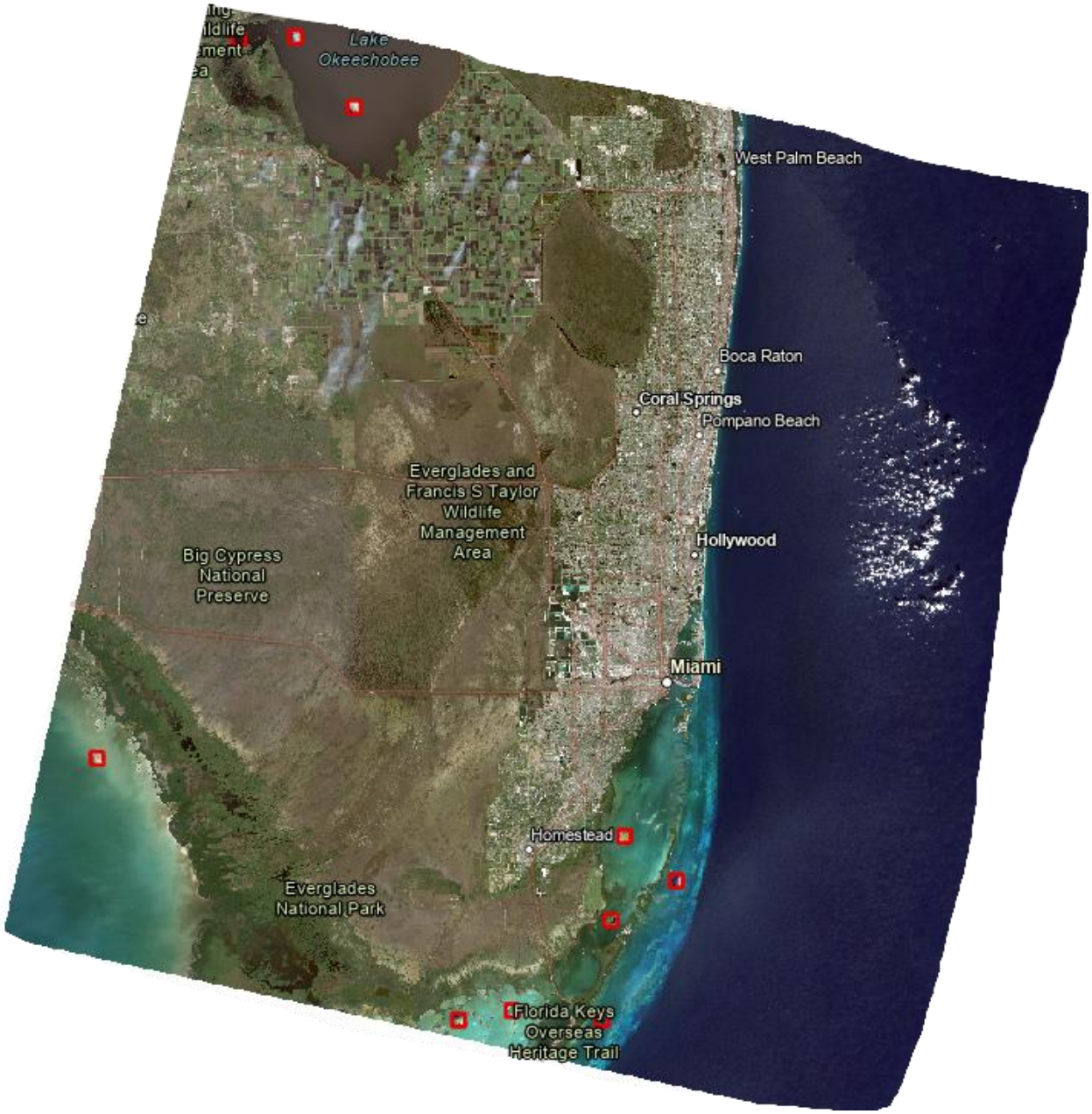


Figure 30 Landsat 8 Collection 1 Level-1 scene LC08_L1TP_022030_20230621_20230630_02_T1 showing a portion of the southern tip of Florida. An ESRI basemap was overlayed to display the names of features in the image. Sites selected for analysis are highlighted in red. This image courtesy of (U.S. Geological Survey, 2023).

Figure 30 shows one of the scenes that I used to select my water images. My primary target was Lake Okeechobee, however the waters of coastal Florida had different colors due to a variety of parameters such as fertilizer runoff, algal blooms, and depth. I could have just chosen to focus on Lake Okeechobee but since the lake's water is mostly uniform, I opted to select from multiple regions in the image to aid the model's ability to identify different types of water. It was also difficult to find scenes that were perfectly centered over the area of interest and thus selecting images from different parts of

the scene could help to balance any sort of distortions or degradations caused by the Landsat satellites. With waterbodies such as the great lakes, I opted to select images from both sides of the U.S./Canada border for similar reasons. Some of the smaller, secondary waterbodies such as Pymatuning Reservoir were large enough to clip out regions that were mostly water. Waterbodies that weren't large enough were ignored in the interest of time.

I selected forests by looking at the largest national forests in the country by area. The sites that I selected were from the George Washington National Forest, Jefferson National Forest, Quachita National Forest, Targhee National Forest, Boise National Forest, Payette National Forest, and Mt. Baker National Forest. The Landsat scenes were collected from May through September to ensure that the leaves were in full bloom. Care was taken to ensure that the clipped images were optically pure, however some of the images had portions of roads or buildings. This is exemplified in Figure 31.

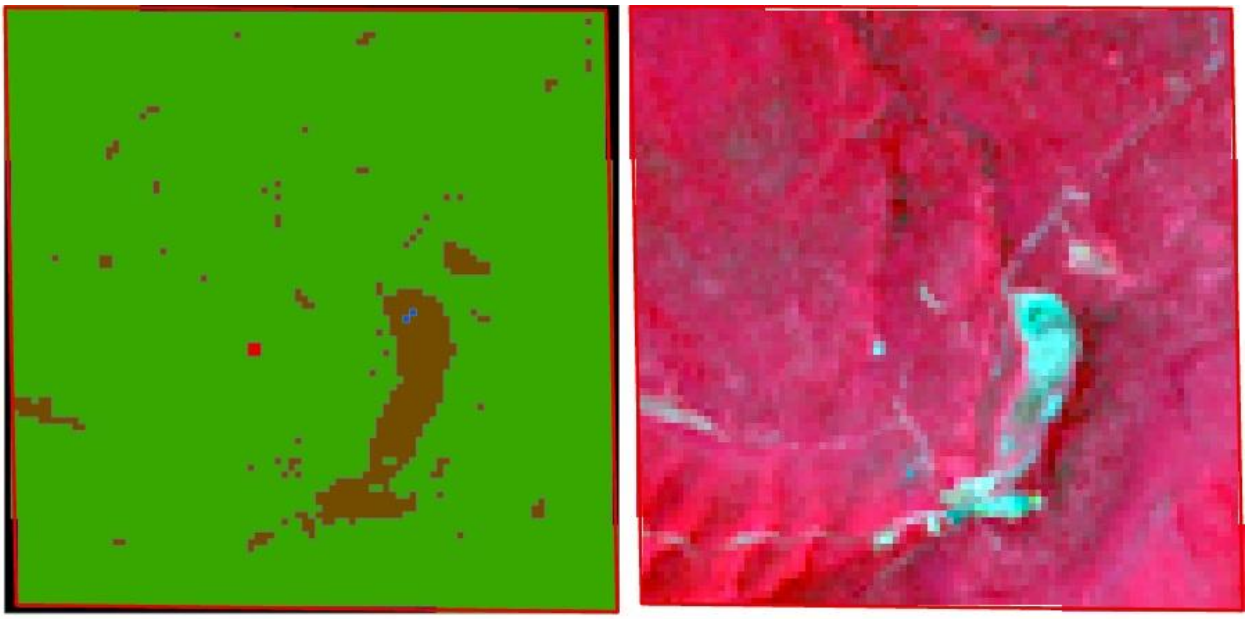


Figure 31 A side-by-side comparison of a classified image and Landsat composite CIR image of the same scene. Forest is shown in green, bare earth in brown, urban in red, water in blue, and no data in black. The Landsat image was created from LC08_L1TP_017034_20230618_20230623_02_T1 (U.S. Geological Survey, 2023).

Here we can see grayish streaks in the image that are roads, however they were hard to classify. In the interest of time some of the pixels were classified as bare earth while others were classified as forest. This image was classified with the following thresholds as per Table 3.

Landcover class	Remote Sensing Index	Range
water	MNDWI	0, 1.0
bare earth	BSI	-0.2, 0.0
forest	NDVI	0.46, 1.0

Table 3 Remote sensing thresholds used to classify the CIR composite in Figure 31.

I was able to classify the water using MNDWI, bare earth with BSI, and forest with NDVI. As per Table 3, water was between 0 and 1.0 for MNDWI, bare earth was between -0.2 and 0.0 using BSI and forest was between 0.46 and 1.0 using NDVI. I wasn't able to delineate between bare earth and the urban structure, so I manually classified it as urban because it was obviously a structure surrounded by forest. The remaining forest sites were similarly classified.

Sites for cropland were identified by first searching for the largest farms by area in the country. This lead to articles such as O'Keefe's article on agriculture.com (O'Keefe, 2019), (Farms, n.d.), (Whiteclouds, n.d.). After getting a general idea of where to look for the farms, I was able to find suitable sites on USGS Earth Explorer. I tried to find the Simplot family farmland in western Idaho and eastern Washington. The Boswell family farmland in the San Joaquin Valley, the Fanjul family farmland in the Florida Everglades Agricultural Area, the Offutt family farmland based in Fargo, North Dakota, and the farmland of Stewart & Linda Resnick in Kern County California and Texas. I limited my search to the months of May-July because these are the months when crops tend to be in full leaf. However, those months are a generalization and certain crops are in full leaf outside of that range. For example, winter wheat in western Kansas shows full leaf from May-July while corn matures between August and September in the Midwest (Campbell & Wynne, 2011).

Sites for bare earth were selected based on an internet search for the largest deserts in North America (List of North American Deserts, 2025). I used the Chihuahuan Desert, Sonoran Desert, Mojave Desert, Great Basin Desert, and Central Basin and Range. I took care to select sites with minimal vegetation. While the Landsat imagery appeared to show bear earth, displaying the images in CIR detected traces of vegetation in some of the locations.

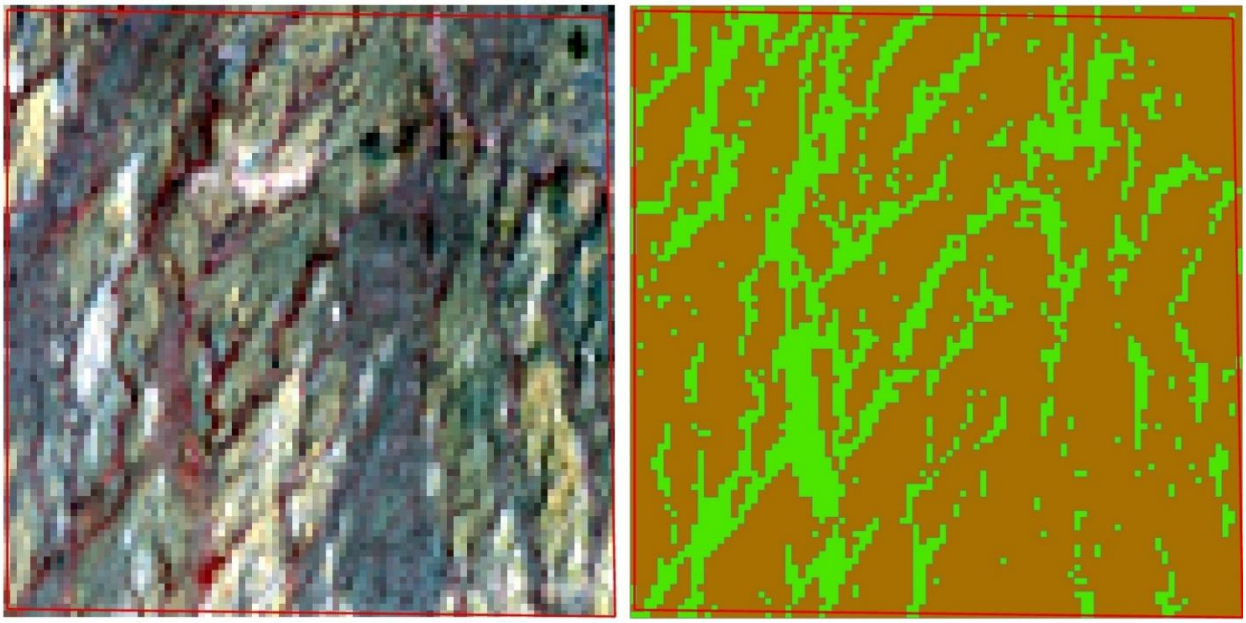


Figure 32 A side-by-side comparison of a CIR scene from Landsat imagery of the Sonoran Desert with a classified image of the same scene. In the classified image, green represents vegetation and brown represents bare earth. The image was created from LC09_L1TP_037037_20230708_20230709_02_T1 (U.S. Geological Survey, 2023).

Figure 33 A true color image of a scene from the Sonoran Desert. This image was created from Landsat 9 imagery LC09_L1TP_037037_20230708_20230709_02_T1 (U.S. Geological Survey, 2023).

Figure 32 and Figure 33 show the same scene of the Sonoran Desert. Figure 33 shows the scene in true color while Figure 32 shows the same scene in CIR and a classified version of the image. The true color image shows a scene that looks devoid of life. It is completely brown, which is what one would expect of a scene from the Sonoran Desert. However, the color infrared image shows some

streaks of red in the image. These streaks are possibly arroyos with vegetation growing alongside. This sort of vegetation was classified as grassland unless I had sufficient cause to classify it as forest.

Locations of grasslands were chosen based on an internet search of the largest national grasslands (National Grassland, 2025) (The Largest National Grasslands In The United States, n.d.). I selected the grasslands that were the largest by area, namely, Little Missouri National Grassland, Buffalo Gap National Grassland, Thunder Basin National Grassland, Butte Valley National Grassland, Pawnee National Grassland, and Grand River National Grassland. Scenes were compiled from their surroundings. Grasslands were hard to classify because they were harder to identify in satellite imagery.

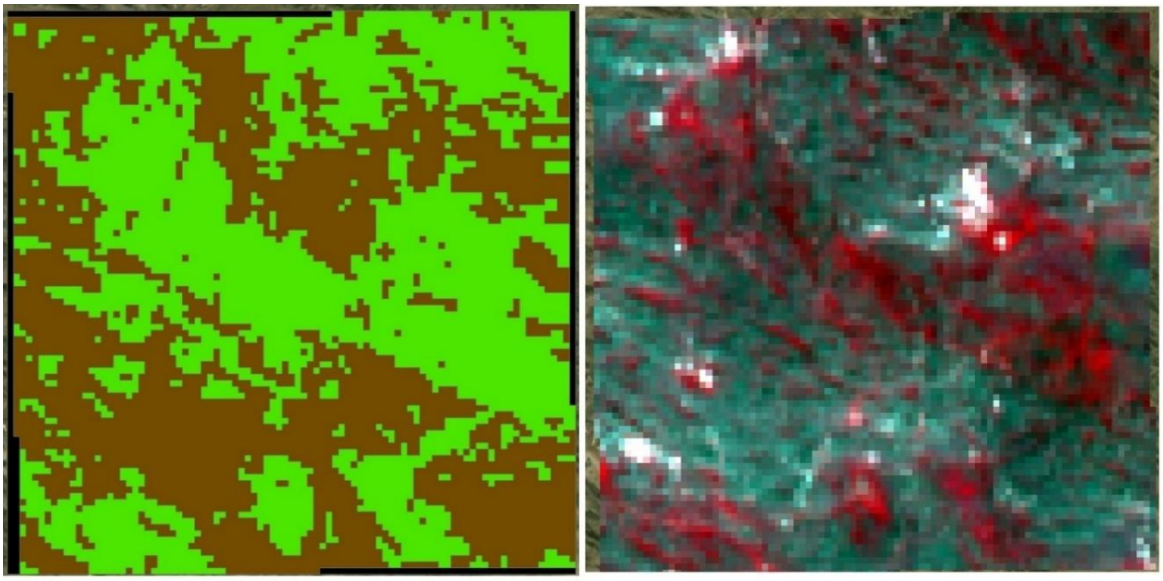


Figure 34 A side-by-side comparison of a classified image and a CIR composite of a Landsat scene. Vegetation is displayed in green and bare earth in brown. The images were created from LC09_L1TP_032030_20230619_20230619_02_T1.

Figure 34 shows images from a Landsat composite CIR image and a classified image of a scene from Dry Valley Nebraska. For this classification, BSI and SAVI were used to classify the image. I was able to use Google Maps to verify that the vegetation was grassland.

Figure 35 A street view image from Google Maps showing Dry Valley Nebraska as a grassland. This image was captured during September of 2023. (Google, 2023).

The use of Google Street View helped to confirm vegetation as grassland opposed to forest. However, the use of Google Street View was limited in that it only allowed vies from the road and in some situations, as pictured in Figure 35, one cannot move the view down the road. Figure 35 was captured at the intersection of Highway 97 and Dry Valley Road, however, Figure 34 was about 4 miles down Dry Valley Road.

Cities were selected based on an internet search of the largest cities by area in the United States of America (Wikipedia, 2025). The cities that I used were New York City, Los Angeles, Chicago, Houston, and Phoenix. The cities were easy to find, however they had mixed landcover classes. Some of the pixels themselves seem to be mixed.

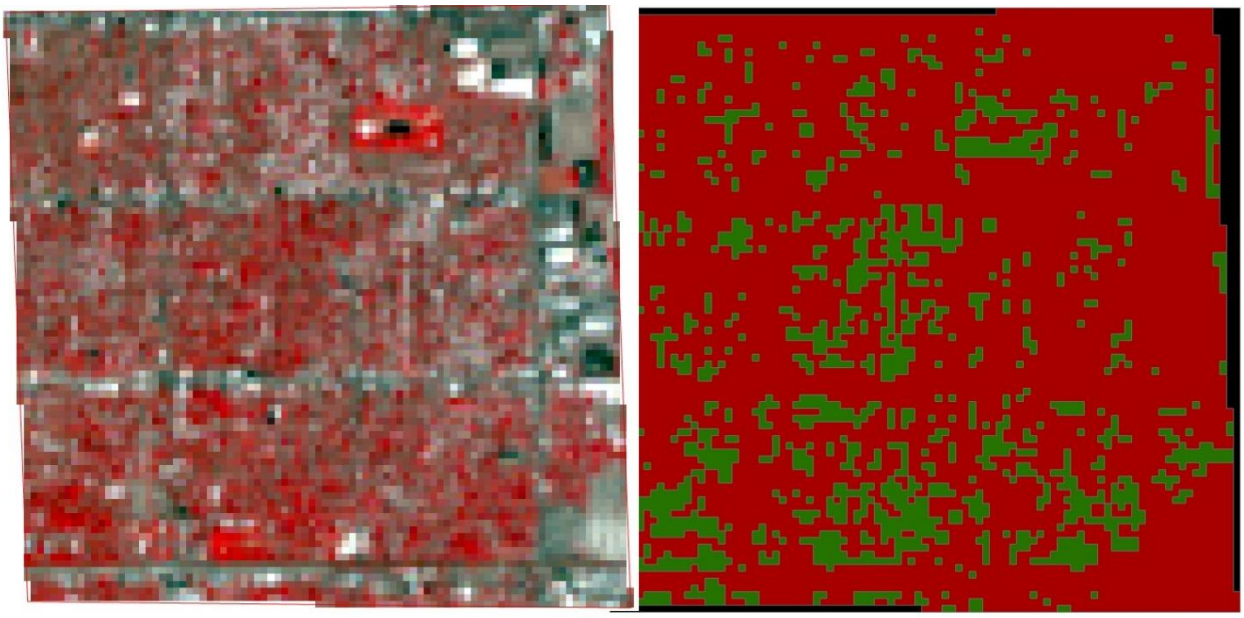


Figure 36 A side-by-side comparison of a CIR Landsat scene and a classified image. In the classified image, vegetation is displayed in green and urban features are displayed in red. The Landsat scene was created from LC08_L1TP_022031_20230621_20230630_02_T1.

Figure 36 shows a set of images of a suburb of Chicago. In a suburban setting, there tends to be lots of vegetation because people like to have grass filled yards with shade-trees as well as parks. This makes it hard to extract features from the image. For this particular image, urban features were classified using NDBI from -0.1, 0.35. Forest was classified using SAVI from 0.25 to 0.82. In general, all vegetation in urban areas was assumed to be forest unless I could confirm that it was grass such as a park or baseball field.

Sites with snow were selected based upon an internet search of the snowiest places in the United States (Wikipedia, 2025). The locales that I selected were in and around Paradise, Washington, Timberline lodge ski resort in Oregon, Alta Utah, Soda Springs California, and Valdez, Alaska. Images from these sites were selected during the months of December-March to ensure that snowfall was present.

Figure 37 Shows a Landsat scene in true color. The Landsat scene was created from LC08_L1TP_026028_20230225_20230301_02_T1.

Figure 37 shows a Landsat image of Grindstone Lake in Northwoods Beach, Wisconsin. I intended to classify this image as snow and bare earth with any non-snow feature being classified as bare earth. However, I didn't have time to classify the snow images.

I have a total of 50 snow images, 52 grassland images, 50 urban images, 49 bare earth images, 50 forest images, 51 crop images, and 55 water images. Because the snow images weren't classified, the dataset consists of 307 images, each of which was about 90 pixels squared. Due to the small size of the dataset, and the no data regions around some of the images, data augmentation was used to clip

out smaller images and create additional images by translating and flipping the images. Each image was augmented to produce 10 images, resulting in 3,070 images for training the model. This number was chosen to ensure that my setup would be able to train the model.

I used eCognition to classify the images. The classified images were output as shapefiles. These shapefiles had to be converted to raster images. I used ArcPy's FeatureToRaster function to convert the shapefiles to raster images (Esri, 2024). I set the cell size to 30 because that was the ground sample distance of the original Landsat images.

4.3 Image Augmentation

Image augmentation is the use of small, random changes in an image to create new images based. These random changes can take the form of changes in color, orientation, flipping, and cropping. It is a technique that can be employed to expand a limited dataset because neural networks tend to require large datasets. My code performs horizontal or horizontal flips 50% of the time. Translations of up to 10% are done 50% of the time. Setting scale_limit = 0 ensures that all of my images have the same ground sample distance of 30 meters. I set rotate_limit to 0 because I didn't want to risk any of my images having too much of the no data category. I didn't consider changes in color because the goal of my research is to train the model to classify images based on spectral values. Cropping wasn't considered because my images have to be the same size and I didn't want to risk ruining the ground sample distance or add in extra "no data" classes to my images.

4.4 Reclassifying Images

The classes that I used in eCognition to output my classified images was "forest", "water", "grassland", "crop", "urban", and "bare earth". They had to be reclassified because neural networks can only recognize numeric labels. Listing 1 shows the code that I used to reclassify my images.

Listing 1 Python code to reclassify landcover classes to numeric values.

```
class_value_mapping = {

    "bare earth": 0,

    "urban": 1,
```

```
        "water": 2,
        "forest": 3,
        "vegetation": 4,
        "grassland": 4,
        "crop": 5,
        "ignore": 9,
    }
```

This allowed every text label to be converted to a numeric label on the fly.

Chapter 5 Methodology

5.1 Overview

This chapter describes the experimental methodology used to evaluate my transformer. The transformer performs landcover classification using multispectral remote sensing imagery for six landcover classes: bare earth, urban, water, forest, vegetation, and crop. This methodology will include dataset preparation, model training, hyperparameter tuning, ablation studies, and performance evaluation.

5.2 General Workflow

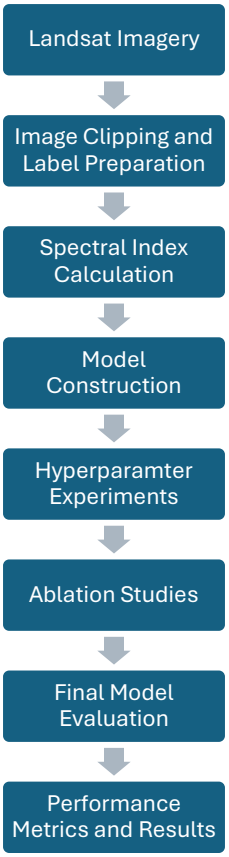


Figure 38 A flowchart showing the workflow of the model.

Figure 38 shows how everything was accomplished to complete the model, from selecting scenes of Landsat imagery to performing the final experiments for the model. First the images are selected. After clipping the images and classifying them to prepare labels the model was created. I then had to create my spectral index bank and ClassTransformer. Then I ran my ablation studies to optimize the hyperparameters. This led to the final tests of the model where I compared its

performance with the thresholds and adaptive weights to the model without those helpers. The final results would indicate whether or not my threshold logic helped the model to perform image classification.

5.3 Overview of Experimental Variables

Parameter	**Value**
Patch size	8
Patch stride	4
Model Depth	48
Number of heads	4
Number of Layers	2
Dropout	0.05
Lambda weight	1.0
Soft scale	2.0
Hard weight	2.0
Hard scale	5.0
Adaptive cap	10.0
Smoothing	0.0
Learning rate	5e-5
Minimum learning rate*	1e-2
Weight decay	5e-5
Decay rate*	0.8
Batch size	12
Maximum normalization	8.0
Maximum normalization high cap	8.0
Maximum normalization low cap	3.0
rho	1.2
quantile	0.9
beta	0.99
β_1	0.9
β_2	0.95
Warmup epochs	1
Ender	5

Table 4 List of all of the hyperparameters and their associated values used as standard values.

Table 4 shows all of the hyperparameters of my model and their initial values. My model has a number of different hyperparameters that can be used to tweak the model's performance. Table 4 shows the hyperparameters that I experimented upon.

The adaptive cap is the maximum value for the adaptive weights. Smoothing is related to the confidence of the model. It tells the model how confident it should be in each prediction. With smoothing set to 0.0 the model assumes that the correct class is 1.0 and the incorrect class is 0.0. With a nonzero value of smoothing each wrong class would be given a nonzero probability equal to smoothing divided by the number of wrong classes. The learning rate determines the size of the weight updates during the optimizer steps. Weight decay limits the size of the model weights during the optimizer updates to reduce the chances of overfitting. Batch size controls the number of images that are loaded into the model before the weights are updated. A larger batch size would produce smoother training and faster training but it also uses more GPU memory. Max normalization is the largest allowed gradient before the optimizer updates the weights. The max normalization, cap hi, is the value that I set as the ceiling for gradient clipping. The max normalization, cap lo, is the value that I set as the floor for gradient clipping. P, or rho, is a scaling factor for the adaptive cap. Quantile controls how a percentage value is taken from the values in window. A low value of quantile, such as 0.25, results in lower recent value from the window while a high value of quantile results in a higher recent value from the window. Here window is a set of true norm values. β_1 and β_2 are memory settings for my optimizer. β_1 controls how much the optimizer remembers the slope of the gradients. A larger value of β_1 results in a smoother slope. β_2 controls how much the optimizer remembers the magnitude of the squared gradients. Warmup epochs are the epochs that allow gradients to stabilize before real training occurs. During the warmup epochs the learning rate is a fraction of its normal value. Ender is the value that I set to end the training loop. It is the largest percent difference that I allow between successive epochs.

5.4 Spectral Index Calculations

The model uses the remote sensing indices to perform landcover classification. I calculated the automated water extraction index as

$$AWEI_{nsh} = 4 \times (\rho_{band2} - \rho_{band5}) - (0.25 \times \rho_{band4} + 2.75 \times \rho_{band7}) \quad (16)$$

where ρ is the reflectance value of spectral bands of Landsat 5 TM. Band 2 is the green band, band 4 corresponds to NIR, band 5 to SWIR and band 7 to SWIR (Feyisa, Meilby, Fensholt, & Proud, 2014). This index helps to distinguish water pixels from non-water pixels by forcing non-water values below 0 and water values above 0. I substituted in the equivalent bands for Landsat 8 & 9 OLI. The bare soil index, BSI, is given by the following

$$BSI = \frac{(SWIR1 + Red) - (NIR + Blue)}{(SWIR1 + Red) + (NIR + Blue)} \tag{17}$$

The bare soil index helps to classify bare earth. Bare soil values range from -1 to 1 where values near 1 indicate bare earth (Nguyen, Chidthaisong, Kieu Diem, & Huo, 2021). The normalized difference built-up index, NDBI, can be calculated as

$$NDBI = \frac{SWIR1 - NIR}{SWIR1 + NIR}. \tag{18}$$

The enhanced built-up & bareness index, EBBI, was calculated by

$$EBBI = \frac{SWIR1 - NIR}{\sqrt{10 \times SWIR1 + TIRS1}}. \tag{19}$$

The difference built-up index, DBI, was calculated by

$$DBI = \frac{Blue - TIRS1}{Blue + TIRS1} - NDVI. \tag{20}$$

The Index Based Built-up Index, IBI, is given by

$$IBI = \frac{NDBI - \frac{SAVI + MNDWI}{2}}{N\ DBI + \frac{SAVI + MNDWI}{2}} \tag{21}$$

The Urban Index, UI, is given by

$$\frac{SWIR2 - NIR}{SWIR2 + NIR} \tag{22}$$

The Normalized Difference Built-up and Bare Land Index, NDBal, was calculated by

$$NDBal = \frac{SWIR1 - TIRS1}{SWIR1 + TIRS1}. \quad (23)$$

Equations 18-23 were taken from Shahfahad, et al. MNDWI is the Modified Normalized Difference Water Index. It enhances open water features and diminishes built-up features that can be mistaken for open water by other indices (Environmental Systems Research Institute, n.d.). It is given by

$$MNDWI = \frac{Green - SWIR}{Green + SWIR} \quad (24)$$

I used the SWIR1 band in the calculation. The Soil-Adjusted Vegetation Index, SAVI, is similar to NDVI but it accounts for the brightness of the soil in areas with low amounts of vegetation. It outputs values between -1.0, and 1.0. It is given by

$$SAVI = 1.5 \times \frac{NIR - Red}{NIR + Red + 0.5} \quad (25)$$

for Landsat 8 and 9 imagery (U.S. Geological Survey, n.d.). The Normalized Difference Snow Index (NDSI) is used to emphasize snow. It is given by (U.S. Geological Survey, n.d.)

$$NDSI = \frac{Green - SWIR1}{Green + SWIR1}. \quad (26)$$

The Modified Normalized Difference Impervious Surface Index (MNDISI) is given by (Sun, Wang, Guo, & Shang, 2017)

$$NDISI = \frac{T_b - \frac{MNDWI + NIR + SWIR1}{3}}{T_b + \frac{MNDWI + NIR + SWIR1}{3}} \quad (27)$$

where T_b is given by

$$T_b = \frac{K2}{ln\left(\frac{K_1}{L_\lambda} + 1\right)}. \quad (28)$$

T_b is the effective at-sensor brightness temperature, K2 is a calibration constant, K_1 is a calibration constant, and L_λ is the spectral radiance at the sensor's aperture (Chander, Markham, & Helder, 2009). L_λ is given by

$$L_\lambda = M_L \times Q_{cal} + A_L \tag{29}$$

M_L is the radiance multiplicative rescaling factor, Q_{cal} is the Level-1 pixel value in digital numbers, and A_L is the radiance additive rescaling factor (Bah, Norouzi, Prakash, Blake, & Khanbilvardi, 2022). I used Landsat 8 scene L1TP_016040_20200501_20200820_02_T1 to get the values for K_1, K2, M_L and A_L of 774.8853, 1321.0789, 3.3420 x 10 -4, and 0.1000, respectively. A better method would be to calculate it for each Landsat scene but I assumed that using constant values would be okay for this analysis. The Built-Up Index, BU, was erroneously calculated as

$$BU = \frac{SWIR1 + NIR - Red + Blue}{SWIR1 + NIR + Red + Blue}. \tag{30}$$

The Built-Up Index wasn't used in classifying any of the images. It was only used as a parameter for the model. The only indices used in training were AWEI, BSI, MNDWI, NDBI, NDVI, and SAVI. The other indices were only used to further aide the in classifying the various image bands.

5.5 Model Architecture

My model is a transformer referred to as ClassTransformer. The model first applies a convolutional stem to the input imagery, then divides the feature map into patches using the patch embedding layer. These patches are then sent to the transformer encoder layers allowing the model to learn spatial and spectral relationships across the image. A final classification layer then produces patch-level class predictions for the landcover classes.

In neural networks, weights are used to update the loss. The model seeks to find the smallest loss via back propagation by using a loss function. I used PyTorch's CrossEntropyLoss. Cross entropy is used because it enables a model to predict different classes. It rewards the model for predicting a target class. Cross entropy is ideal because it assigns 1 class per pixel. It is given by

$$l_n = -w_{y_n} log \frac{\exp(x_{n,y_n})}{\sum_{c=1}^{C} \exp(x_{n,c})} \cdot 1\{y_n \neq ignore\_index\} \tag{31}$$

Where C is the number of classes, x is the input, y is the label, w is the weight and N is the batch size (PyTorch, n.d.). For the weights, I used an inverse-class frequency weighting normalized to a mean of

1. In this case I gave classes with low pixel counts a higher weight than classes with large numbers of pixels.

I chose to use the remote sensing thresholds to help teach the model how to classify different landcover classes. Because neural networks learn by updating their weights, I thought that weights could be used with the thresholds. I wanted the model to use the thresholds as a guide in how to use the remote sensing indices. The model would be rewarded when it properly classified a feature within the bounds of the threshold values. The model would similarly be punished when it made errors of omission or commission. Errors of omission occur when the model gives input pixels the wrong class, for example labeling a water pixel as bare earth. Thus, water pixels are omitted from the water class. Errors of commission are those in which the model adds pixels to the wrong class, for example, classifying bare earth pixels as water pixels. Thus, non-water pixels are committed to the water class (Campbell & Wynne, 2011). When the model performs either an error of omission or commission it should be penalized. I decided to reward or penalize the model by using weights.

I used a system of hard and soft constraints to penalize and reward my model respectively. Since the model learns by minimizing the loss, I decided that the loss should decrease if the model uses the thresholds effectively, negative loss, and increase if the model doesn't. Thus, the loss would be given by

$$L = l_n + h\sum_{b,c} P_{b,c} - \lambda\sum_{b,c} B_{b,c} \tag{32}$$

where l_n is the Cross Entropy Loss given by Eq. 34. The first term, $h\sum_{b,c} P_{b,c}$ is the penalty for the model and $\lambda\sum_{b.c} B_{b,c}$ is the reward. The reward is always negative because it helps reduce the model's total loss, while the penalty is always positive such that the model is penalized for its mistakes. P is penalty for the model and B is the bonus, b is an index over the items in the batch and c is one of the landcover classes. The coefficients h and λ are hyperparameters that I use to tune the impact of the threshold constraints. The h represents the tuning knob for the hard constraint and the λ is for the soft constraint. The hard constraint penalizes the model if a small number of pixels satisfy the threshold values.

The soft constraint takes the form of

$$B_{b,c} = \mu_{b,c}\pi_{b,c}\lambda_{b,c} \tag{33}$$

Where B is the bonus. The hard constraint takes the form of

$$P_{b,c} = \left(1 - \mu_{b,c}\right)\left(1 - \pi_{b,c}\right)h_{b,c} \tag{34}$$

I wanted to reward the model for properly classifying a feature within the bounds of the threshold values. It would still be able to classify features that are outside of the threshold values but doing so would result in a penalty.

The remote sensing indices were used to create thresholds for the model to use to help identify the various landcover classes. If the model correctly identifies a pixel that is within the bounds of the threshold it should be rewarded. The opposite is also true. If the model incorrectly classifies a pixel that is within the bounds of the threshold it should be penalized. The model must have the freedom to classify pixels that are outside of the thresholds because no classification is perfect, but it must be penalized if it chooses incorrectly. The only way to penalize or reward the model is by adjusting the weights. The goal of the model is to minimize the weight which is what happens during backpropagation. I have added a term to the weight to penalize and reward the model based on its use of the thresholds.

5.6 Training Procedure

For each experiment, I conducted 3 trials, each of which used a different random seed from the set {1,2,3} for each of the trials. I initialized the model with Kaiming weights to make my results more reproducible. I also used an Xavier distribution when initializing my fully connected layers or the patch embedding layers and multilayer perceptron, and final attention head. My experiments were designed to run until the percent difference was less than 5% for the last 3 epochs. I limited the trials to 20 epochs in the interest of time. I used the default values in Table 4 at the beginning of the experiment and only changed the hyperparameter in question. After finding an optimal value for the hyperparameter I used that optimal value in all subsequent experiments. For the final experiments, I allowed them to run for 100 epochs or until the percent difference over 3 epochs was 2% or smaller. In these experiments I allowed the model to run with the best set of hyperparameters and had 3

different sets: with thresholds and adaptive weight logic, with thresholds and constant weight logic, and without thresholds.

5.7 Evaluation Metrics

The accuracy of my model was performed by using a confusion matrix or error matrix. The confusion matrix shows the errors for each category as well as the misclassifications by category. To compile the error matrix, the images, labels, and model predictions are required. The images and labels are treated as ground truth. When the model makes a prediction, it is compared to the image-label pair to determine if the prediction is accurate. Pixels that are accurately labeled are added as well as those pixels that are mislabeled. As mentioned earlier, pixels that are mislabeled fall into categories, errors of omission and errors of commission.

	Bare earth	urban	water	forest	vegetation	crop	Row total
Bare earth	809314	63696	0	149049	128454	109919	1260432
Urban	0	602536	1469	0	2660	0	606665
Water	427	2545	621627	0	1333	192	626124
Forest	702	0	0	364419	1555	329	367005
Vegetation	22957	23906	2	94855	342157	17889	501766
crop	40384	0	0	5	0	401895	442284
Column total	873784	692683	623098	608328	476159	530224	3804276

Table 5 An error or confusion matrix.

Table 5 shows an error matrix. The values along the diagonal, shown in red, represent the total number of pixels that were correctly classified into their respective class. The errors of commission are reported in the columns of the matrix. For bare earth, 0 pixels were mistakenly classified as urban, 427 pixels as water, 702 pixels as forest, 22957 pixels as vegetation, and 40384 pixels as crop. The errors of commission are in the rows of the matrix. They represent pixels that were mistakenly classified as the expected class, for example water pixels classified as bare earth. As per Table 5 in the bare earth class, 63696 urbans pixels, 0 water pixels, 149049 forest pixels, 128454 vegetation pixels

and 109919 crop pixels were mistakenly classified as bare earth. The overall accuracy is determined by adding the red values and dividing by the total number of pixels, the intersection of row and column total. This gives an accuracy of 82.59%. The error matrix was calculated using scikit-learn and it doesn't add the row and column totals. To save the time that it would take to manually calculate the overall accuracy for every trial in every experiment I also used scikit-learn's classification report. The classification report summarizes the results of the error matrix.

	precision	recall	F1-score	support
Bare earth	.93	.64	.76	1260432
Urban	.87	.99	.93	606665
Water	1.00	.99	1.00	626124
Forest	.60	.99	.75	367005
vegetation	.72	.68	.70	501766
crop	.76	.91	.83	442284
accuracy			.83	3804276
Macro avg	.81	.87	.83	3804276
Weighted avg	.85	.83	.82	3804276

Table 6 A classification report.

Table 6 shows a classification report generated by scikit-learn. In this case I have taken the liberty of replacing the class labels with their respective class names. Precision is the ratio of correct predictions to the total number of predictions classified as correct. It is given by

$$Precision = \frac{correctly\ classified\ actual\ positives}{everything\ classified\ as\ positive} = \frac{TP}{TP + FP} \tag{35}$$

Recall is the ratio of correct predictions to the total number of predictions classified as correct. It is given by

$$Recall = \frac{correctly\ classified\ actual\ positives}{all\ actual\ positives} = \frac{TP}{TP + FN} \tag{36}$$

TP, is the true positive, FP is the false positive, and FN is the false negative. These values come directly from the error matrix. The true positive, TP, is given by the number of correctly classified pixels of a given class and FP, the false positive, would be the errors of commission. Thus, the precision for bare earth would be the quotient of the correctly classified pixels for bare earth, 809314, and the row total for bare earth, 873784. The false negative, FN, is given by the sum of the errors of omission. The recall for bare earth would be the number of correctly classified pixels divided by the column total or 809314 / 1260432. In remote sensing, the precision is the user's accuracy and recall is the producer's accuracy. The accuracy in the classification report is given by

$$Accuracy = \frac{correct\ classifications}{total\ classifications} = \frac{TP + TN}{TP + \ TN + FP + FN} \tag{37}$$

Where TN is the number of true negatives or everything that isn't classified as the given class in the error matrix (Google Developers, 2026). The F1-scores are given by

$$F1 = 2 \times \frac{precision \times recall}{precison + recall} \tag{38}$$

by Taha and Hanbury (Taha & Hanbury, 2015). I decided to use the macro average of the F1-score as the metric for my experiments because it factors all of the classes into a single metric. I also used the individual F1-scores when the macro average F1-score was tied with that from another trial to help select the best value for the hyperparameters.

Chapter 6 Results

I began by experimenting by running some tests to see if having the learning rate have a cosine decay to 1e-7 would improve the performance of the model. I ran 3 trials with the cosine decay and 3 without. I used a different random seed from the set {1,2,3} for each of the trials. I initialized the model with Kaiming weights to make my results more reproducible. I also used an Xavier distribution when initializing my fully connected layers or the patch embedding layers and multilayer perceptron, and final attention head. The results were averaged to produce Figure 39.

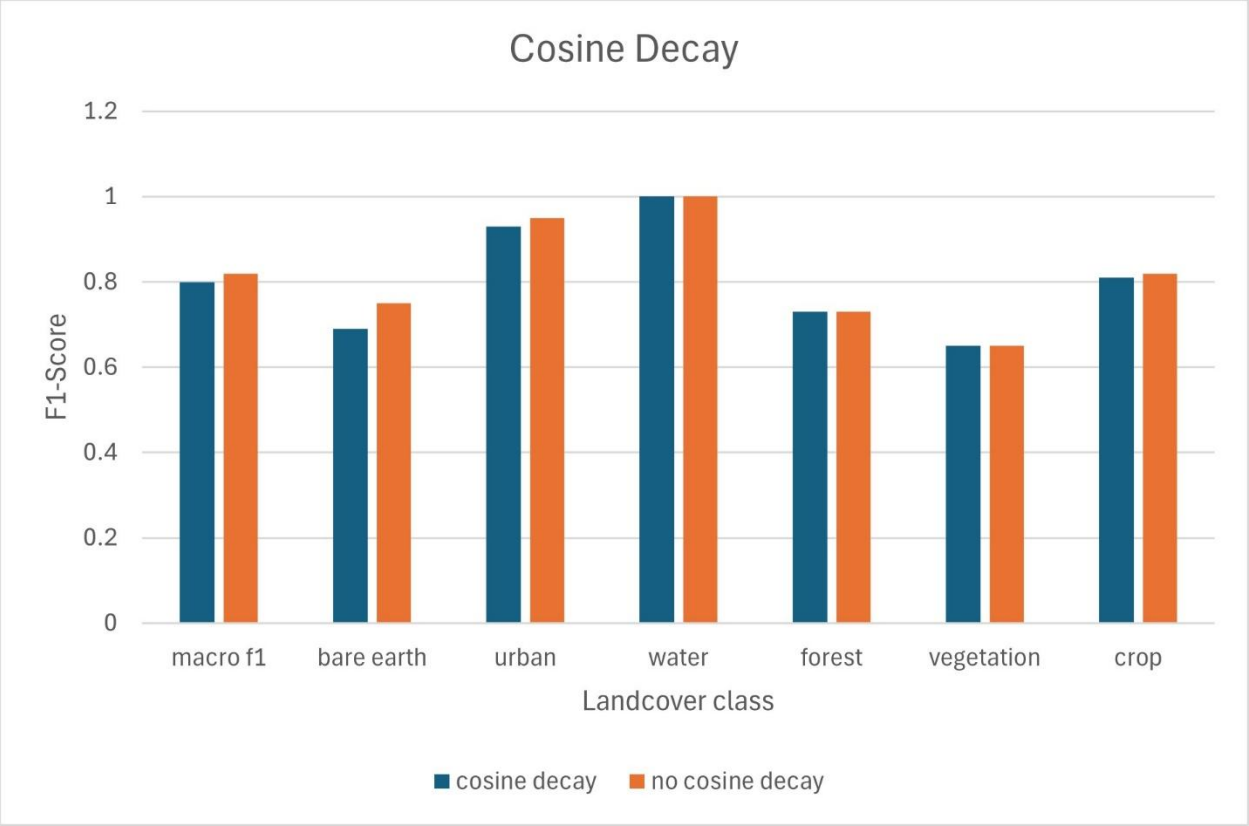


Figure 39 An image showing the results of an ablation test for cosine decay. Constant learning rate is shown in blue while the learning rate with a cosine decay is shown in orange.

Figure 39 shows that a constant learning rate is better than one with a cosine decay. The largest gain is in the bare earth class with a difference of 6%. The smallest gain is in the crop class

with a difference of 1%. Since there is no gain to be had from using a cosine decay, it will be dropped in subsequent experiments.

The next ablation study focuses on the number of warmup epochs. A warmup epoch allows the model to run at a learning rate that is smaller than normal. At the beginning of training, gradients can be quite large and more damaging. The weights of the model are also randomized which can lead to noisy gradients. Having warmup epochs can help stabilize the gradients prior to training the model. I tested numbers from 0-5 warmup epochs. These results are illustrated in Figure 40.

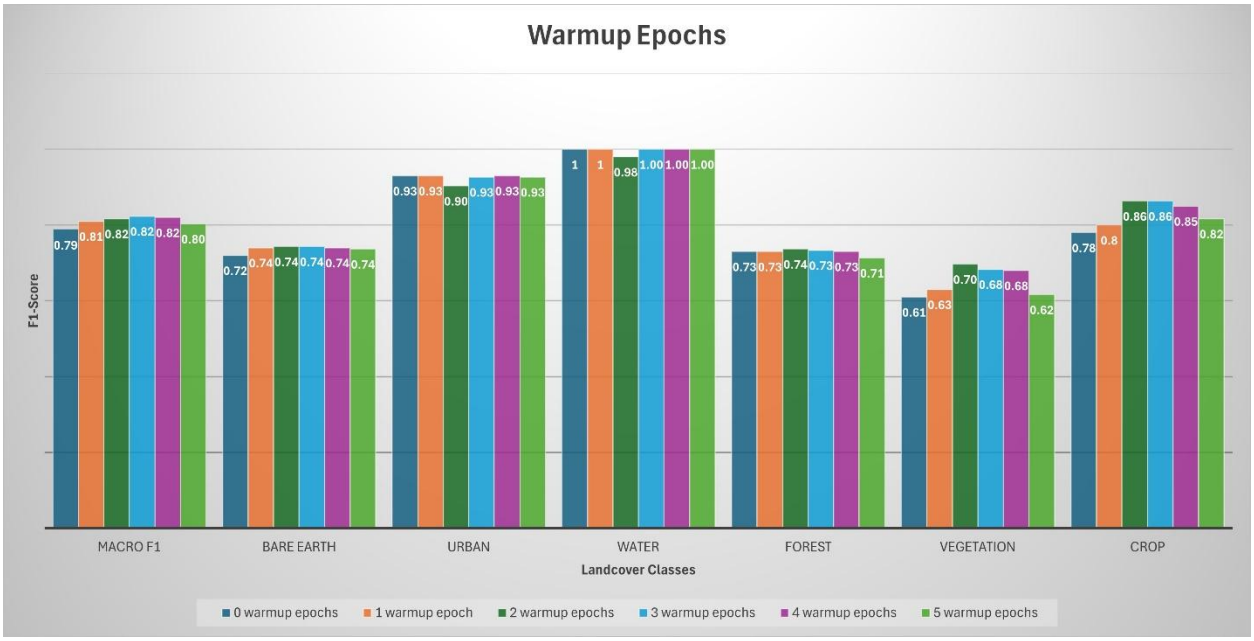


Figure 40 An ablation study to determine the number of warmup epochs. 0 warmup epochs is shown in blue, 1 warmup epoch in orange, 2 warmup epochs in green, 3 warmup epochs in light blue, 4 warmup epochs in purple, and 5 warmup epochs in light green.

Figure 40 shows that the macro F1-score peaks for 3 warmup epochs. However, the values were calculated to 2 significant figures so we should consider all of the warmup epochs with an average percentage of 82, namely 2-4 warmup epochs. When comparing 2 warmup epochs to 3 warmup epochs we find that the urban class has gains of 3%, the water class gains 2%, the forest class loses 1%, and vegetation loses 2% when 3 warmup epochs are chosen. In other words, choosing 2 warmup epochs would result in an advantage for the forest, and vegetation classes at the cost of a lower percentage for the urban and water classes. Selecting 3 warmup epochs would result in

strengthening the stronger urban and water classes while minimizing the damage to the forest and vegetation classes. In my opinion, 3 warmup epochs are better than 2.

When comparing 3 warming up epochs to 4 warmup epochs, we find that the only difference between them is the crop class. Using 3 warmup epochs produces 86% for crop compared to 85% for 4 warmup epochs. Therefore, 3 warmup epochs is the best choice.

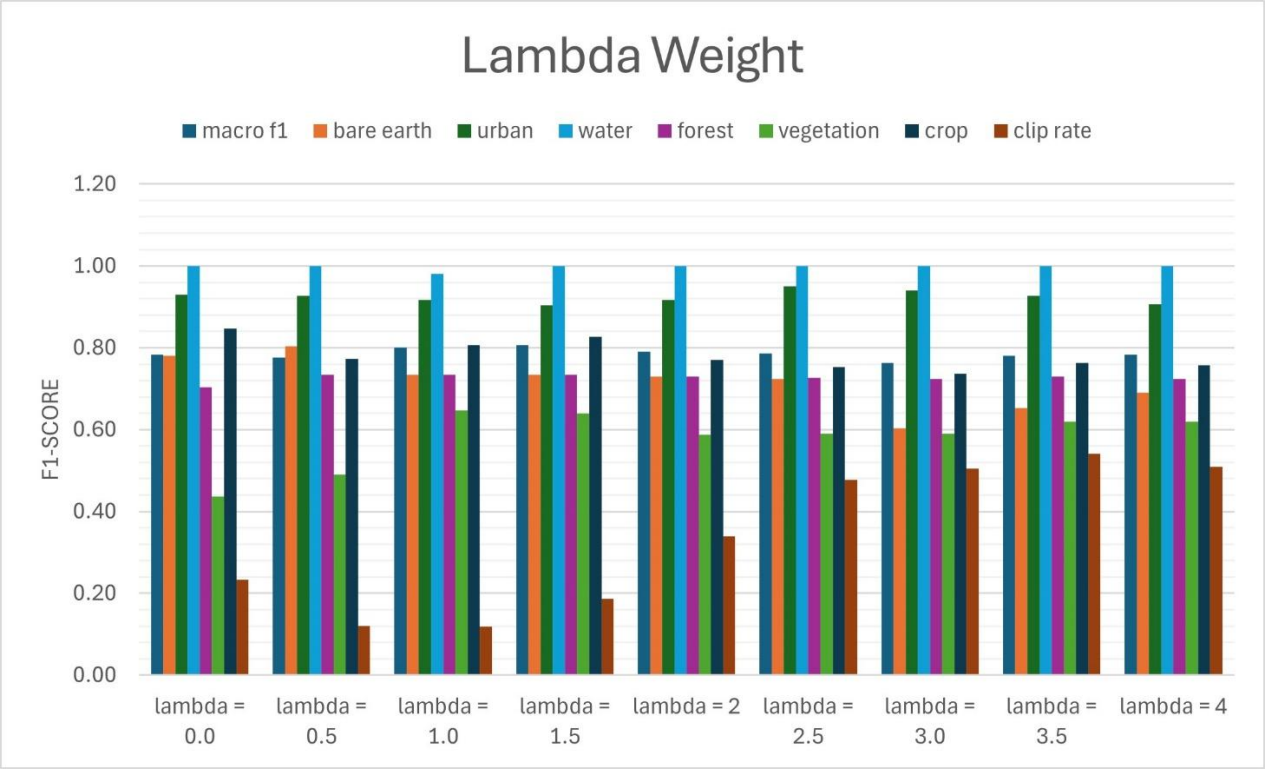


Figure 41 An ablation study showing how the F1-score changes for different values of lambda. The value of 1.5 has the highest F1-Score.

Figure 41 shows the results of an ablation study for the lambda weight. The results indicate that a value of 1.5 has the highest F1-score, and thus overall accuracy of 0.81. As Lambda increases from 0 to 1.5 the F1-score increases until it plateaus at 1.5. After 1.5, it begins to decrease again. The clip rate has a minimum at lambda = 1.0 but then begins to steadily increase again until lambda is 4.0. I stopped my experiments at this point because the clip rate was getting quite high and the F1 score was decreasing. Because of this I decided to look at values of lambda between 1.0 and 1.5 to see if 1.5 was a true peak or if I could tease out another value.

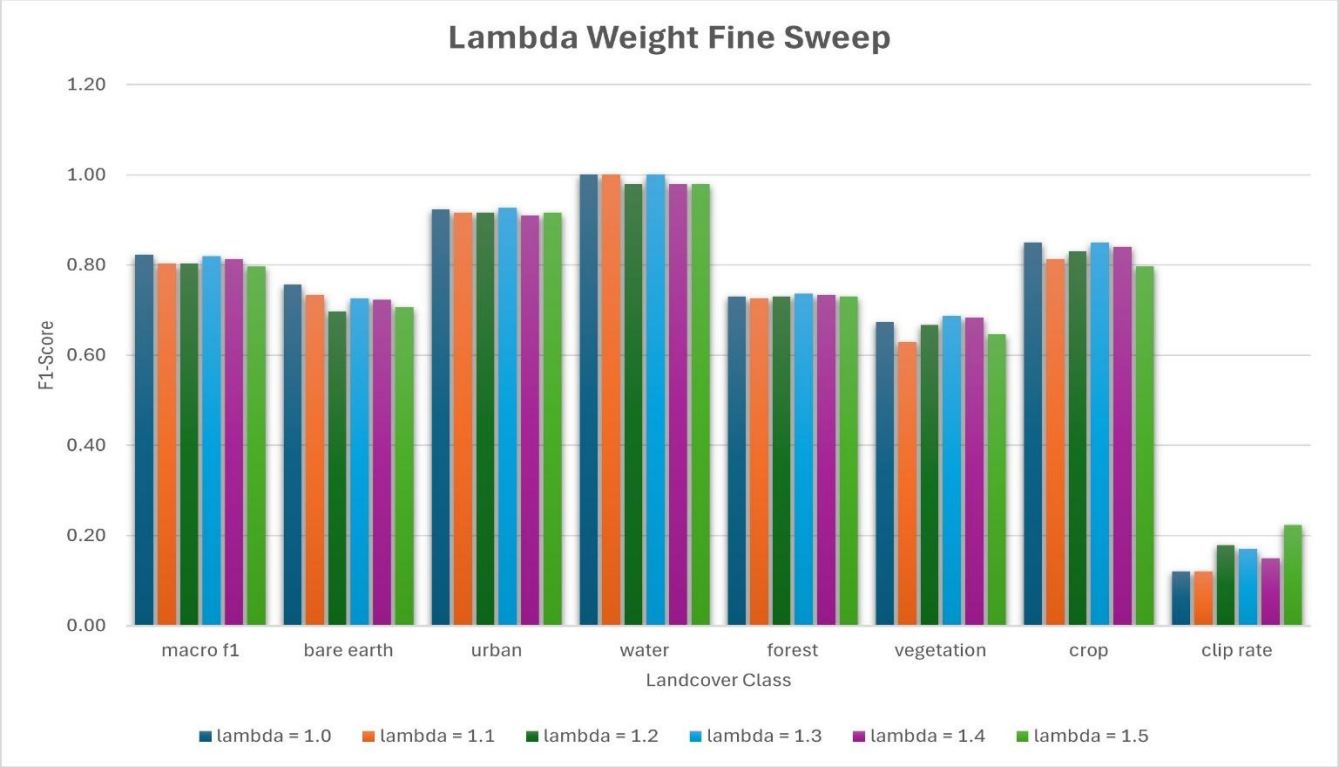


Figure 42 An ablation study showing how the F1-score changes for various values of lambda between 0 and 1.2.

Figure 42 shows the results as lambda is incremented in tenth-integer intervals. The highest average F1-score occurs when lambda is 1.0 and 1.3 with a value of 0.82. Comparing these two values of lambda, when lambda is 1.0, bare earth shows an improvement of 3 points, at the expense of vegetation and forest losing 1 percentage point each. The largest difference between the two values is the clip rate which is 5 percentage points lower when lambda is 1 opposed to lambda being 1.3. Based on these results, I chose 1.0 as the value of lambda.

The next hyperparameter that I focused on is the soft scale. Lambda weight controls the effect of the award given to the model for correct predictions. The soft scale hyperparameter moderates the effect of the adaptive factor to the lambda weight. As the main loss fluctuates the effect of lambda weight scales based on the soft scale of the adaptive lambda. I ran some experiments to optimize the value of the soft scale by choosing integer values from 0 – 5.

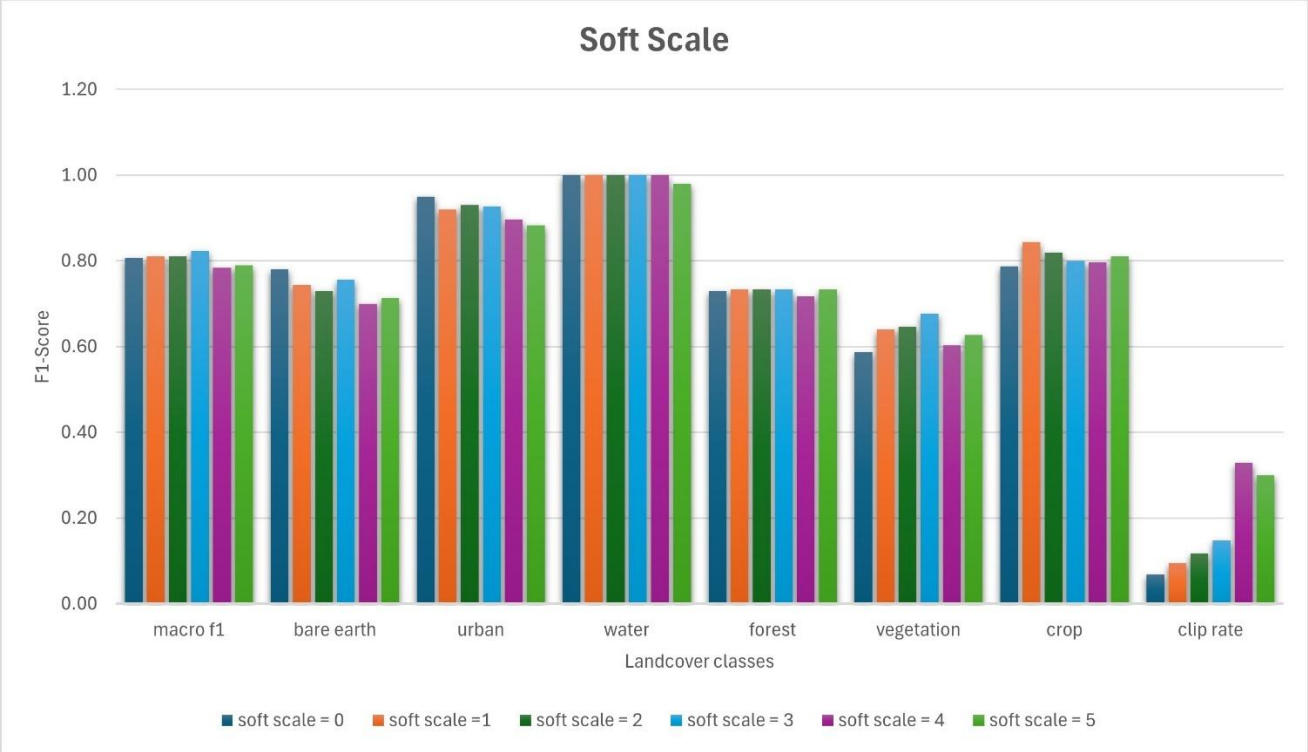


Figure 43 An ablation study of the soft scale for the adaptive lambda of the model. A value of 3.0 results in the highest F1-score of 0.82.

Figure 43 shows the results of the ablation study on various values of the soft scale. The highest F1-score of 0.82 results from a soft scale value of 3.0. As the soft scale increases from 0 the F1-score increases as well and reaches a maximum when the soft scale is 3.0. The clip rate also increases steadily as the soft scale increases. Because of the F1-scores, I decided to perform a finer sweep using increments of 0.25 from 2.5 to 3.5.

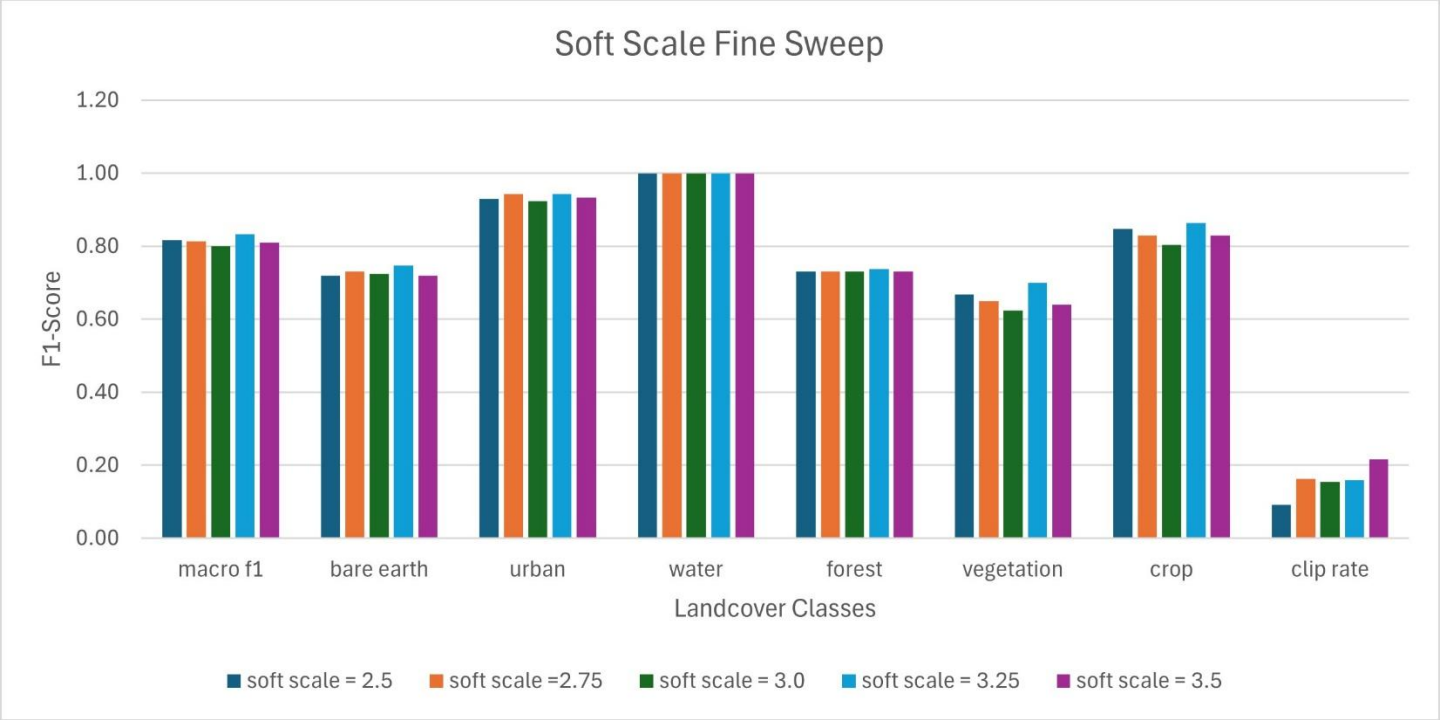


Figure 44 An ablation study showing the results of a fine sweep for the soft scale. A value of 3.25 results in the highest F1-score of 0.83.

As per Figure 44, the fine sweep of the soft scale shows that there is a maximum value between 3.0 and 3.5. Thus, the fine sweep was justified. When the soft scale is 3.25, the F1-score is 83% which is higher than 80% when the soft scale is 3.0 and 81% when the soft scale is 3.5.

Figure 45 An ablation study showing how the F1-score changes for various values of the hard weight.

Figure 45Figure 45 shows how the F1-score changes for various values of the hard weight. When the hard weight is 2.0, the average F1-score is 0.82 which is the highest out of the entire study. Based on these results I decided to look at values of the hard weight near 2.0 to see if 2.0 was a local maximum or if there were any higher values between 2.0 and 3.0.

Figure 46 An ablation study showing the results of a finer sweep using increments of 0.25 for the hard weight.

During this experiment, the code ran for longer than I intended but it led to some interesting results. As per Figure 46, the best F1-score of 83% occurs when the hard weight is 4.25, however at this value the clip rate is 28% which is rather high. The 2nd place value of 82% occurs at values of 2.25, 2.75, and 3.25. 2.25 has a clip rate of 17%, 2.75 has a clip rate of 18% and 3.75 has a clip rate of 23%. Because of this I dropped 3.25 and focused on 2.25 and 2.75. At 2.25 the vegetation and crop values are higher than those of 2.75 so 2.25 will be used for the rest of my experiments.

The next hyperparameter that I optimized was the hard scale. It modulates the strength of the adaptive hard weight by controlling how it responds to deviations from the margin. For these experiments the margin is 0.1.

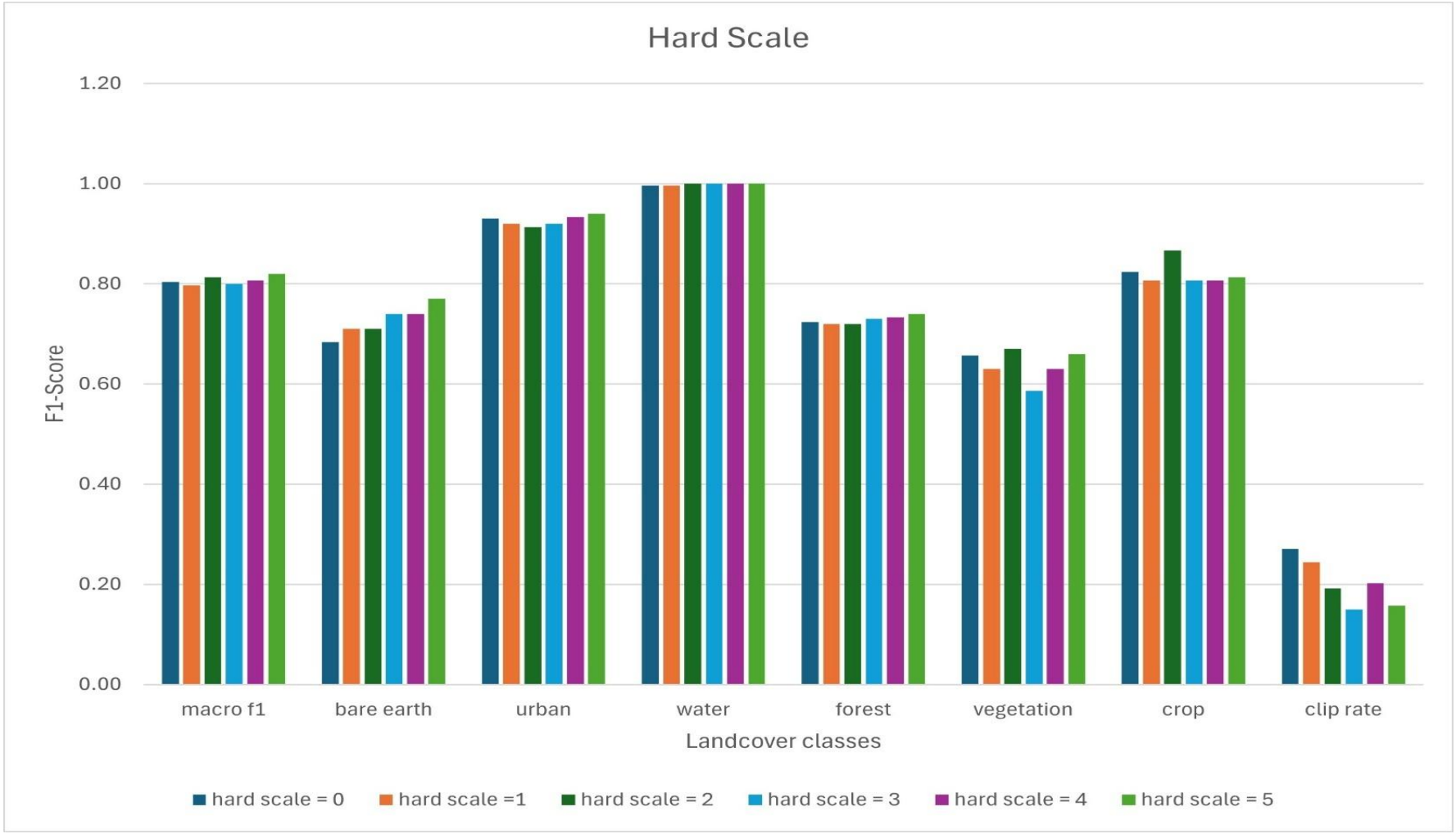


Figure 47 An ablation study showing how the hard scale affects the F1-score.

As per Figure 47, the hard scale was incremented by 1.0 for each experiment. The highest F1-score was obtained when the hard scale was at 5.0. The F1-scores were increasing up to 82% at 5.0 so I decided to perform a fine sweep around 5.0.

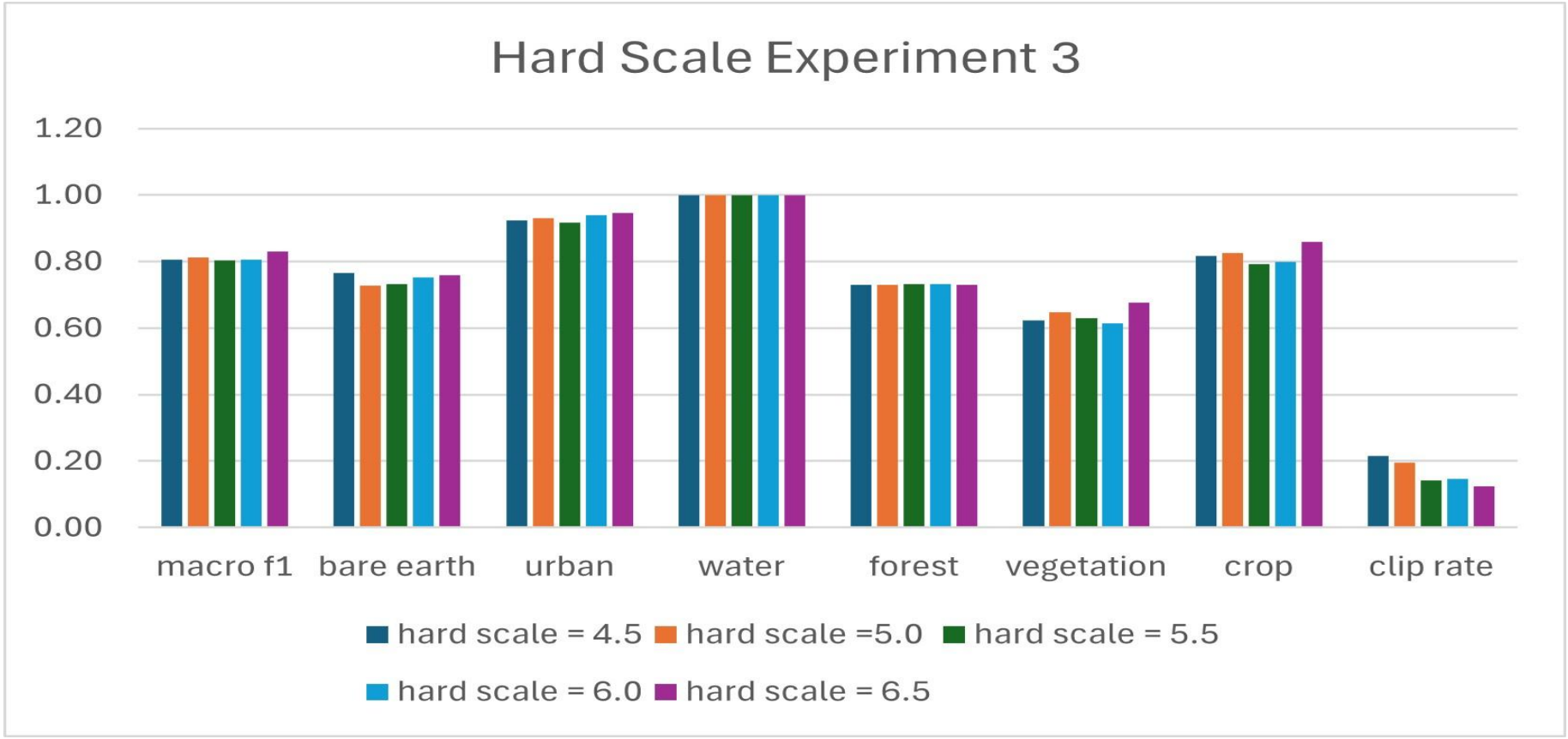


Figure 48 An ablation study showing a fine sweep of the hard scale using increments of 0.25.

As per Figure 48, the best F1-score of 82% occurs when the hard scale is set to 6.5. Because of this I decided to perform another sweep around 6.5.

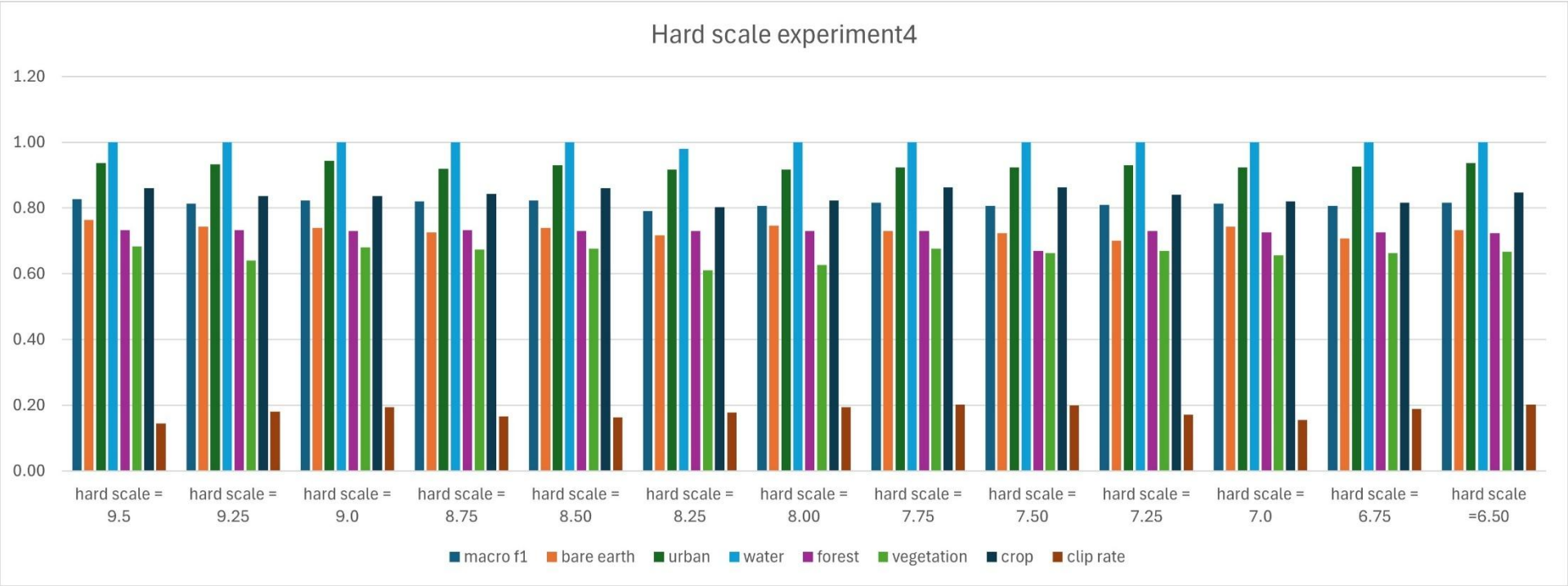


Figure 49 A set of experiments showing how accuracy of the model changes as the hard scale changes from 6.50 – 9.50.

Figure 49 shows that the best F1 score of 83% occurs at a hard scale of 9.5. Going forward, I'll use 9.5 as the value for the hard scale. If time permits I'll repeat the hard scale experiments with the best values from Figure 45Figure 47 - Figure 49 to double check my results.

The next hyperparameter that I optimized was the adaptive cap. It serves as a ceiling for the adaptive weights, thus ensuring that they don't become too large and overpower the main loss.

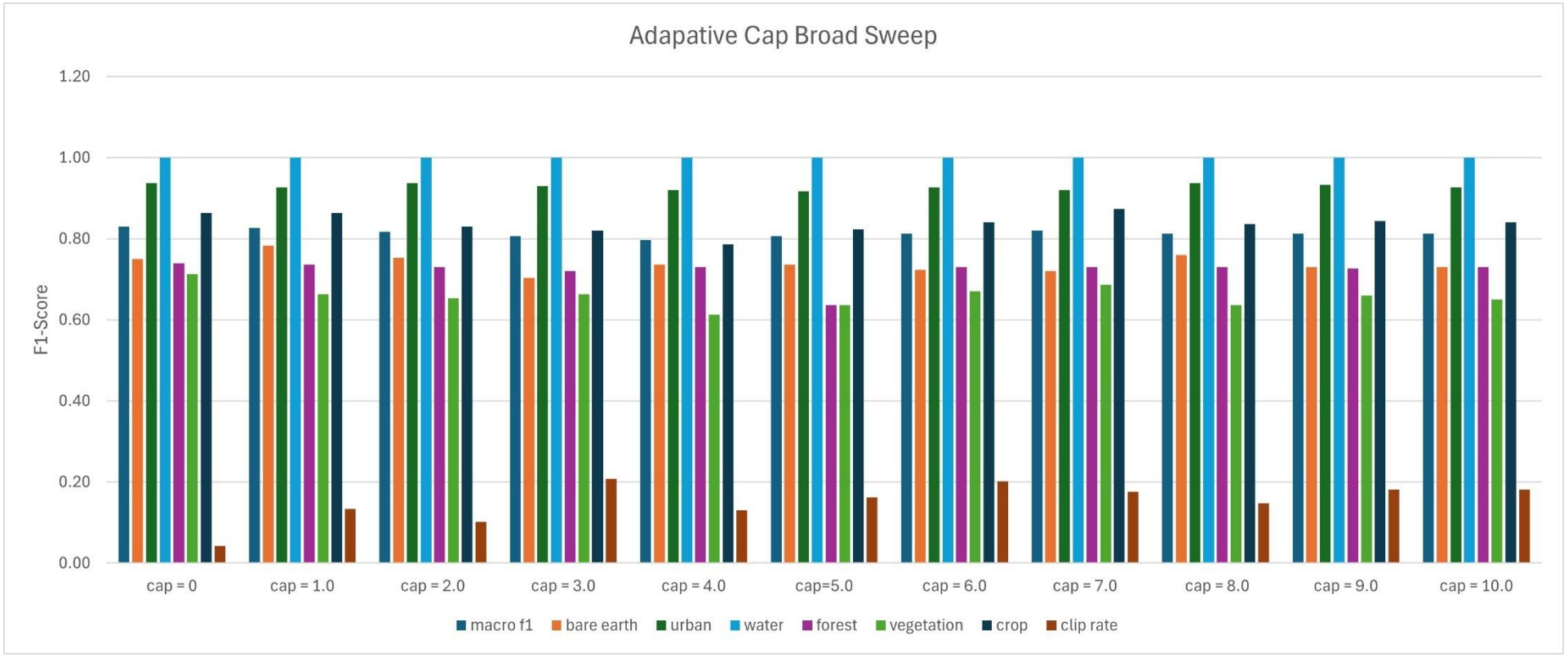


Figure 50 An ablation studying showing the effects of the adaptive cap on the F1-Score.

Figure 53Figure 50 shows how changing the adaptive cap affects the performance of the model. The model does best when the adaptive cap is 0 and 1.0, both tying with an F1-score of 83%. Because of this I ran a fine sweep focusing on values between 0.25 and 2.0 with increments of 0.25.

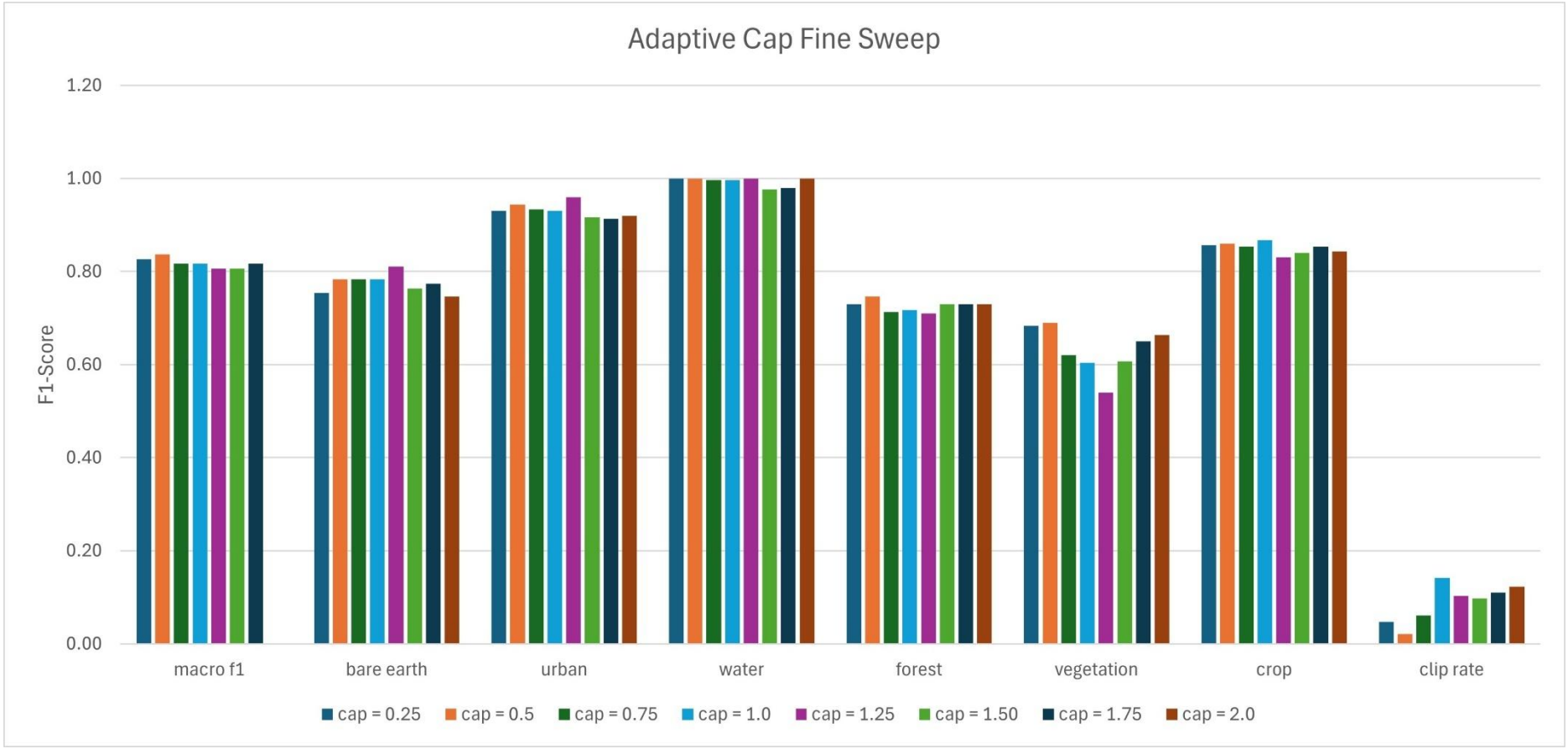


Figure 51 An ablation study showing how changing the adaptive cap affects the accuracy of the model.

Figure 51 shows the accuracy changes when the adaptive cap is increased from 0.25 to 2 in steps of 0.25. In this case, an adaptive cap of 0.5 has the largest F1-score and the smallest clipping rate. All subsequent experiments will have the adaptive cap set to 0.5.

The value for the margin was originally set to 0.1. I ran a set of experiments to determine the optimal value of margin. I tested values from 0 – 0.7.

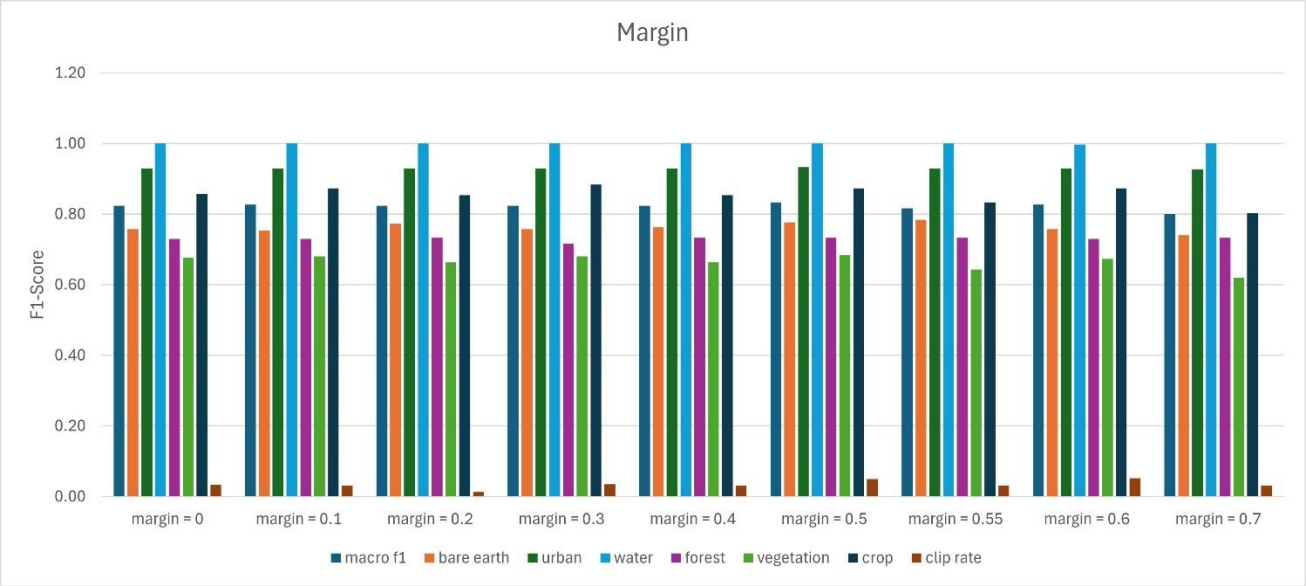


Figure 52 An ablation study showing how changing the value of margin affects the F1 score.

As per Figure 52, the highest F1 scores occur when margin is set to 0.5 and 0.6. When comparing the results from these two experiments, setting margin equal to 0.5 results in higher bare earth and vegetation accuracies. Subsequent experiments will use 0.5 as the value for margin.

Figure 53 An ablation study showing how the F1-score changes for various values of patch size and patch stride.

Figure 53 shows the relationship between the F1-score and the patch size and patch stride. For these tests, the batch size was set using $B = B_0 \left(\frac{P}{P_0}\right)^2$ to keep the effective number of pixels equal per optimizer step. Here B_0 is the default value that I used as the batch size, 20, p_0 is the original patch size of 10, and p is the new patch size. This stems from the fact that the number of patches, $N = \frac{HW}{P^2}$, where H is the height of the image, and W is the width of the image (Dosovitskiy, et al., 2020, p. 3). The patch size and stride were set such that the overlap was 50% for every value of the patch size. The patch size of 16 is in dark blue, 14 is in orange, 12 is in dark green, 10 is in sky blue, 8 is in magenta and 6 is in light green. The graph is organized to show how the F1-score changes for each landcover class as well as the average F1-score. The highest average F1-scores occur when the patch size is 14 and 10 with an average F1-score of 83%. A patch size of 14 results in the highest values for urban, water, vegetation, and crop. When the patch size is 10, the bare earth class improves by 2% to 77% and the forest class improves by 1% to 74% but vegetation decreases from 70% to 68%. I decided to

use the patch size of 14 because of the slightly higher vegetation results. The next set of tests will attempt to determine if the 50% overlap is the best by altering the stride from 2 to 14.

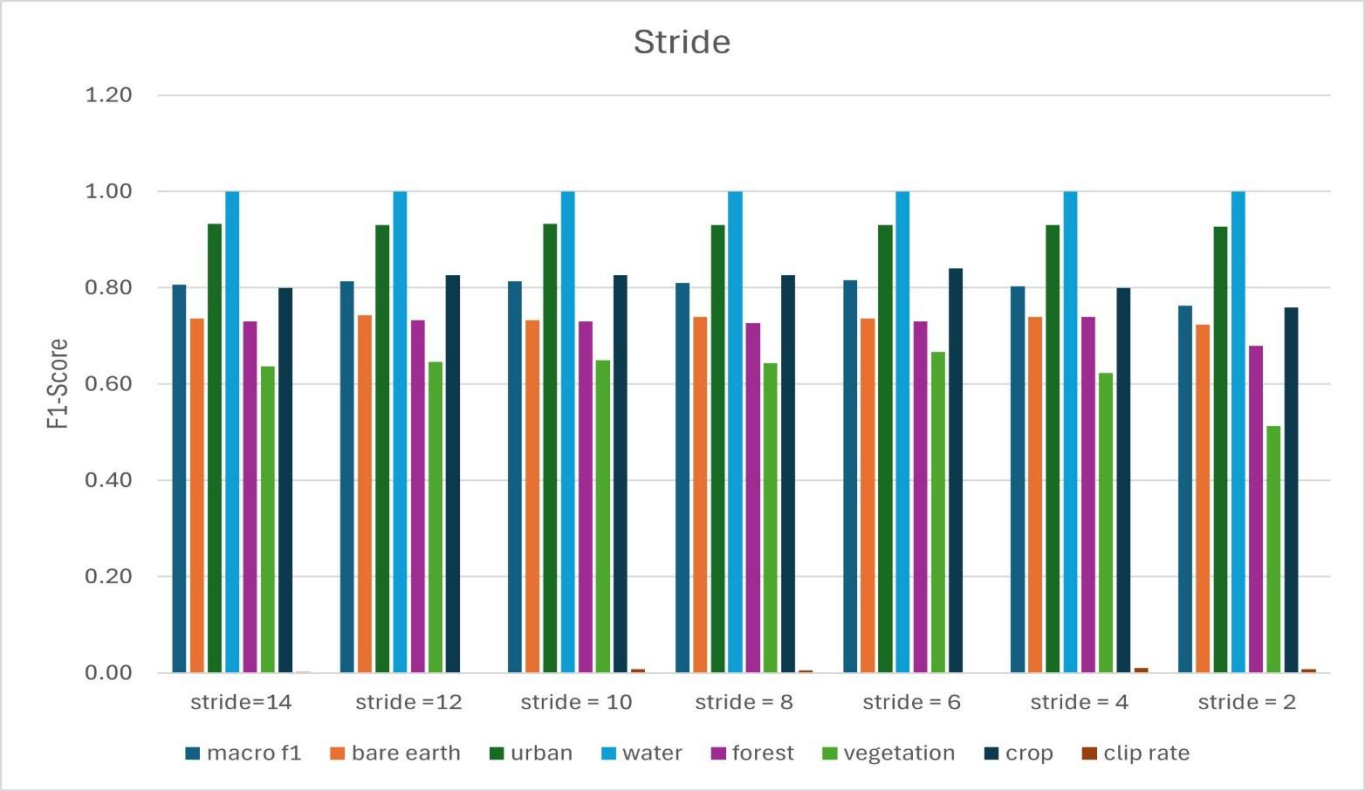


Figure 54 The results of an ablation study in which the patch size was kept constant and the stride was adjusted from 0% to 86% overlap. A stride of 14 corresponds to 0% overlap, 12 corresponds to 14.3%, 10 corresponds to 28.6% overlap, 8 corresponds to 42.9% overlap, 6 corresponds to 57.1% overlap, 4 corresponds to 71.4% overlap, and 2 corresponds to 85.7% overlap.

Figure 54 shows the results of varying the patch size with a constant stride. Based on these results, a stride of 6 with a patch size of 14, results in a 57.1% overlap and produces the highest macro F1-score of 82%. This combination produces the best results for all classes and it has the lowest clip rate. The next set of experiments will focus on the learning rate.

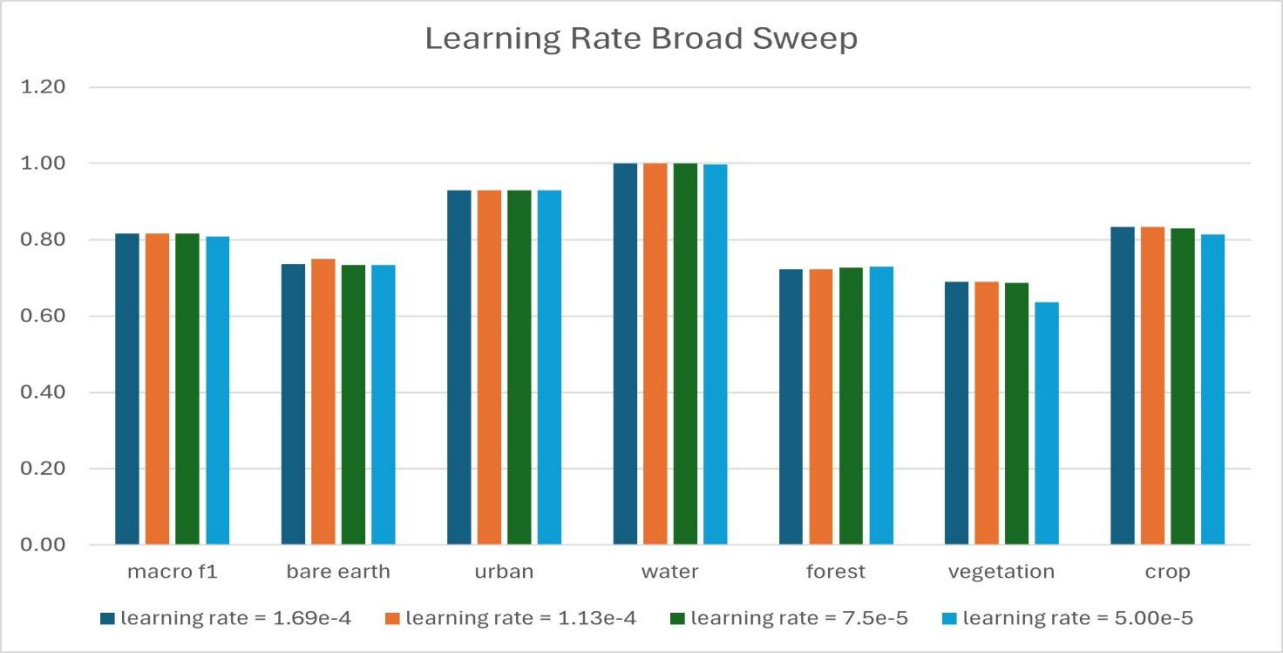


Figure 55 An ablation study showing the effects of the learning rate on F1-scores. The learning rate ranges from 1.69e-4 – 7.5e-5.

Figure 55 shows the results of broadly changing the learning rate. The macro-F1 scores are tied for all values of the learning rate save for 5.0 e-5. The values for 7.5 e-5, 1.13 e-4, and 1.69e-4 are all identical save for the bare earth and forest classes. 1.13e-4 has the largest value for bare earth at 75% compared to 74% and 73% for 1.69e-4 and 7.5e-5 respectively. For the forest class 1.69e-4 and 1.13 e-4 have 72% while 7.5e-5 has 73%. Based on these results a fine sweep around 1.13 e-4 would be best for the learning rate because of the higher percentage gain for the bare earth class. Based on these results I performed a sweep from 8.45e-05 to 3.97e-04.

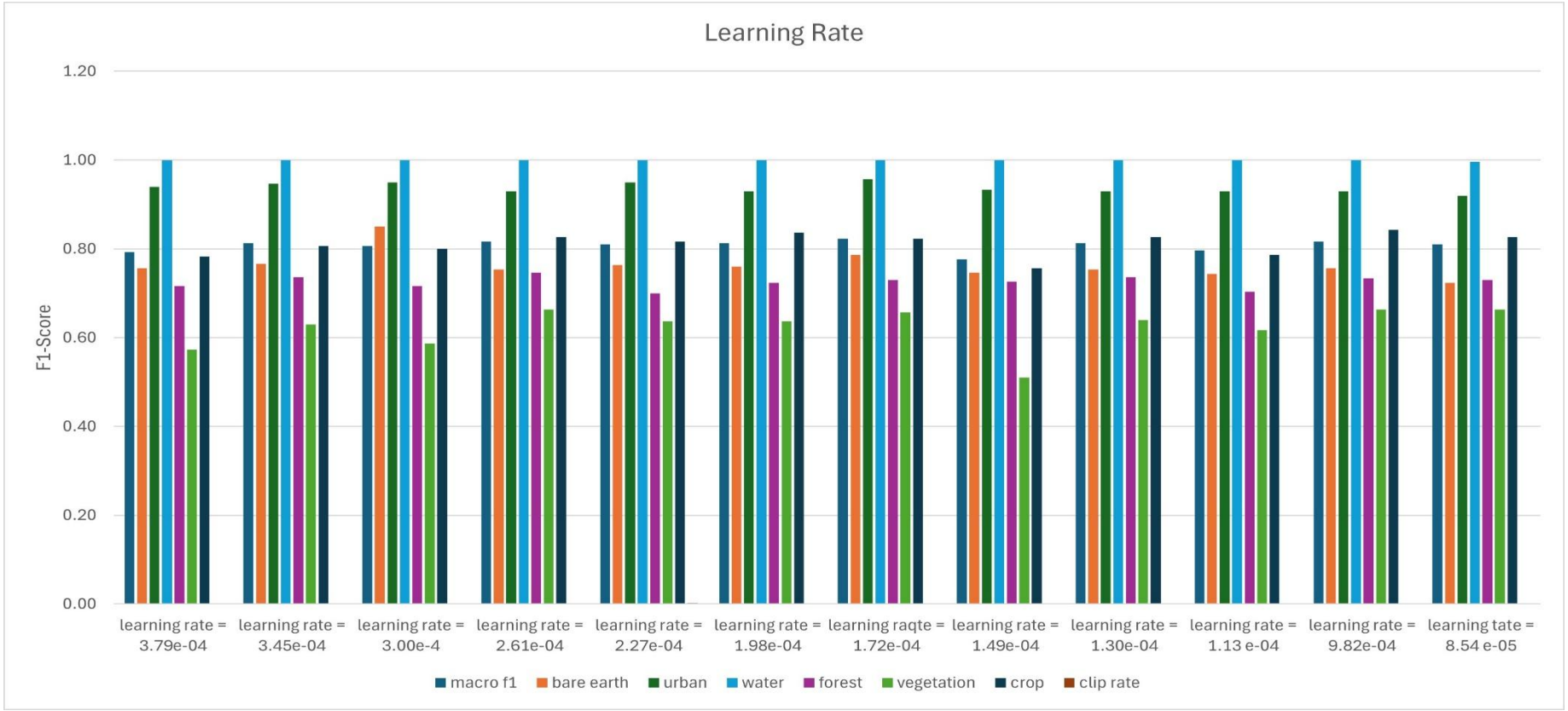


Figure 56 A fine sweep of the learning rate from 8.54e-05 through 3.79e-04.

Figure 56 shows the results of my 2nd set of experiments on the learning rate. This time there was a three-way tie for the best F1 score of 82% when the learning rate is 2.61e-04, 1.72e-04, and 9.82e-04. Looking at this in reverse order, having a learning rate of 9.82e-04 results in the highest values for crop at 84% compared to 82% at 1.72e-04 and 83% at 2.61e-04. Using a learning rate of 1.72e-04 results in the highest value for bare earth at 79% and for urban at 96%. Using a learning rate of 2.61e-04 results in the largest value for forest at 75%. All of the other values were the same for the 3 learning rates. Based on these results I selected 1.72e-04 as the best learning rate because it has the best bare earth and urban values at the expense of 2% for crop. All subsequent experiments will use 1.72e-04 as the learning rate. The next set of experiments focused on weight decay.

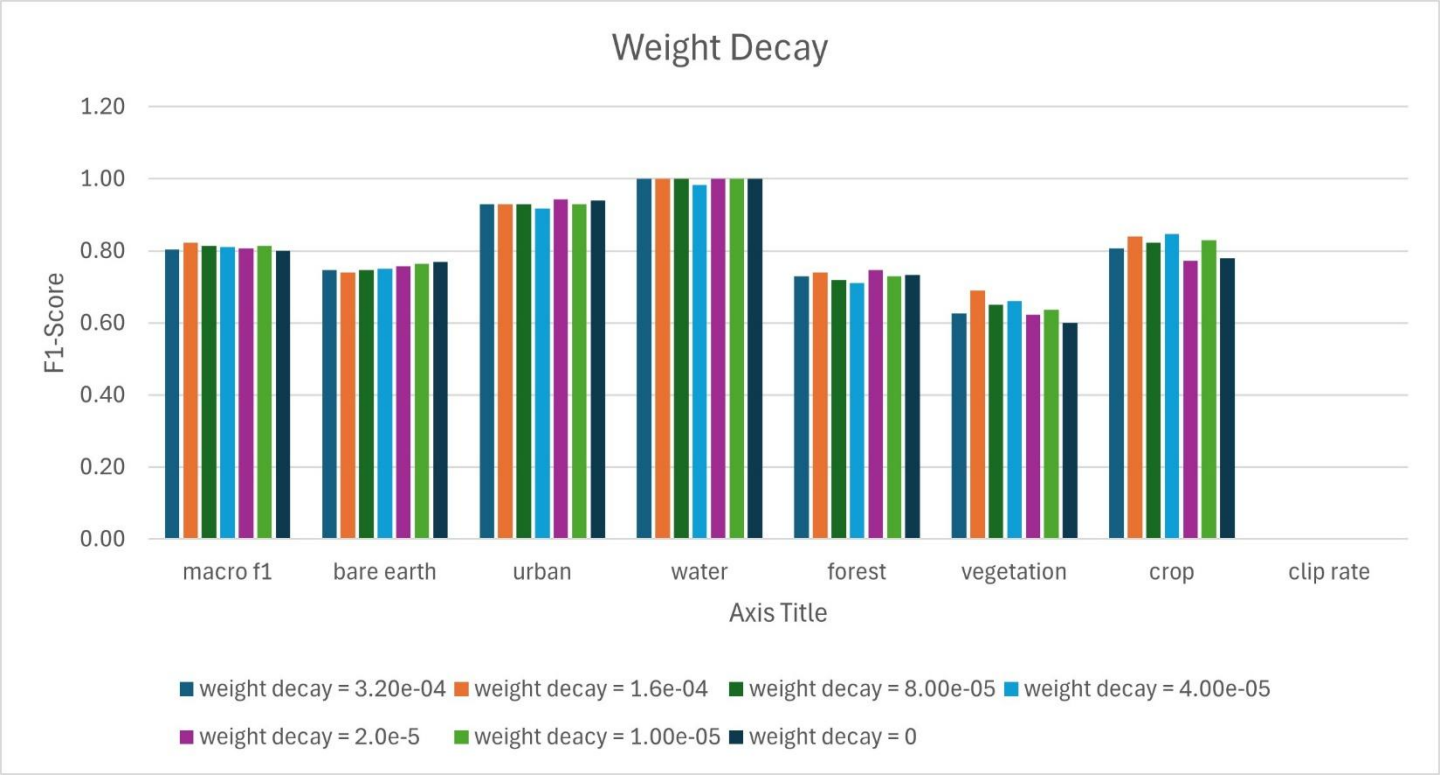


Figure 57 An ablation study showing the effects of various values for weight decay on the F1-Score. The values for the weight decay range from 0 to 1.00e-5.

Figure 57 shows the effects of weight decay on the model. Using a weight decay of 1.6e-04 results in the highest F1-score. This value will be used in all subsequent experiments.

My values of dropout were elements of the set from 0-5.0 in half integer increments.

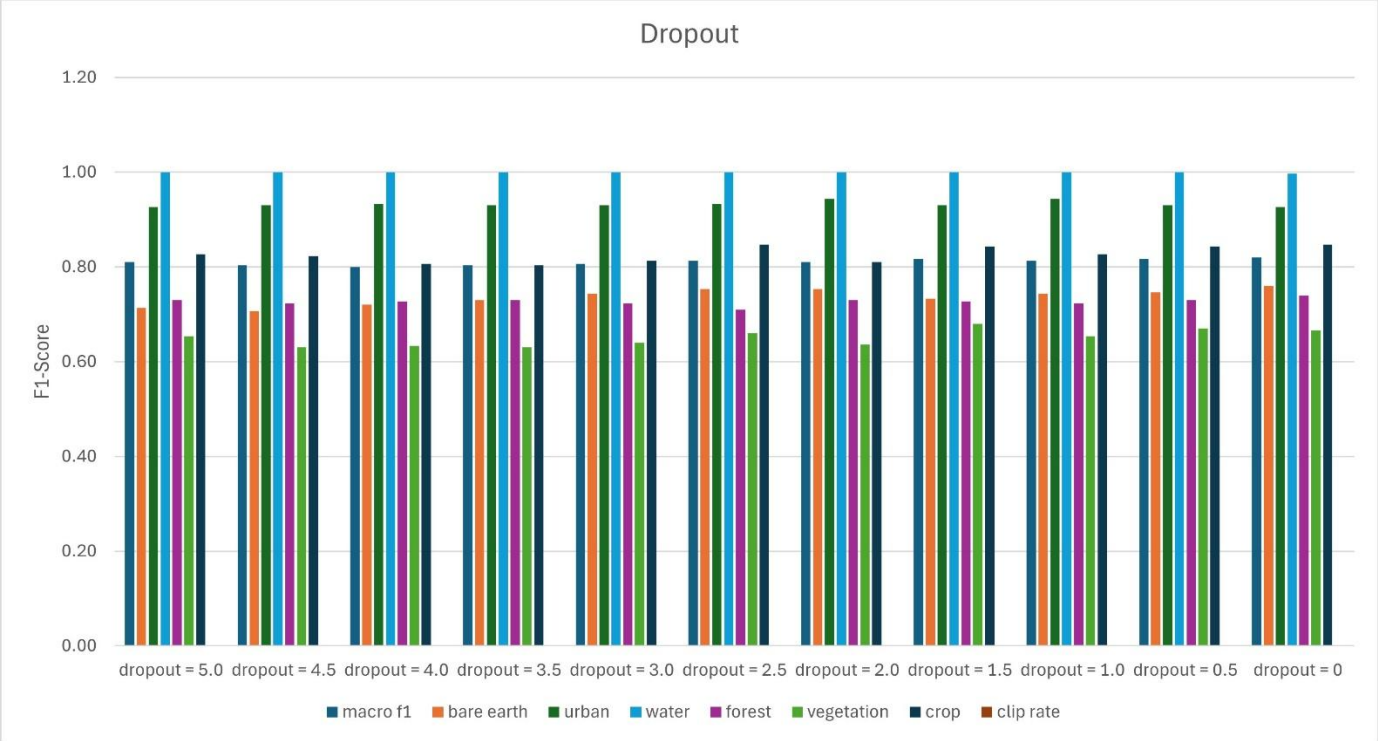


Figure 58 An ablation study showing the effects on the F1-score as the dropout varies.

As per Figure 58 there was no clear winner in these experiments because the macro F1-score had a 3-way tie when the dropout had values of 0, 0.5, and 1.5. A dropout of 0.0 results in the highest bare earth, forest, and crop classes at 76%, 74%, and 85%, respectively. When the dropout is 0.5 none of the classes have a maximum value over the others. When the dropout is 1.5 results in vegetation having an optimal value of 68%. In this case, I opted for using a value of 1.5 because of the higher vegetation value.

The next set of experiments focused on the attention heads. I was originally using 2 heads so I decided to run the experiments on elements of the set {2, 3, 4, 6, 8, 12}.

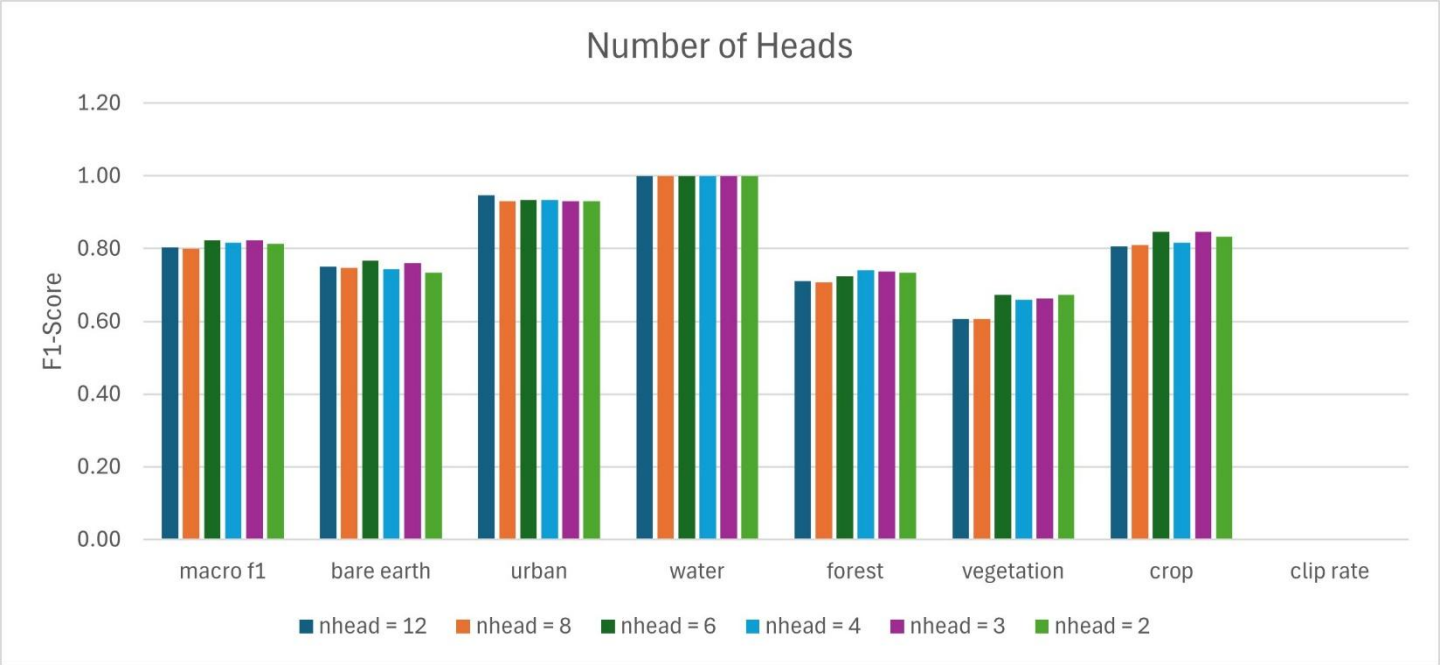


Figure 59 An ablation study showing how the number of attention layers affects the F1-score.

As per Figure 59, there is a three-way tie for the best F1-score of 82% between 3, 4, and 6 attention heads. Choosing 6 attention heads results in the highest value for bare earth and vegetation at 77% and 67%, respectively. Choosing 4 attention heads results in the highest value for forest at 74% which is tied with 3 attention heads. At 3 attention heads, forest and crop have the highest values at 74% and 85%, respectively. I decided to choose 3 attention heads because it has the best forest and crop values. It has the 2nd best vegetation value of 66% which is only 1 percentage point shy of the best value of 67%. 3 attention heads will be used in all subsequent experiments.

The next set of experiments focused on the number of attention layers. Since my default number of layers was 2, I ran these experiments on the elements of the set {1, 2, 3, 4}.

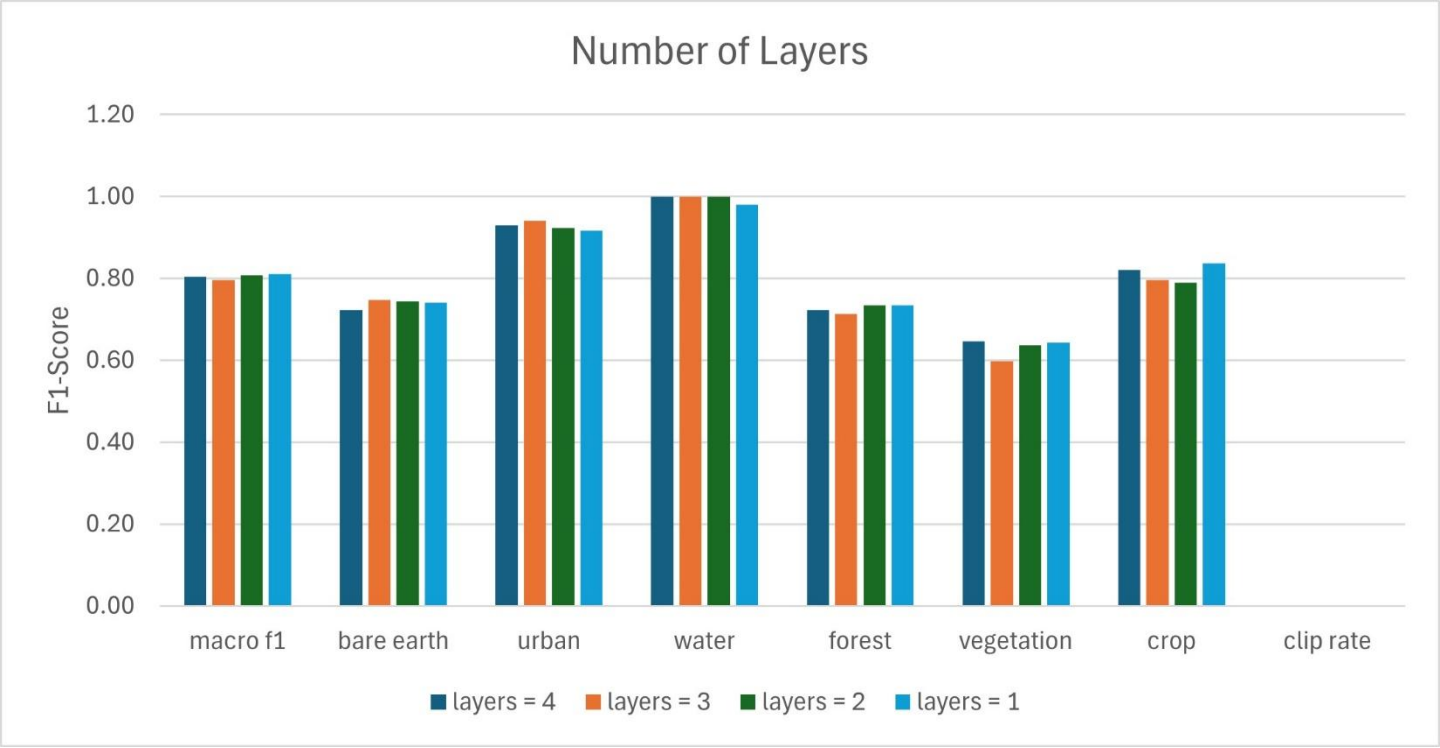


Figure 60 An ablation study showing how the F1-scores change as a result of changes in the number of attention layers.

Figure 60 shows a tie for the best F1-score of 81% between 1 and 2 layers. Having 2 attention layers results in a higher value for the water class at 100% while having 1 attention layer results in 98% for water and 84% for the crop class. Because water is normally at 100%, I consider a 2% drop in water is worth the 5% gain for the crop class. Thus, 1 attention layer will be used in subsequent experiments.

The next set of experiments were used to optimize β, particularly β_2 which controls how quickly the optimizer changes the step size for each parameter based on the gradient magnitude. As per **Error! Reference source not found.** β_1 and β_2 were set to 0.9 and 0.95, respectively.

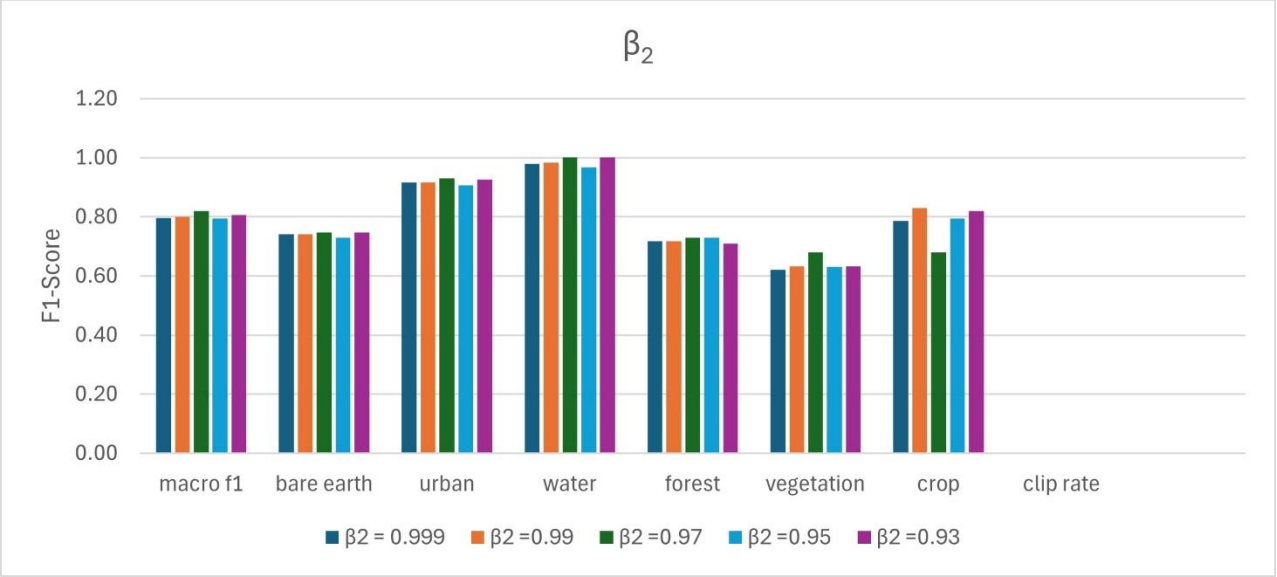


Figure 61 The results of an experiment for different values of β_2 showing 0.97 to be the best value of β_2.

As per Figure 61, the best value for β_2 is 0.97 as it results in the highest F1-score from all of the experiments.

With all of the main hyperparameters tuned, I decided to revisit the hard scale because of the difficulty in finding the optimal value. In this experiment, I only tested the values that had a F1-score of 82% or higher, namely ,5.50, 6.50, 7.75, 8.75, 9.0, and 9.50.

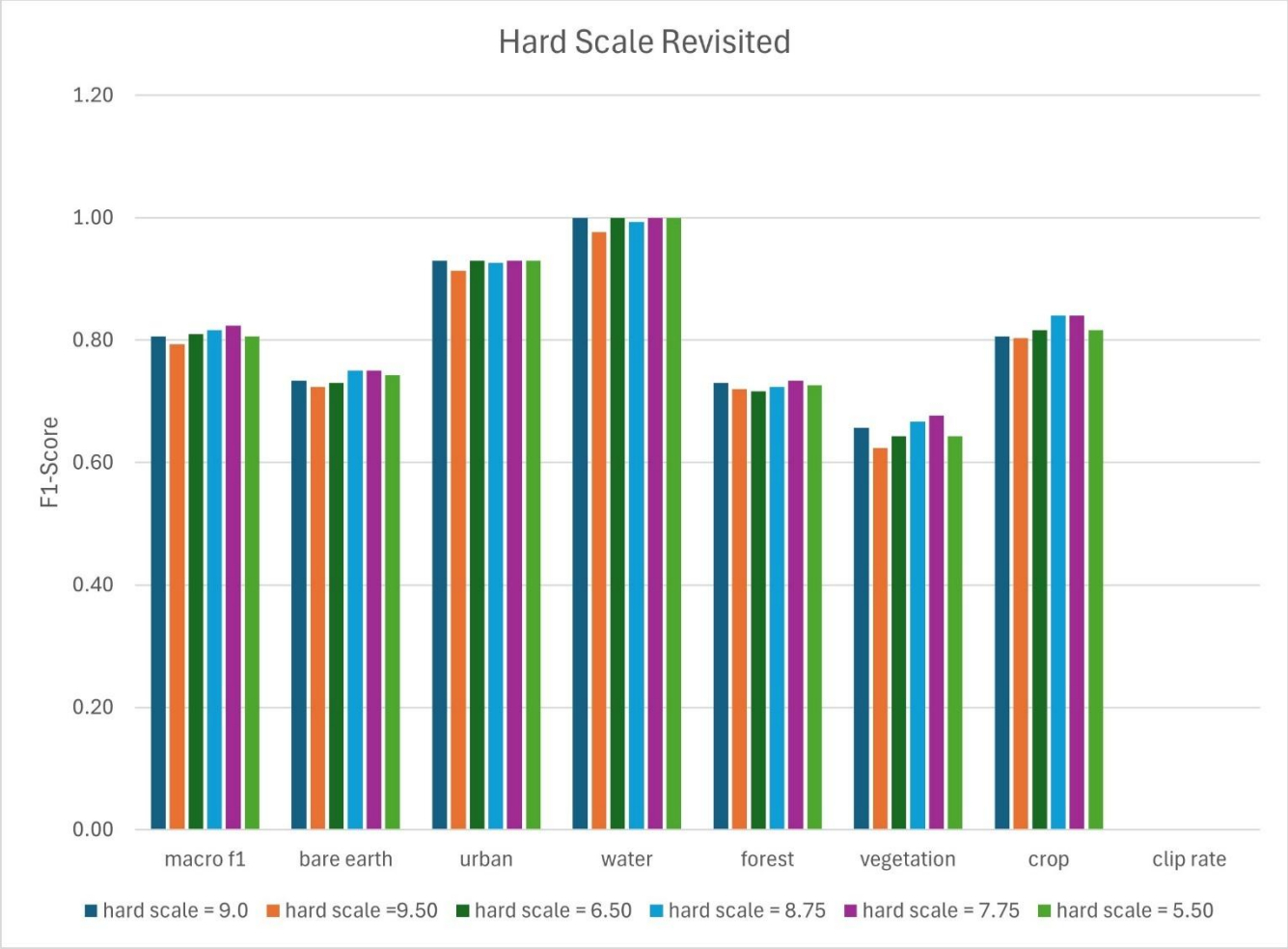


Figure 62 An experiment showing how the hard scale affects the F1-score. The value of 7.75 results in the highest accuracy of 82%.

Figure 62 shows that the best value for the hard scale is 7.75. The disparity between these experiments and previous ones could be due to a fault in the model, randomness in selecting training and test data, or to optimizing the other hyperparameters. In any event, I decided to use 7.75 as the value for hard scale in my final experiments.

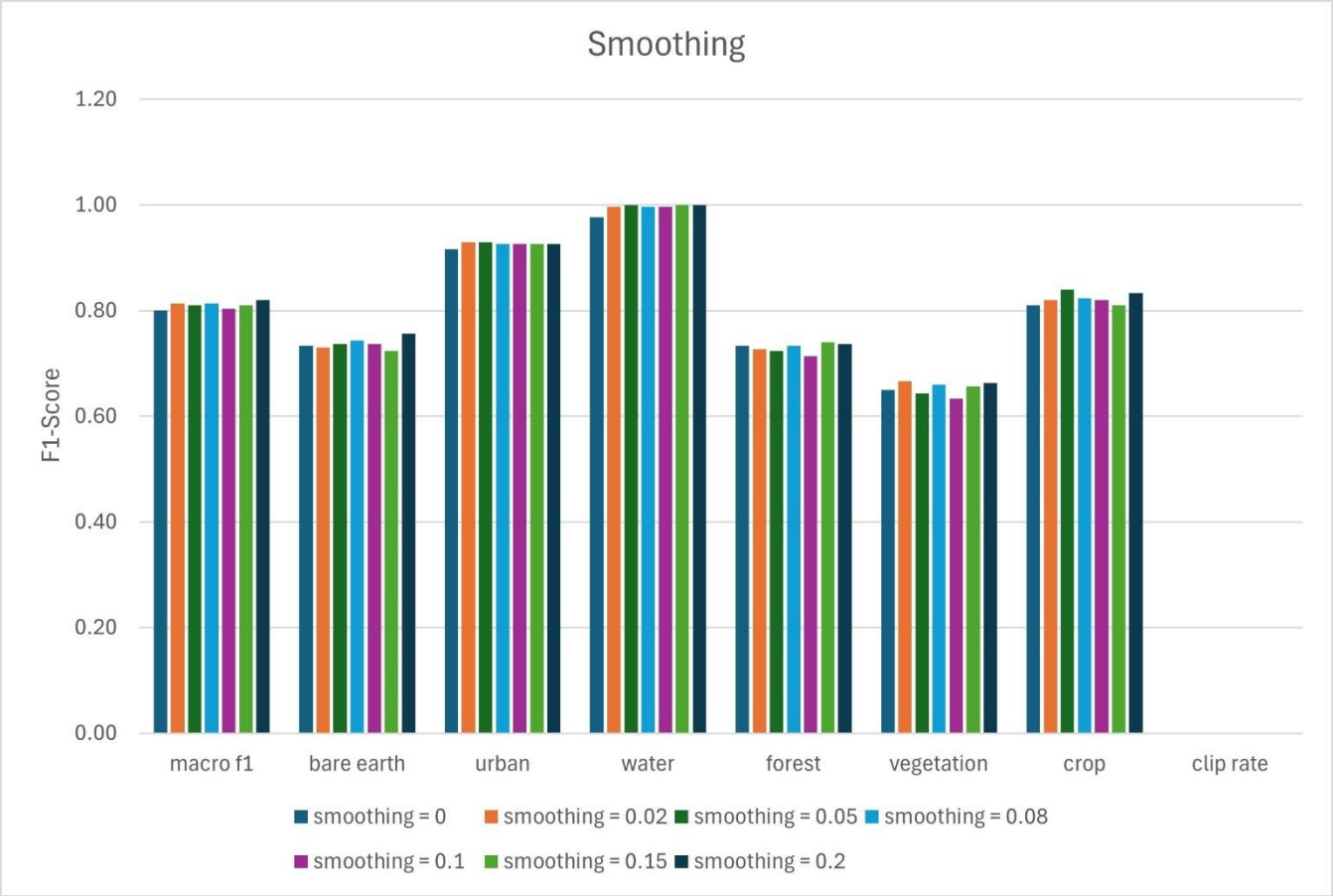


Figure 63 The results of experiments to determine the optimal value of smoothing.

Figure 63 shows the results of my smoothing experiments. Based on these results, 0.2 is the optimal value for smoothing. It results in the highest F1-score of 82%. Values of 0.1 and 0 result in 80% while the other values result in F1-scores of 81%.

Parameter	**Value**
Patch size	14
Patch stride	6
Model Depth	72
Number of heads	3
Number of Layers	1
Dropout	0.15
Lambda weight	1.0
Soft scale	3.25
Hard weight	2.25

Hard scale	7.75
Adaptive cap	0.5
Smoothing	0.2
Learning rate	1.72e-4
Minimum learning rate*	1.72e-4
Weight decay	1.6e-4
Decay rate*	0.8
Batch size	12
Maximum normalization	8.0
Maximum normalization high cap	8.0
Maximum normalization low cap	3.0
rho	1.2
quantile	0.9
beta	0.99
β_1	0.9
β_2	0.97
Warmup epochs	3
Ender	5

Table 7 The final values of the hyperparameters.

Now that all of the hyperparameters have been optimized I began the final experiments. The final values of the hyperparameters are shown in Table 7. For these experiments I tested to see if the use of thresholds and adaptive weights would enhance the model's performance compared to not using them at all. For these experiments I increased the maximum number of epochs to 100 just in case. In most of my experiments the number of epochs was less than 10 epochs per trial so I doubt that I'd ever get to 100 epochs. I also decreased the acceptable extent for the percent difference between epochs from 5% to 2%. Lastly, I decided to run each trial twice so that I could see if there were any changes amongst individual trials. I ended up averaging these results resulting in Figure 64.

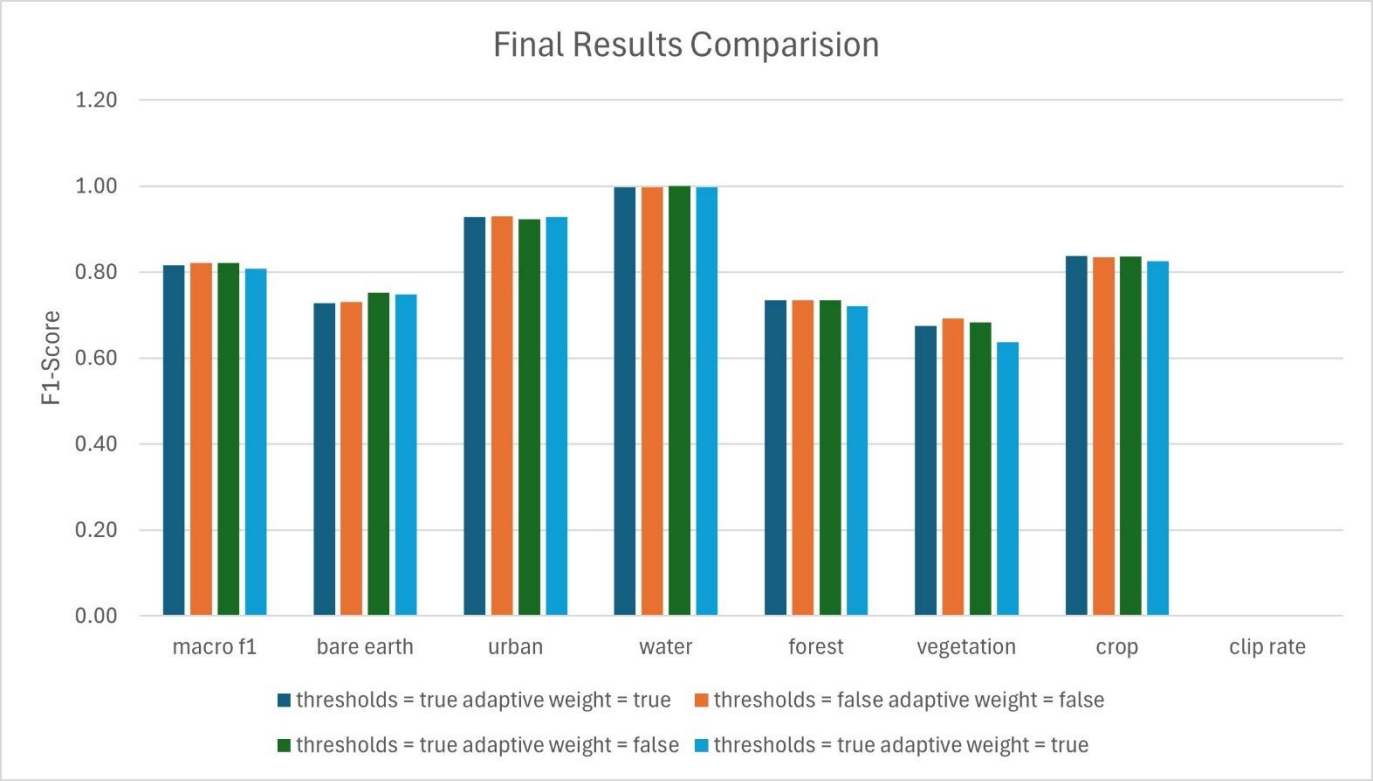


Figure 64 The final results of my experiments. False for thresholds and adaptive weight refers to the base model, while true for thresholds or adaptive weights refers to the use of thresholds.

Here, thresholds refers to using the remote sensing thresholds and giving the model rewards and penalties depending on how the model performs classifications. Adaptive weight refers to adjusting the strength of the thresholds depending on how well the model performs. For example, if the model correctly classifies a pixel using the thresholds it gets a reward to lower the loss but the weight shouldn't become negative, so it gives a smaller reward for lower values of the main loss. For my adaptive weight, I noticed that I was using the last value for the bonuses and penalties instead of averaging those values over all of the threshold entries in the entire batch. So I reran the experiment when thresholds and adaptive weights are in use. This run is represented by the navy blue in Figure 64. Based on these results the use of adaptive weights hasn't improved the model's performance.

Because of those results I revisited the code for my model to perform implementation checks. I standardized the random seed to ensure that my results could be easily reproduced. I also moved the exponential moving average model outside of the epoch loop to ensure that that EMA weights were accumulating over the full training session. I further refined the optimizer and scheduler logic by ensuring that the optimizer was being called before the scheduler. I also checked the parameters of

the optimizer to ensure that they were consistent throughout the program. Lastly, I reviewed my gradient accumulation logic and removed an extra gradient initialization. This ensures that the gradients accumulate properly before the optimizer step. Then I reran the final results of the model with and without my threshold logic. I ran each experiment twice with the same random seeds. I allowed them to run for up to 100 epochs, with a loop condition to stop after 5 epochs and the percent difference of the past 3 runs was less than 2%. The results are as follows.

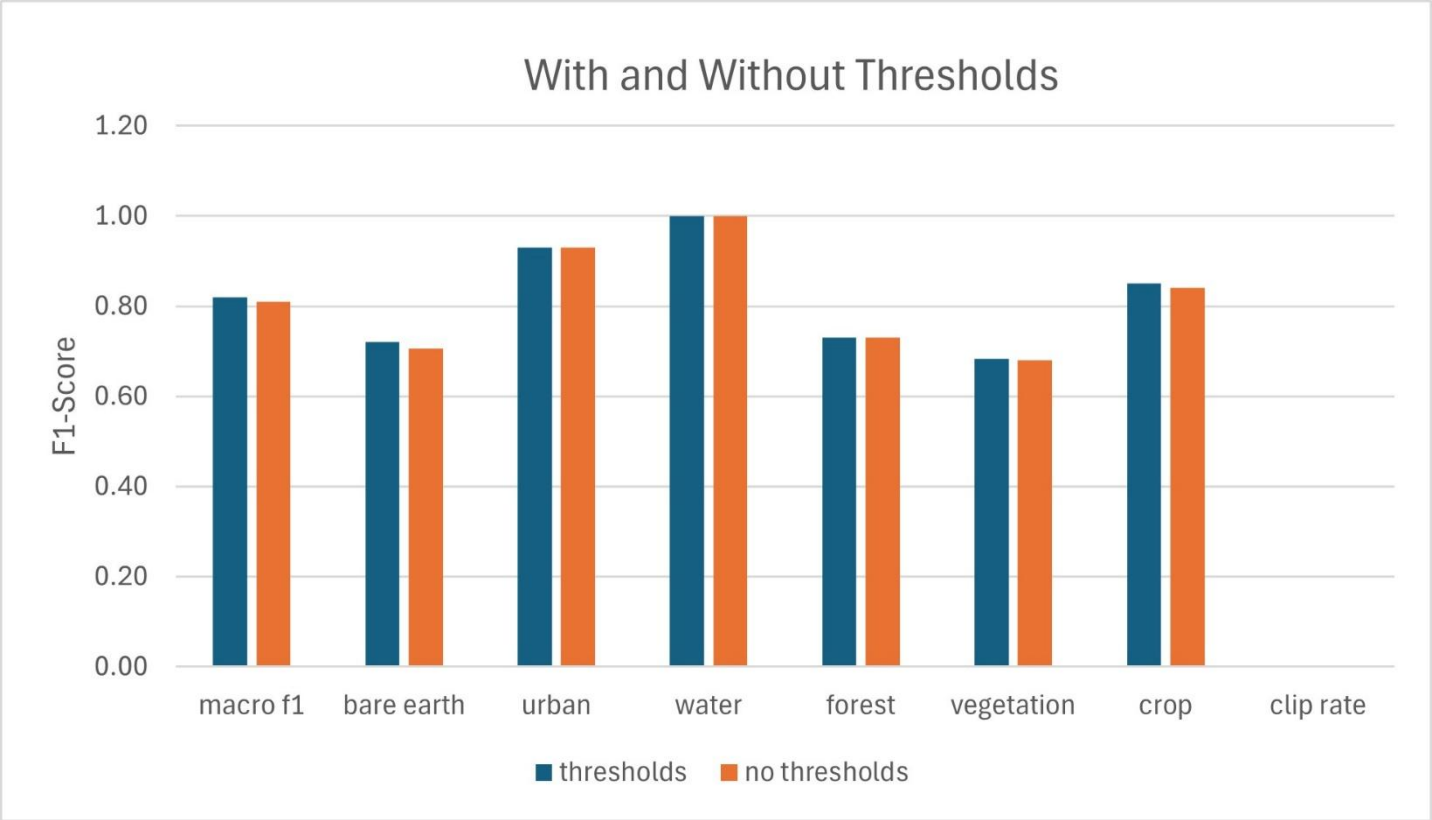


Figure 65 Ablation study showing the final results of the model using all of the results from the previous ablation studies.

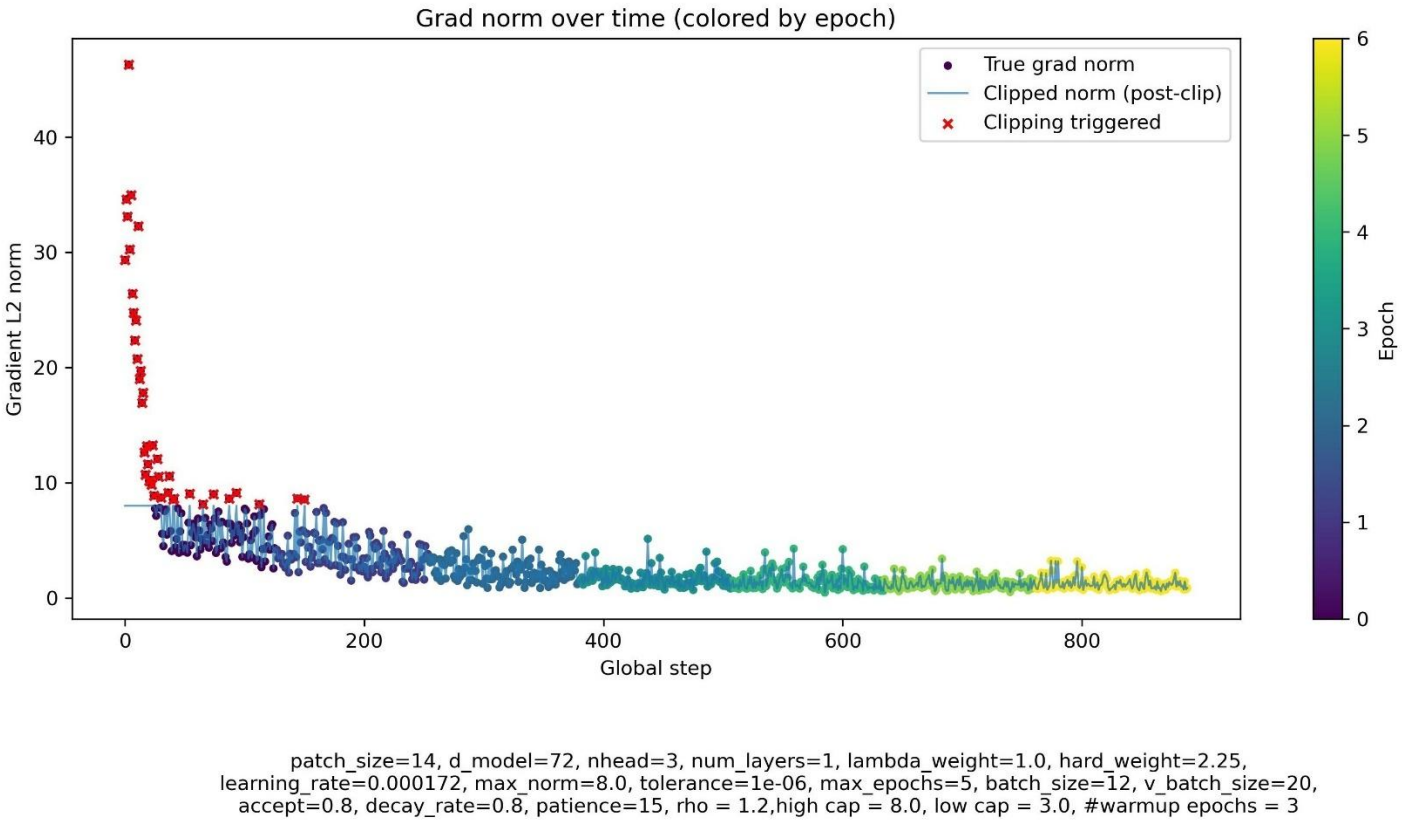
As per Figure 65, the model performs best with the thresholds. In this case, the F1-scores for each individual seed was the same for both runs which makes me confident that the results are more reproducible. The model clearly performs better when the thresholds are used. Overall, the model performs 1% better with the thresholds. For bare earth, there is a 1% improvement and crop had a 1% improvement. The results for urban, water, forest, and vegetation were identical. Using a random seed of 1 resulted in 8 epochs of training with and without using thresholds. The other random seeds resulted only 7 epochs of training. Just looking at the results for random seed 1, training began at 17:38:32 and ended at 19:11:34. This is a total training time of 1 hour 33 minutes and 2 seconds. In

comparison, when my threshold logic isn't used my experiment for random seed 1 began at 10:02:58 and ended at 11:17:56 resulting in a time of 1 hour 14 minutes and 58 seconds. Using my thresholds resulted in a gain of 1% in overall accuracy for 18 minutes and 4 seconds of extra time.

	Macro F1	Bare Earth	Urban	Water	Forest	Vegetation	Crop
With Thresholds	0.82	0.71	0.93	1.00	0.73	0.69	0.85
	0.81	0.70	0.93	1.00	0.73	0.67	0.84
	0.83	0.75	0.93	1.00	0.73	0.69	0.86
Average	0.82	0.72	0.93	1.00	0.73	0.68	0.85
Without Thresholds	.81	.70	0.93	1.00	0.73	0.68	0.84
	.80	.68	0.93	1.00	0.73	0.67	0.82
	.82	.74	0.93	1.00	0.73	0.69	0.86
Average	0.81	0.71	0.93	1.00	0.73	0.68	0.84

Table 8 The results of the final experiments for all three random seeds with and without thresholds.

Table 8 shows the results of my final experiments in order based on their random seed. It is curious to note that each random seed performed 1% better with thresholds. Table 8 also reveals that the best seed from the three that I chose was random seed 3.



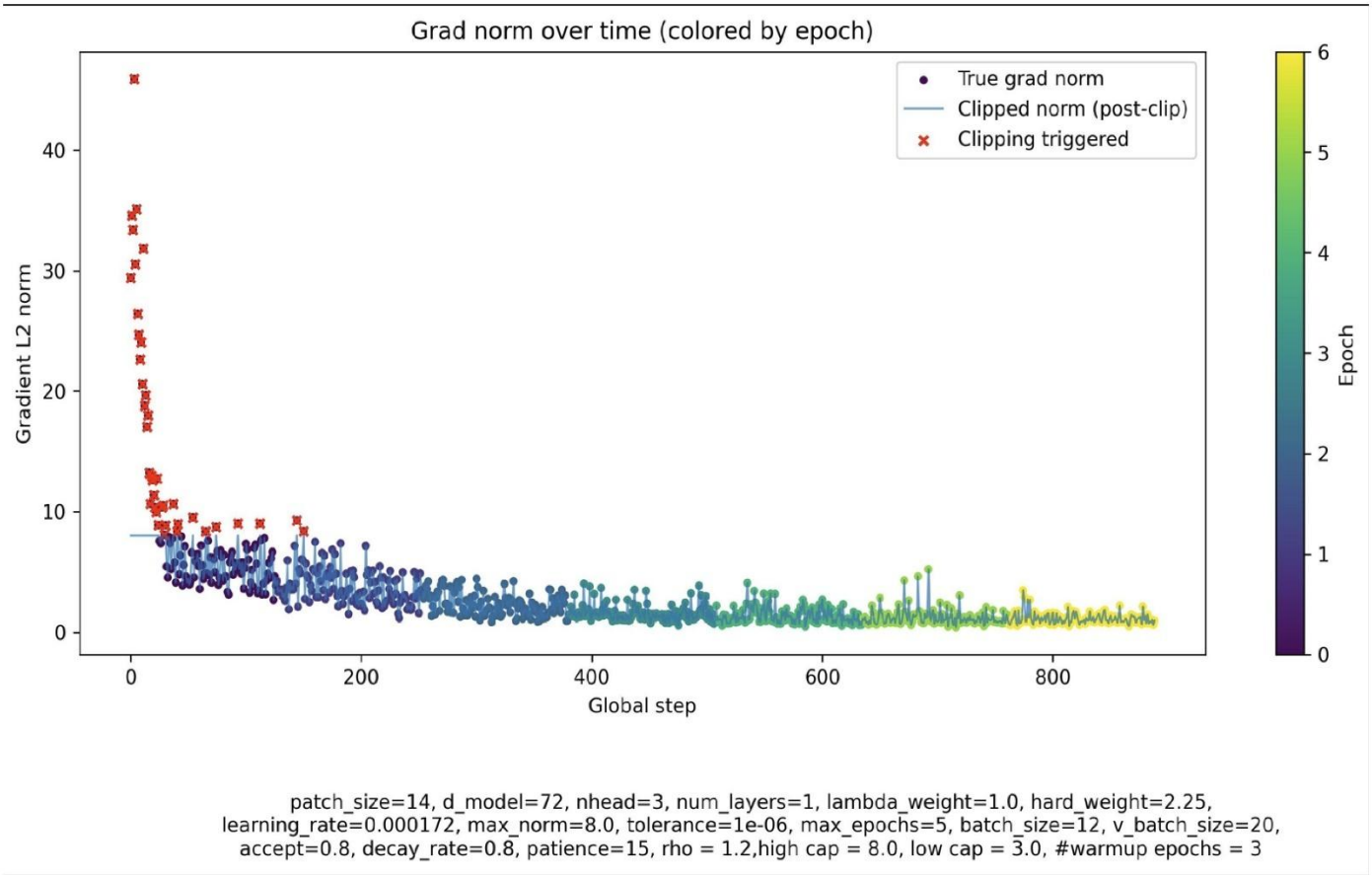


Figure 66 An image showing the gradient throughout the epochs for random seed 3. The top image uses thresholds while the bottom image doesn't use thresholds.

Figure 66 shows the results of my final experiments for random seed 3 with and without thresholds. The biggest difference between the two graphs occurs after 600 steps. From 600 - 900 steps. The overall curve is flatter and has spikes in the gradient. Without thresholds, the curve has more spikes, particularly between 600 and 800 steps.

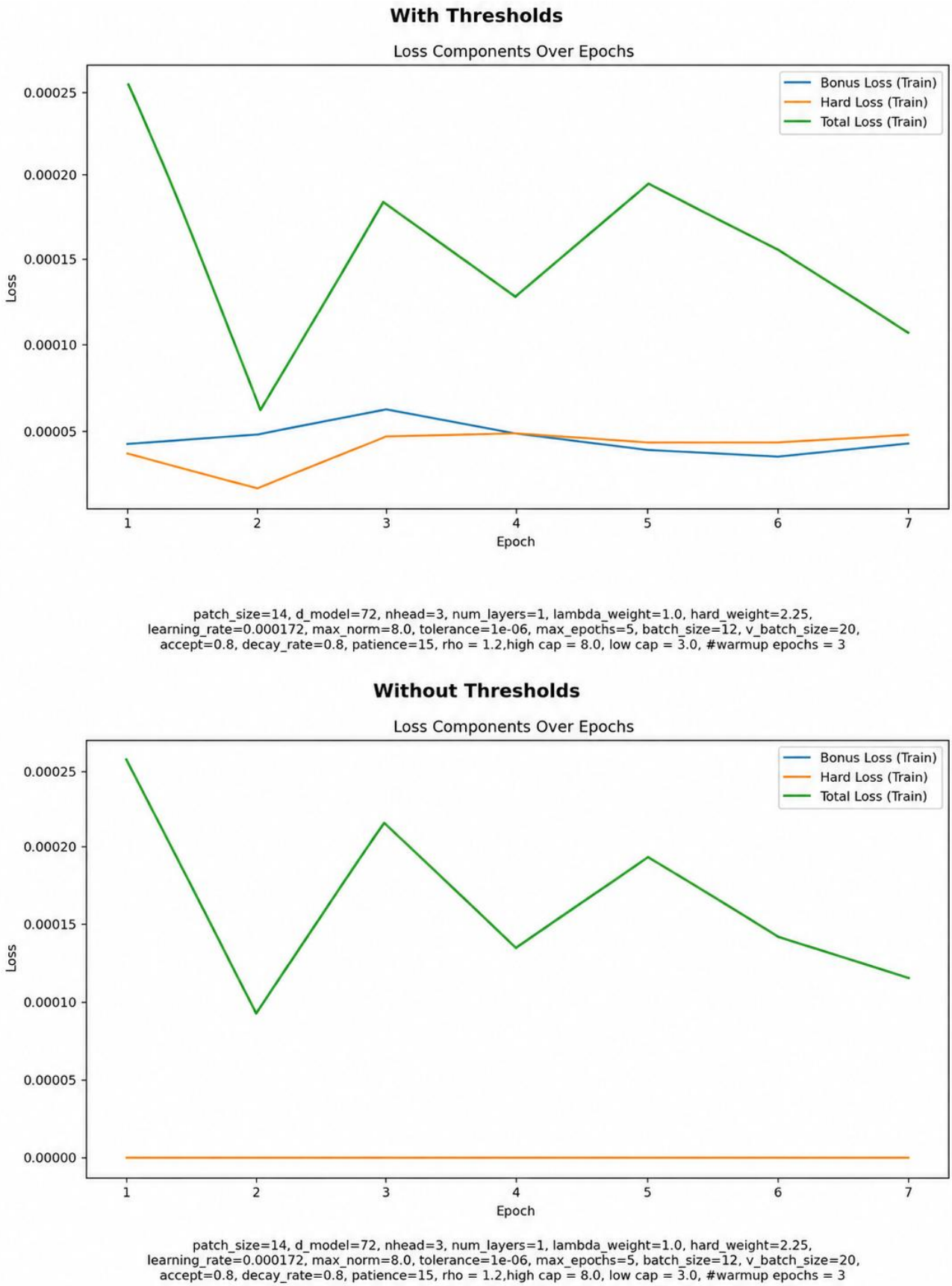


Figure 67 An image showing how the use of thresholds with adaptive weights affects the training loss.

Figure 67 shows how the use of thresholds affects the training loss. The top image uses thresholds while the bottom doesn't. The loss starts off in the same manner for both cases. However, when thresholds aren't used the loss is higher at the 3rd epoch going over 20e-4 while it doesn't when thresholds are used. Moreover, when thresholds are used the end of the curve, from epochs 6-7, produces a flatter curve than the one that doesn't use thresholds.

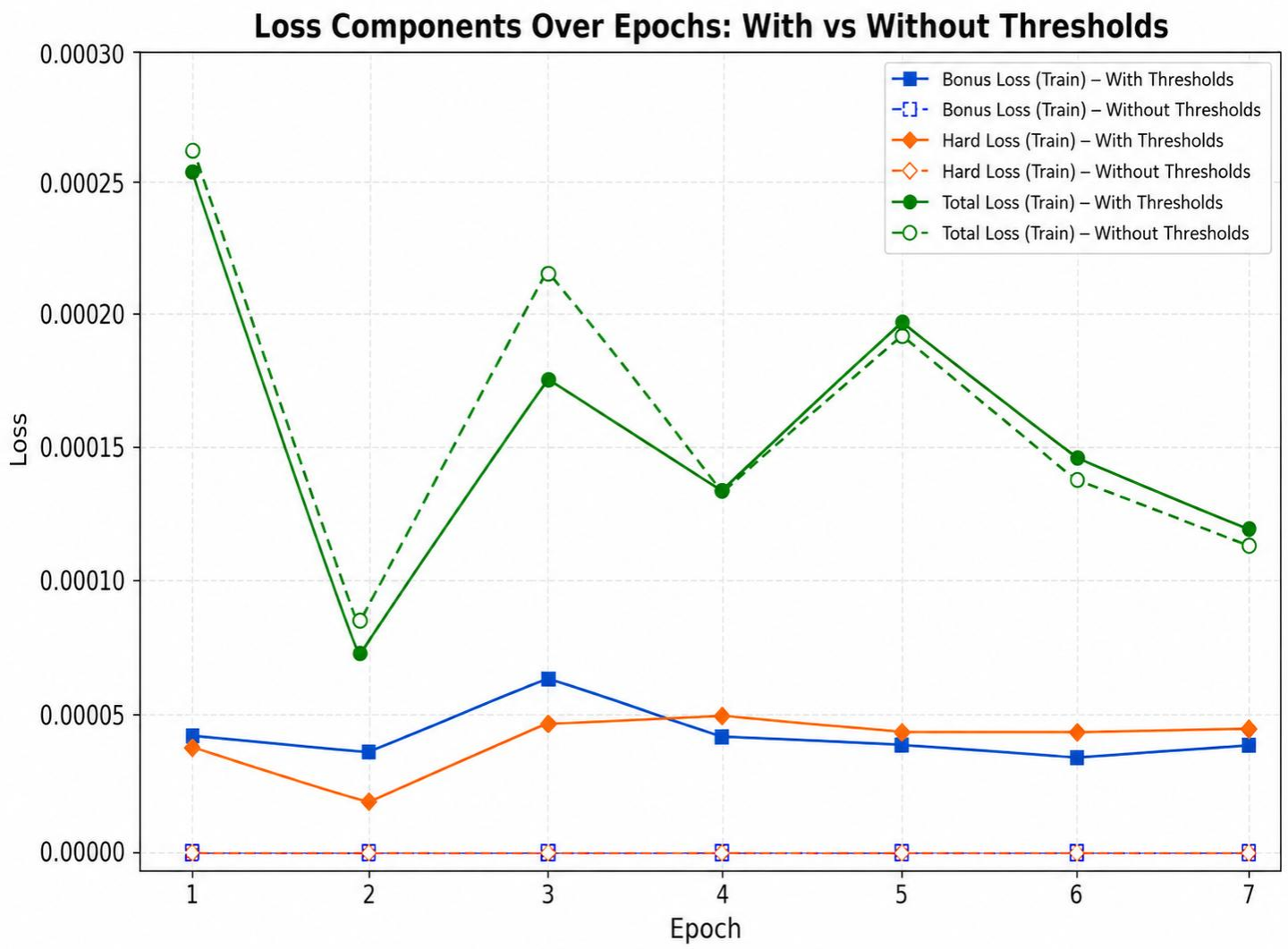


Figure 68 An image that shows the loss components with and without thresholds overlayed on the same graph.

Figure 68 shows the total loss with and without thresholds on the same graph. In this case we can clearly see that the total loss is almost always greater when thresholds aren't used.

Chapter 7 Conclusion

The goal of this thesis was to determine if the use of remote sensing thresholds would improve the performance of a neural network. While textbooks have ranges of values for various types of landcover classes, just using those values blindly won't yield the best results because of differences in the sensor, atmospheric conditions, the natural variability in land cover classes, anthropogenic changes in landcover classes, and the manmade delineation between classes such as bare earth and concrete. A human analyst is required to ensure that classes are correctly classified. A body of water may have vegetation in it that could make it harder to classify using the remote sensing indices alone, but the analyst would make the correct determination. Similarly, the roofs of buildings share spectral patterns with rocks or bare earth which can make them hard to properly classify. I have attempted to create a transformer that can use the remote sensing indices to perform such a classification.

Furthermore, the transformer's accuracy is higher when it uses the remote sensing indices than when it doesn't. The transformer does this by employing a system of rewards and penalties to the loss. The model's loss decreases when it makes an accurate prediction and increases when it doesn't. A direct consequence of using the remote sensing indices is that it enables one to understand how the model is classifying images by telling us which indices it used and which values were used for those indices.

The transformer is capable of achieving an accuracy of 82% using the training data. This is 1% better than when it doesn't use the remote sensing indices. The transformer performs best for the bare earth and crop classes with respective gains of 1% on average for 3 different random seeds.

This thesis reflects how my training data and transformer can improve the ability to properly classify remote sensing imagery faster than traditional methods. The dataset that I created uses Landsat 8 & 9 OLI imagery. While other datasets use Landsat or similar imagery, my dataset uses the NIR, SWIR 1, SWIR 2, TIRS 1, and TIRS 2 bands in addition to the traditional Red, Green, and Blue bands. I created 50 image-label pairs for 7 landcover classes with 10 additional label-image pairs for validation. I also used the remote sensing indices to classify each of the image and save those values in csv files. I didn't finish classifying all of the images for the snow class so their results were not used in this thesis, however, the model can be extended to account for other classes easily enough as long as they have threshold values accompanying them.

The transformer that I created uses those multi-spectral image-label pairs and the threshold values that I used to classify the images. As mentioned previously, it does so by assigning penalties and rewards to the model based on its predictions. The transformer is not only aware of the threshold values, it knows when it makes a good prediction or a bad prediction.

Limitations

The model has, like all things, some limitations. As mentioned previously, I didn't have time to classify any of the snow images. They would have been a wonderful addition to the model by allowing it to classify snow and ice it could be used to track the extent of sea ice and the motion of glaciers. Another limitation is the classification of the images. The classification could have been performed more accurately if time allotted as it was a lot of images to classify by myself. Another limitation was the crop data, I should have tried getting more imagery from different times in the planting season. Some of the fields were apparently already harvested while some were just beginning to show some growth. Because the model was validated using multispectral Landsat imagery, it may not work well for 3-band imagery such as NAIP aerial imagery. Furthermore, it should work with any other 30 meter resolution multispectral imagery, but it hasn't been tested on images from any other sensors such as Sentinel.

In regards to the model itself, the random seeds weren't used properly during most of the experiments. Thus, the results of those experiments will be harder to reproduce. There was also an issue with the exponential moving average, EMA, which was initialized once per epoch instead of per random seed during those experiments. The gradient accumulation logic also had to be revised after those experiments. Thus, there could be a better set of hyperparameters for training the model.

Error! Reference source not found. proves that there are better values for the hyperparameters. Reducing the patch size resulted in higher accuracies and better resolution in the model's predictions. Further research should reinvestigate the optimal values for the hyperparameters.

The reward and penalty system was somewhat coarse in that it gave the same reward and penalty to all classes in the model, a better approach would be to have a different lambda weight and hard weight for each class, but that would increase training time.

Another limitation is the reliance on ArcPy and E-Cognition. The code itself requires an ArcGIS license to use ArcPy. The model can be modified to work without it, however. The dataset requires E-Cognition to classify the images using the remote sensing indices, or a similar program. Since both ArcGIS and E-Cognition are proprietary, there are costs associated with their usage.

Future Work

This work has lots of potential for future work. The dataset could be improved by either performing the classification more accurately or by adding additional landcover classes such as the unfinished snow class. The crop class could be expanded to include more images with a higher crop percentage. Another interesting experiment would be to run the model without augmented data to see how the use of thresholds would compare with a smaller dataset. Also, the model should be rerun to optimize the values of all of the hyperparameters.

The transformer itself should be tested on a variety of images, ranging from multispectral to traditional RGB. It should also be tested on a variety of different spatial resolutions to see how the accuracy scales from low to high resolution imagery. If the model gives a lower accuracy on, for example, NAIP imagery, then the training data could be modified to include some NAIP imagery along with the thresholds used to classify the images.

The transformer should be modified to allow for tunable rewards and penalties for each class. It would be a lot of work because each class would have two additional hyperparameters that require training.

Retraining the transformer would be interesting because it would allow me to see if my mistake with the gradient accumulation would have thrown off any of the other hyperparameters. Furthermore, the other hyperparameters should be experimented on to see if any changes can increase the accuracy of the model.

This remote sensing transformer has proven to improve the accuracy when using the remote sensing thresholds than without. This indicates that smaller datasets can attain higher accuracies by using similar methods opposed to relying on large datasets.

Appendix A: Transformer Class Code

This is the code for my vision transformer that was used for land cover classification as per Chapter 5.

```
class ClassTransformer(nn.Module):
    def __init__(self,
                 selected_indices,
                 d_model=d_model,
                 patch_size = patch_size,
                 nhead=nhead, num_layers=num_layers,
                 num_classes = num_classes,
                class_names = class_names,
                 patch_stride = None):
        super().__init__()
        self.selected_indices = selected_indices
        self.index_bank = Spectral_Index_Bank()
        self.num_classes = num_classes
        self.patch_size = patch_size
        self.class_names = class_names
        self.patch_stride = patch_stride if patch_stride is not None else patch_size


        #The number of input chanells equals the numer of remote sensing indices plus the seperate bands from
the image.
        num_input_channels = len(selected_indices) + 8


        # GroupNorm is stabler than Batch Norm for small batches
        #Picks a group count that is the greatesst common divisor of the input channels and 32.
        #32 tends to work well with GroupNorm.
        gn_groups = max(1, gcd(num_input_channels, 32))
        self.input_norm = nn.GroupNorm(gn_groups, num_input_channels)


        #Convolutional feature extractor
        self.conv = nn.Conv2d(num_input_channels, d_model, kernel_size=3, padding=1)


        # Batch-first patch embedding with positional encodings
        self.patch_embed = PatchEmbedding(
            in_channels=d_model,
            patch_size = self.patch_size,
            d_model = d_model,
            stride = self.patch_stride
        )
```

```
        # Transformer encoder, pre-norm, with GELU with multilayer perceptron
        encoder_layer = TransformerEncoderLayer(
            d_model=d_model,
            nhead=nhead,
            dim_feedforward=4 * d_model,
            dropout=dropout,
            activation="gelu",
            batch_first=True,
            norm_first=True,
        )
        for m in encoder_layer.modules():
            if isinstance(m, nn.LayerNorm):
                m.eps = 1e-5

        # Silence nested-tensor warning while keeping norm_first=True
        self.transformer = TransformerEncoder(encoder_layer, num_layers=num_layers,
enable_nested_tensor=False)

        # Extra token pre-norm with a failsafe to prevent NANs
        self.token_norm = nn.LayerNorm(d_model, eps=1e-5)

        #classifer head
        self.classifier = nn.Linear(d_model, num_classes)

    def forward(self, x):

        #Failsafe check for NANs
        if torch.isnan(x).any():
            print("Input has nans!")
            print("min:", x.min(), "max:", x.max(), "mean:", x.mean())
            raise ValueError("nans in input!")

        # Tensor containing remote sensing indices and their names
        index_stack, names = self.index_bank(x)

        #=== keep oroginal spatial size to crop back later
        #H_in, W_in = x.shape [-2:]

        # Feature extractor with failsafes for NANs
        x = self.input_norm(x)
        features = self.conv(x)  # [B, D, H, W]
```

```
if torch.isnan(features).any():
    raise RuntimeError("NaN after conv")

#Get the height and width of the features tensor.
#Should have shape [B, D, H, W]
H_in, W_in = features.shape[-2:]

# Patches -> tokens [B, S, D]
#Create tokens from the feature map.
#Gives the patch grid and the padded patch sizes from patch embedding.
#Has a failsafe for NANs.
tokens, Hp, Wp, Hpad, Wpad = self.patch_embed(features)
if torch.isnan(tokens).any():
    raise RuntimeError("NaN in tokens")

#Normalizes and scales tokens.
tokens = self.token_norm(tokens) * (1.0 / math.sqrt(features.size(1)))



#Feed tokens to the transformer.
#Failsafe for NANs.
encoded = self.transformer(tokens)       # [B, S, D]
if torch.isnan(encoded).any():
    raise RuntimeError("NaN in transformer output")

#Classify each patch token.
#Failsafe for NANs.
patch_logits = self.classifier(encoded)  # [B, S, C]
if torch.isnan(patch_logits).any():
    raise RuntimeError("NaN in patch logits")

#Get the batch size
B = patch_logits.size(0)

#Turn the patch logits back into a 2d patch grid.
patch_map = patch_logits.transpose(1, 2).reshape(B, self.num_classes, Hp, Wp)

#Upsample patch grid logits to pixel grid logits using the padded patch size.
pixel_logits_pad = F.interpolate(patch_map, size=(Hpad, Wpad), mode="bilinear", align_corners=False)

#Gets the spatial size of the input tensor
H,W = x.shape[-2:]
```

```
#Crops off the extra padding so that the input and output features have the same size.
pixel_logits = pixel_logits_pad[..., :H_in, :W_in]

return pixel_logits
```

Appendix B: Spectral Index Bank Class Code

This is the code for my spectral index bank. Herein are the raw bands from the Landsat imagery as well as the remote sensing indices that were calculated based on those bands.

```
class Spectral_Index_Bank(nn.Module):
    def __init__(self):
        super().__init__()
    def forward(self,x):
        scale_factor = 1.0#10000.0 #normalize the reflectance bands
        blue = x[:,0]/scale_factor
        green = x[:,1]/scale_factor
        red = x[:,2]/scale_factor
        nir = x[:,3]/scale_factor
        swir1 =x[:,4]/scale_factor
        swir2 = x[:,5]/scale_factor
        tirs1 = x[:,6]/1000.0 # normalization for Kelvin



        k2 = 1321.0789
        k1 = 774.8853
        ML =  3.3420E-04
        AL = 0.10000

        # Helper function to avoid divide-by-zero
        def safe_divide(numerator, denominator, eps=1e-6):
            return numerator / (denominator + eps)

        # Calculate each index
        awei = 4 * (green - swir1) - (0.25 * nir + 2.75 * swir2)
        ndvi = safe_divide(nir - red, nir + red)
        ndvi_test = safe_divide(nir - red, nir + red)
        bsi = safe_divide((swir1 + red) - (nir + blue), (swir1 + red) + (nir + blue))
        bui = safe_divide((swir1 + nir) - (red + blue), swir1 + nir + red + blue)
        bu = bui
        ndbi = safe_divide(swir1 - nir, swir1 + nir)
        mndwi = safe_divide(green - swir1, green + swir1)
        ndwi = safe_divide(green - nir, green + nir)
        ndsi = safe_divide(green - swir1, green + swir1)
        savi = 1.5 * safe_divide(nir - red, nir + red + 0.5)
        ui = safe_divide(swir2 - nir, swir2 + nir)
```

```
        ndbal = safe_divide(swir1 - tirs1, swir1 + tirs1)
        evi = 2.5 * safe_divide(nir - red, nir + (6 * red) - (7.5 * blue) + 1)
        dbi =safe_divide(blue -tirs1, blue + tirs1) - ndvi
        ebbi = safe_divide((swir1 - nir), torch.sqrt(10 * swir1 + tirs1))



        # Brightness temperature and other temperature based indices
        T = k2 / torch.log((k1 / (tirs1 + 1e-6)) + 1)
        L = ML * tirs1 + AL  # Spectral radiance

        # Indices that depend on other indices
        ndsi_mean = (mndwi + nir + swir1) / 3
        ndisi = safe_divide(T - ndsi_mean, T + ndsi_mean)
        #ndisi = ndsi_mean

        # IBI depends on NDBI, SAVI, MNDWI
        ibi_numerator = ndbi - (savi + mndwi) / 2
        ibi_denominator = ndbi + (savi + mndwi) / 2
        ibi = safe_divide(ibi_numerator, ibi_denominator)

        indices = torch.stack([
            awei, bsi, bu, dbi, ebbi, evi, ibi, mndwi, ndbal, ndbi, ndisi, ndsi, ndvi, ndwi, savi, ui],
dim=1)

        ndvi_stacked = indices[:,12]
        savi_stacked = indices[:,14]

        index_names = ["AWEI", "BSI", "BU", "DBI", "EBBI", "EVI", "IBI", "MNDWI", "NDBAL", "NDBI",
"NDISI","NDSI", "NDVI", "NDWI", "SAVI", "UI"]

        return indices, index_names
```

Appendix C: ArcPy Raster Dataset Class Code

This is the code for my dataset. It takes in the raw images, the classified raster images, the remote sensing thresholds, and the remote sensing indices. It returns a tensor stack containing the images and the remote sensing indices, the labels, and the threshold values for the remote sensing indices.

```
class ArcPyRasterDataset(Dataset):
    def __init__(self, image_paths, raster_paths, thresholds_dict, num_classes,crop_size = (80,80), transform=None, use_thresholds = True, spectral_bank = Spectral_Index_Bank(),
                 classaware = False, rare_ids = (0,5), rare_prob = 0.7, min_pix_map = None, tries = 20, jitter = 0.25, ignore_index = 255, global_invfreq = None):
        self.image_paths = image_paths
        self.raster_paths = raster_paths
        self.thresholds_dict = thresholds_dict if thresholds_dict is not None else{}
        self.thresholds_dict = {
            os.path.normpath(k).lower():v
            for k,v in thresholds_dict.items()
        } if thresholds_dict is not None else {}
        self.crop_size = crop_size
        self.transform = transform
        self.use_thresholds = use_thresholds
        self.num_classes = num_classes
        self.spectral_bank = spectral_bank
        self.thresholds_dict2 = thresholds_dict if thresholds_dict is not None else {}
        self.ignore_index = ignore_index
        self.classaware = classaware
        self.rare_ids = set(rare_ids)
        self.rare_prob = float(rare_prob)
        self.min_pix_map = dict(min_pix_map or {})
        self.tries = int(tries)
        self.jitter = float(jitter)
        self.global_invfreq = (np.asarray(global_invfreq, dtype = np.float32) if global_invfreq is not None else None)

    def __len__(self):
        return len(self.image_paths)



    def center_crop(self, img_tensor, lbl_tensor):
        _,h,w = img_tensor.shape
        th,tw = self.crop_size
```

```
    if h < th or w < tw:
        raise ValueError(f" Image too small for cropping: got {h}x{w}, need {th}x{tw}")

        #calculate center crop

    top = (h-th)//2
    left = (w-tw)//2
    img_cropped = img_tensor[:, top:top+th, left:left+tw]
    lbl_cropped = lbl_tensor[top:top + th, left:left+tw]
    return img_cropped, lbl_cropped



def _random_crop(self, img, lbl):
    th, tw = self.crop_size
    H, W = lbl.shape[-2], lbl.shape[-1]
    i = np.random.randint(0, max(1, H - th + 1))
    j = np.random.randint(0, max(1, W - tw + 1))
    return img[:, i:i+th, j:j+tw], lbl[i:i+th, j:j+tw]

def _crop_around_class(self, img, lbl, target_id, min_pix=200, tries=20, jitter=0.25):
    th, tw = self.crop_size
    H, W = lbl.shape[-2], lbl.shape[-1]
    ys, xs = torch.nonzero(lbl == target_id, as_tuple=True)
    if ys.numel() == 0:
        return self._random_crop(img, lbl)

    k = torch.randint(0, ys.numel(), (1,)).item()
    i = int(np.clip(int(ys[k]) - th//2, 0, max(0, H - th)))
    j = int(np.clip(int(xs[k]) - tw//2, 0, max(0, W - tw)))

    for _ in range(tries):
        patch_lbl = lbl[i:i+th, j:j+tw]
        if (patch_lbl == target_id).sum().item() >= min_pix:
            return img[:, i:i+th, j:j+tw], patch_lbl
        di = int(np.random.uniform(-th*jitter, th*jitter))
        dj = int(np.random.uniform(-tw*jitter, tw*jitter))
        i = int(np.clip(i + di, 0, max(0, H - th)))
        j = int(np.clip(j + dj, 0, max(0, W - tw)))

    return img[:, i:i+th, j:j+tw], patch_lbl  # fallback

def _choose_target_id(self, lbl):
```

```
        # classes present (drop ignore)
        present = torch.unique(lbl[lbl != self.ignore_index]).tolist()
        if not present:
            return None
        # prioritize rare_ids if present
        present_rare = [c for c in present if c in self.rare_ids]
        if present_rare and np.random.rand() < self.rare_prob:
            if self.global_invfreq is not None:
                w = np.array([self.global_invfreq[c] for c in present_rare], dtype=np.float64)
                p = (w / w.sum()) if w.sum() > 0 else None
                return int(np.random.choice(present_rare, p=p))
            return int(np.random.choice(present_rare))

        # otherwise weighted by global rarity if provided
        if self.global_invfreq is not None:
            w = np.array([self.global_invfreq[c] for c in present], dtype=np.float64)
            p = (w / w.sum()) if w.sum() > 0 else None
            return int(np.random.choice(present, p=p))

        return None



    def __getitem__(self, idx):
        try:
            #full_path_key = self.image_paths[idx]
            if idx >= len(self.image_paths) or idx >= len(self.raster_paths):
                raise IndexError(f"Index {idx} out of range for dataset of size {len(self.image_paths)} with
{len(self.raster_paths)} raster paths")


            full_path_key = os.path.normpath(self.raster_paths[idx]).lower()
            path = self.image_paths[idx]
            bands = [arcpy.RasterToNumPyArray(arcpy.Raster(path + f"/Band_{i}"))for i in range(1, 8)]  #
using bandns 2 - 7 and 10-11 from Landsat 8 & 9
            img = np.stack(bands, axis=0)
            img_tensor = torch.tensor(img).float()


            #normalize
            img_tensor = img_tensor / 65535.0


            #load labeled images
            label_path_key = self.raster_paths[idx]
            label_raster = arcpy.RasterToNumPyArray(self.raster_paths[idx])
            label_tensor = torch.tensor(label_raster, dtype=torch.long)
```

```
            #=== remap label values
#=== skip classes 6 and 10
#remap_array = torch.full((256,), IGNORE_LABEL, dtype=torch.long)
remap_array = torch.full((256,), self.ignore_index, dtype=torch.long)


label_map = {0:0, 1:1, 2:2, 3:3, 4:4, 5:5, 6:6, 7:6,8:6, 9:6}
for old, new in label_map.items():
    remap_array[old] = new
label_tensor=remap_array[label_tensor]


#safety mask
valid = (label_tensor >= 0) & (label_tensor < self.num_classes)
label_tensor = torch.where(valid, label_tensor, torch.full_like(label_tensor, self.ignore_index))

# labels should be 2D for cropping logic
lbl2d = label_tensor.squeeze(0)  # [H,W]

th, tw = self.crop_size
_, H, W = img_tensor.shape

if self.classaware:
    # choose a class to focus on (returns None if nothing usable)
    target_id = self._choose_target_id(lbl2d)
    if target_id is None:
        # random crop fallback
        img_patch, lbl_patch = self._random_crop(img_tensor, lbl2d)  # img[C,th,tw], lbl[th,tw]
    else:
        # how many pixels of that class we want to see in the patch (tune per class)
        min_pix = int(self.min_pix_map.get(target_id, 200))
        img_patch, lbl_patch = self._crop_around_class(
            img_tensor, lbl2d, target_id,
            min_pix=min_pix, tries=self.tries, jitter=self.jitter
        )
else:
    # deterministic center crop for both tensors
    if H < th or W < tw:
        raise ValueError(f"Image too small for cropping: got {H}x{W}, need {th}x{tw}")
    top  = (H - th) // 2
    left = (W - tw) // 2
```

```
            img_patch = img_tensor[:, top:top+th, left:left+tw]  # [C,th,tw]
            lbl_patch = lbl2d[top:top+th, left:left+tw]          # [th,tw]

        # restore channel dimensions labels to  [1,H,W]
        label_tensor = lbl_patch.unsqueeze(0)  # [1,th,tw]
        img_tensor   = img_patch               # [C,th,tw]
        # --- end CROPPING ---







        #ignore values that are out of range
        mask_valid = (label_tensor >= 0) & (label_tensor <  self.num_classes)
        label_tensor = torch.where(
            mask_valid,
            label_tensor,
            #torch.full_like(label_tensor, IGNORE_LABEL)
            torch.full_like(label_tensor, self.ignore_index)
        )

        #=== compute indices, shape = [16, H, W]
        img_tensor_batched = img_tensor.unsqueeze(0) #shape should be [1,7,H,W]
        indices, _ = self.spectral_bank(img_tensor_batched)
        indices = indices.squeeze(0)
        ndvi = indices[12]
        #print(f"NDVI (after squeeze) in ArcPyRasterDataset range: min={ndvi.min().item():.4f},
max={ndvi.max().item():.4f}")



        #=== combine indices, shape = [16, H, W]
        combined = torch.cat([img_tensor, indices], dim = 0)

        #=== set invalid labels to IGNORE
        mask_valid = torch.isin(label_tensor, torch.tensor(list(label_map.values())))
        label_tensor = torch.where(mask_valid, label_tensor, torch.full_like(label_tensor,
self.ignore_index))



        #get thresholds
        #full_path_key comes from the clipped images folder which is a different directory than the
raster folder
        #solution try to use a path key corresponding to the raster path
```

```
            if self.use_thresholds:
                full_path_key = os.path.normpath(self.raster_paths[idx]).lower()
                thresholds = self.thresholds_dict.get(full_path_key,[])
                threshold_entries = self.thresholds_dict.get(full_path_key, [])
                threshold_O = self.thresholds_dict2.get(full_path_key)

                if full_path_key not in self.thresholds_dict:
                    print(f"[MISSING] {full_path_key}")
                    print("Available keys sample:")
                    for i, key in enumerate(list(self.thresholds_dict.keys())[:5]):
                        print(f"  {i+1}. {key}")
                    matches = difflib.get_close_matches(full_path_key, list(self.thresholds_dict.keys()),
n=3, cutoff=0.5)
                    print("Did you mean:")
                    for match in matches:
                        print(f"  {match}")
                    raise KeyError(f"Thresholds not found for file: {full_path_key}")

                threshold_tensors = []
                index_names = ["AWEI", "BSI", "BU", "DBI", "EBBI", "EVI", "IBI", "MNDWI", "NDBAL", "NDBI",
"NDISI","NDSI", "NDVI", "NDWI", "SAVI", "UI"]

                for entry in threshold_entries:
                    num_indices = len(index_names)
                    threshold_tensor = torch.full((num_indices, 2), float('nan'), dtype=torch.float32)
                    thresholds_list = []
                    for i, spec_range in enumerate (entry['indices']):

                        if not spec_range:
                            continue

                        spec_range = spec_range.strip()
                        if spec_range.lower() in {"nan", "none", "null"}:
                            continue
                        if ',' in spec_range:
                            try:
                                cleaned_parts = []
                                for part in spec_range.split(','):
                                    cleaned_parts.append(
                                        part.replace("min=", "").replace("max=", "").strip()
                                    )

                                min_val, max_val = map(float, cleaned_parts)
```

```
                            threshold_tensor[i] = torch.tensor([min_val, max_val], dtype=torch.float32)

                        except Exception as e:
                            print(f"Failed parsing index {i} for {entry['class']}: {spec_range} ({e})")
                        except ValueError:
                            print(f" Skipping invalid threshold for class {entry['class']}:
'{spec_range}'")
                            continue

                    threshold_tensors.append(threshold_tensor)

                #=== final thresholds shape [number of classes per file, number of indices, 2
                thresholds = torch.stack(threshold_tensors, dim=0) if threshold_tensors else
torch.zeros((0,0,2), dtype=torch.float32)
            else:
                thresholds = torch.zeros((0,0,2), dtype=torch.float32)
                threshold_O = []
            if self.transform:
                combined = self.transform(combined)

            return combined, label_tensor, thresholds, full_path_key, label_path_key, threshold_O

        except Exception as e:
            print(f"Error at index {idx} for file: {self.image_paths[idx]}")
            raise e
```

Appendix D: Thresholds If Statement Code

This is the code pertaining to my use of thresholds. Awards are given when image patches for a given class are within range of the threshold. Penalties are awarded when image patches for a given class fall outside of the range of the threshold values.

```
                if use_thresholds:

                    #initialize the loss variables with a value of 0.0
                    bonus_loss = torch.tensor(0.0, device=device)
                    hard_loss = torch.tensor(0.0, device=device)
                    patch_loss = torch.tensor(0.0, device=device)

                    #pull index_stack and index_names from the model
                    index_stack, index_names = model.index_bank(images)

                    #=== clamp index stack to ensure it doesn't blow up by setting nan to 0, and inf to 1e6
                    index_stack = torch.nan_to_num(index_stack, nan=0.0, posinf=1e6, neginf=-1e6)

                    #test the values of index_stacked
                    r'''
                    ndvi_stacked_gogeta = index_stack[:,1]
                    savi_stacked_gogeta = index_stack[:,8]
                    print(f"NDVI (in model) range: min={ndvi_stacked_gogeta.min().item():.4f},
max={ndvi_stacked_gogeta.max().item():.4f}")
                    print(f"SAVI (in model) range: min={savi_stacked_gogeta.min().item():.4f},
max={savi_stacked_gogeta.max().item():.4f}")
                    '''
                    #map each remote sensing index to a numerical value
                    name_to_idx = {name:i for i, name in enumerate(index_names)}

                    #initialize dictionaries to calculate the penalities and bonuses per class
                    class_bonus_losses = {class_ : 0.0 for class_ in model.class_names}
                    class_hard_losses = {class_ : 0.0 for class_ in model.class_names}

                    #=== initialize threshold loss per image
                    batch_bonus_loss = torch.tensor(0.0, device = device)
                    batch_hard_loss = torch.tensor(0.0, device=device)

                    #initalize the number of images with bvalid thresholds
                    valid_image_count = 0
```

```
####==== make softmax
probs = torch.softmax(outputs,dim=1)


#=== create class aliases for image classification
class_aliases = {
    "bare earth" : "bare earth",
    "bare eath" : "bare earth",
    "urban" : "urban",
    "forest" : "forest",
    "foreset" : "forest",
    "grass" : "vegetation",
    "vegetation" : "vegetation",
    "grassland" : "vegetation",
    "crop" : "crop",
    "water" : "water",
    "ignore" : "IGNORE LABEL"
}

#loop through all of the images
for b in range(images.size(0)):

    #pull thresholds fromthe model
    thresholds = threshold_O[b]

    #skip images that don't have thresholds and print the filepath
    if thresholds is None:
        print(f" Skipping image at batch index {b} due to missing thresholds.")
        print(f" Problem file: {full_path_key[b]}")
        continue

    #initialize variables for indicies, bonuses, and penalties
    valid_idxs = []
    bonuses = []
    bonuses_b =[]
    penalties = []
    penalties_b = []
    valid_idxs_b = []
    temp_threshold_dict = {}

    #loop through every entry in thresholds
```

```
                    for entry in thresholds:

                        #get the class name for the entry
                        class_name = entry['class']

                        #clean the class name
                        raw_class = str(entry.get("class", "")).strip().lower().replace("_", " ")

                        #use class_aliases to map raw class to the correct class name
                        mapped_name = class_aliases.get(raw_class, raw_class)

                        #check to ensure that mapped name exists and one of the class names
                        if mapped_name is None or mapped_name not in model.class_names:
                            #print(f"     Skipping {class_name}, not in model.class_names")
                            print(f"     Skipping {raw_class}, not in model.class_names")
                            print(f" mapped_class is {mapped_name}")
                            print(f" class names are {model.class_names}")
                            continue

                        #print(f"class name = {class_name} , mapped name = {mapped_name}")
                        if not mapped_name:
                            #print(f"  Skipping `{class_name}`: no alias defined.")
                            print(f"  Skipping `{raw_class}`: no alias defined.")
                            continue

                        if mapped_name not in class_names:
                            #print(f"  Skipping `{class_name}` (mapped to `{mapped_name}`): not in
model.class_names.")
                            print(f"  Skipping `{raw_class}` (mapped to `{mapped_name}`): not in
model.class_names.")
                            continue

                        #retrieve the class ID based on mapped_name
                        class_idx = model.class_names.index(mapped_name)
                        #print(f"the mapped class result is {model.class_names.index(mapped_name)} and
the class names result is {model.class_names.index(class_name)}")

                        #check to ensure class name matches mapped name
                        if raw_class == mapped_name:
                            #print(f"{class_name} is in the model's class names.")
                        else:
                            #print(f" Apparently {class_name} is not equal to {mapped_name}.")
                            print(f" Apparently {raw_class} is not equal to {mapped_name}.")
```

```
    #print(f" the index is {class_idx} and the class name is {class_name}")
    print(f" the index is {class_idx} and the class name is {raw_class}")
    print(f" the mapped class is {mapped_name}")
    print("_" * 80)

#==== build mask position in indices list
#extract indicies from index_stack
idx_b = index_stack[b]

#create mask for values in range of threshold values for an image
in_range_mask = build_in_range_mask_from_indices_entry(entry, idx_b, index_names)

#ensure that the mask exists otherwise skip to the next entry
if in_range_mask is None:
    # no valid thresholds for this entry
    continue

#create the mean of of the mask and convert boolean to binary values
patch_in_range_mask_mean = in_range_mask.float().mean()

#clamp values between 0 and 1
patch_in_range_mask_mean =patch_in_range_mask_mean.clamp(0.0,1.0)

#calcualtes the amount of our range and clamps it between 0 and 1
por = (1.0 - patch_in_range_mask_mean).clamp(0.0,1.0)

#converts in range mask to binary mask
in_range_mask = in_range_mask.float()

#find indices for entry
indices = entry['indices']

temp_threshold_list = []

#======= class aware penalty code
#move in range mask to cuda and convert to float
in_range = in_range_mask.to(probs.device, dtype=torch.float32)

#create mask of pixels out of range of the thresholds
out_range = 1.0 - in_range

#define the labels and pixels for the image
```

```
                    labels_b = labels[b]
                    valid_b  = (labels_b != IGNORE_LABEL)

                    #pull the probability for the image
                    probs_b  = probs[b]

                    #create a label aware mask of pixels that are not in the class class_idx, c
                    gt_not_c = (valid_b & (labels_b != class_idx)).to(probs_b.dtype)

                    #find the probability for current class class_idx, c
                    p_c = probs_b[class_idx]

                    #===== get info for adaptive scaling
                    #percentage of pixels within thresholds
                    pin = in_range.float().mean()

                    #percentage of pixels outside threshold clamped between 0 and 1
                    por = (1.0 - pin).clamp(0.0, 1.0)

                    #calculate safe denominators for the hard and soft weights to avoid division by 0
                    den_soft = (in_range * valid_b).sum().clamp_min(1.0)
                    den_hard = (out_range * gt_not_c).sum().clamp_min(1.0)

                    #calculate adaptive weights
                    adaptive_lambda = compute_adaptive_lambda(main_loss, scale = soft_scale)
                    adaptive_hard   = compute_adaptive_hard_weight(por, scale = hard_scale, margin =
margin)

                    #clamp the adaptive weights from 0 to adaptive_cap
                    adaptive_lambda = torch.clamp(adaptive_lambda, 0.0, adaptive_cap)
                    adaptive_hard   = torch.clamp(adaptive_hard,   0.0, adaptive_cap)

                    #use the adaptive weights to calcualte the bonus and penalty for the class
                    bonus_val   = adaptive_lambda * (p_c * in_range * valid_b.to(p_c.dtype)).sum() /
den_soft
                    penalty_val = adaptive_hard   * (p_c * out_range * gt_not_c).sum() / den_hard

                    #add the bonuses, penalties, and current index to a list
                    bonuses_b.append(bonus_val)
                    penalties_b.append(penalty_val)
                    valid_idxs_b.append(class_idx)

                    #find the output of the image  for a specific index
```

```
                    class_output = outputs[b, class_idx]

                    #finds the mean of the output
                    class_mean_output = class_output.mean()


                #====aggregate per image if valid classes are found
                #check to see if bonuses exist
                if bonuses_b:

                    #create tensors of the bonuses and penalties
                    bonuses_tensor = torch.stack(bonuses_b) #bonuses across the entire image
                    penalties_tensor = torch.stack(penalties_b) #penalties across the entire image
                    image_bonus = torch.stack(bonuses_b).mean() #mean of bonuses across the image
                    image_penalty = torch.stack(penalties_b).mean() ##mean of penalties across the
image

                    #count the number of times bonuses were applied
                    batch_bonus_loss += image_bonus
                    batch_hard_loss += image_penalty

                    #count the number of valid images
                    valid_image_count += 1

                    #iterates over each class and counts the bonuses and penalties
                    for cls_idx, bval, pval in zip(valid_idxs_b, bonuses_tensor, penalties_tensor):

                        #pull class name from model
                        cn = model.class_names[cls_idx]

                        #accumulate the conuses and penalites, then puts them on cpu
                        class_bonus_sums[cn]   += float(bval.detach().cpu())
                        class_penalty_sums[cn] += float(pval.detach().cpu())

                        #count the number of times each class appears
                        class_counts[cn]       += 1



                #=== average across entire batch
                if valid_image_count > 0:
```

```
                #avearge the bonus and hard losses over the valid images
                bonus_loss = batch_bonus_loss / valid_image_count
                hard_loss = batch_hard_loss / valid_image_count
            else:
                #assign 0 to bonus and hard loss
                bonus_loss = outputs.new_zeros(())
                hard_loss = outputs.new_zeros(())

        #calculate total loss with bonus and hard loss
        if adaptive_weight:
            total_loss = main_loss - bonus_loss + hard_loss

        else:

            total_loss = main_loss - lambda_weight * bonus_loss + hard_weight * hard_loss

    else:
        #calculate total loss without threshold logic
        total_loss = main_loss
```

Appendix E: Supplementary Files and Content

My supplementary files include the code that I used to train my model as both Python script and a Jupyter Notebook. Thresholds.csv contains the remote sensing thresholds for the hard and soft constraints. Imagery.zip contains all of the original training and validation images as well as the augmented images and labels. These files can be located on ScholarSphere with the title of "Multispectral Landsat Imagery for AI Training" (Simmons, 2026).

Bibliography

Bah, A. R., Norouzi, H., Prakash, S., Blake, R., & Khanbilvardi, R. (2022). Spatial Downscaling of GOES-R Land Surface Temperature over Urban Regions: A Case Study for New York City. *Atmosphere, 13*(2). doi:https://doi.org/10.3390/atmos13020332

Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural Machine Translation by Jointly Learning to Align and Translate. *arXiv*. doi:https://doi.org/10.48550/arXiv.1409.0473

Blaus, B. (2013, December 20). *Multipolar Neuron*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Neuron#/media/File:Blausen_0657_MultipolarNeuron.png

Buslaev, A., Parinov, A., Khvedchenya, E., Iglobikov, V. I., & Kalinin, A. A. (2020). Albumentations: Fast and Flexible Image Augmentations. *Information, 11*(2). doi:https://doi.org/10.3390/info11020125

Caltech Infrared Processing and Analysis Center. (n.d.). *Infrared regeions of the elctromagnetic spectrum*. Retrieved from Caltech: https://archive.ph/20120529003352/http://www.ipac.caltech.edu/Outreach/Edu/Regions/irregions.html

Campbell, J. B., & Wynne, R. H. (2011). *Introduction to Remote Sensing* (5th ed.). New York: The Guilford Press.

Chander, G., Markham, B. L., & Helder, D. L. (2009). Summary of current radiometric calibration coefficients for Landsat MSS, TM, ETM+, and EO-1 ALI sensors. *Remote Sensing Environment, 113*(5), 893-903. doi:https://doi.org/10.1016/j.rse.2009.01.007

Curran, P. (1980, September). Multispectral remote sensing of vegetation amount. *Progress in Physical Geography, 4*(3), 315-341. doi:https://doi-org.ezaccess.libraries.psu.edu/10.1177/030913338000400301

Dive Into Deep Learning. (n.d.). *Image Augmentation*. Retrieved Octtober 9, 2025, from Dive Into Deep Learning: https://d2l.ai/chapter_computer-vision/image-augmentation.html

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., . . . Houlsby, N. (2020). An Image Is Worth 16 x 16 Words: Transformers for Image Recognition at Scale. *arXiv*.

Environmental Systems Research Institute. (n.d.). *Indices Gallery--ArcGIS Pro Documentation*. Retrieved September 29, 2024, from Esri: https://pro.arcgis.com/en/pro-app/latest/help/data/imagery/indices-gallery.htm

Erikson, S. (2000). *Deadhouse Gates.* New York: Tom Doherty Associates.

Esri. (2024). ArcGIS Pro (Version 3.4.3). Environmental Systems Research Institute. Retrieved from https://www.esri.com

Farms, S. (n.d.). *Sandcreek Farms*. Retrieved September 29, 2024, from What Is The LARGEST Farm In The US? Top 9: https://www.agriculture.com/farm-management/farm-land/top-5-farms-with-the-largest-acreage-in-the-us

fdeloche. (2017, June 19). *Recurrent neural network*. Retrieved June 6, 2025, from Wikipedia: https://en.wikipedia.org/wiki/Recurrent_neural_network#/media/File:Recurrent_neural_network_unfold.svg

Feyisa, G. L., Meilby, H., Fensholt, R., & Proud, S. R. (2014, January). Automated Water Extraction Index: A new technique for surface water mapping using Landsat imagery. *Remote Sensing of Environment, 140*, 23-35. doi:https://doi.org/10.1016/j.rse.2013.08.029

Feynman, R. P. (2011). *Six Easy Pieces.* New York: Basic Books.

Géron, A. (2023). *Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow.* Sebastopol: O'Reilly Media.

Google. (2023, September). Retrieved October 5, 2025, from https://maps.app.goo.gl/6MRAAeogzPYw526c8

Google Developers. (2026, January 12). *Classification: Accuracy, recall, precision, and related metrics*. Retrieved June 5, 2026, from Google for Developers: https://developers.google.com/machine-learning/crash-course/classification/accuracy-precision-recall

Google Developers. (2026, January 1). *Thresholds and the confusion matrix*. Retrieved from Google for Developers: https://developers.google.com/machine-learning/crash-course/classification/thresholding

Greene, J. (2014, September 18). *Infrared photography*. Retrieved May 28, 2025, from Wikipedia: https://upload.wikimedia.org/wikipedia/commons/thumb/9/90/Hollywood_Hills_California.jpg/1280px-Hollywood_Hills_California.jpg

Gribbin, J. (1998). *Q is For Quantum.* New York: The Free Press.

Hagan, M. T., Demuth, H. B., Beale, M. H., & De Jesús, O. (2014). *Neural Network Design.* Martin Hagan. Retrieved from https://hagan.okstate.edu/NNDesign.pdf

Hardesty, L. (2017, April 14). *Explained: Neural Networks Ballyhooed artificial-intelligence technique known as "Deep learning" revives 70-year-old idea*. Retrieved from MIT News: https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414

Harsh. (n.d.). *Understanding Aritifical Neural network (ANN)*. Retrieved June 4, 2025, from CodeSpeedy: https://www.codespeedy.com/understanding-artificial-neural-network-ann/

Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation, 9*(8), 1735-1780.

Inductiveload, & NASA. (2025, May 7). *Electromagnetic Spectrum*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Electromagnetic_spectrum

Introduction to Artifical Neural Networks and the Perceptron. (n.d.). Retrieved June 5, 2025, from Quantstart: https://www.quantstart.com/articles/introduction-to-artificial-neural-networks-and-the-perceptron/

Introduction to Recurrent Neural Networks. (2025, May 23). Retrieved June 7, 2025, from Geeks for Geeks: https://www.geeksforgeeks.org/introduction-to-recurrent-neural-network/

Jahangeer, N., Ganesan, S., & Begum, A. A. (2022). A Study on Different Deep Learning Algorithms Used in Deep Neural Nets: MLP SOM and DBN. *Wireless Personal Communications*. doi:http://dx.doi.org/10.1007/s11277-022-10079-4

Janga, B., Asamani, G. P., Sun, Z., & Cristea, N. (2023). A Review of Practical AI for Remote Sensing in Earth Sciences. *Remote Sensing*. doi:https://doi.org/10.3390/rs15164112

Jensen, J. R. (2014). *Remote Sensing of the Environment: An Earth Resource Perspective* (Second Edition ed.). Essex: Pearson Education Limited.

Kalaitzis, F., Bayaraa, M., & Rossi, C. (n.d.). State of AI for Earth Observation. *Satellite Applications Catapult*. Retrieved from https://sa.catapult.org.uk/wp-content/uploads/2022/09/State-of-AI-for-Earth-Observation.pdf

Karpathy, A. (2015, May 21). *The Unreasonable Effectiveness of Recurrent Neural Networks*. Retrieved June 8, 2025, from GitHub: https://karpathy.github.io/2015/05/21/rnn-effectiveness/

Landsat Missions. (n.d.). *Landsat I*. Retrieved May 31, 2025, from USGS: https://www.usgs.gov/landsat-missions/landsat-1

Larson, R., Hostetler, R. P., & Edwards, B. H. (2003). *Calculus: Early Transcendental Functions.* Boston: Houghton Mifflin Company.

List of North American Deserts. (2025, September 30). Retrieved September 29, 2024, from Wikipedia: https://en.wikipedia.org/wiki/List_of_North_American_deserts

Lo, A. (2021, Dec 18). *Weight Decay and Its Peculiar Effects*. Retrieved June 10, 2025, from Towards Data Science: https://towardsdatascience.com/weight-decay-and-its-peculiar-effects-66e0aee3e7b8/

Madran, P., Singh, V., Singh, D. P., Diwakar, M., Pant, B., & Kishor, A. (2022). A Hybrid Deep Learning Approach for ECG-Based Arrhythmia Classification. *Bioengineering, 9*(152). doi:http://dx.doi.org/10.3390/bioengineering9040152

Maggiori, E., Tarabalka, Y., Charpiat, G., & Alliez, P. (2017). Convolutional Neural Networks for Large-Scale Remote Sensing Image Classification. *IEEE Transactions on Geoscience and Remote Sensing, 55*(2), 645-657. doi:https://doi.org/10.1109/TGRS.2016.2612821

McCulloch, W. S., & Pitts, W. (1943). A Logical Calculus of the Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biophysics*, 115-133.

Meigs, A. D., Otten, J., & Cherezova, T. Y. (1998). Ultraspectral Imging: A New Contribution To Global Virtual Presence. *IEEE Aerospace Conference Proceedings, 2*, 5-12.

Minsky, M. L., & Papert, S. A. (1988). *Perceptrons: An Introduction to Computational Geometry*. MIT Press.

Mishra, D. (2020, May 9). *Weight Decay == L2 Regularization*. Retrieved June 10, 2025, from Towards Data Science: https://towardsdatascience.com/weight-decay-l2-regularization-90a9e17713cd/

Mzid, N., Pignatti, S., Huang, W., & Casa, R. (2021, January). An Analysis of Bare Soil Occurence in Arable Croplands for Remote Sensing Topsoil Applications. *Remoe Sensing*. doi:https://doi.org/10.3390/rs13030474

National Grassland. (2025, August 18). Retrieved September 29, 2024, from Wikipedia: https://en.wikipedia.org/wiki/National_grassland

Nguyen, C. T., Chidthaisong, A., Kieu Diem, P., & Huo, L.-Z. (2021). A Modified Bare Soil Index to Identify Bare Land Features during Agricultural Fallow-Period in Southeast Asia Using Landsat 8. *Land, 10*(3). doi:https://doi.org/10.3390/land10030231

Nhu, C.-N., & Park, M. (2022). Dynamic Network SLice Scaling Assisted by Attention-Based Prediction in 5G Core Network. *IEEE Access*, 72955*72972. doi:http://dx.doi.org/10.1109/ACCESS.2022.3190640

Office of Theses and Disserations. (n.d.). *Submission Procedure*. Retrieved May 16, 2025, from Pennsylvania State University: https://gradschool.psu.edu/assets/uploads/documents/Thesis-and-Dissertation-Handbook.pdf

O'Keefe, E. (2019, September 28). *Top 5 farms with the largest acreage in the U.S.* Retrieved October 5, 2024, from Successful Farming: https://www.agriculture.com/farm-management/farm-land/top-5-farms-with-the-largest-acreage-in-the-us

Oni, D. (2023, October 21). *The implications of increasing Earth observation data*. Retrieved April 27, 2024, from NSR An Analysys Mason Company: https://www.nsr.com/the-implications-of-increasing-earth-observation-data/

Panahi, H., Azizi, Z., Kiadaliri, H., Ali Almodaresi, S., & Aghamohamadi, H. (2024). Bare Soil detecting algorithms in western iran woodlands usiong remote sensing. *Smart Agricultural Technology*, 100429. doi:https://doi.org/10.1016/j.atech.2024.100429

PyTorch. (n.d.). *CrossEntropyLoss*. Retrieved October 23, 2025, from PyTorch: https://docs.pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html

Rana, M., & Kharel, S. (2019). Feature Extraction For Urban And Agricultural Domains Using Ecognition Developer. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 609-615. doi:https://doi.org/10.5194/isprs-archives-XLII-3-W6-609-2019

Shahfahad, Mourya, M., Kumari, B., Tayyab, M., Paarcha, A., Asif, & Rahman, A. (2021). Indices based assessment of built-up density and urban expansion of fast growing Surat city using multi-temporal Landsat data sets. *GeoJournal*, 1607-1623. doi:https://doi.org/10.1007/s10708-020-10148-w

Sharma, P. (n.d.). *Convolution Filters / FIlters in CNN*. Retrieved June 6, 2025, from OpenGenus: https://iq.opengenus.org/convolution-filters/

Simmons, S. O. (2026, July 14). *Multispectral Landsat Imagery for AI Training [Dataset]*. doi:https://doi.org/10.26207/nckg-h496

Sun, Z., Wang, C., Guo, H., & Shang, R. (2017). A Modified Normalized Difference Impervious Surface Index (MNDISI) for AUtomatic Urban Mapping from Landsat Imagery. *Remote Sensing, 9*(9). doi:https://doi.org/10.3390/rs9090942

Taha, A. A., & Hanbury, A. (2015). Metrics for evaluting 3D medical image segmentation: analysis, selection, and tool. *BMC Medical Imaging, 15*(29). doi:10.1186/s12880-015-0068-x

The Largest National Grasslands In The United States. (n.d.). Retrieved September 29, 2024, from World Atlas: https://www.worldatlas.com/articles/the-largest-national-grasslands-in-the-united-states.html

Touvron, H., Cord, M., Douze, M., Massa, F., Sablayrolles, A., & Jégou, H. (2021). Training data-efficient image transformers & distillation through attention. *arXiv*.

Trokas. (2023). *DNN (Deep Neural Networks)*. Retrieved June 5, 2025, from GitHub: https://trokas.github.io/ai_primer/DNN.html

Trott, S. (2024, January 17). *Perceptrons, XOR, and the first "AI winter"*. Retrieved June 6, 2025, from Substack: https://seantrott.substack.com/p/perceptrons-xor-and-the-first-ai

U.S Geological Survey. (2023, 07 28). Landsat 8 OLI/TIRS C1 Level-1 data (path 041, row 032). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2014, 05 16). Landsat 8 OLI/TIRS C1 Level-1 data (path 025, row 039). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2016, 05 07). Landsat 8 OLI/TIRS C1 Level-1 data (path 023, row 039). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2016, 06 07). Landsat 8 OLI/TIRS C1 Level-1 data (path 032, row 030). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2018, 05 26). Landsat 8 OLI/TIRS C1 Level-1 data (path 034, row 030). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2020, 07 19). Landsat 8 OLI/TIRS C1 Level-1 data (path 041, row 029). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2022, 06 22). Landsat 8 OLI/TIRS C1 Level-1 data (path 034, row 029). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2022, 07 07). Landsat 8 OLI/TIRS C1 Level-1 data (path 043, row 034). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2022, 07 08). Landsat 9 OLI/TIRS C1 Level-1 data (path 042, row 035). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2023, 01 11). Landsat 8 OLI/TIRS C1 Level-1 data (path 015, row 042). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2023, 06 18). Landsat 8 OLI/TIRS C1 Level-1 data (path 017, row 034). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2023, 06 21). Landsat 8 OLI/TIRS C1 Level-1 data (path 022, row 031). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2023, 18 15). Landsat 8 OLI/TIRS C1 Level-1 data (path 023, row 027). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2023, 08 28). Landsat 8 OLI/TIRS C1 Level-1 data (path 029, row 035). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2023, 06 13). Landsat 8 OLI/TIRS C1 Level-1 data (path 030, row 027). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2023, 06 30). Landsat 8 OLI/TIRS C1 Level-1 data (path 037, row 037),. Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2023, 07 12). Landsat 8 OLI/TIRS C1 Level-1 data (path 041, row 036). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2023, 07 23). Landsat 9 OLI/TIRS C1 Level-1 data (path 014, row 032). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2023, 06 01). Landsat 9 OLI/TIRS C1 Level-1 data (path 018, row 031). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2023, 06 21). Landsat 9 OLI/TIRS C1 Level-1 data (path 022, row 030). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2023, 08 16). Landsat 9 OLI/TIRS C1 Level-1 data (path 038, row 030). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2023, 09 08). Landsat 9 OLI/TIRS C1 Level-1 data (path 039, row 031). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2023, 07 13). Landsat 9 OLI/TIRS C1 Level-1 data (path 040, row 036). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2023, 07 27). Landsat 9 OLI/TIRS C1 Level-1 data (path 042, row 030). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2023, 06 30). Landsat 9 OLI/TIRS C1 Level-1 data (path 045, row 026). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2023, 07 21). Landsat 9 OLI/TIRS C1 Level-1 data(path 032, row 038). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2023, 07 08). Landsat 9 OLI/TIRS C1-Level-1 data (path 037, row 037). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2024, 06 12). Landsat 8 OLI/TIRS C1 Level-1 data (path 025, row 033). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2024, 06 06). Landsat 8 OLI/TIRS C1 Level-1 data (path 026, row 036). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2024, 07 12). Landsat 8 OLI/TIRS C1 Level-1 data (path 027, row 034). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2024, 08 17). Landsat 8 OLI/TIRS C1 Level-1 data (path 039, row 029). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2024, 06 06). Landsat 9 OLI/TIRS C1 Level-1 data (path 031, row 029). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (2024, 08 25). Landsat 9 OLI/TIRS C1 Level-1 data (path039, row037). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (n.d.). Landsat 9 OLI/TIRS C1 Level-1 data (path 016, row 042). Retrieved from https://earthexplorer.usgs.gov

U.S. Geological Survey. (n.d.). *Landsat Soil Adjusted Vegetation Index*. Retrieved October 3, 2024, from U.S. Geological Survey: https://www.usgs.gov/landsat-missions/landsat-soil-adjusted-vegetation-index

U.S. Geological Survey. (n.d.). *Normalized Difference Snow Index*. Retrieved October 3, 2024, from U. S. Geological Survey: https://www.usgs.gov/landsat-missions/normalized-difference-snow-index

USGS. (2025, August 1). *What are the band designations for the Landsat satellites?* Retrieved September 24, 2024, from USGS: https://www.usgs.gov/faqs/what-are-band-designations-landsat-satellites

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., . . . Polosukhin, I. (2017). Attention Is All You Need. *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 6000-6010.

Whiteclouds. (n.d.). Retrieved September 29, 2024, from Top 10 Largest Farms in the United States: https://www.agriculture.com/farm-management/farm-land/top-5-farms-with-the-largest-acreage-in-the-us

Wikipedia. (2025, March 12). *List of snowiest places in the United States by state*. Retrieved September 29, 2024, from Wikipedia: https://en.wikipedia.org/wiki/List_of_snowiest_places_in_the_United_States_by_state

Wikipedia. (2025, September 12). *List of United States citites by area*. Retrieved September 29, 2024, from Wikipedia: https://en.wikipedia.org/wiki/List_of_United_States_cities_by_area

Wikipedia Contributors. (n.d.). *Generalization Error*. Retrieved June 10, 2025, from Wikipedia: https://en.wikipedia.org/wiki/Generalization_error

Wikipedia contributors. (n.d.). *Neuron*. Retrieved June 5, 2025, from Wikipedia: https://en.wikipedia.org/wiki/Neuron

Xu, K., Ba, J. L., Kiros, R., Cho, K., Courville, A., Salakhutdinov, R., . . . Bengio, Y. (2015). Show, Attend, and Tell: Neural Image Caption Genereation with Visual Attention. *Proceedings of the 32nd International Conference on Machine Learning*, 2048-2057. doi:https://doi.org/10.48550/arXiv.1502.03044

you. (n.d.). Retrieved June 8, 2025, from Dictionary.com: https://www.dictionary.com/browse/you

Zhai, X., Kolesnikov, A., Houlsby, N., & Beyer, L. (2021). Scaling Vision Transformers. *arXiv*.

Zhou, B. (n.d.). 5.3 Attention Mechanism. University Park, Philadelphia, USA. Retrieved June 8, 2025, from https://psu.instructure.com/courses/2391983/pages/5-dot-3-attention-mechanism?module_item_id=44369150