Temp Final Report

Nash Rickert

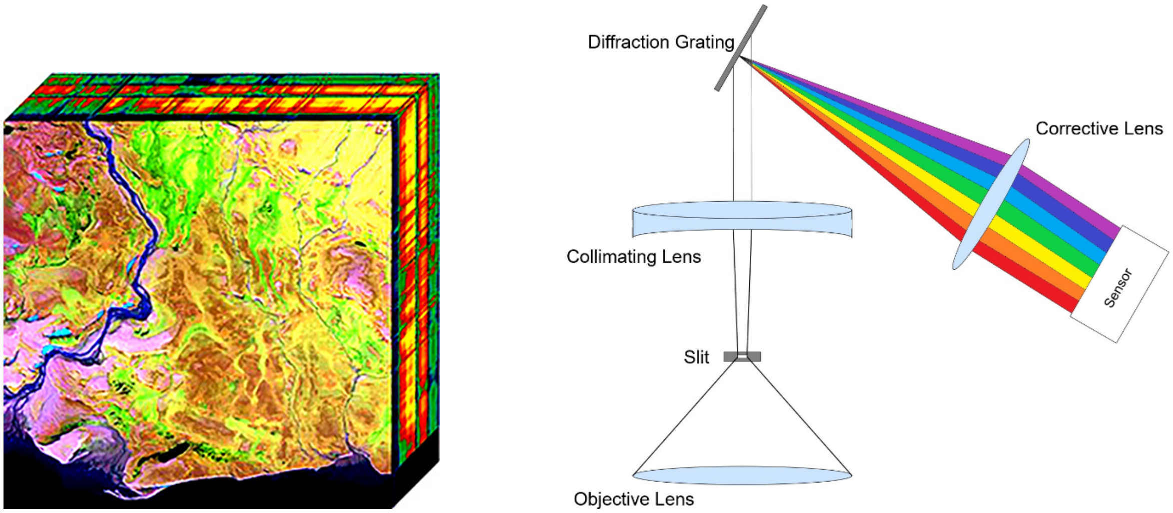
July 21, 2025

1 Abstract

2 Project Overview

2.1 Hyperspectral Imaging

While a familiar RGB image samples only the three spectral bands associated with the colors red, green, and blue in the visible spectra to form an image, a hyperspectral imager uses diffraction to sample a continuous range of spectral bands, including those falling outside of the visible spectrum. This is useful for many tasks such as environmental monitoring because hyperspectral images can account for characteristics of the data that would not exist in an RGB image. For example, the chlorophyll in vegetation absorbs most visible light, but its reflectance changes rapidly near the infrared range of the spectrum. This phenomenon is called the 'red edge', and makes identifying vegetation relatively straightforward using hyperspectral imaging. For our project, we are seeking to leverage these advantages to identify forest regions with an adbundance of ground fuel, which indicates an elevated risk of forest fires.



(a) A hyperspectral data cube [1]

(b) The optics of a hyperspectral imager [2]

Figure 1: The depth of a hyperspectral data cube represents the many color channels that make up each pixel

As a result of the continuous sampling of the spectrum, hyperspectral images contain much more data than a traditional image. The sensor we are using collects data on 1608 color channels for each pixel compared to the 3 channels of an RGB image. While binning can reduce the number of channels we utilize and size of our data as a result, the sheer quantity of hyperspectral data is a major obstacle to its effective analysis.

2.2 Real Time Processing

Traditionally, hyperspectral data is captured in the field and returned to a lab for processing. This is process can be length and tedious due to the inherently large size of hyperspectral data – the captured data may be many gigabytes or even terabytes in size, meaning the processing stage can take days or weeks. For time-sensitive environmental tasks, it would be preferable to process data as the sensor receives it. This would allow field workers to utilize data for decisionmaking as they receive it and would eliminate the need to store data persistently. For example, when classifying which areas of a forest need to be cleared of ground fuel, workers could fly a drone equipped with a sensor and real-time processor; upon completion, they could immediately begin clearing debris.

ML classification tasks are generally done with GPUs due to their parallelism and efficiency at completing arithmetic operations. For this reason, a strong infrastructure exists for training and running models on a GPU, making their design and implementation relatively easy. Unfortunately, GPUs have one main drawback that makes them unsuitable for our purposes. In the process of completing inference on data received from a sensor, it is necessary for the CPU to write data from the sensor into DRAM, load the data from DRAM into the GPU, perform the inference operation, and then finally write results back to DRAM. The overhead of writing and loading the data and writing the results back to DRAM is inordinate when our goal is to complete inference at the same rate that data is collected. Furthermore, cache misses and refresh rates make the processing speed of a GPU nondeterministic and unpredicatble. Thus we choose to use a field-programmable gate array (FPGA) for our project. An FPGA has the advantage that data can be streamed in directly from the sensor and pipelined deterministically through the fabric of our circuit. Therefore the main challenge of this project is to effectively leverage the advantages of custom hardware to achieve rapid inference. While potentially advantageous, this entails many more difficulties and pitfalls than one would typically encounter when using a GPU.

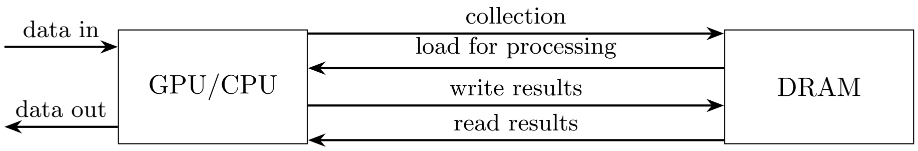


Figure 2: Latency overhead of using a GPU for computation

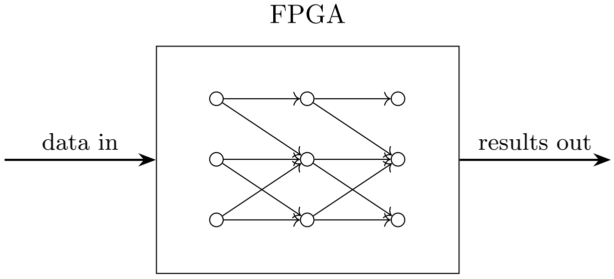


Figure 3: Lack of latency overhead of an FPGA inference stream

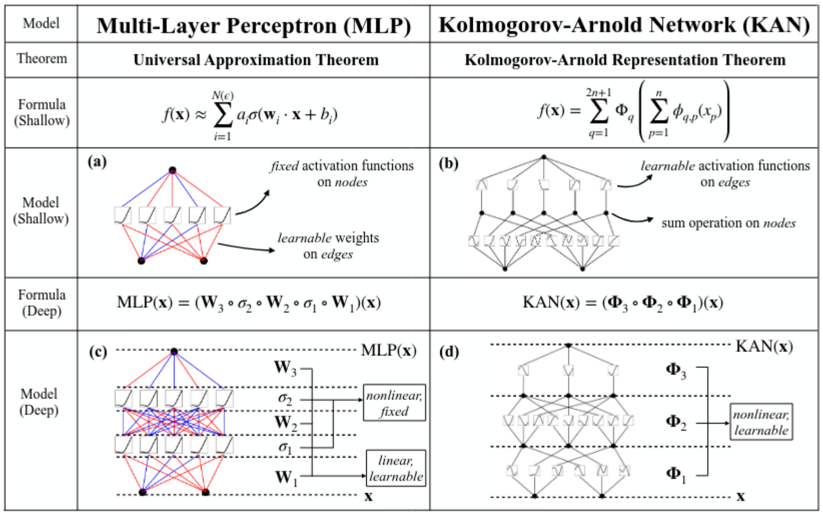
2.3 Model Architectures and Considerations

Achieving real time hyperspectral image processing requires not just efficient hardware design, but also consideration of the model architectures that will work most effectively on an FPGA. Because of our control over hardware , we might be able to achieve latency improvements at the marginal expense of model accuracy. Furthermore, we can't rely on conventional wisdom regarding runtime on a GPU and we aren't concerned about the training time of our model. We can accept an excessively long model training time if it results in inference runtime improvement since the true bottleneck to the success of the project is in inference latency. In comparison, training is largely irrelevant and can be done with many more resources without issue.

It is also necessary to consider the memory usage of our model. A convolutional neural network (CNN) that needs to store a large amount of data in DRAM for its kernel passthrough is inefficient because of the latency involved in storing and accessing that memory. The memory could be stored in local static FPGA memory, but only so much of that memory is available and it is also necessary to use it to store model weights. We could attempt to pipeline data from DRAM in such a way that there is no latency overhead involved in accessing it, but this adds additional complexity to our hardware design and implementation. Thus we would like to design and use a model that is fast, accurate, small, and leverages the computational advantages of a programmable circuit.

2.4 Kolmogorov-Arnold Networks

A Kolmorgorov-Arnold Network (KAN) is a type of neural network that directly learns nonlinear activation functions between each layer of the network rather than learning weights for linear transformations and applying a fixed nonlinear activation function between each layer [3].



Model	Multi-Layer Perceptron (MLP)	Kolmogorov-Arnold Network (KAN)
Theorem	Universal Approximation Theorem	Kolmogorov-Arnold Representation Theorem
Formula (Shallow)	$f(\mathbf{x}) \approx \sum_{i=1}^{N(\epsilon)} a_i \sigma(\mathbf{w}_i \cdot \mathbf{x} + b_i)$	$f(\mathbf{x}) = \sum_{q=1}^{2n+1} \Phi_q \left(\sum_{p=1}^{n} \phi_{q,p}(x_p) \right)$
Model (Shallow)	(a)	(b)
Formula (Deep)	$\mathrm{MLP}(\mathbf{x}) = (\mathbf{W}_3 \circ \sigma_2 \circ \mathbf{W}_2 \circ \sigma_1 \circ \mathbf{W}_1)(\mathbf{x})$	$\mathrm{KAN}(\mathbf{x}) = (\mathbf{\Phi}_3 \circ \mathbf{\Phi}_2 \circ \mathbf{\Phi}_1)(\mathbf{x})$
Model (Deep)	(c)	(d)

Figure 0.1: Multi-Layer Perceptrons (MLPs) vs. Kolmogorov-Arnold Networks (KANs)

Figure 4: A visualization of the differences between a KAN and traditional MLP [3]

KANs generally perform similarly to multilayer-perceptron networks (MLPs) on most tasks; whether they will find widespread use is still largely unknown given their recent development in the field of machine learning. However, for us they have an extremely appealing property: We can achieve performance improvements by encoding their nonlinear functions as lookup tables within the fabric of our FPGA. What this means is that for a given nonlinear function over some fixed domain (note that this domain is determined based on the inputs the function sees during training – thus it shifts to properly accomodate incoming data), we can choose some granularity for a mesh, calculate outputs across that mesh, and then store those results in a lookup table. Then for some input, we can interpolate between the entries of our lookup table to determine an approximate result. Because the KAN is composed entirely of nonlinear activation functions, this extremely fast lookup process encompasses the entirety of model inference, potentially granting significant speed improvements within the fabric of our FPGA.

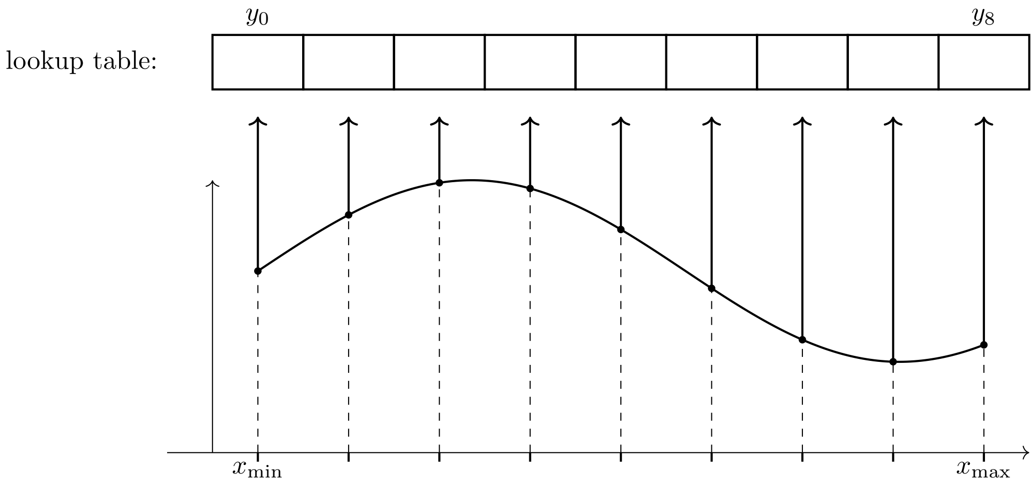


Figure 5: An example 8 entry lookup table over a nonlinear function

2.4.1 Lookup Tables in an FPGA

In general, the lookup tables for a KAN require a significant amount of memory. For a fully connected KAN with layers consisting of 5 layers and 200, 32, 32, 32, and 16 nodes within each layer respectively, there will be $(200*32)+(32*32)+(32*32)+(32*16) = 8960$ lookup tables. If each has a relatively fine mesh of 4096 elements and each element is 4 bytes, then simply storing the elements of each lookup tables would take $8960 * 4096 * 4$ bytes ≈ 147 Mb. In reality the lookup tables for each layer need not be loaded at the same time, the fixed point types we use will almost certainly be less than 4 bytes, and we can experiment with table granulatities of less than 4096 and see how it affects accuracy, but the fact remains that the memory overhead of lookup tables is significant. Thus it is necessary for out implementation to write a linux device driver that can efficiently load lookup table values from device memory into static fabric memory so that they can be used by our FPGA.

3 Project Work

3.1 CNN Implementation and Benchmarking

At the start of the work, it was important to do performance benchmarking on existing accurate hyperspectral vision models in order to calibrate our expectations of the kind of workload we would encounter when doing real-time inference. We used a previous paper and model created by researchers at Montana State University as our CNN benchmark [4] [5]. Their code uses binning and many convolutional layers to classify pixels of the well known Indian Pines Dataset.

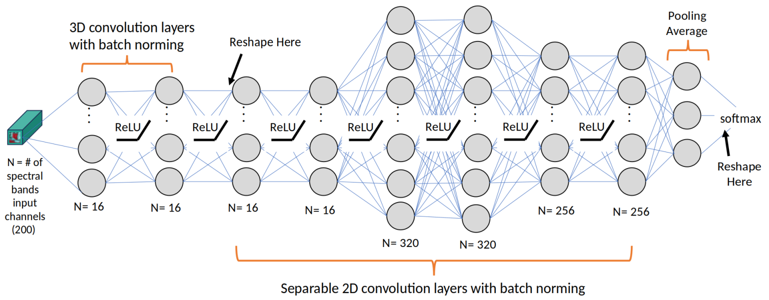

3D convolution layers with batch norming
Reshape Here
Pooling Average
softmax
Reshape Here
N = # of spectral bands input channels (200)
ReLU
ReLU
ReLU
ReLU
ReLU
ReLU
ReLU
ReLU
N= 16
N= 16
N= 16
N= 16
N= 320
N= 320
N= 256
N= 256
Separable 2D convolution layers with batch norming

Figure 6: Visualization of the layers of the CNN model [6]

I recreated their code faithfully in C. The point of this was to get a sense of the number of operations involved in a large convolutional network that might be implemented on our hardware as well as to allow benchmarking on the CPU/GPU of our board, ignoring the fabric. It should be noted that this program naturally ran very slowly compared to the python code due to the lack of optimizations that pytorch implements such as GPU acceleration, parallelism, memory optimization, operator fusion, and more. The CNN model did about 2 billion elementary add and multiply operations for each pixel classified of the dataset.

```
self.conv_layer1 = nn.Sequential(
    nn.Conv3d(in_channels=img_shape[0], out_channels=16, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.BatchNorm3d(16))

self.conv_layer2 = nn.Sequential(
    nn.Conv3d(in_channels=16, out_channels=16, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.BatchNorm3d(16))

self.sepconv1 = nn.Sequential(
    nn.Conv2d(in_channels=16 * img_shape[1], out_channels=16 * img_shape[1], kernel_size=5, padding=2, groups=16 * img_shape[1]),
    nn.ReLU(),
    nn.Conv2d(in_channels=16 * img_shape[1], out_channels=320, kernel_size=1, padding=0),
    nn.ReLU(),
    nn.BatchNorm2d(320))

self.sepconv2 = nn.Sequential(
    nn.Conv2d(in_channels=320, out_channels=320, kernel_size=3, padding=1, stride=stride, groups=320),
    nn.ReLU(),
    nn.Conv2d(in_channels=320, out_channels=256, kernel_size=1, padding=0),
    nn.ReLU(),
    nn.BatchNorm2d(256))

self.sepconv3 = nn.Sequential(
    nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, padding=1, stride=stride, groups=256),
    nn.ReLU(),
    nn.Conv2d(in_channels=256, out_channels=256, kernel_size=1, padding=0),
    nn.ReLU(),
    nn.BatchNorm2d(256))
```

```

self.average = nn.AvgPool2d(kernel_size=out)

if classes == 2:
    self.fc1 = nn.Linear(256, 1)
else:
    self.fc1 = nn.Linear(256, self.classes)
```

```

def forward(self, x):
    # 3D Feature extractor
    x = self.conv_layer1(x)
    x = self.conv_layer2(x)

    # Reshape 3D-2D
    x = reshape(x, (x.shape[0], self.img_shape[1] * 16, self.img_shape[2],
      ↪ self.img_shape[3]))

    # 2D Spatial encoder
    x = self.sepconv1(x)
    x = self.sepconv2(x)
    x = self.sepconv3(x)

    # Global Average Pooling
    x = self.average(x)

    x = reshape(x, (x.shape[0], x.shape[1]))
    x = self.fc1(x)
    return x
```

3.2 KAN Implementation and Benchmarking

3.2.1 Implementation

I likewise implemented a KAN model that was created by former student Dirk Kaiser that did pixel-wise classification on the Indian Pines dataset. The layers of the model consisted of 200, 32, 32, 32, 16 nodes respectively. I implemented the model in C as closely as possible to the fabric implementation; this means that I used lookup tables to perform computations and adder trees to do addition. I used CFFI (C Foreign Function Interface) to create a shared C library that can be called from python to more easily facilitate populating the lookup tables and experimenting with different model architectures. What this means is that a model can be entirely trained in python, and then by evaluating each activation function at an appropriate grid of points in python, we can populate the lookup tables of the C version by calling the appropriate function. Additionally, because the final FPGA implementation will use fixed data types, I implemented scaling so that the lookup tables were populated with scaled values in the range $[0, 1]$ and rescaled outwards. This will save space and complexity in the final implementation.

3.2.2 Lookup Table Implementation

Each lookup table consists of both the actual stored values, as well as meta-data which allows us to calculate the appropriate output value for some input as well as to do the rescaling of our output (if scaling is enabled).

```
struct lkup_tbl {
    float tbl[TBL_SIZE];
    float xmin;                // x val associated with table[0]
```

```c
    float xmax;                 // x val associated with table[TBL_SIZE - 1]
    float xdist;                // dist between x values.
                                // Roughly (xmax - xmin) / TBL_SIZE
    float inv_xdist;            // the reciprocal of xdist for division
    float ymin;
    float ymax;
};
```

Thus to do a lookup on our table, we use **xmin**, **xmax**, **xdist**, and **inv_xdist** to do linear interpolation between the lookup table entries nearest to our input. Note that **inv_xdist** exists only because the division operation is expensive in fabric, so it is better to have it precomputed.

```c
static float lookup(float x, struct lkup_tbl *table) {
    if (x <= table->xmin) {
        return table->tbl[0];
    }
    else if (x >= table->xmax) {
        return table->tbl[TBL_SIZE - 1];
    }
    float idxf = (x - table->xmin) * table->inv_xdist;
    // remainder is now the proportion that x is between idx and idx + 1
    float remainder = fmodf(idxf, 1);
    assert(remainder >= 0);

    int idx = (int) idxf;
    return table->tbl[idx] + ((table->tbl[idx + 1] - table->tbl[idx]) * remainder);
}
```

Each node of our model has both a lookup table and an adder tree associated with it. When we propogate from one layer to another (where a layer is a collection of nodes), we first accumulate the existing values of our adder tree. Then for each lookup table corresponding to a node in the next layer, we find the output based on our accumulated value and place it in the adder tree for that node. This process is fully illustrated in the following two algorithms, which shows a complete neural network forward pass implemented with lookup tables.

Algorithm 1 Forward Algorithm

```
function FORWARD(model, input)
    INITVALS(model.layers[0], input)
    for $i = 1$ to model.len $- 1$ do
        PROPAGATE(model.layers[$i$])
    end for
    ret ← []
    final_layer ← model.layers[model.len $- 1$]
    for all node $\in$ final_layer do
        ret.APPEND(node.val)
    end for
    return ret
end function
```

Algorithm 2 Propagation Algorithm

```
Require: layer is not the final layer of the model
    This function operates without scaling
function PROPAGATE(layer)
    for all node $\in$ layer do
        node.val $\leftarrow$ ACCUMULATE(node.adder_tree)
        for $j = 0$ to node.next_layer.len $- 1$ do
            lookup_table $\leftarrow$ node.tables[$j$]
            target_node $\leftarrow$ node.next_layer.nodes[$j$]
            target_tree $\leftarrow$ target_node.adder_tree
            output $\leftarrow$ LOOKUP(node.val, lookup_table)
            target_tree.inputs[target_tree.ptr] $\leftarrow$ output
            target_tree.ptr $\leftarrow$ target_tree.ptr $+ 1$
        end for
    end for
end function
```

3.2.3 Accuracy Loss and Performance

For a single sample, the forward function with a table size of 4096 takes 0.00085-0.00087 seconds to run in C with my CPU: 12th Gen Intel(R) Core(TM) i7-1260P (16) @ 4.70 GHz. The final results generally vary by 0.1 - 0.01 from the actual python results which generally should not be enough to change the pixel classification, proving the viability of a KAN lookup table implementation.

3.3 Device Driver Development

3.3.1 Motivation

Given the large amount of data necessary to transfer to populate KAN lookup tables in the fabric, it was necessary to create a linux device driver that could transfer memory from system RAM into the static RAM of the FPGA to make it accessible during computation. In particular, my goal was to create a block driver whose block size was the same size as that of static RAM. This would allow an easy interface between the userspace of our system and the RAM of our fabric. Because of the large memory demand of lookup tables as discussed before, it would be necessary to load and unload multiple different sets of lookup tabels in the course of the classification of a single pixel. Thus an interface between user and fpga memory would be necessary in any circumstance to load important memory into the fabric, such as CNN data or model weights, but it is especially important in for our lookup table implementation, which uses large amounts of memory which needs to be loaded and unloaded frequently into fabric memory.

3.3.2 Board and Development Environment Setup

3.3.3 Driver Work

4 Future Work

4.1 KAN Accuracy Testing

Currently the KAN lookup table accomodates easily toggling lookup table value scaling and changing the lookup table sizes at compilation time of the shared library. However it does not currently accomodate changing data type precision or using fixed data types, as would occur in fabric. It would be useful to implement this in some fashion. Additionally, it will be necessary and useful to do parameter sweeps over lookup table size and data type precision to see what the accuracy tradeoffs are for various model sizes. Shrinking these two parameters would mean that we can shrink the amount of memory that our model will need to load from RAM and use, making our final implementation easier to create and more efficient

4.2 Device Driver Overhead

I did not have time this summer to do testing on the time overhead of the device driver or the feasibility of loading and unloading lookup tables many times in the course of the classification of a single pixel. If such loading or unloading has a high latency associated with it and needs to be done frequently enough that it makes efficient pipelining of our data difficult, then it may be best to propogate our data in batches instead of as single pixels, caching the intermediate values. This would allow us to perform computations on multiple pieces of data with a single set of lookup tables before loading the next set to continue the computation. This would make pipelining our data more difficult, but it might be necessary if the amount of loading and unloading of lookup tables is excessive in a single forward pass. This should be considered in the future, especially once we have a better sense of the efficiency of our driver, size of our model and stored data, and difficulty of pipelining the device driver loading ’just in time’.

4.3 Benchingmarking on the FPGA

5 Conclusion

6 Acknowledgements

I would like to acknowledge my PI Ross Snider for the direction and assistance he has provided on the project. I would also like to thank Nat Sweeney, Zackery Backman, and Dirk Kaiser for the work that they have done and continue to do on the project. Finally I would like to thank my fellow REU members for the support and encouragement they have provided throughout the summer.

This material is based upon work supported by the National Science Foundation under Grant No. 2349091.

7 References

References

[1] W. C. contributors, “Hyperspectralcube,” https://commons.wikimedia.org/wiki/File:HyperspectralCube.jpg, 2007.

[2] N. Sweeney, “Development of a smart unmanned system hyperspectral imager (sushi) for ground fuel characterization.”

[3] Z. Liu, Y. Wang, S. Vaidya, F. Ruehle, J. Halverson, M. Soljačić, T. Y. Hou, and M. Tegmark, “Kan: Kolmogorov-arnold networks,” 2025. [Online]. Available: https://arxiv.org/abs/2404.19756

[4] G. Morales, J. Sheppard, R. Logan, and J. Shaw, “Hyperspectral band selection for multispectral image classification with convolutional networks,” in *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE, Jul. 2021, p. 1–8. [Online]. Available: http://dx.doi.org/10.1109/IJCNN52387.2021.9533700

[5] G. Morales, J. W. Sheppard, R. D. Logan, and J. A. Shaw, “Hyperspectral dimensionality reduction based on inter-band redundancy analysis and greedy spectral selection,” *Remote Sensing*, vol. 13, no. 18, 2021. [Online]. Available: https://www.mdpi.com/2072-4292/13/18/3649

[6] D. Kaiser, “Eele490 undergraduate research,” 2025, available at: https://github.com/rksnider/SmartHyperspectralCamera/blob/main/Distillation/Paper.

[7] Y. Wang, X. Yu, Y. Gao, J. Sha, J. Wang, L. Gao, Y. Zhang, and X. Rong, “Spectralkan: Kolmogorov-arnold network for hyperspectral images change detection,” 2024. [Online]. Available: https://arxiv.org/abs/2407.00949

[8] V. Lobanov, N. Firsov, E. Myasnikov, R. Khabibullin, and A. Nikonorov, “Hyperkan: Kolmogorov-arnold networks make hyperspectral image classificators smarter,” 2024. [Online]. Available: https://arxiv.org/abs/2407.05278