Terrain Image Compression Using Neural Nets
Group C

Imagine a situation where humanity has sent a planetary lander/rover to our nearest neighboring star, and we need to deliver topographical data to help the lander choose a landing site. Sending the data costs a fortune per bit, but we have all the computing power of humanity on Earth to help compress the images. Compression algorithms are commonplace, but they are often too generic to be applicable in this situation. Neural nets offer a potential solution. If a small neural net is trained on a singular image, the neural net (and some error correction) is all that needs to be sent to the intergalactic lander. The total size of the compressed image can be calculated with the following equation[1]:

(ERRORBITS * NUMPIXELS + 32 * NUMPARAMETERS) / NUMPIXELS

where ERRORBITS is found using [1]:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}$$

$$ERRORBITS = 2 + \log_2(RMSE)$$

This problem attempts to minimize these equations on several height map tiles taken in the United States by USGS.

Model Design

Before tuning the model's architecture, it is essential to find some basics to set it up for success. Cleaning input data is always a safe place to start, and selecting activation functions that match the problem will help the model succeed. Once a good model foundation has been selected, different layer sizes and training methods can be used to produce further optimizations.

The input is the latitude and longitude of each pixel in an image. Neural networks prefer to work with numbers around 0 ± 1, which is very uncommon when using pure latitude and longitude values. To correct this issue, each image can be shifted to have the values centered around zero. To achieve this shift, keras provides a Normalization layer that can be incorporated into the neural network in between the input and the first hidden layer.

The final models for compression often used two types of activation functions: tanh and something similar to ReLu. Activation functions that output values near zero are popular because they allow for consistent model learning, and force improvements to be more separated between layers. However, near zero output activation functions have one major limitation: they have trouble learning when the input values are high because the slope is so close to zero. Activation

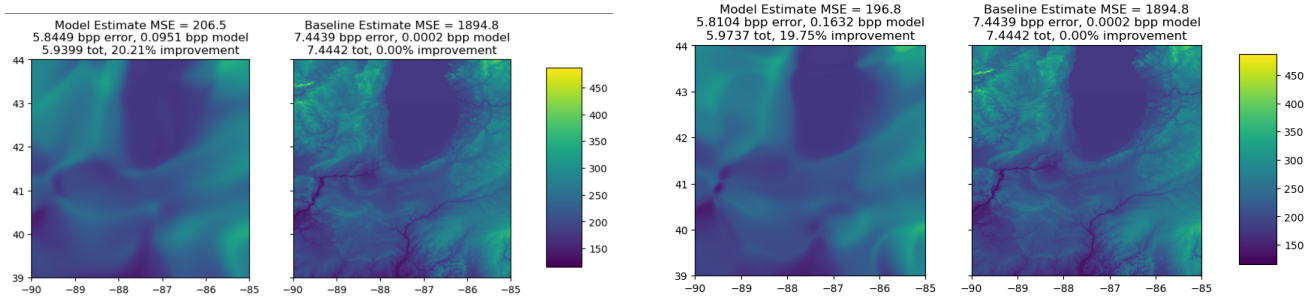
functions like swish and ReLu fix this issue by continuing to have a non zero slope, even as input values get very high[2]. In an attempt to capitalize on the benefits of both of these activation functions, models were selected to have the first couple of hidden layers utilize the tanh function, while the last couple of hidden layers used a function similar to ReLu (ReLu, Swish, Selu, etc).

When compiling a model, there are a wide variety of parameters to change which will affect the learning of the model. These include the loss function, the optimizer, and the learning rate. Our team saw little to no improvement by changing these fields, so they remained as follows: the loss function used is mean squared error, and the optimizer is Adam with a learning rate of 0.01.

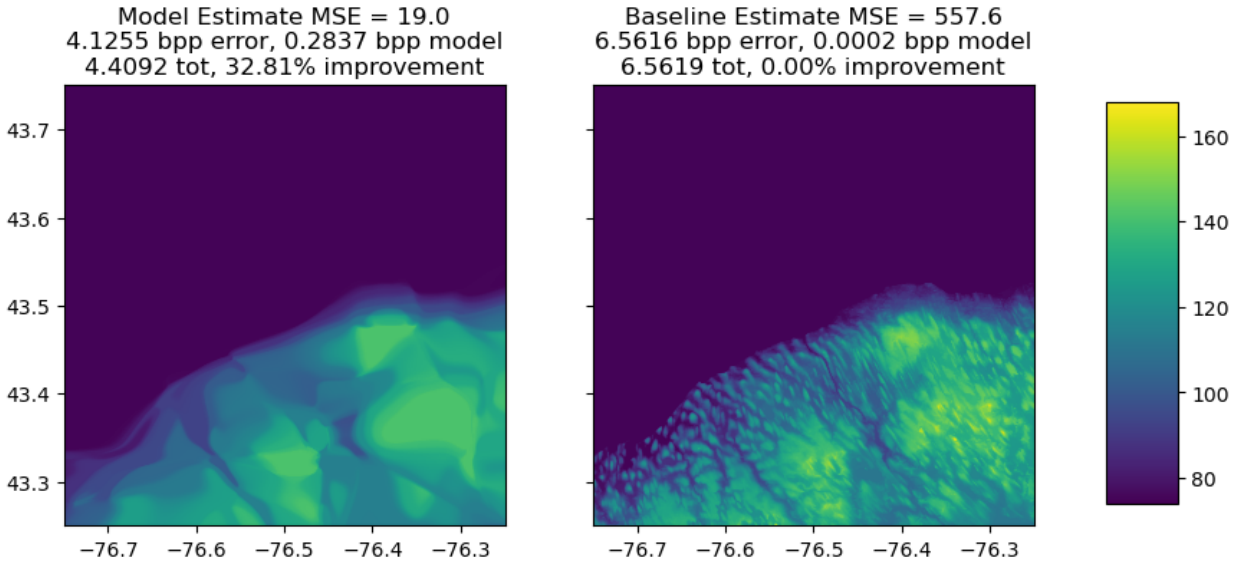
When it comes to training a model, there are several ways to modify the learning style to help quickly improve the model. The batch size and the number of epochs are the two primary parameters to tune. A small batch size means that a small subset of the training data are selected, which means that the model trains on a blurry version of the image. Also, the potential improvement per epoch is higher when the batch size is lower. Using this information, it is helpful to begin training on a small batch size, and then increase the batch size as the improvement per epoch begins to disappear. As the batch sizes get larger, the model trains on a more precise version of the image with smaller improvements. Keras has an early stopping feature that can be helpful to stop each batch size at the optimal number of epochs without unnecessary trial and error.

The final step is to optimize the size and shape of each neural network. Because each additional parameter is detrimental, each neuron has to be worth its value in the amount of improvement it provides. Each model generally stuck to a pyramidal shape for the hidden layers, under the assumption that the level of complexity required to describe the data in each layer should be decreasing. When choosing the size of the first layer, the complexity of the specific image is vital. Some images have vast sections of flat terrain, and these images are more efficiently described by simpler neural networks. On the other hand, an image with tons of variation is better described by a more complex network with a higher number of neurons in the first hidden layer. However, this recommendation should only be used if other methods to increase complexity fail to produce the desired results. In some situations, increasing the number of hidden layers from ~3 to ~6 seems to produce better results per added parameter than a similar change in the size of the hidden layers.

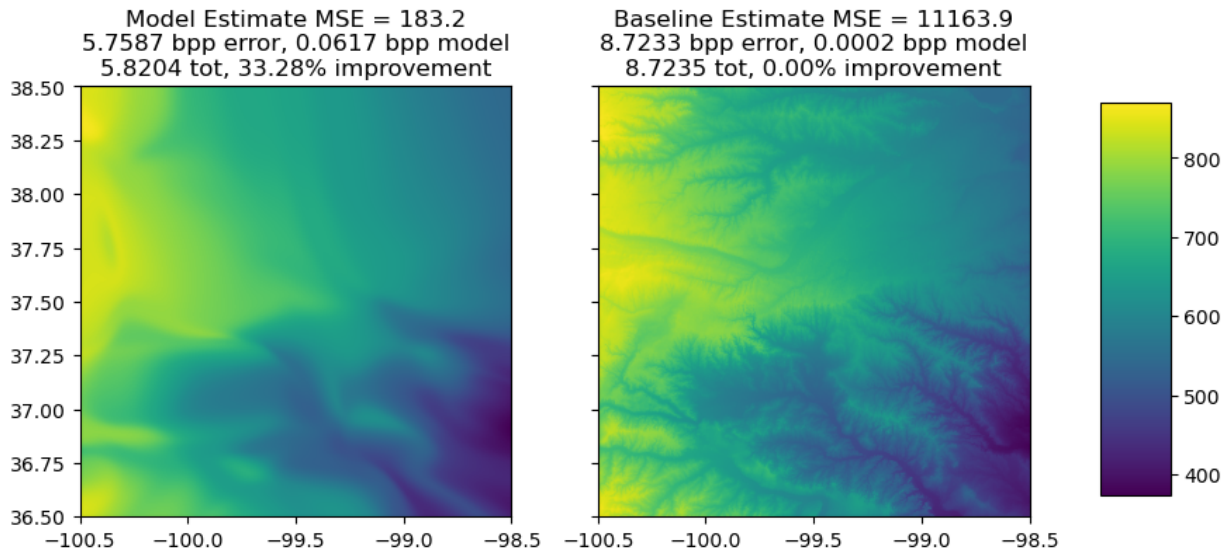
Trained Images



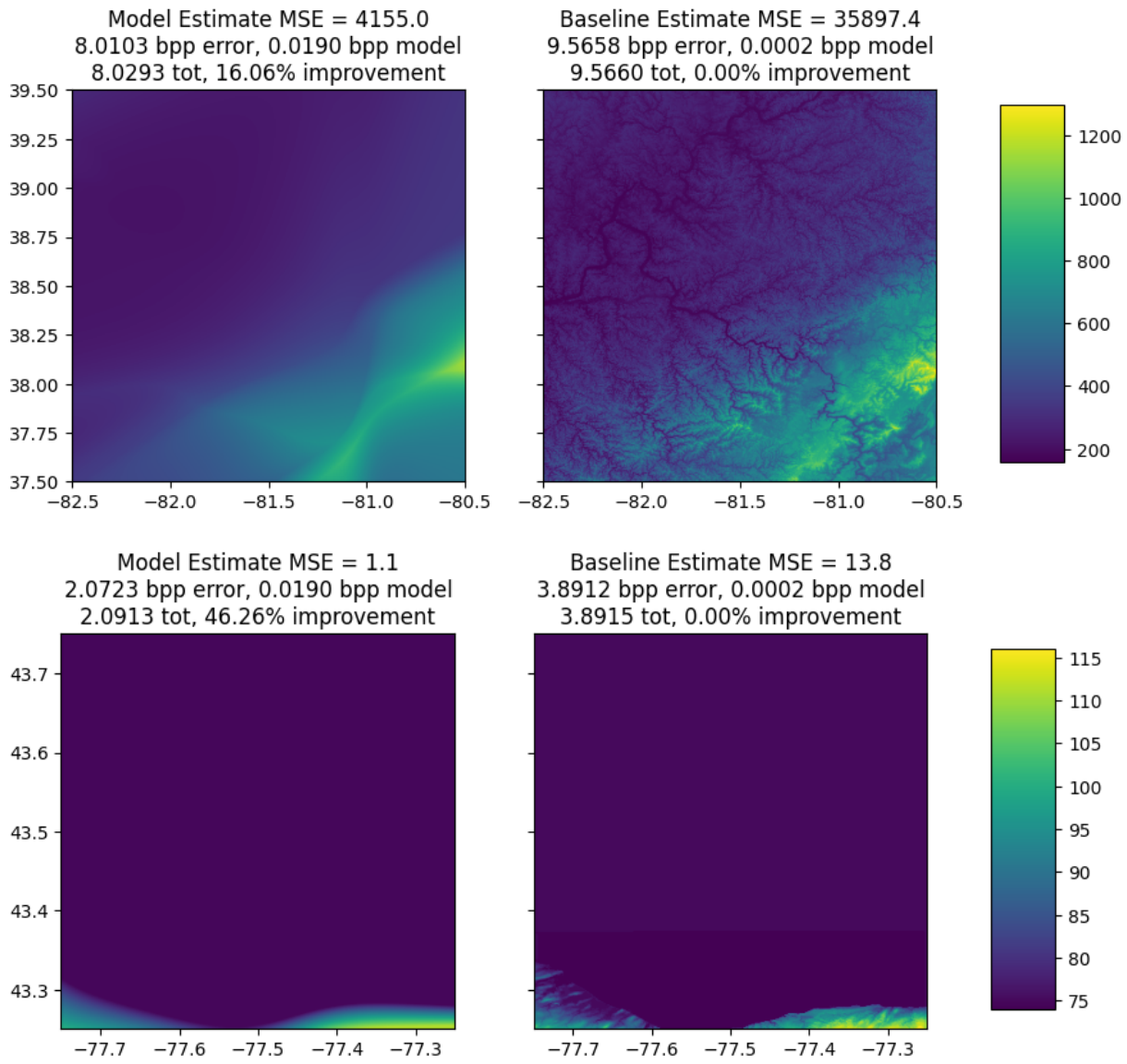
These are the top two results for the group image (-90.0, -85.0, 39.0, 44.0). The model on the left used a deep layer structure of (normalization, 8, 16, 8, 8, 1) and an activation function layout of (swish, tanh, tanh, swish, linear). The model on the left has a deep layer structure of (normalization, 12, 12, 12, 10, 8, 6, 4, 2, 1) and an activation layer layout of (tanh, tanh, tanh, tanh, tanh, tanh, relu, relu, relu).



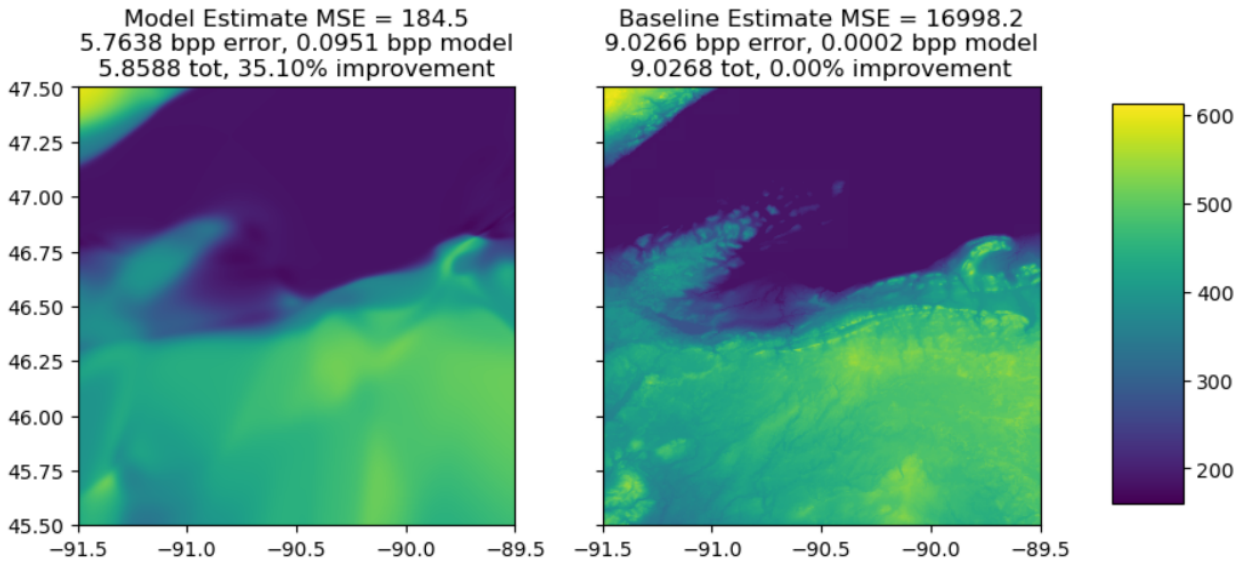
This is the top results for -76.75, -76.25, 43.25, 43.75. The model used a deep layer structure of (normalization, 16, 16, 12, 12, 12, 10, 8, 6, 4, 2, 1) and an activation function layout of (swish, swish, swish, tanh, tanh, tanh, tanh, tanh, tanh, swish, relu).



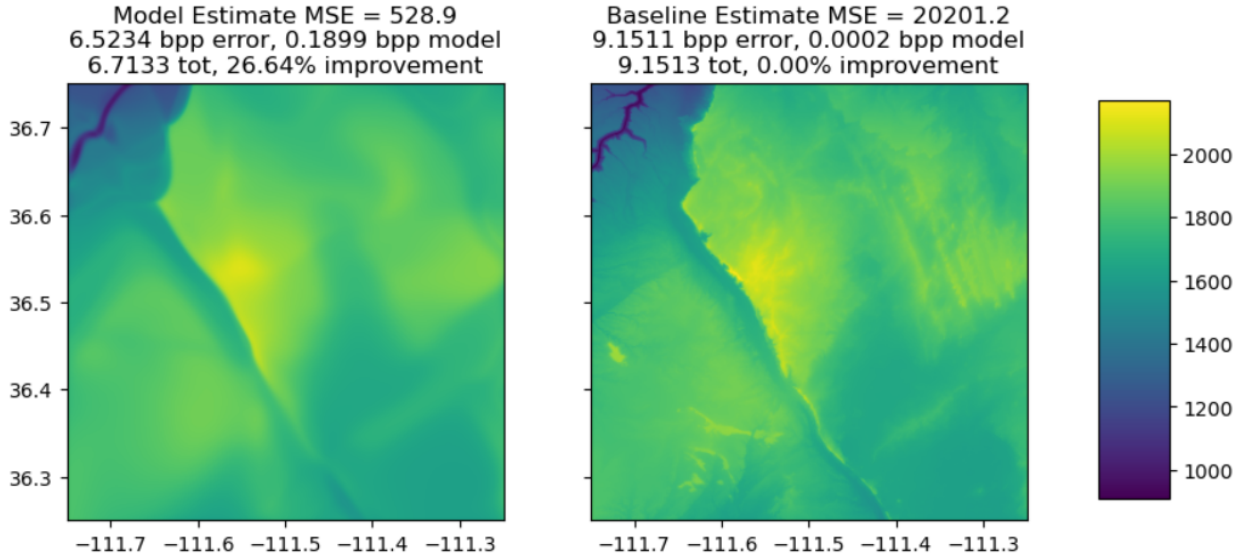
This is the top results for -100.5, -98.5, 36.5, 38.5. The model used a deep layer structure of (normalization, 8, 7, 6, 5, 5, 4, 3, 2, 1) and an activation function layout of (tanh, tanh, tanh, tanh, tanh, tanh, relu, relu, relu).



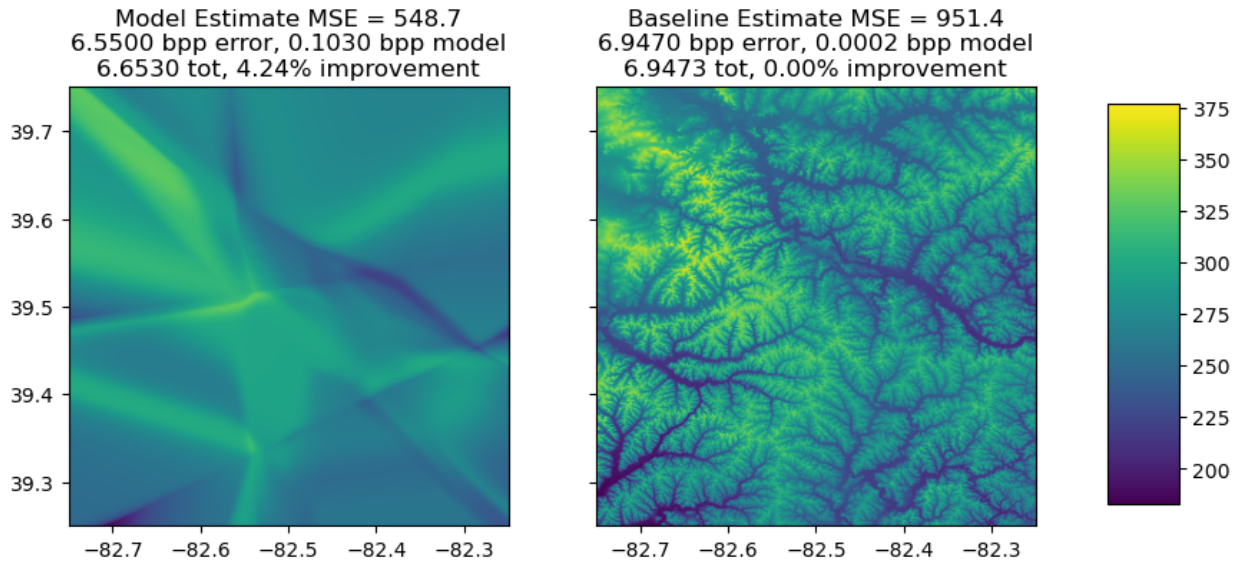
These 2 images both used the same neural network structure. These results are for -77.75, -77.25, 43.25, 43.75 and -82.5, -80.5, 37.5, 39.5. The models used a deep layer structure of (normalization, 8, 4, 2, batch normalization, 1) and an activation function layout of (swish, swish, swish, linear).



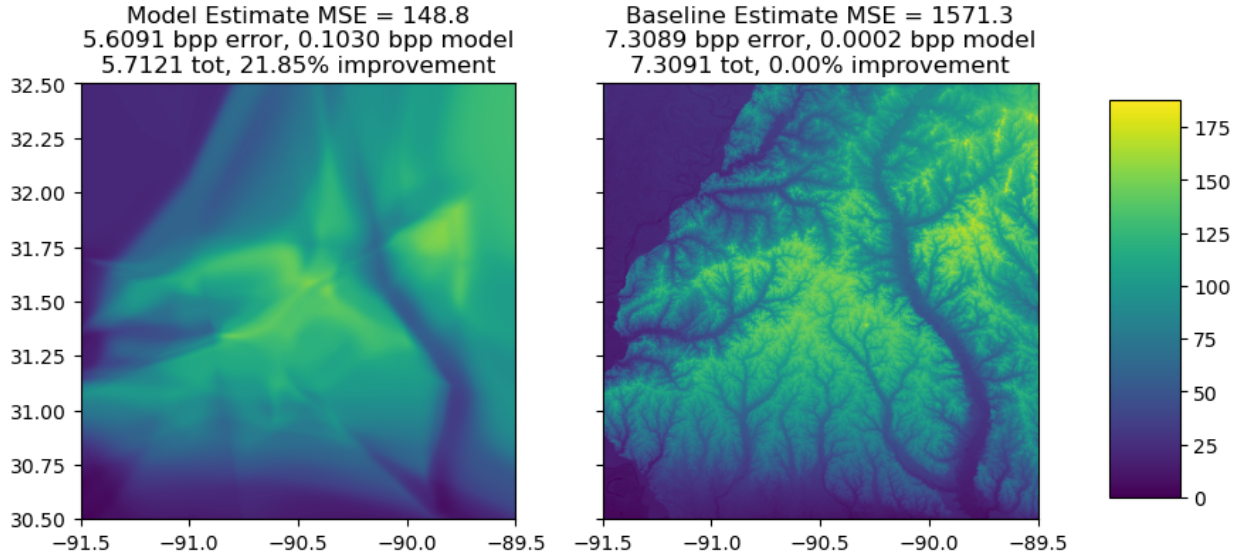
This is the top results for -91.5, -89.5, 45.5, 47.5. The model used a deep layer structure of (normalization, 8, 16, 8, 8, 1) and an activation function layout of (swish, tanh, tanh, swish, linear).



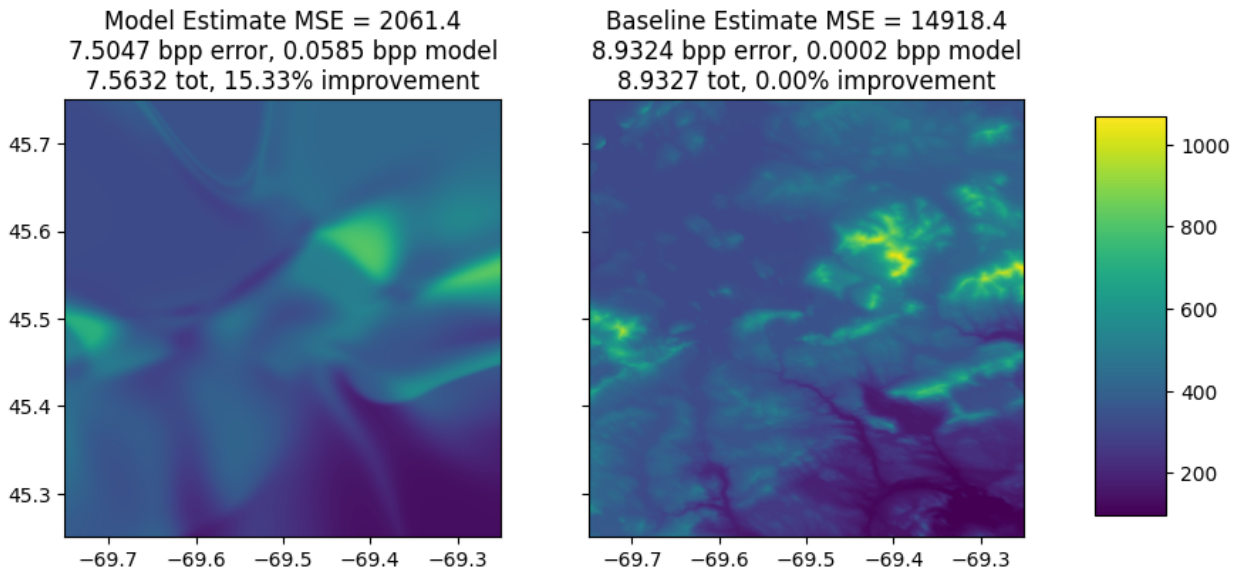
This is the top results for -111.75, -111.25, 36.25, 36.75. The model used a deep layer structure of (normalization, 32, 16, 8, 1) and an activation function layout of (tanh, sigmoid, swish, linear).



This is one of the top results for -87.75, -87.25, 39.25, 39.75. The model used a deep layer structure of (normalization, 32, 4, 2, 1) and an activation function layout of (swish, swish, swish, swish).



This is one of the top results for -91.5, -89.5, 30.5, 32.5. The model used a deep layer structure of (normalization, 16, 8, 4, 4, 1) and an activation function layout of (tanh, swish, swish, swish, linear).



This is one of the top results for -69.75, -69.25, 45.25, 45.75. The model used a deep layer structure of (normalization, 16, 8, 4, 2, batch normalization, 1) and an activation function layout of (swish, swish, tanh, swish, relu).

Some of these models perform better than others, and they all have different shapes. The model for -77.75, -77.25, 43.25, 43.75 extraordinarily well, which means that it is tuned very well for that specific image. The image contains a lot of flat space, so it is reasonable to conclude that a small model can represent the image with enough accuracy to be efficient. Many of the other models could be drastically improved if each model was better tuned to each image. Given more time, team C could have analysed the differences in their models to better produce specific models for each image.

References

[1] Terrain Assignment PDF, Mitch Parry, CS 4531
[2] An Introduction to the ReLU Activation Function, Bharath Krishnamurthy, https://builtin.com/machine-learning/relu-activation-function