Semantic Segmentation of Building Footprints Using U-Net

Amber Walker

May 2024

1 Introduction

In this homework assignment, I perform semantic segmentation to extract building footprints from aerial images. Essentially, I create a model that's able to predict whether each pixel in an image belongs to a building or not, using a U-Net convolutional neural network architecture.

1.1 U-net Model

What is a U-net model? U-Net is a type of convolutional neural network (CNN) designed specifically for image segmentation tasks. It was originally developed for Biomedical image segmentation [Ronneberger, Fischer, and Brox 2015].

The reason for its name is because of its unique symmetric architecture, with an encoder (contracting path) and a decoder (expanding path) which gives it a U-shaped appearance.

The encoder is made up of multiple convolutional layers followed by max-pooling layers, where each convolutional layer applies a set of filters to the input, followed by an activation function. The max-pooling layers help to reduce the spatial dimension of the input, which helps the network capture more broad, general patterns while also reducing computation resources.

The decoder consists of upsampling layers, which are basically the opposite of pooling layers (they increase the spatial dimensions of the input feature maps and are often implemented with transposed convolutions), followed by convolutional layers.

The purpose of the decoder is to reconstruct the spatial dimensions of the input while refining the features to produce a detailed segmentation map.

One of the unique features of U-Net is the use of skip connections that link corresponding layers in the encoder and decoder. These connections concatenate feature maps from the encoder to the decoder, providing the decoder with high-resolution features from the encoder. This helps the network retain fine details and improves the accuracy of the segmentation.

The final layer of the U-Net typically uses a 1x1 convolution to map the feature representation to the desired number of output channels (e.g., one channel for binary segmentation). A sigmoid activation function is often applied to the output layer for binary segmentation tasks, producing a probability map where each pixel represents the likelihood of belonging to a specific class.

2 Data Preparation

The first step was to load a dataset of 3347 color raster images, each with dimensions of 256x256x3 pixels, representing areas of 300 square meters in the state of Massachusetts. Corresponding to each image, there is a binary masks indicating building footprints. These masks are derived from OpenStreetMap data.

Next I normalized the pixel values to the range [0, 1], by dividing each pixel value by the maximum possible value, which is 255. Then check the dimensions and display several matching image and label pairs to ensure correspondence.

Below is a code snippet and a sample image.

```python
# Check dimensions
print('Train Images Shape:', train_images.shape)
print('Train Labels Shape:', train_labels.shape)
print('Validation Images Shape:', val_images.shape)
print('Validation Labels Shape:', val_labels.shape)
print('Test Images Shape:', test_images.shape)
print('Test Labels Shape:', test_labels.shape)

# Display several matching image and label pairs
n_samples = 3
sample_indices = np.random.choice(train_images.shape[0], n_samples, replace=False)

for idx in sample_indices:
    compare([train_images[idx], train_labels[idx]], titles=['Image', 'Label'])
```

2.1 Explore Data

```
Train Images Shape: (234, 256, 256, 3)
Train Labels Shape: (234, 256, 256, 1)
Validation Images Shape: (50, 256, 256, 3)
Validation Labels Shape: (50, 256, 256, 1)
Test Images Shape: (50, 256, 256, 3)
Test Labels Shape: (50, 256, 256, 1)
```

The dataset is split into training, validation, and test sets with 234 training images, 50 validation images, and 50 test images. This suggests a training-validation-test split ratio, approximately 70%-15%-15%.

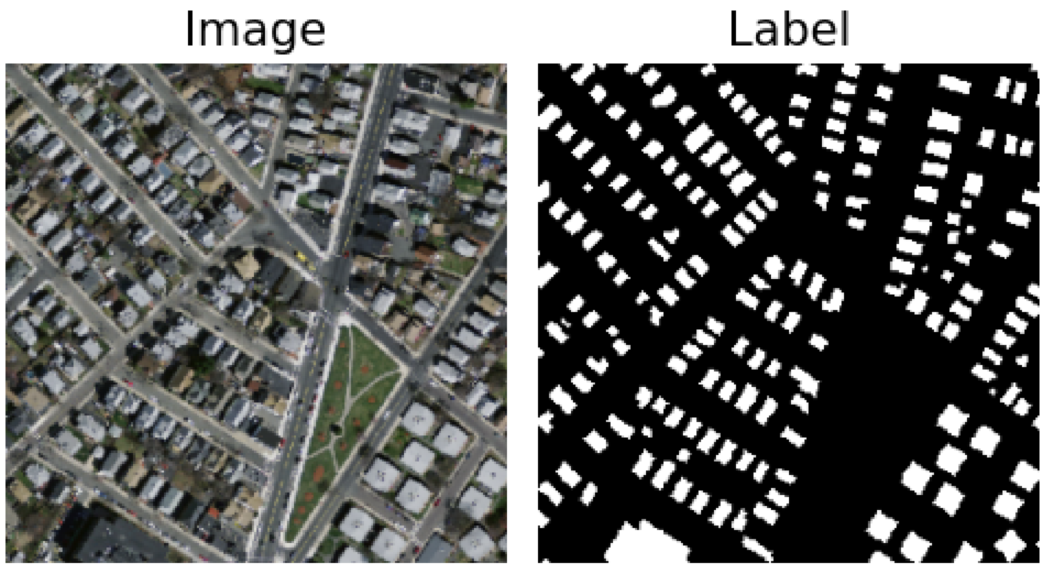

Figure 1: Input-output pair used in semantic segmentation tasks with the U-Net model.

In Figure 1, on the left is the original input image and on the right is the binary mask corresponding to the left sub-image. The white regions represent the building footprints. Each white pixel indicates that the corresponding pixel in the input image is part of a building. The black regions represent non-building areas, such as roads, open spaces, and vegetation.

2.2 Class Distribution

Next, I want to see the class distribution to see if the classes are balanced or unbalanced.

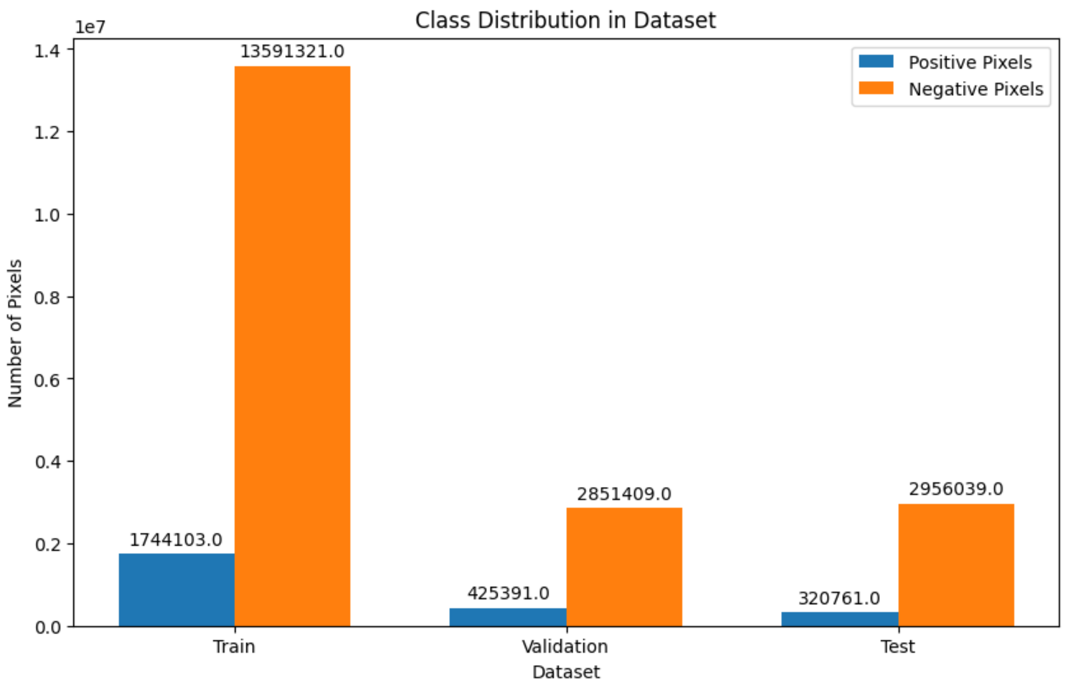

Figure 2: Plot of the class distribution

In Figure 2, we can see our classes are imbalanced and the negative pixels outweigh the positive pixels. This makes sense as the pixels outside of the buildings in the images make up more space than the objects themselves.

3 Simplified U-Net Model

In this section we use the Keras functional API to define a simplified U-net model.

3.1 Hyperparameter Search

Before defining my model, I set up Keras Tuner to perform the hyperparameter search using the `Hyperband` algorithm. The search is conducted over multiple activation functions for the hidden layers, different types of kernel initializers, optimizers, learning rates and number of filters.

3.1.1 Activation Functions

Activation functions introduce non-linearity into the model, enabling it to learn complex patterns. The activation functions considered are:

- **ReLU (Rectified Linear Unit)**: Defined as $f(x) = \max(0, x)$. It is simple and effective, often leading to faster training and better performance.
- **Tanh (Hyperbolic Tangent)**: Defined as $f(x) = \tanh(x)$. It outputs values between -1 and 1, making it zero-centered.
- **Sigmoid**: Defined as $f(x) = \frac{1}{1+e^{-x}}$. It outputs values between 0 and 1, commonly used for binary classification.
- **Leaky ReLU**: A variation of ReLU where $f(x) = x$ for $x \geq 0$ and $f(x) = \alpha x$ for $x < 0$. The parameter α controls the slope for negative values, mitigating the dying ReLU problem. (We also search for the best value of α).

3.1.2 Kernel Initializers

Kernel initializers determine the initial values of the weights in the network, impacting the convergence and stability of the model:

- **Glorot Uniform (Xavier Uniform)**: Initializes weights from a uniform distribution within $\left[-\sqrt{\frac{6}{n_{in}+n_{out}}}, \sqrt{\frac{6}{n_{in}+n_{out}}}\right]$.
- **He Normal**: Initializes weights from a normal distribution with a standard deviation of $\sqrt{\frac{2}{n_{in}}}$, suited for ReLU activations.
- **LeCun Normal**: Initializes weights from a normal distribution with a standard deviation of $\sqrt{\frac{1}{n_{in}}}$, often used for Leaky ReLU and sigmoid activations.

3.1.3 Optimizers

Optimizers update the model's weights based on the computed gradients. The optimizers I consider are:

- **Adam (Adaptive Moment Estimation)**: Combines the advantages of two extensions of stochastic gradient descent, AdaGrad and RMSProp. It computes adaptive learning rates for each parameter.
- **SGD (Stochastic Gradient Descent)**: Updates the parameters by following the negative gradient of the loss function.
- **RMSprop (Root Mean Square Propagation)**: Divides the learning rate by an exponentially decaying average of squared gradients, adapting the learning rate for each parameter.

3.1.4 Learning Rate

The learning rate controls the step size during the optimization process. Choosing the right learning rate can improve the convergence speed and performance of the model. If it's too small, it will take a longer time to converge. If it's too large it might oscillate and never converge.

3.1.5 Results

The optimal hyperparameters found by Keras Tuner are as follows:

- **Activation Function**: leaky_relu
- **alpha for Leaky ReLU**: 0.21000000000000002
- **Kernel Initializer**:he_normal
- **Learning Rate**: 0.0011349803487026022
- **Optimizer**: adam

These hyperparameters are then used to build and compile the final simplified U-net model.

3.2 Train

The next step is to initialise and compile the model using the chosen optimization algorithm.

```
#initialize the model
unet_simp_model = unet_simp(input_size=(256, 256, 3))
adam = Adam(learning_rate=best_hps.get('learning_rate'))

#compile the model
unet_simp_model.compile(optimizer=adam,
                    loss='binary_crossentropy',
                    metrics=['accuracy', Precision(), Recall(), AUC()])
```

```

unet_simp_model.summary()
```

Below is a summary of the model, including the output shape of each layers and number of params.

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 256, 256, 3)	0	[]
conv2d (Conv2D)	(None, 256, 256, 32)	896	['input_1[0][0]']
activation (Activation)	(None, 256, 256, 32)	0	['conv2d[0][0]']
conv2d_1 (Conv2D)	(None, 256, 256, 32)	9248	['activation[0][0]']
activation_1 (Activation)	(None, 256, 256, 32)	0	['conv2d_1[0][0]']
max_pooling2d (MaxPooling2D)	(None, 128, 128, 32)	0	['activation_1[0][0]']
conv2d_2 (Conv2D)	(None, 128, 128, 48)	13872	['max_pooling2d[0][0]']
activation_2 (Activation)	(None, 128, 128, 48)	0	['conv2d_2[0][0]']
conv2d_3 (Conv2D)	(None, 128, 128, 48)	20784	['activation_2[0][0]']
activation_3 (Activation)	(None, 128, 128, 48)	0	['conv2d_3[0][0]']
max_pooling2d_1 (MaxPooling2D)	(None, 64, 64, 48)	0	['activation_3[0][0]']
conv2d_4 (Conv2D)	(None, 64, 64, 32)	13856	['max_pooling2d_1[0][0]']
activation_4 (Activation)	(None, 64, 64, 32)	0	['conv2d_4[0][0]']
conv2d_5 (Conv2D)	(None, 64, 64, 32)	9248	['activation_4[0][0]']
activation_5 (Activation)	(None, 64, 64, 32)	0	['conv2d_5[0][0]']
max_pooling2d_2 (MaxPooling2D)	(None, 32, 32, 32)	0	['activation_5[0][0]']
conv2d_6 (Conv2D)	(None, 32, 32, 64)	18496	['max_pooling2d_2[0][0]']
activation_6 (Activation)	(None, 32, 32, 64)	0	['conv2d_6[0][0]']
conv2d_7 (Conv2D)	(None, 32, 32, 64)	36928	['activation_6[0][0]']
activation_7 (Activation)	(None, 32, 32, 64)	0	['conv2d_7[0][0]']
max_pooling2d_3 (MaxPooling2D)	(None, 16, 16, 64)	0	['activation_7[0][0]']
conv2d_8 (Conv2D)	(None, 16, 16, 384)	221568	['max_pooling2d_3[0][0]']
activation_8 (Activation)	(None, 16, 16, 384)	0	['conv2d_8[0][0]']
conv2d_9 (Conv2D)	(None, 16, 16, 384)	1327488	['activation_8[0][0]']
activation_9 (Activation)	(None, 16, 16, 384)	0	['conv2d_9[0][0]']
conv2d_transpose (Conv2DTranspose)	(None, 32, 32, 64)	98368	['activation_9[0][0]']
concatenate (Concatenate)	(None, 32, 32, 128)	0	['conv2d_transpose[0][0]', 'activation_7[0][0]']
conv2d_10 (Conv2D)	(None, 32, 32, 64)	73792	['concatenate[0][0]']
activation_10 (Activation)	(None, 32, 32, 64)	0	['conv2d_10[0][0]']
conv2d_11 (Conv2D)	(None, 32, 32, 64)	36928	['activation_10[0][0]']
activation_11 (Activation)	(None, 32, 32, 64)	0	['conv2d_11[0][0]']
conv2d_transpose_1 (Conv2DTranspose)	(None, 64, 64, 32)	8224	['activation_11[0][0]']
concatenate_1 (Concatenate)	(None, 64, 64, 64)	0	['conv2d_transpose_1[0][0]', 'activation_5[0][0]']
conv2d_12 (Conv2D)	(None, 64, 64, 32)	18464	['concatenate_1[0][0]']
activation_12 (Activation)	(None, 64, 64, 32)	0	['conv2d_12[0][0]']
conv2d_13 (Conv2D)	(None, 64, 64, 32)	9248	['activation_12[0][0]']
activation_13 (Activation)	(None, 64, 64, 32)	0	['conv2d_13[0][0]']
conv2d_transpose_2 (Conv2DTranspose)	(None, 128, 128, 48)	6192	['activation_13[0][0]']
concatenate_2 (Concatenate)	(None, 128, 128, 96)	0	['conv2d_transpose_2[0][0]', 'activation_3[0][0]']
conv2d_14 (Conv2D)	(None, 128, 128, 48)	41520	['concatenate_2[0][0]']
activation_14 (Activation)	(None, 128, 128, 48)	0	['conv2d_14[0][0]']
conv2d_15 (Conv2D)	(None, 128, 128, 48)	20784	['activation_14[0][0]']
activation_15 (Activation)	(None, 128, 128, 48)	0	['conv2d_15[0][0]']
conv2d_transpose_3 (Conv2DTranspose)	(None, 256, 256, 32)	6176	['activation_15[0][0]']
concatenate_3 (Concatenate)	(None, 256, 256, 64)	0	['conv2d_transpose_3[0][0]', 'activation_1[0][0]']
conv2d_16 (Conv2D)	(None, 256, 256, 32)	18464	['concatenate_3[0][0]']
activation_16 (Activation)	(None, 256, 256, 32)	0	['conv2d_16[0][0]']
conv2d_17 (Conv2D)	(None, 256, 256, 32)	9248	['activation_16[0][0]']
activation_17 (Activation)	(None, 256, 256, 32)	0	['conv2d_17[0][0]']
conv2d_18 (Conv2D)	(None, 256, 256, 1)	33	['activation_17[0][0]']
Total params:	**2019825 (7.71 MB)**		
Trainable params:	**2019825 (7.71 MB)**		
Non-trainable params:	**0 (0.00 Byte)**		

3.3 Estimate Model Parameters

The next step is to estimate the model parameters using the training and the validation sample. I use batch-size of 32 and 20 training epoch as well as an early-stopping callback with a patience parameter of 3 to recover the parameters that minimise the error on the validation sample.

```
#early stopping
early_stopping = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)

#train
history = unet_simp_model.fit(
    train_images, train_labels,
    validation_data=(val_images, val_labels),
    batch_size=32,
    epochs=20,
    callbacks=[early_stopping]
)
```

In order to understand and visualize the evolution of the training and validation metrics during training, I use the provided ‘display_history‘ utility function provided by the professor.

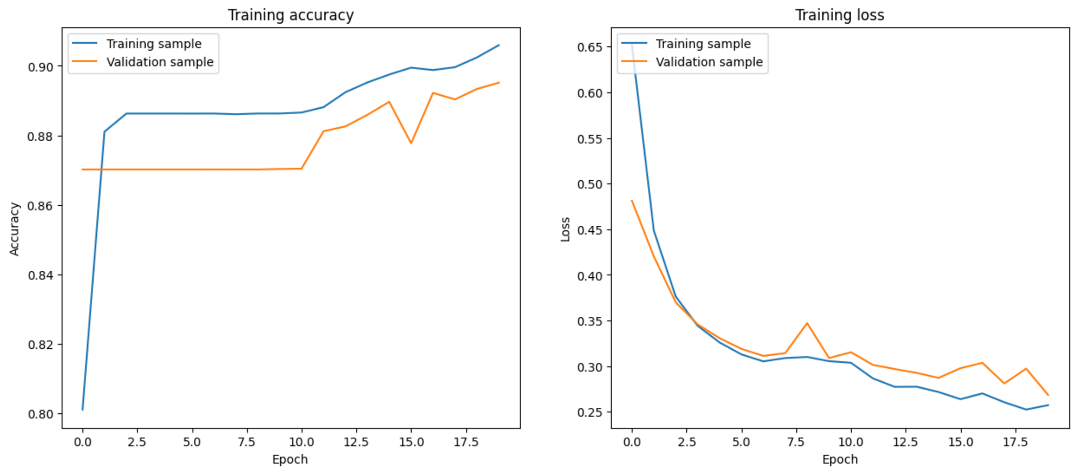

Figure 3: Plot of the class distribution

The training and validation metrics plotted in Figure 3 provide insights into the model’s performance during training. Key observations:

- **Training Accuracy vs. Validation Accuracy:**
  - The training accuracy increases steadily and surpasses the validation accuracy throughout the training process.
  - This gap between training and validation accuracy suggests that the model is learning the training data well but is not generalizing as effectively to the validation data.
- **Training Loss vs. Validation Loss:**
  - The training loss decreases rapidly at the beginning and continues to decrease, indicating that the model is fitting the training data progressively better.
  - The validation loss also decreases initially, showing improvement on unseen data. However, it starts to fluctuate and eventually stops decreasing significantly after a certain number of epochs.
  - When the validation loss is lower than the training loss or starts to increase while the training loss continues to decrease, it is a strong indicator of overfitting.

Next, I evaluate the model's generalization performance.

```python
#evaluate the model on the test set
test_loss, test_accuracy, test_precision, test_recall, test_auc = unet_simp_model.evaluate(test_images,
    test_labels, verbose=1)

print(f"Test Loss: {test_loss}")
print(f"Test Accuracy: {test_accuracy}")
print(f"Test Precision: {test_precision}")
print(f"Test Recall: {test_recall}")
print(f"Test AUC: {test_auc}")
```

The performance of the model on the test dataset is summarized in Table 1. Each metric provides us with insights into different parts of the model's performance.

Metric	**Value**
Test Loss	0.225
Test Accuracy	91.5%
Test Precision	60.1%
Test Recall	39.1%
Test AUC	0.886

Table 1: Model evaluation metrics on the test dataset.

3.3.1 Interpretation

- **Test Loss:** The test loss is 0.225, indicating that the model's predictions are relatively close to the actual values.
- **Test Accuracy:** The model achieves an accuracy of 91.5%, meaning it correctly classifies 91.5% of the test samples.
- **Test Precision:** With a precision of 60.1%, the model correctly predicts the positive class 60.1% of the time when it makes a positive prediction.
- **Test Recall:** The recall is pretty low at 39.10%, indicating that the model classifies only 39.10% of the actual positive cases. This suggests the model may be missing many true positives.
- **Test AUC:** The AUC score of 0.8862 demonstrates a high ability of the model to distinguish between positive and negative classes.

While the model shows high accuracy and AUC, the low recall indicates a significant number of actual positive cases are not being identified. To address this, we could consider adjusting the decision threshold, using data augmentation, or applying oversampling techniques to improve recall without severely impacting precision.

3.4 Predict

Next I compute the predicted probabilities for the test images and turn the probabilities into binary predictions using a threshold of 0.5. I display prediction statistics using the provided 'display_statistics' utility for several test images.

```python
test_preds = unet_simp_model.predict(test_images, batch_size=32, verbose="auto", steps=None,
    callbacks=None)
test_label_predict = (test_preds > 0.5).astype(bool)

sample = 0, 1, 2
for i in sample:
  display_statistics(test_images[i], test_labels[i], test_preds[i], test_label_predict[i])
```

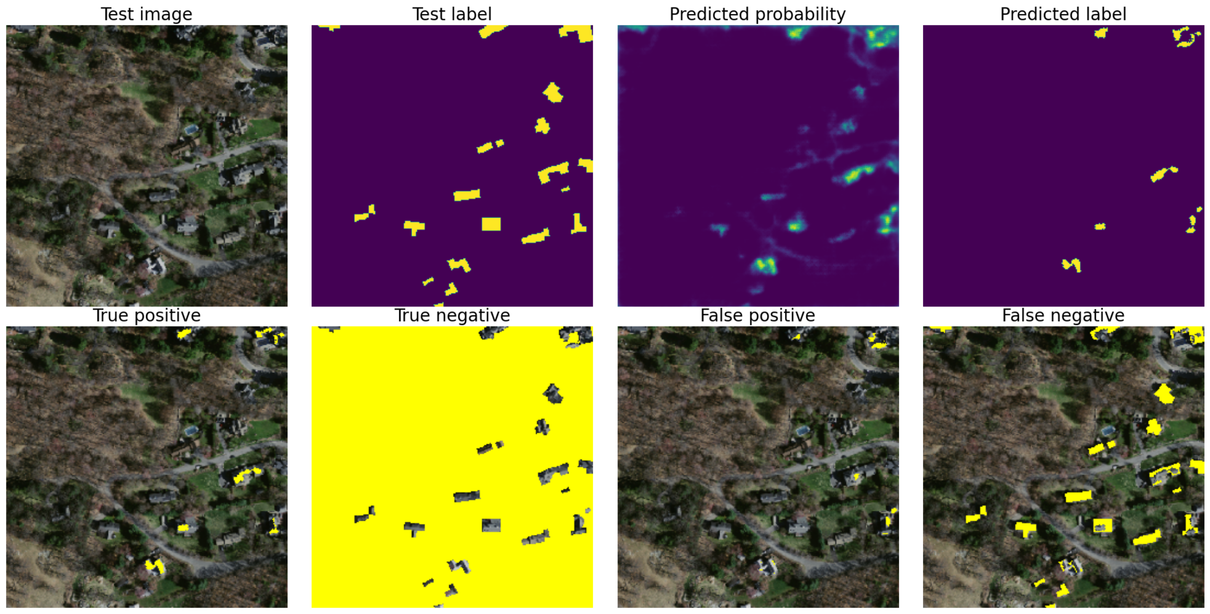

Figure 4: Display pred stats: sample 1

- **Test Image:** The original aerial image from the test dataset.
- **Test Label:** Ground truth labels indicating building locations.
- **Predicted Probability:** Model's predicted probabilities for each pixel being a building.
- **Predicted Label:** Binarized predictions indicating buildings and non-buildings.
- **True Positive:** Regions correctly identified as buildings by the model.
- **True Negative:** Regions correctly identified as non-buildings by the model.
- **False Positive:** Regions incorrectly identified as buildings by the model.
- **False Negative:** Regions where buildings were missed by the model.

The simple model with a 0.5 threshold shows good performance in correctly identifying many buildings (true positives) and non-building areas (true negatives) in Figure 4. However, since there are false positives and false negatives this means there is room for improvement. The other two samples can be found in the Appendix.

For better visualization, I am using a function (borrowed from the Advanced NLP course) to plot the accuracy, precision, recall and F1.

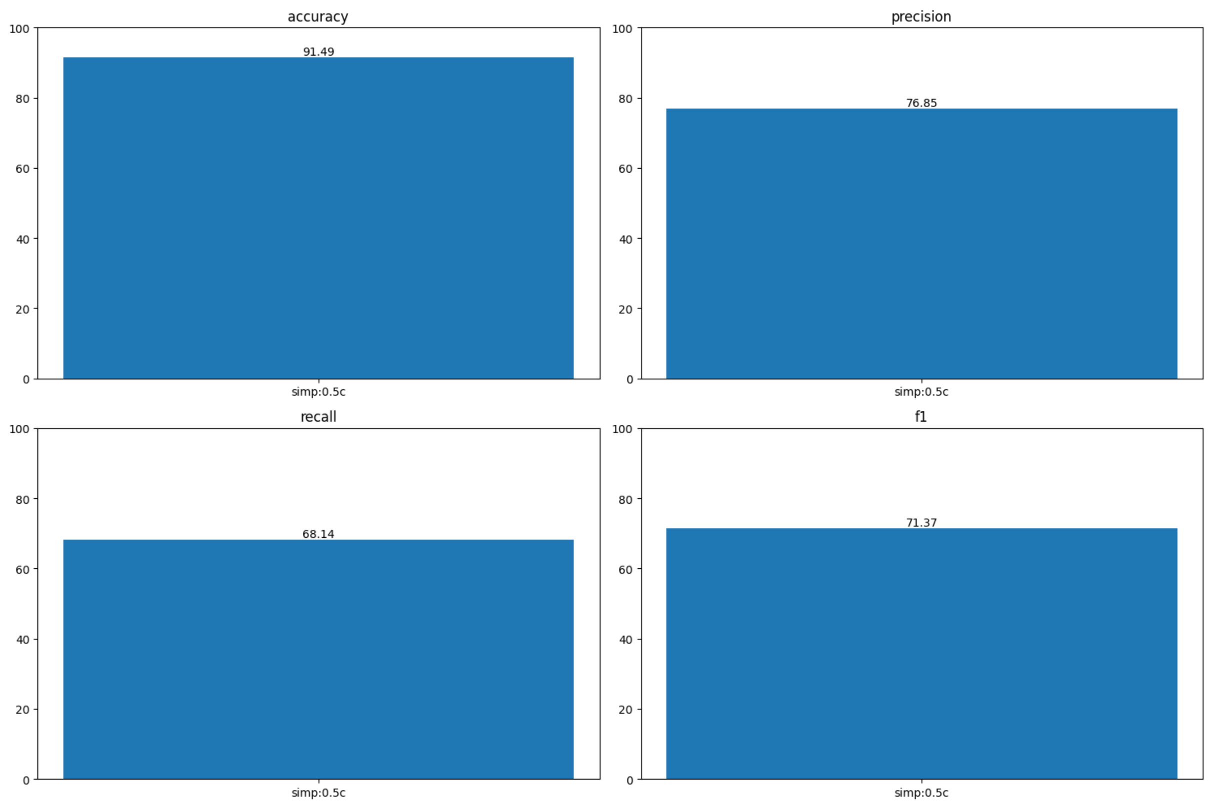

Figure 5: Metrics for U-net model at 0.5 threshold

3.5 Optimal Threshold

Assuming that we give equal importance to reducing false positives and false negatives, use a decision probability threshold that strikes the best balance between these two quantities.

To find the decision probability threshold that balances FN and FP, I developed a function to evaluate how the model performs on multiple different threshold values. The one that gives the smallest difference between the number of FP and FN will be the winner.

```python
#function to find the best threshold value
def find_best_c(val_proba_predict):
  c = np.linspace(0, 1, 100)
  fp_count = []
  fn_count = []

  for value in c:
    val_label_predict = (val_proba_predict > value).astype(bool)
    fp = np.sum(np.logical_and(np.invert(val_labels.astype(bool)), val_label_predict))
    fn = np.sum(np.logical_and(val_labels.astype(bool), np.invert(val_label_predict)))
    fp_count.append(fp)
    fn_count.append(fn)

  #select the best value
  fp_count = np.array(fp_count)
  fn_count = np.array(fn_count)
  diff = np.abs(fp_count - fn_count)
  best_c_index = np.argmin(diff)
  best_c = c[best_c_index]
  print(f"Best cutoff threshold: {best_c}")
```

```
    return best_c
```

The best threshold value in this case was 0.32. Setting the new threshold and computing the predicted labels with the new threshold, I received the following new metrics.

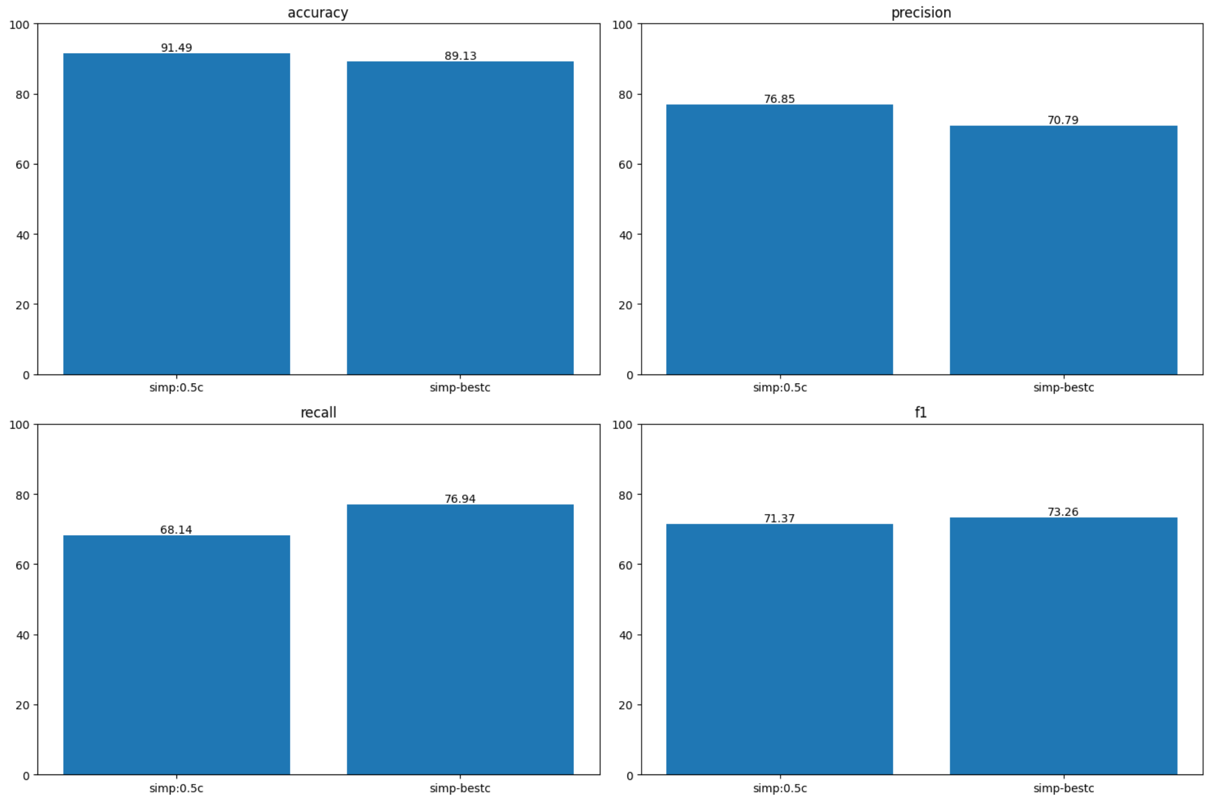

Figure 6: Metrics for U-net model at 0.22 threshold

Both models perform well in terms of overall accuracy, with the mode with the 0.5 threshold showing slightly higher accuracy. The optimal threshold model has higher precision, indicating better performance in minimizing false positives.

The model with the optimal threshold has higher recall, indicating better performance in identifying actual positives and minimizing false negatives.

The F1 score, which balances precision and recall, is higher for the model with the optimal threshold indicating it has a better overall balance between these metrics.

3.5.1 Conclusion

- For applications where reducing false negatives is critical, using the optimal threshold is preferred due to its higher recall.
- For applications where reducing false positives is critical, using 0.5 threshold is preferred due to its higher precision.
- Overall, the optimal threshold offers a better balance between precision and recall, making it more suitable for scenarios where both metrics are important.

4 Increase Predictive Performance

In order to try to increase the predictive performance of the model, I tried a few different methods. I modified the network with batch-normalization and then spatial dropout layers. Lastly, I used image augmentation techniques to increase the robustness of the model.

4.1 Batch Normalization

Batch normalization is a technique used to improve the training of deep neural networks by normalizing the input of each layer. It stabilizes and accelerates the training process by reducing the internal covariate shift, which is the change in the distribution of network activations due to changes in network parameters during training. In this project, batch normalization was applied after each convolutional layer in the U-Net simplified model. By normalizing the outputs of the convolutional layers, the hope is that the model learns more efficiently and converges faster.

Metric	**Value**
Test Loss	0.310
Test Accuracy	90.2%
Test Precision	66.9%
Test Recall	0.06%
Test AUC	0.638

Table 2: Model with Batch Normalization evaluation metrics on the test dataset.

4.2 Spatial Dropout

Spatial dropout is a regularization technique specifically designed for convolutional neural networks. Unlike standard dropout, which randomly drops individual neurons, spatial dropout drops entire feature maps, preventing adjacent neurons from becoming overly reliant on each other. This improves the robustness of the model and prevents it from overfitting. In this project, spatial dropout was applied after certain convolutional layers in the U-Net model. By randomly dropping entire feature maps during training, the hope is that the model is encouraged to learn more diverse and robust features.

Metric	**Value**
Test Loss	0.278
Test Accuracy	90.2%
Test Precision	37.3%
Test Recall	0.00592%
Test AUC	0.835

Table 3: Model with Spatial Dropout evaluation metrics on the test dataset.

4.3 Data Augmentation

Data augmentation is a strategy to artificially increase the diversity of the training data by applying various transformations such as rotations, translations, shear, and zoom. This helps improve the model's ability to generalize to new, unseen data. In this project, data augmentation was implemented using the `ImageDataGenerator` from Keras, which applied random rotations, width and height shifts, shearing, zooming, and horizontal flips to the training images. By augmenting the training data, the model is exposed to a wider variety of input conditions, in hopes to improve its robustness and ability to handle different scenarios in the test data.

Metric	Value
Test Loss	0.272
Test Accuracy	90.2%
Test Precision	0.0%
Test Recall	0.0%
Test AUC	0.826

Table 4: Model with Data Augmentation evaluation metrics on the test dataset.

5 Performance Comparison of Models with Different Techniques

Figure 7 shows the comparison of accuracy, precision, recall, and F1 score for various models and configurations: ”simp:0.5c”(Simplified mode with 0.5 threshold), ”simp-bestc” (Simplified model with optimal threshold, ”bn” (Model with batch normalization), ”sd” (Model with spatial dropout), and ”aug-data” (Model with data augmentation).

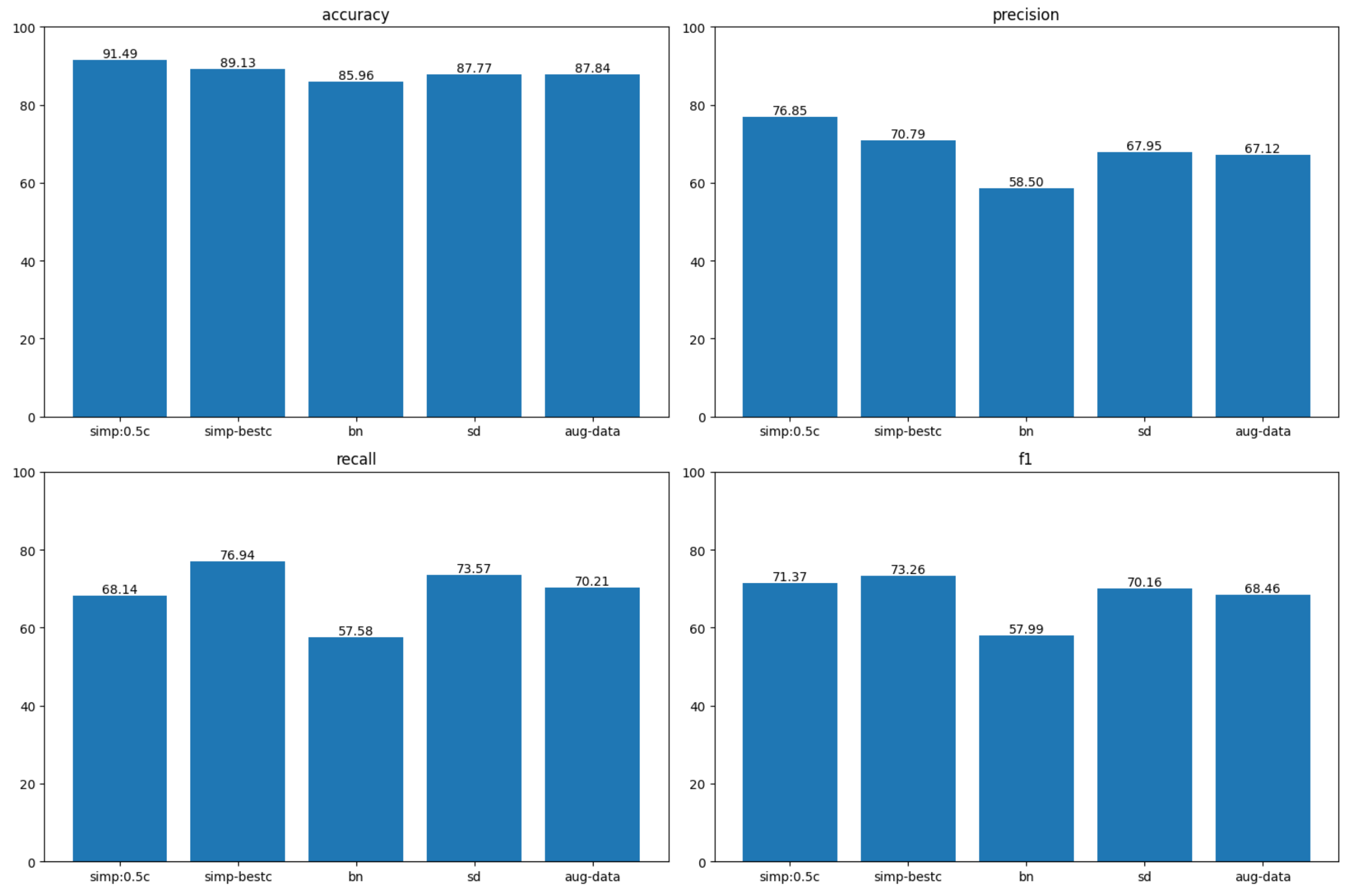

Figure 7: Comparison of performance metrics for different model configurations.

5.1 Accuracy

- **Highest:** Simplified U-net model with 0.5 threshold (91.49%)
- **Lowest:** Model with Batch Normalization (85.96%)
- **Interpretation:** The simple U-net model with 0.5 threshold correctly classifies the most samples overall, while batch normalization results in the lowest accuracy.

5.2 Precision

- **Highest:** Simplified U-net model with 0.5 threshold (76.85%)
- **Lowest:** Model with Batch Normalization (58.50%)
- **Interpretation:** Simplified U-net model with 0.5 threshold is best at minimizing false positives, whereas batch normalization produces the most false positives.

5.3 Recall

- **Highest:** Simplified U-net model with optimal threshold (76.94%)
- **Lowest:** Model with Batch Normalization (57.58%)
- **Interpretation:** Simplified U-net model with optimal threshold identifies the most true positives, while batch normalization misses the most true positives.

5.4 F1 Score

- **Highest:** Simplified U-net model with optimal threshold (73.26%)
- **Lowest:** Model with Batch Normalization (57.99%)
- **Interpretation:** Simplified U-net model with optimal threshold has the best balance between precision and recall, whereas batch normalization struggles the most.

5.5 Conclusion

- **Simplified U-net model with 0.5 threshold :** Best for reducing false positives with the highest accuracy and precision.
- **Simplified U-net model with optimal threshold:** Best for reducing false negatives with the highest recall and F1 score.
- **Batch Normalization (bn):** Needs improvement, with the lowest performance across metrics.
- **Spatial Dropout (sd):** Effective for improving recall but needs better precision.
- **Data Augmentation (aug-data):** Provides a good balance between precision and recall, improving model robustness.

Overall, it seems the simplified U-net model performed the best. With more time and compute units, I would like to delve deeper into building the model with more layers and focus more on tuning the parameters. If I were concerned with reducing false negatives (which would most likely be the case in situations such as correctly identifying buildings, think of aircrafts landings), I would opt for the simplified model with the optimal threshold as this has the highest recall (ability to correctly identify real positive instances).