Evaluating Semantic Segmentation Performance with Various CNN Architectures for PASCAL VOC-2007

Hargen Zheng
Halıcıoğlu Data Science Institute
University of California, San Diego
San Diego, CA 92092
yoz018@ucsd.edu

Chuong Nguyen
Dept. of Computer Science and Engineering
University of California, San Diego
San Diego, CA 92092
chn021@ucsd.edu

Nathaniel del Rosario
Halıcıoğlu Data Science Institute
University of California, San Diego
San Diego, CA 92092
nadelrosario@ucsd.edu

Ziyue Liu
Dept. of Electrical and Computer Engineering
University of California, San Diego
San Diego, CA 92092
zil085@ucsd.edu

Adam Tran
Department of Mathematics
University of California, San Diego
San Diego, CA 92092
ant010@ucsd.edu

Abstract

We utilize the PASCAL VOC-2007 dataset for pixel-level semantic segmentation, featuring pixelwise annotations for 21 object categories (including the background category). Our initial model employs a basic encoder-decoder architecture, utilizing convolution and transpose convolution layers with ReLU activation functions. The initial loss criterion is set as unweighted Cross Entropy, and batch normalization is applied, with no additional techniques like max-pooling or dropout. This baseline model yields a 0.056613 IoU and 0.734628 pixel accuracy but tends to predict most pixels as *background*. To address this, we implemented cosine annealing, data augmentation (random rotation, flip, scaling), weighted cross entropy, and dropout. These enhancements result in an improved IOU of approximately 0.07. Finally, we benchmark our model against `FCN ResNet-101` and `UNet` in terms of IOU and pixel accuracy. Notably, `fcn_resnet101` achieves a higher IOU of around 0.3. However, this is influenced by limited training data—our dataset comprises only 209 training images. Through data augmentation, we expand the training set to 836 images. Overall the highest performing model we trained in regards to the pixel accuracy and IoU metrics was the UNet which achieved over 0.754065 pixel accuracy and 0.0705095 IoU, which surpassed the baseline benchmark. In comparison to the transfer learning Fully Convolutional Network ResNet-101 model, which is pretrained on COCO dataset that has similar classification task,

CSE 151B: Deep Learning. PA2, Fully Convolutional Network for Semantic Segmentation.

the accuracy was still lower than ResNet's performance of 0.873404 and 0.330007 IoU, which was expected with a model designed with more complex architecture and similar classification task.

1 Introduction

The task of object recognition has been around for many decades, and with many different advancements in the past twenty years, as well as the increasing number of applications and use cases in everyday life, computer vision has become an increasingly important and complex field of artificial intelligence. In this report, we learn about and experiment with different architectures for Convolutional Neural Networks to perform semantic segmentation for image classification and object recognition.

To begin, we used PyTorch for all of the experiments, taking advantage of the built in methods to help build different architectures for the networks, as well as implementing methods beyond native stochastic gradient descent and random weight initialization. Specifically, we used batch normalization between the layers of the CNN instead of in the raw data; the design choice behind doing this for mini-batches instead of the full data set was to speed up training and use higher learning rates, intuitively making learning 'easier'. This is because batch normalization reduces internal covariate shift. Applying this at each layer ensures that the mean and variance at each layer stay the same, reducing the change in the distribution at each new layer, and thus leading to a more robust network. Such approach is useful especially when the network is very deep, which is something we later experiment with.

The network weights were initialized using Xavier Initialization, which combats the problem of too large or too small initial weights. Too large or too small initial weights can cause activation outputs to completely vanish during the forward pass, which is why we choose to use Xavier Initialization. This method sets a layer's weights to values chosen from a random uniform distribution bounded between $\pm\frac{\sqrt{6}}{\sqrt{n_i+n_{i+1}}}$ where n_i is the number of outgoing connections and n_{i+1} is the number of outgoing connections.

In the later sections, we discuss more of the specific parameters, methods, and experiments involved in the construction of a model that can successfully detect and classify objects in images.

2 Related Work

For a particular image in the dataset, pixels for training are predominately classified as background. Using any normal Convolutional Neural Network without taking the class imbalance into account will force the model to mainly predict background pixels. To mitigate the class infrequency, the following paper [2] suggested to use the following loss function: Focal Loss, Tversky Loss, Focal Tversky Loss, and Weighted Cross Entropy Loss in which we attempted to implement all of them. However, we discovered that our model frequently failed to perform as best as the baseline while implementing the listed loss functions except for the Weighted Cross Entropy. The Weighted cross entropy performs relatively well in our dataset because it assigns weights according to the frequency of the feature in the image. Meaning features that appear least over the entire dataset will get more weights. According to [3], this is an effective method because given a rare feature in an image, this loss function forces the network to learn the feature explicitly rather than randomly guessing the frequent features like the background by giving higher punishments/penalties when calculating the error. Hence, we decided to stick with the Weighted Cross Entropy to mitigate our imbalance dataset. Despite doing this, the IoU result only improves by a slight margin compared to the baseline performance. We then went back to take a look at data augmentation, [1], because we realized that our dataset is very small and thus our network was not learning as much. Initially, we only augmented our data once, which increased our dataset size by a bit, but the performance remains relatively the same. We then decided to perform out data augmentations 4 times to significantly increase our dataset, and the performance/IoU improved by roughly 13%.

We reference the seminal paper 'U-Net: Convolutional Networks for Biomedical Image Segmentation' [4] to reconstruct the UNet architecture, elaborated in Section 3.3. This serves as one of the contemporary models for model comparison.

3 Methods

3.1 Baseline Model

The input to the network is an input image with three channels for the RGB values. These are then passed to an encoder that consists of a series of convolutional layers (conv1 to conv5) with increasing numbers of filters (32, 64, 128, 256, 512), each followed by batch normalization (bnd1 to bnd5) and a ReLU activation function. After this the model consists of transposed convolutional layers (deconv1 to deconv5) with a decreasing numbers of filters (512, 256, 128, 64, 32) to increase the dimensions of the next inputs to the next layers, each followed by batch normalization (bn1 to bn5) and again a ReLU activation. The second decoder half of the layers serve to upsample the feature maps back to the original input size. The final convolutional layer (the classifier layer) with a kernel size of 1 maps the 32-channel feature map to the number of classes in the dataset. Subsequently, the output of this final convolutional layer is processed through a fully connected layer, producing predictions in a $Batch\_size \times N\_classes \times Height \times Width$ tensor.

The optimizer we used was the AdamW built in method from PyTorch, and the training loop incorporates batch normalization with early stopping on the validation accuracy. AdamW works by implementing weight decay or regularization only after controlling the parameter wise step size, and thus making the regularization term proportional to the weight itself instead of including it in the moving averages of the weights. This allows for the weights to tend less likely towards a larger scale, and therefore lead to a better generalizing model since smaller weights are preferred. Because of this, we chose AdamW over native Adam and SGD to hopefully increase generalization, even among a smaller dataset. The results are below in the results section.

3.2 Improvements Over Baseline

The baseline model has a relatively high pixel accuracy but low IoU value. Since the pixel labels are pretty imbalanced where *background* label is dominating, our model might just be predicting pixels to background to achieve the high pixel accuracy. To improve the baseline model performance, we tried to incorporate learning rate scheduler, data augmentation, and designing loss function.

Firstly, we discuss the details of **learning rate scheduler approach**. As instructed, we tried cosine annealing learning rate scheduler, which decreases the learning rate within a window, then reset the learning rate to the original value, and repeat. Mathematically, the learning rate is updated as follows:

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{T_{cur}}{T_{max}}\pi\right)\right), \quad T_{cur} \neq (2k+1)T_{max}; \tag{1}$$

$$\eta_{t+1} = \eta_t + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 - \cos\left(\frac{1}{T_{max}}\pi\right)\right), \quad T_{cur} = (2k+1)T_{max}, \tag{2}$$

where we set $\eta_{max} = 1e-2, \eta_{min} = 1e-5, T_{max} = 20$ in all our experiments. The optimizer we pass into the learning rate scheduler is described in the baseline section.

The cosine annealing learning rate scheduler helps us move towards minimum of loss function at a decreased speed, while making sure we are not stuck at the local minimum by suddenly resetting the learning rate to the original value after some period.

Since the size of our training dataset is pretty small, we tried to apply **data augmentation** techniques to expand the training dataset, in hope that we can obtain a better performance by exposing our model to more variations of the input images. We randomly crop 224×224 subimages from the original image. This helps us gain different crops of the input image and thus generating more training examples for our model. We set *antialias* to *True* to smooth out the jagged edges on curved lines and diagonals. Besides, we also applied random rotation with degrees between -180 degrees and 180 degrees. This helps us rotating the images in all possible angles and provides different orientations of the same objects in the training set. Finally, we flip images horizontally with the default probability of 0.5. Once a given image is flipped horizontally, the orientation of a given object is changed, thus providing more information for our models to recognize the object.

In all cases of data augmentation, we hope to achieve a better model performance by generating artificial training examples that would provide more information about our training dataset to the model, which helps the model learn decision boundaries better and thus having a better overall performance. With the above three different augmentation techniques, we are able to add three times more size of the original training dataset, which expands the training set by a factor of four.

Due to the nature of our dataset and because we are classifying each and every pixel, there is a huge class imbalance in the labels. Around 73.6% of the pixels in the training dataset are labeled as background pixels. Naturally, if our model were to classifying every single pixel as background, we would get a pretty high pixel accuracy. To combat the class imbalance and improve our model, we decided to **apply different weights** to each class and pass the resulting weights to the cross entropy loss, so we can still use a popular loss function for the multi-class classification task, while making the loss function customized to our imbalanced dataset. To obtain the weights for individual classes, we count the number of occurrences of each class in the training labels. For class category i, the weight is given by

$$\log\left(\frac{1}{\max\{c_0, c_1, \ldots, c_{20}\}} \frac{\sum_j c_j}{c_i}\right), \tag{3}$$

where c_i represents the pixel counts of class appearing in training set labels.

Firstly, we divide the sum of total counts by class count for each class category. By this computation, dominant classes would have lower weights. Then, we normalize the weight to be from 0 to 1 by dividing the result by the maximum of class counts. Finally, log is taken to scale the class weights. By doing so, classes with a smaller frequency would have a higher weight, leading to a higher penalty if misclassified. This forces our model to focus more on the less frequent classes and solves the problem of class imbalance.

Note that for model improvements, the later techniques build on previous ones. The sequence of techniques we tried is learning rate scheduler, data augmentation, and finally loss function redesign. This means while experimenting with data augmentation, we have learning rate scheduler turned on. Also, when we tried the redesign of loss function, we apply both the learning rate scheduler and data augmentation techniques.

3.3 Experimentation Methods

(5a)

Recall that the baseline architecture consisted of the last layer of the network being a Conv2d layer, with a standard Cross Entropy Loss using the built in PyTorch method. The weight initialization was kept as the Xavier method. Additionally we kept the AdamW optimizer method standard as well. With this in mind the first initial architecture involving dropout was similar to the baseline, however we expand on this in the next two paragraphs to highligh significant changes and the incentives behind them.

The first initial architecture change that we attempted was implementing dropout. This involved randomly turning off nodes at each layer with a uniform probability of .5 with the hopes of increasing generalization and reducing overfitting in the baseline model. Overfitting was not a huge problem to begin with considering the baseline validation accuracy was upper bounded at 73 percent and the training accuarcy was upper bounded at 76 percent. After implementing dropout in between the encoder and decoder, the performance improved by .1 percent on the validation set from 72.88 to 72.89 percent.

Table 1: Baseline w/ Dropout Architecture

Layer	**In-Channels**	**Out-Channels**	**Stride**	**Kernel Size**	**Padding**	**Activation**
Conv1	3	64	2	3	1	ReLU
Conv2	64	128	2	3	1	ReLU
Conv3	128	256	2	3	1	ReLU
Conv4	256	512	2	3	1	ReLU
Conv5	512	512	2	3	1	ReLU
Conv6	512	512	2	3	1	ReLU
Conv7	512	512	2	3	1	ReLU
Deconv1	512	512	2	3	1	ReLU
Deconv2	512	512	2	3	1	ReLU
Deconv3	512	512	2	3	1	ReLU
Deconv4	512	256	2	3	1	ReLU
Deconv5	256	128	2	3	1	ReLU
Deconv6	128	64	2	3	1	ReLU
Deconv7	64	n_class	2	3	1	-

The second architecture kept the dropout implementation from the previous architecture. Instead, we have reduced the probability to 0.3, as we are going to add dropout after each convolutional layer. Additionally, to minimize overfitting, we have also added two maxpooling layers to add some regularization and mitigate any variations on the input dataset that could significantly impact the performance of our network despite sharing similar features and input values. Regarding the activation functions, we changed Relu to LeakyRelu for learning purposes during back propagation when values to activation function is negative, as the regular Relu would just otherwise skip over a particular perceptron as the gradient is 0.

Table 2: Baseline w/ Dropout Architecture and Maxpooling + Leaky Relu

Layer	In-Channels	Out-Channels	Stride	Kernel Size	Padding	Dilation	Activation
Conv1	3	32	2	3	1	1	LeakyReLU
Maxpooling	32	32	1	2	1	1	LeakyReLU
Conv2	32	64	1	3	1	1	LeakyReLU
Maxpooling	64	64	1	2	1	1	LeakyReLU
Conv2	32	64	1	3	1	1	LeakyReLU
Conv3	64	128	1	3	1	1	LeakyReLU
Conv4	128	256	1	3	1	1	LeakyReLU
Conv5	256	512	1	3	1	1	LeakyReLU
Deconv1	512	512	1	3	1	1	LeakyReLU
Deconv2	512	256	1	3	1	1	LeakyReLU
Deconv3	256	128	1	3	1	1	LeakyReLU
Deconv4	128	64	1	3	1	1	LeakyReLU
Deconv5	64	32	1	3	1	1	-

Transfer Learning with FCN ResNet101 (5b)

As our training dataset is super small, it is a good idea to carry out transfer learning, where we leverage the power of another deep model that is pretrained on a similar task. In our case, we selected Fully Convolutional Network with a ResNet-101 (FCN ResNet-101) backbone, which is pretrained on the COCO dataset that has the same number of class categories as VOC 2007. To address class imbalance issues, we applied the all three techniques discussed in section 3.2, namely cosine annealing learning rate scheduler, data augmentation, and cross entropy loss function with designed class weights. We continued to use Xavier weight initialization and AdamW optimizer to maintain consistency so that we can effectively compare the experimentation results with our baseline model.

Table 3: FCN ResNet-101 Architecture

Layer (type (var_name))	**Input Shape**	**Output Shape**	**Kernel Size**	**Activation**
Conv2d	[16, 3, 224, 224]	[16, 64, 112, 112]	3	N/A
BatchNorm2d	[16, 64, 112, 112]	[16, 64, 112, 112]	N/A	ReLU
Maxpool2d	[16, 64, 112, 112]	[16, 64, 56, 56]	2	N/A
(layer1) Bottleneck (0)	[16,64,56,56]	[16, 256, 56, 56]	3	ReLU
Bottleneck (1)	[16,256,56,56]	[16, 256, 56, 56]	3	ReLU
Bottleneck (2)	[16,256,56,56]	[16, 256, 56, 56]	3	ReLU
(layer2) Bottleneck (0)	[16,256,56,56]	[16, 512, 28, 28]	3	ReLU
Bottleneck (1)	[16,512,28,28]	[16, 512, 28, 28]	3	ReLU
Bottleneck (2)	[16,512,28,28]	[16, 512, 28, 28]	3	ReLU
Bottleneck (3)	[16,512,28,28]	[16, 512, 28, 28]	3	ReLU
(layer 3) Bottleneck (0)	[16,512,28,28]	[16, 1024, 28, 28]	3	ReLU
Bottleneck (1)	[16,1024,28,28]	[16, 1024, 28, 28]	3	ReLU
...	...	...	...	...
Bottleneck (21)	[16,1024,28,28]	[16, 1024, 28, 28]	3	ReLU
Bottleneck (22)	[16,1024,28,28]	[16, 1024, 28, 28]	3	ReLU
(layer 4) Bottleneck (0)	[16,1024,28,28]	[16, 2048, 28, 28]	3	ReLU
Bottleneck (1)	[16,2048,28,28]	[16, 2048, 28, 28]	3	ReLU
Bottleneck (2)	[16,2048,28,28]	[16, 2048, 28, 28]	3	ReLU
Conv2d	[16, 2048, 28, 28]	[16, 512, 28, 28]	3	N/A
BatchNorm2d	[16, 512, 28, 28]	[16, 512, 28, 28]	N/A	ReLU
Dropout	[16, 512, 28, 28]	[16, 512, 28, 28]	3	N/A
Conv2d	[16, 512, 28, 28]	[16, 21, 28, 28]	3	N/A

UNET (5c)

We also tried out the UNet architecture. As described in the UNet paper[5], in the downsampling part of the network, each convolution layer (without padding) is followed by a rectified linear unit (ReLU). To improve the model performance, we add a batch normalization after each convolutional layer but before the ReLU activation. After each two full operation of convolutional layers, which consists of a part of a convolutional block, we apply a 2×2 max pooling with stride 2. We repeat the process 5 times, where in the last time we repeat the unsampling block, we do not apply max pooling. Then, we apply upsampling with skip connections from downsampling layers with crop following similar procedure. At the end, we apply a 1×1 convolution to map the feature vector to the vector with the size of classes.

The architecture is summarized as follows.

Table 4: UNet Architecture

Layer	In-Channels	Out-Channels	Stride	Kernel Size	Padding	Activation
Conv2d(e11)	3	64	0	3	1	ReLU
Conv2d(e12)	64	64	0	3	1	ReLU
Maxpool1	64	64	2	2	0	-
Conv2d(e21)	64	128	0	3	1	ReLU
Conv2d(e22)	128	128	0	3	1	ReLU
Maxpool2	128	128	2	2	0	-
...	...	...	...	...	...	...
Conv2d(e51)	512	1024	0	3	1	ReLU
Conv2d(e52)	1024	1024	0	3	1	ReLU
upconv1	1024	512	2	2	0	-
Conv2d	1024 (concatenate with e42)	512	2	2	0	ReLU
Conv2d	512	512	2	2	0	ReLU
upconv2	512	256	2	2	0	-
Conv2d	512 (concatenate with e32)	256	2	2	0	ReLU
Conv2d	256	256	2	2	0	ReLU
...	...	...	...	...	...	...
outconv	64	21	1	1	0	-

4 Results

In this section, we present the loss against number of epochs for each individual model we have experimented. The plots follows with captions to denote different experiments.

Figure 1: Baseline Model Plot

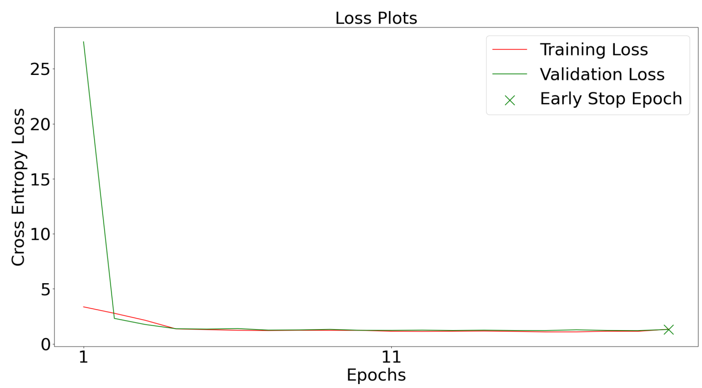


Figure 2: Baseline Model with Learning Rate Scheduler Plot

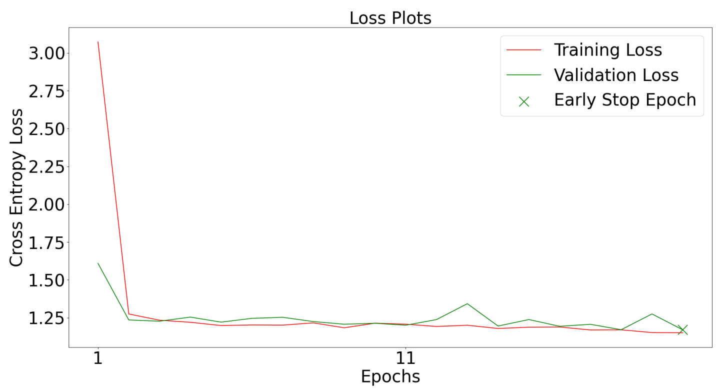


Figure 3: Baseline Model with Data Augmentation Plot

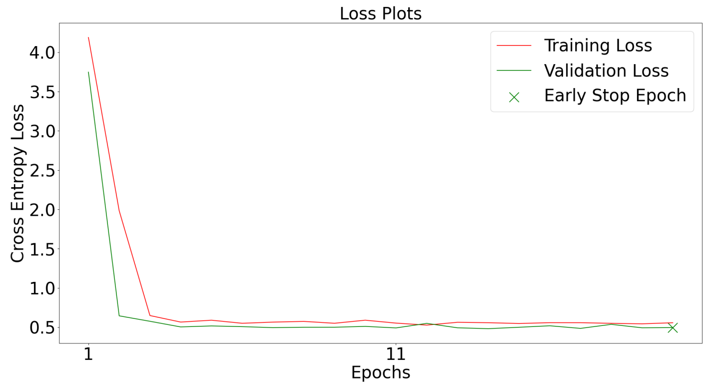


Figure 4: Designed Loss Plot

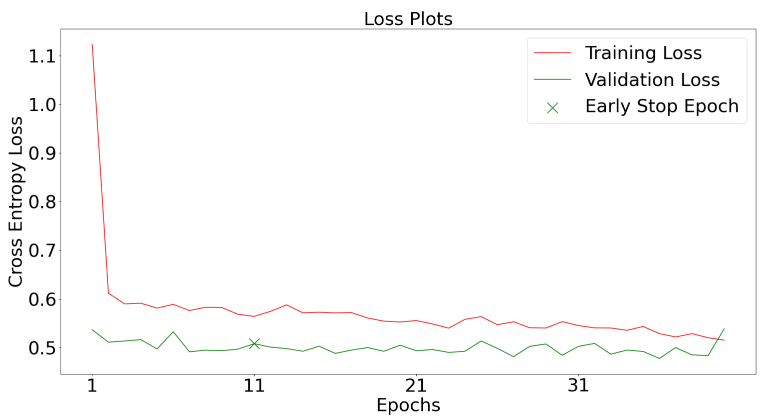


Figure 5: Baseline with Drop-Out

Figure 6: FCN ResNet101 Model Plot

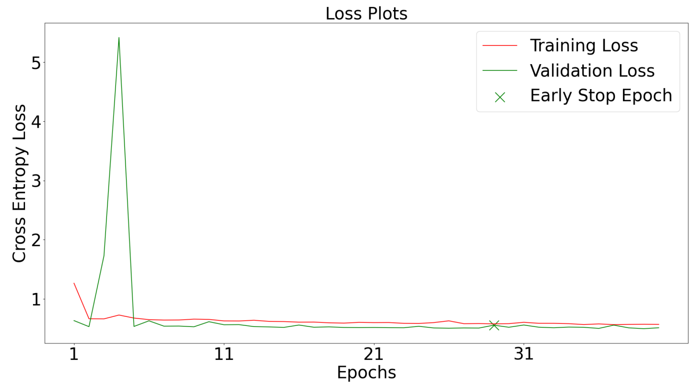


Figure 7: UNet Model Plot

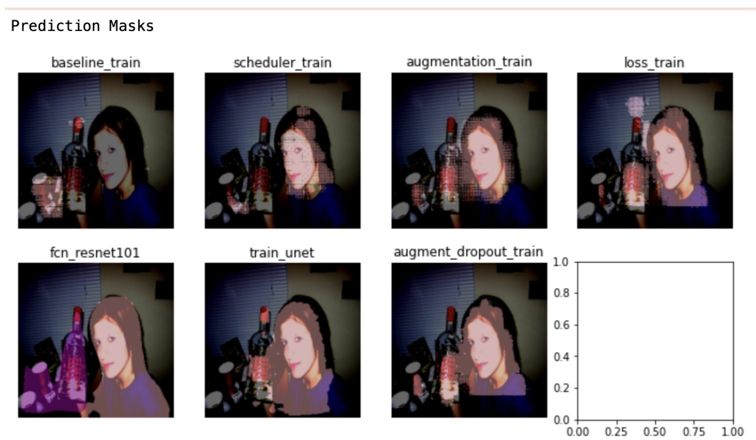


Figure 8: Visualization Plot

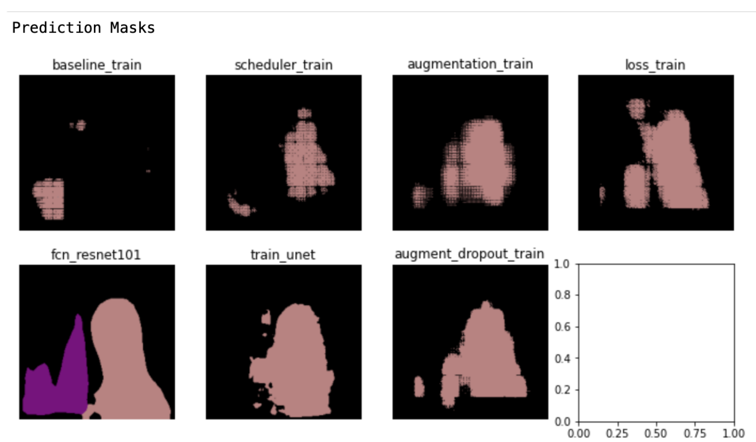


Figure 9: Visualization Plot

Our models' results are given by the table below:

Table 5: Models and Their Performance

Model Name	Validation Accuracy	Validation IoU
Baseline (3)	0.734628	0.0566138
Baseline w/ Dropout (5a)	0.728932	0.0542647
Baseline w/ Dropout & Data Augmentation (5a)	0.7307808	0.0643877
Baseline w/ Scheduler (4a)	0.745939	0.0560166
Baseline w/ Data Augmentation (4b)	0.749721	0.0686395
Baseline w/ Loss Function (4c)	0.750202	0.0679313
FCN ResNet-101	0.873404	0.330007
UNet	0.754065	0.0705095

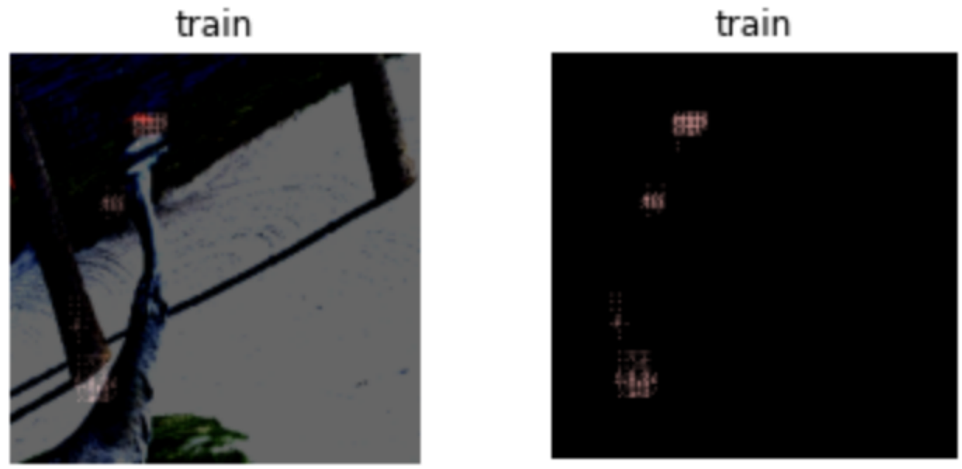
Note that since pixel accuracy might not be a good evaluation for our problem, we choose to prioritize IoU measure. For the entire training statistics, we show the best IoU value in the above table, with the corresponding pixel accuracy in the same epoch.

5 Discussion

5.1 Q3

An important thing to note about evaluation is that IoU is the preferred evaluation metric for this dataset. Because of the class imbalance nature of the data, accuracy is less reliable of a metric compared to IoU.

The baseline model achieved an accuracy of 73.46% with a standard deviation of 5.66%. The drawback of the baseline model is straightforward. First, we only have limited training data before applying data augmentation to the baseline model in Q4. There are only 209 images for the training, and most of the pixels (30182641 pixels) are labeled as background in the first batch, and 2769488 pixels are labeled as a person in the first batch, compared to only 188751 pixels as birds, and 152089 labeled as a potted plant. As we use the regular entropy loss function, the baseline model will be pushed to learn to identify most pixels as background and human, and on many occasions, we notice that the baseline model can identify the significant object in the image but falsely label them as persons. For instance, consider an image of a bird. Although the baseline model can accurately identify the main object and its background, it struggles to distinguish between a bird and a human.



(a) Accuracy Against Epoch (b) Loss Against Epoch

Figure 10: Softmax Regression Experiment Plots

Table 6: FCN ResNet-101 Architecture

Layer (type (var_name))	Input Shape	Output Shape	Kernel Size	Activation
Conv2d	[16, 3, 224, 224]	[16, 64, 112, 112]	3	N/A
BatchNorm2d	[16, 64, 112, 112]	[16, 64, 112, 112]	N/A	ReLU
(layer1) Bottleneck (0)	[16,64,56,56]	[16, 256, 56, 56]	3	ReLU
...	...	...	...	...
ConvTranspose2d (deconv1)	[16, 512, 7, 7]	[16, 512, 7, 7]	3	N/A
BatchNorm2d	[16, 512, 7, 7]	[16, 512, 7, 7]	N/A	ReLU
ConvTranspose2d (deconv2)	[16, 512, 7, 7]	[16, 256, 14, 14]	3	N/A
BatchNorm2d	[16, 256, 14, 14]	[16, 256, 14, 14]	N/A	ReLU
...	...	...	...	...
Conv2d (classifier)	[16, 32, 224, 224]	[16, 21, 224, 224]	3	N/A

Hence, class underrepresentation is not only limited to the relation between background class and other classes, but the underrepresentation between each non-background still needs to be improved.

In light of the baseline model's architecture, the network's encoder part uses the three-by-three convolutional filter to reduce the spatial resolution of the input feature maps. This downsampling operation helps extract high-level features and reduce the computational load. When applying transpose convolution as upsampling to restore spatial, without proper cropping, the upsampling process may not fully retain the spatial information from the feature information decoded in the downsampling process, resulting in our prediction not aligning with the original spatial locations of features. This misalignment can lead to a loss of precise spatial information, affecting the accurate localization of objects, which is especially evident in our model as we need to transpose a seven-by-seven feature map to a 224 by 224 prediction.

5.2 Q4

Comparing two architectures, one employing a scheduler on top of AdamW optimization and Xavier weight initialization, and the other omitting the scheduler, several observations emerge. The use of a scheduler, which introduces additional decay, does not lead to an acceleration of the learning process within the same number of epochs. In fact, it results in a longer training duration compared to the baseline without a scheduler. Despite this extended training time, the performance of both models ultimately converges to similar levels, just one later than the other in the same measure of timesteps. Specifically, the baseline with the scheduler achieves an accuracy of 0.745939 and IoU of .05601 compared to the baseline without the scheduler (0.734628 0.0566138), indicating that the added complexity of the scheduler does not significantly enhance final model performance, considering the trade-off in training efficiency and speed. In conclusion, these methods still outperformed the baseline model, and were the closest models to the UNet in terms of Pixel Validation accuracy, so they were still worth training and evaluating.

For data augmentation, cropping, rotation and horizontal flip all provided more detailed information about each object for our baseline model. By changing the orientations of individual objects in a single image, we artificially generate three times of the original size more training examples for our model to train on. Just like any machine learning and deep learning tasks, more training examples helps the model better understand the task and learn better as a result. Compared with the original baseline model, data augmentation helps us get to roughly 0.686 average IoU and 0.749721 pixel accuracy value. We see much improvements compared with the baseline model as our model is able to learn much more from "more information" in the training dataset.

For weighted cross entropy loss function, we also see a slight improvment over the baseline model. Due to the weighted loss, our model penalizes the loss function more when we make wrong predictions for infrequent labels, whereas it contributes less penalty to wrong predictions for majority classes, especially the background. This places more emphasis on the less frequent classes and helps us overcome the class imbalance issue.

5.3 Q5

For experimentation and development on the baseline model (question 5), we tried many different architectures. The first method was only implementing dropout on top of the baseline model, which as stated in 3.3, only yielded a .1 percent improvement on the validation accuracy. IoU actually decreased from .056 to .054 with dropout, suggesting that the benefit in accuracy was not worth the loss in IoU. Specifically, dropout was implemented between the encoder and decoder, and not between every single layer to save computational resources. The reason for trying this architecture method was to reduce overfitting in the baseline and increase generalization, however dropout alone did realize this hypothesis. As we will see in other architectures, combining dropout with other methods such as augmentation and the UNet yielded better results, suggesting dropout alone was not enough to warrant an improvement in IoU or Accuracy.

The second architecture we used incorporated two max pooling layers. We intentionally did this, as we figured that applying max pooling layers will help significantly reduce the impact of data augmentations, which will in turn improve the generalization of the network on new dataset. Additionally, maxpooling puts great emphasis on regularizing features in our dataset and prevented out network from overfitting. We also decided to implement LeakyRelu instead of regular Relu for our activation function. We chose to do this because we want to enable some learning in the activation while doing back propagation when the value inserted into the activation function is negative. Finally, we added dropout at every single layer in our network. Despite already having maxpooling to help with preventing overfitting, by implementing dropout will make our network more robust with missing features, as some features will be set to 0 add a certain layer. As the network can't learn full features at all time will allow our network to become less reliance on perfect inputs for prediction and to be more confident at predicting some of the features relevant to in the inputs when our dataset become noisy. Despite our second architecture seems to work logically, datahub just wouldn't run and we keep getting gpu reached limits. Hence, we can't provide any data/ results for this architecture.

Next, we applied transfer learning and see how much impact small dataset has on our model performance, as pretrained models on similar tasks would usually generalize well on other similar tasks with small training dataset. Here, we applied Fully Convolutional Network with backbone of ResNet-101. The model is pretrained on the COCO dataset, which also has 21 classes, yet it has much more training data. Before training, we hoped that the transfer learning model would perform much better by leveraing more training data from similar task. After training, we see that both pixel accuracy and average IoU are improved by a large margin compared with the baseline model – roughly 0.873 and 0.33 respectively. The main reason would be that COCO dataset is also used for image classification task. Therefore, even we freeze all layers besides the classifier layer, the intermediate layers are still relevant for the task and can compute much useful information for the final output layer to make relatively accurate predictions. Plus, by leveraging data augmentation and weighted cross entropy loss techniques, our training data could perform much better compared with the na"ive baseline model, thus helping the classifier layer to learn just enough to make reasonably predictions for our semantic segmentation task.

For the final part, we leverage the structure of UNet architecture, which consists of a series of encoding blocks, followed by a series of decoder blocks. Let's first see the architecture of the UNet from the paper[4] as follows:

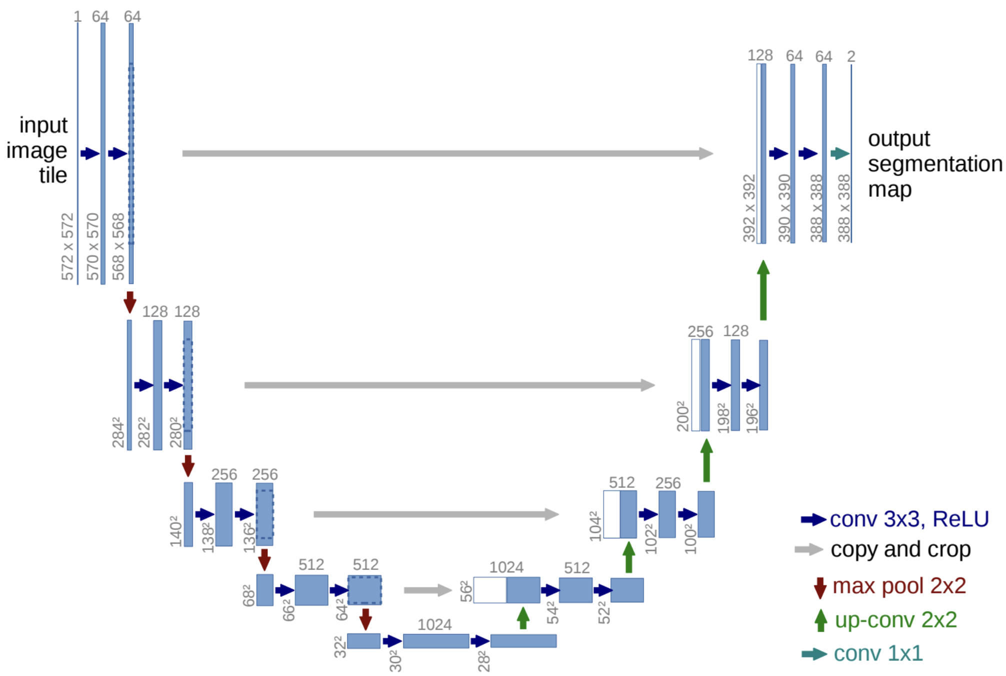


Figure 11: UNet Architecture

From the experimental results, we can see that UNet architecture gives us better performances in both the pixel accuracy and the average IoU result. One of the reasons could be the depth of the architecture. Since now we have more repetition of convolutional blocks of the structure *conv2d→batchNorm→ReLU*, the model learns, layer by layer, more information about the training dataset through nonlinearity. Batch normalization also helps much with our model performance more. Due to the small size of training data, it might be hard for our model to decode the information and make relatively accurate predictions. By the inherent structure of UNet architecture, we are able to retain more information from earlier layers of the network thanks to the skip connection from encoder blocks to the decoder blocks. However, UNet has much parameters to learn and the size of our training data is still small even after data augmentation, we are not able to learn as much as the FCN ResNet-101 model due to the inherent problem we face when solving for the task. Also, our loss function might not be good enough to solve the imbalanced dataset issue, thus skip connections might pass more noise to the deeper layers, thus confusing our model to make wrong predictions, especially for dominating classes – this might be the reason why our IoU does not improve much.

6 Team Contributions

6.1 Nathaniel del Rosario

I worked on implementing new architectures to improve upon the baseline FCN model. This consisted of writing new code and running the training procedure on the new design as well as noting its performance. The new models I tested were dropout, dropout ensembled with input image transformation, as well as a very deep CNN with dropout. Additionally, I helped create the outline for the report, wrote part of the abstract and the introduction paragraphs, wrote the Models and Performance Table, part 5a), wrote and formatted some of the tables, wrote about the design choices behind AdamW, Xavier Weight Initialization, Batch Normalization, and Dropout, and summarized the baseline model in 3.1 as well as included the tables for problems 3 and 4a. I also wrote the discussion for Q4 and Q5.

6.2 Hargen Zheng

Our initial data augmentation method is not working properly to generate four times the original size of training data. Therefore, I modified our approach to perform data augmentation and it worked. Besides, I worked on designing heuristics to find weights for each class, so we can alleviate the problem of super imbalanced dataset. In addition to work on improving the baseline model, I also worked on the experiments with FCN ResNet-101 model transfer learning and UNet model.

6.3 Chuong Nguyen

I was very interested in expanding the convolutional neural network to improve the performance of our network. I was originally going for 10 layers in the encoder and 10 layers in the decoder, alongside max pooling, and dropout at each layer. However, it was very consuming regarding resources and memory. Then, I tried working on just adding the dropout on every layers, max pooling on two layers, and used leakyrelu for the activation function. I have included some of the related works and how extensive readings inspired some of our implementation for this network. Regarding experimentation, I tried looking and implementing different loss functions to counter imbalance dataset and discovered the ineffectiveness of these loss functions on our network.

6.4 Ziyue Liu

I finished the baseline model and implemented the baseline training and baseline model with the scheduler and used them as a template for the training procedure of Q4 and Q5. I also implemented iou and pixel_accuracy functions for displaying training and validation accuracy and plot and table_creating functions in the util file and implemented a visualization file for displaying pixel labeling.

6.5 Adam Tran

I worked on improving the baseline model, mainly on problems 4b and 4c of the assignment. This included writing code to augment the training data with transformed images. I also improved the existing base transformation code. Secondly, to deal with the class imbalance, I calculated and coded in weights for the Cross Entropy Loss Function. In addition to working on problems 4b and 4c, I wrote the sections for these problems in the write up and helped with writing other miscellaneous parts of the write up, such as proofreading and fixing errors.

References

[1] Luis Perez and Jason Wang. The effectiveness of data augmentation in image classification using deep learning, 2017.

[2] Trong Huy Phan and Kazuma Yamamoto. Resolving class imbalance in object detection with weighted cross entropy losses. *arXiv preprint arXiv:2006.01413*, 2020.

[3] Mohammad Reza Rezaei-Dastjerdehei, Amirmohammad Mijani, and Emad Fatemizadeh. Addressing imbalance in multi-label classification using weighted cross entropy loss function. In *2020 27th National and 5th International Iranian Conference on Biomedical Engineering (ICBME)*, pages 333–338, 2020.

[4] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.

[5] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18*, pages 234–241. Springer, 2015.