Convolutional Network Implementation for Semantic Segmentation using Pytorch

Andrew Ghafari
Department of Computer Science
University of California San Diego
aghafari@ucsd.edu

Jay Jhaveri
Department of Computer Science
University of California San Diego
jjhaveri@ucsd.edu

Abstract

In this third assignment, we were given the PASCAL VOC-2007 dataset, and were tasked to perform pixel-level segmentation. This dataset is a very popular one in the computer vision spectrum, and was created as part of the annual Visual Object Classes Challenge, and served as the dataset of that competition from 2005 to 2012. The original dataset contains more than 9000 images, that are annotated with object bounding boxes for 20 different categories such as car, cat, dog, boat and many more. We haven't been given the whole dataset but only roughly 600 images split evenly in thirds between training, validation and testing. Before giving a rundown of the assignment given and how we approached the task at hand, let us first talk about the evaluation metrics. We had 2 evaluation metrics, pixel accuracy and intersection over union (IoU). The assignment went as follows: we first started by implementing a baseline model that had briefly the following architecture: 5 convolution layers, and 5 deconvolution layers each followed by a batch normalization. This gave us a loss of 1.363, a pixel accuracy of 72.08% and an IoU of 0.055, which were close to expectations. Then we started improving it by doing multiple tricks. The first was using a cosine annealing learning rate scheduler. This lowered our overall pixel acc to 71.58%, increased our loss to 1.34 and our average IoU to 0.0613. After that, we implemented data augmentation techniques mainly mirror flips, rotating the images and cropping them. This gave us better loss function, by decreasing it to 1.316, better IoU, increasing it to 0.0639 but lower pixel acc, 70.98%. Finally we used a weighted loss, which was specifically done to target the class imbalance problem that we were facing. This approach gave us a 2.23 loss function value, a better IoU of almost 0.068 and a lower pixel accuracy of 69.1%. The last problem was basically a way for us to try different architectures and apply transfer learning. We first came up with a personal network, and we did that by making significant changes to the network we had previously. This approach gave us a worse loss of almost 4.4, but better IoU, reaching now 0.0765 and lower pixel acc of 69%. Next thing is that we applied transfer learning to ResNet18 model. We had to do some changes to make it appropriate to the task we have, and these changes were removing the last two fully connected layers, locking the resnet's weights and adding instead a 4 layer deconvolution network each with a ReLU activation function and batch normalization. This approach also increase dour IoU significantly, making it go to 15%, a good pixel acc of 74.4% and a good loss function as well of 1.8. Finally, we implemented the U-Net architecture that was mentioned in [Paper name]. The architecture of U-Net model was this. This final scheme gave us a loss function of 2.2, an IoU of 0.087 and a pixel acc of 67.747%.

1 Introduction

As we have mentioned in the abstract, we were given the PASCAL VOC-2007 dataset which was designed to provide a benchmark for object detection and recognition algorithms, and it has been used in a wide range of research studies and competitions. It contains more than 9,000 images, each of which has been annotated with object bounding boxes and class labels for 20 different object categories. We have been given 632 images out of those 9000, and they were split in almost equal fashion, with 209 of them being training images, 213 constituting the validation set and the test set having 210 images. This partition was particularly odd for us given that we always see was more training samples than validation or testing ones, and also the overall number was kind of low, which may have had an influence over the performance of the model. Let us now introduce the evaluation metrics used for this assignment, which are pixel accuracy and intersection over union. The first just shows the percent of correct predictions which is simply the total number of correct predictions (where the predicted pixel and the target pixel match), over the total number of samples. With this evaluation metric, a main issue arises, and that is class imbalance. If our classes are highly imbalanced, it implies that one or more classes are dominant in an image, while some other classes form a minor part of the image. As a result, the network may significantly reduce its error by simply labeling everything with the majority class. That is what was happening to us, with the network getting away with a better 'loss' by just predicting everything as background (label 0) whereas the image had multiple labels. Since class imbalance is a common occurrence in many real-world datasets, it cannot be overlooked. Therefore, alternative metrics that are more effective at handling this problem are necessary. One of them being IoU, which is a commonly used evaluation metric for semantic segmentation tasks in computer vision. It is a measure of the similarity between the predicted segmentation mask and the ground truth mask. The IoU metric is calculated by taking the intersection of the predicted mask and the ground truth mask, and dividing it by the union of the two masks. Specifically, IoU is defined as the area of overlap between the predicted and ground truth masks, divided by the total area comprised by the two masks. In other words, IoU measures the percentage of overlap between the predicted mask and the ground truth mask, and ranges from 0 to 1. A value of 1 indicates a perfect match between the predicted and ground truth masks, while a value of 0 indicates no overlap at all. Throughout this paper, we will be providing the three metrics that we used, loss, pixel accuracy and IoU. We just want to point out that a decrease in pixel accuracy doesn't necessarily mean a 'worse' model, since the class imbalance used to drive the network to only predict 0s and get away with a better loss and higher pixel accuracy.

We approached the task by doing the following. We started with a vanilla baseline implementation that had the following architecture: 5 convolution layers, and 5 deconvolution layers each followed by a batch normalization. This gave us a loss of 1.363, a pixel accuracy of 72.08% and an IoU of 0.055. In addition to that, the weights were initialized using the Xavier weight initialization technique, which is a very popular one in deep learning, since it has been proven to improve the convergence speed and the overall performance of the network. Mathematically, the weight initialization sets the initial weights of each neuron in a layer to be drawn from a Gaussian distribution with a mean of 0 and a variance of 1/n, where n is the number of inputs to the layer. This means that each neuron in the layer will receive a weight that is scaled according to the number of inputs it receives. In other words, neurons with more inputs will have smaller weights, while neurons with fewer inputs will have larger weights. We also used batch normalization, which is a way to normalize the inputs to each layer of a network, mainly to overcome the problem of internal covariant shift, which ultimately makes the model more stable and helps yield better results. After that we resorted to different techniques to try and improve on the vanilla implementation that we had. We did so by first implementing the cosine annealing learning rate scheduler. The idea behind the cosine annealing learning rate scheduler is to gradually decrease the learning rate over time in a cyclical pattern. This is done by applying a cosine function to the learning rate, which starts at a maximum value and gradually decreases to a minimum value, before increasing back to the maximum value. The benefits of the cosine annealing learning rate scheduler are two-fold. First, it can help the model converge more quickly by starting with a high learning rate that enables it to make large weight updates early on. Second, it can help the model avoid getting stuck in local optima by periodically increasing the learning rate, which can help it explore different parts of the weight space. With the help of it, we were able to achieve better IoU results. Next we augmented the dataset by performing multiple transformations upon it, including mirror flipping, rotating them by an arbitrary angle and also cropping the original images, up until we had a new dataset of roughly 3x the size of the original data we had. We also made the same transformation to the labels of these images to keep everything in sync. Lastly, we changed our loss function to incite the model to predict something other than the dominant dataset, so we created a weighted loss function that has weights that are disproportional to the frequency of the label in the dataset, i.e. weighing infrequent classes more. In the last problem we had a fun exploration session. We first created our own model, and we did that by changing on the original architecture of question 4-c. We added a few layers, and changed kernel sizes and activation function, which got us better IoU. We will discuss the new architecture more in depth in the following appropriate sections. After that we took the ResNet18 model and applied transfer learning techniques on it to make it suitable for this problem. We removed the last two layers of it and replaced them with deconv layers to learn localization along with classification and also changed to number of final output channels to fit the number of classes we had. Finally we implemented the U-Net architecture that is found in [1].

2 Related work

We first would like to thank Prof Cottrell for the slides posted as we heavily relied on them while doing this programming assignment. More specifically, lecture 5 and 6 from the Lecture Notes Winter 2023 on Piazza. In addition to that, we would also like the thank the TAs for the help they provided whether on piazza or in person during office hours.

We would also like to mention some of the articles and blogs that we read to get a better understanding of some of the concepts: The first one is the U-Net paper Ronneberger et al [1]. This paper was crucial for us to understand the U-Net architecture, to be able to recreate it. It is a very interesting paper and has a very detailed description of the model they are proposing, which came in handy when recreating it. Next, we want to mention some medium articles that explained important concepts throughout this PA, especially in the improvements parts for the baseline model. Shirvastava walked us through dealing with class imbalance by introducing a weighted loss function.[2]. Same thing also happened with data augmentation with Ray [3], where he walked us through the concepts and benefits of data augmentation. Last paper we would like to mention the ResNet18 documentation in Pytorch, which was really helpful when implementing transfer learning on this architecture [4].

3 Methods

3.1 Baseline

Our baseline model's architecture has multiple convolution layers. The first one has 3 input channels (one for each R,G,B) and 32 output channels, with a kernel size of 3x3, a stride of 2, padding of 1 and dilation of 1.

Each one of these convolution layers that I will discuss next, is followed by a proper ReLU activation function even the deconv ones and then an appropriate batch. The layers that follow all have the same pattern with different input and output dimensions. Conv2 has 32 inputs (equivalent to Conv1's output dimension), and 64 output dimensions. Conv3 has 64 inputs and 128 output dimensions. Conv4 has 128 input channels, and 256 output ones. Lastly Conv5 has 256 input channels and 512 output ones. That is the encoder part of the network. This gets fed to a decoder part, were we have 5 deconvolution layers starting with a 512x512 then a 512x256, then a 256x128, then a 128x64, and a 64x32 and finally a 32x21 (which is the classifier convolution layer, with 21 being the number of classes that we have to predict). The activation function of the last layer is a softmax since we are performing a multi class classification task, but after watching the recommended videos to get started on pytorch, we discovered that softmax is automatically implemented when doing multi class cross entropy. Our loss function is indeed that (Multi-class cross entropy). We used also Xavier weight initialization, which we have discussed the properties and benefits before hand in the introduction section. We also tried the two proposed adaptive learning rate techniques (gradient optimizer) Adam and AdamW with the latter giving the better performance so it was our technique of choice.

3.2 Improvements over baseline

After completing the baseline model and checking its performance metrics, we decided to perform multiple improvements on it to get a better overall model. The improvements we did can be divided into three main parts. The first thing we did is add a cosine annealing learning rate scheduler. It is a common technique used in deep learning to adjust the value of the learning rate when training a neural network. It reduced the learning rate by a cyclical manner following a cosine curve, hence the name. It usually helps the model converge faster to the optimal solution by skipping some local minimums.

The second thing we did is some data augmentation techniques. Given that the training set consisted of only around 200 images which is quite low, and may have been a factor in our model poor performance. We decided to augment the size of it by 3 times, making the new total training set 3 times the size of the original one. We did that by using mirror flipping, rotating some images by arbitrary degrees and cropping some other images.

The last thing we did was a change to the loss function that we had previously and turned it into a weighted one, where the weights on the class is inversely proportional to its frequency, which helps in solving the imbalanced dataset that we have and forces the model to predict things other than the most dominant class (‘0’ background).

All these improvements definitely helped better the performance of the model.

3.3 Experimentation

Table 1: Custom Network (Q. 5_a)

Layer of Network		Layer Properties									
Layer No.	Type of layer	In-Channel	Out-Channel	Output Img Dims	Kernel Size	Padding	Stride	Dilation	Outpadding	Activation	Batch Normalization
1	Convolution	3	32	108	11x11	1	2	1	-	ReLU	Yes
2	Convolution	32	64	54	3x3	1	2	1	-	ReLU	Yes
3	Max Pool	64	64	27	2x2	-	1	-	-	-	-
4	Convolution	64	128	14	3x3	1	2	1	-	ReLU	Yes
5	Convolution	128	256	7	3x3	1	2	1	-	ReLU	Yes
6	Max Pool	256	256	4	2x2	-	1	-	-	-	-
7	Convolution	256	512	2	3x3	1	2	1	-	ReLU	Yes
8	Convolution	512	1024	1	3x3	1	2	1	-	ReLU	Yes
9	Deconvolution	1024	512	2	3x3	1	2	1	1	ReLU	Yes
10	Deconvolution	512	256	3	3x3	1	2	1	0	ReLU	Yes
11	UnPooling	256	256	7	2x2	-	2	-	-	-	-
12	Deconvolution	256	128	14	3x3	1	2	1	1	ReLU	Yes
13	Deconvolution	128	64	27	7x7	1	2	1	0	ReLU	Yes
14	UnPooling	64	64	54	2x2	-	2	-	-	-	-
15	Deconvolution	64	32	108	3z3	1	2	1	1	ReLU	Yes
16	Deconvolution	32	32	244	12x12	1	2	1	0	ReLU	Yes
17	Convolution	32	21	244	1x1	-	-	-	-	-	No

Table 2: Resnet18 Transfer Learning (Q. 5_b)

Layer of Network		Layer Properties							
Layer No.	**Type of layer**	**In-Channel**	**Out-Channel**	**Kernel Size**	**Padding**	**Stride**	**Dilation**	**Activation**	**Batch Normalization**
1	Resnet18 without the FC and Avg pool layers	3	512	-	-	-	-	-	-
2	Deconvolution	512	512	3x3	1	2	1	ReLU	Yes
3	Deconvolution	512	256	3x3	1	2	1	ReLU	Yes
4	Deconvolution	256	128	3x3	1	2	1	ReLU	Yes
5	Deconvolution	128	64	3x3	1	2	1	ReLU	Yes
6	Deconvolution	64	32	3x3	1	2	1	ReLU	Yes
7	Convolution	32	21	1x1	-	-	-	-	No

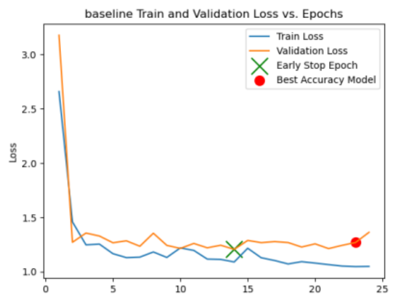
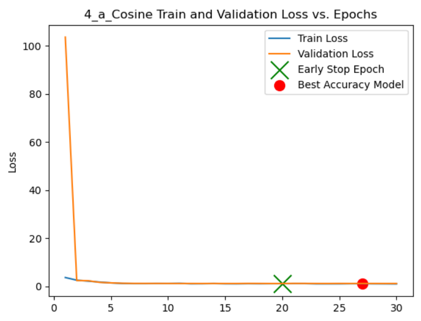
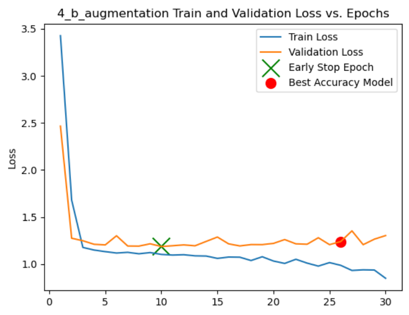
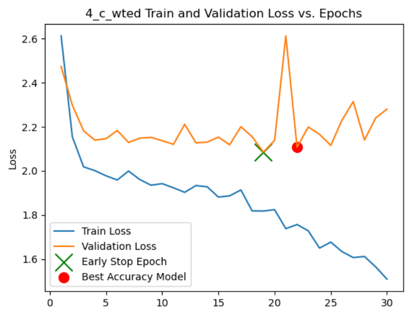
Table 3: U-Net Architecture (Q. 5_c)

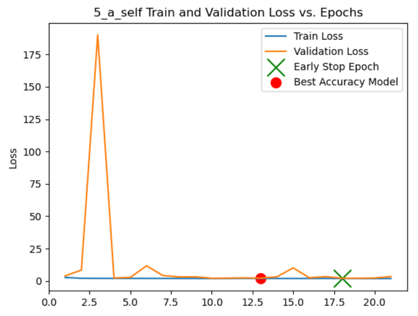
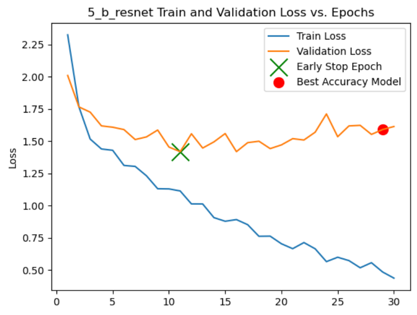
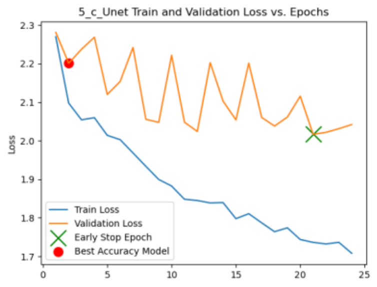
Layer of Network		Layer Properties							
Layer No.	**Type of layer**	**In-Channel**	**Out-Channel**	**Kernel Size**	**Padding**	**Stride**	**Dilation**	**Activation**	**Batch Normalization**
1	Convolution	3	64	3x3	1	1	-	ReLU	Yes
	Convolution	64	64	3x3	1	1	-	ReLU	Yes
M1	MaxPool	64	64	2x2	1	2	-	-	No
2	Convolution	64	128	3x3	1	1	-	ReLU	Yes
	Convolution	128	128	3x3	1	1	-	ReLU	Yes
M2	MaxPool	128	128	2x2	1	2	-	-	No
3	Convolution	128	256	3x3	1	1	-	ReLU	Yes
	Convolution	256	256	3x3	1	1	-	ReLU	Yes
M3	MaxPool	256	256	2x2	1	2	-	-	No
4	Convolution	256	512	3x3	1	1	-	ReLU	Yes
	Convolution	512	512	3x3	1	1	-	ReLU	Yes
M4	MaxPool	512	512	2x2	1	2	-	-	No
5	Convolution	512	1024	3x3	1	1	-	ReLU	Yes
	Convolution	1024	1024	3x3	1	1	-	ReLU	Yes
6	Deconvolution	1024	512	2x2	-	2	-	-	No
C6	Convolution (M4.concat(6))	1024	512	3x3	1	1	-	ReLU	Yes
	Convolution	512	512	3x3	1	1	-	ReLU	Yes
7	Deconvolution	512	256	2x2	-	2	-	-	No
C7	Convolution (M3.concat(7))	512	256	3x3	1	1	-	ReLU	Yes
	Convolution	256	256	3x3	1	1	-	ReLU	Yes
8	Deconvolution	256	128	2x2	-	2	-	-	No
C8	Convolution (M2.concat(8))	256	128	3x3	1	1	-	ReLU	Yes
	Convolution	128	128	3x3	1	1	-	ReLU	Yes
9	Deconvolution	128	64	2x2	-	2	-	-	No
C9	Convolution (M1.concat(9))	128	64	3x3	1	1	-	ReLU	Yes
	Convolution	64	64	3x3	1	1	-	ReLU	Yes
7	Convolution	32	21	1x1	-	1	-	-	No

We would like to note that we also implemented the same enhancements that we used to make our baseline model better here in part 5.a, 5.b and 5.c. In more details, we added cosine annealing learning rate scheduler, used an adamW optimizer, added early stopping with a patience of 3 to prevent overfitting, and used the same data augmentation techniques that we discussed earlier (flipping, rotating, cropping) to increase the training size of our model (x3 total size with respect to the original training size). We also used the same weighted loss function we created above to deal with the class imbalance of the dataset.

4 Results

Table 4: Results & Loss Plots of all Architectures

Model Name	Plot	Test Loss	Average IoU (Val)	Pixel Acc (Val) %
Baseline		1.372173	0.05893	72.0788
4_a_lossSched		1.33896	0.06136	71.584
4_b_Augmentation		1.31664	0.06393	70.9877
4_c_WtedLoss		2.2286	0.067859	69.10516

5_a_OurModel		4.426594	0.0765846	69.02857
5_b_Resnet18		1.8001	0.15017	74.44799
5_c_UNET		2.3586649	0.08326066	72.3275459

Please note that in these figures, the red dot corresponds to the best accuracy model achieved based on the IoU metric, whereas the early stopping is based on the loss metric.

4.2 Visualization results

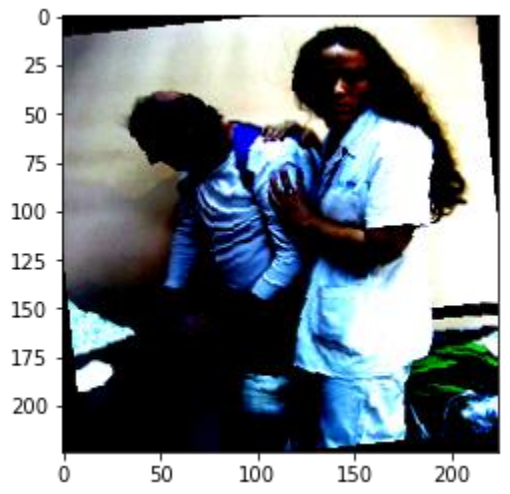


Figure 1: Original Image (Post Augmentations)

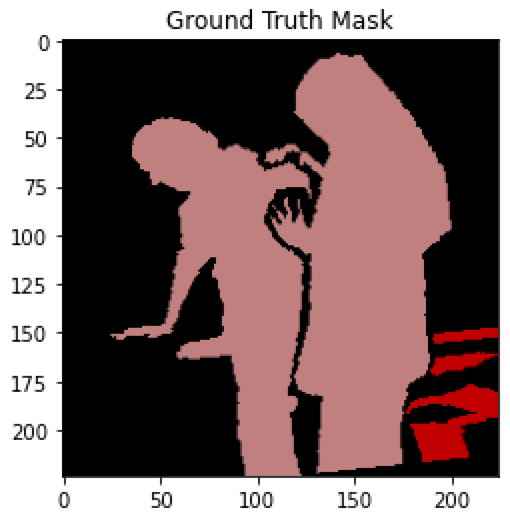


Figure 2: Ground Truth Mask

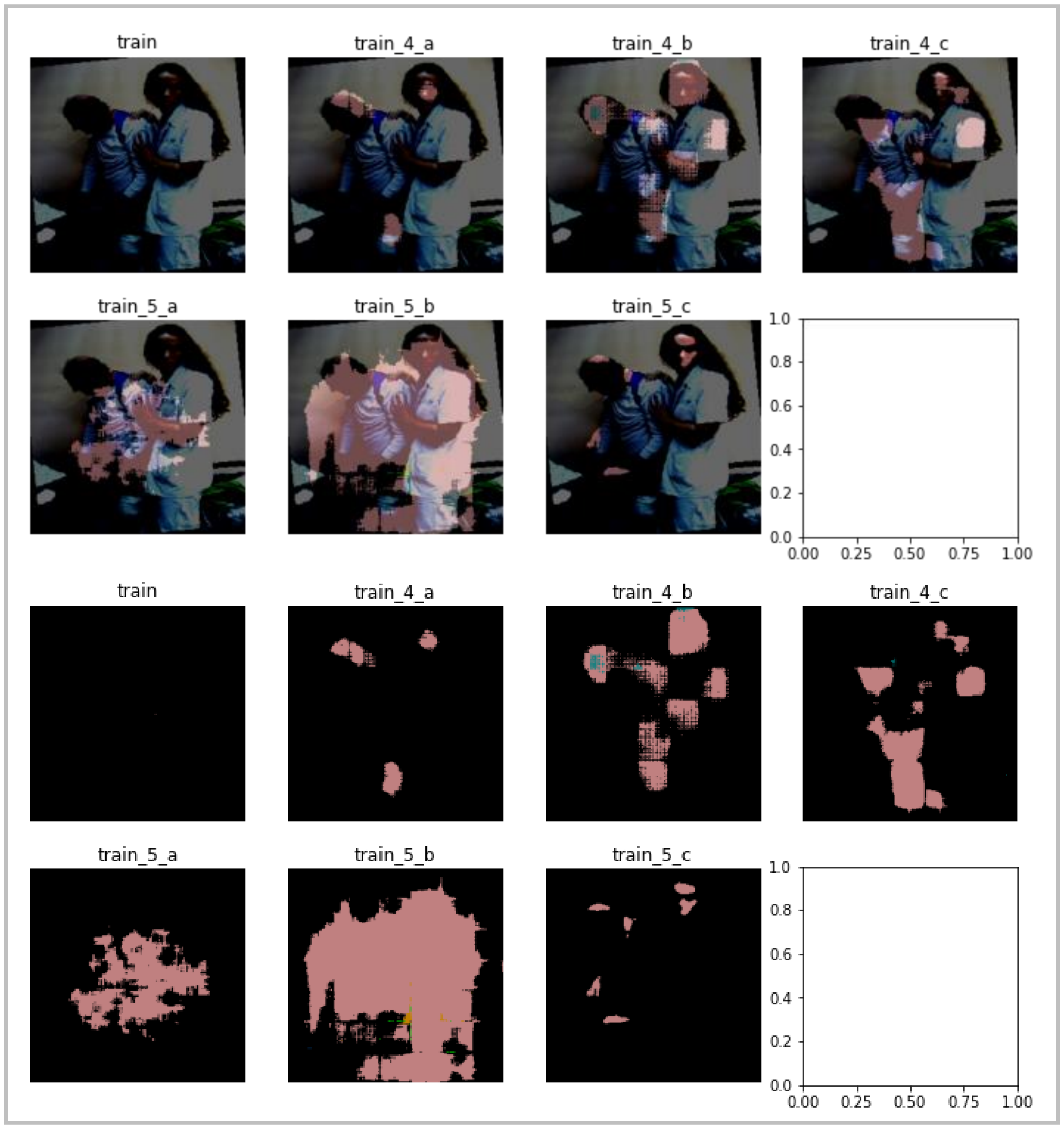


Figure 3: Architecture Predictions (RGB)

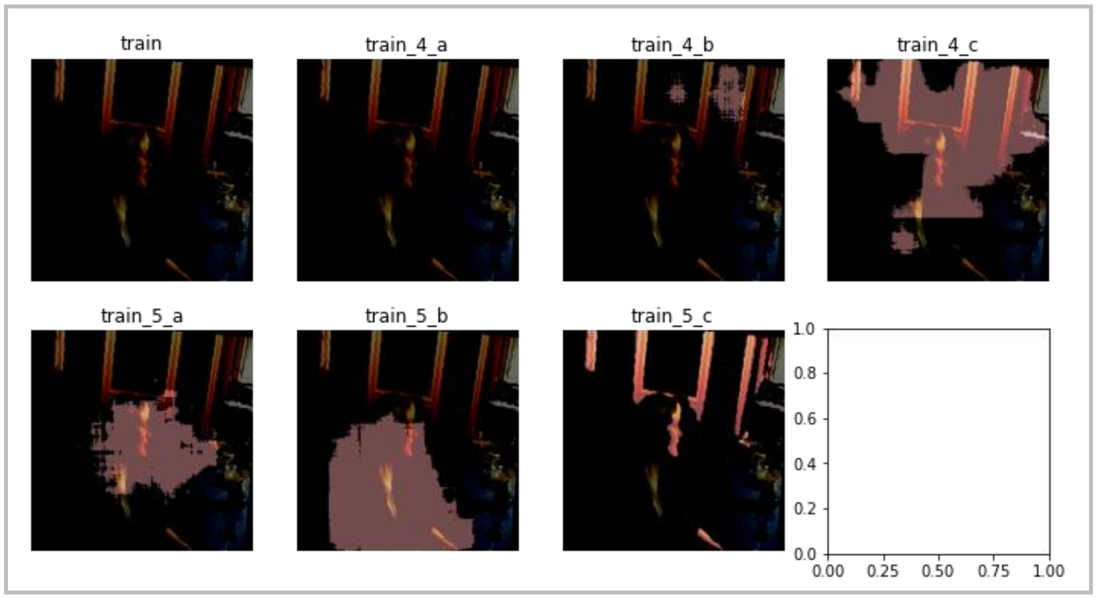
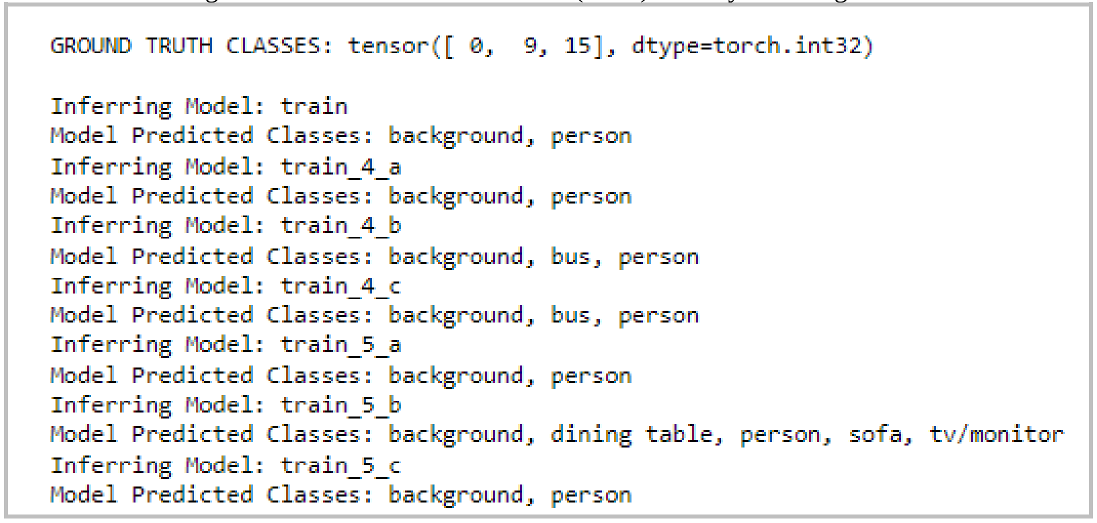


Figure 3: Architecture Predictions (RGB) overlayed on Figure 1.



```
GROUND TRUTH CLASSES: tensor([ 0,  9, 15], dtype=torch.int32)

Inferring Model: train
Model Predicted Classes: background, person
Inferring Model: train_4_a
Model Predicted Classes: background, person
Inferring Model: train_4_b
Model Predicted Classes: background, bus, person
Inferring Model: train_4_c
Model Predicted Classes: background, bus, person
Inferring Model: train_5_a
Model Predicted Classes: background, person
Inferring Model: train_5_b
Model Predicted Classes: background, dining table, person, sofa, tv/monitor
Inferring Model: train_5_c
Model Predicted Classes: background, person
```

Figure 4: Prediction results in tabular format

5 Discussion

5.1 Baseline Model

In the first part of this assignment, in Q3, we implemented a vanilla network which we considered our 'Baseline Model'. We have previously discussed the architecture and attributes of it, so we are not going to repeat that again. But we want to say that given that vanilla implementation, the model performed up to expectations (IoU of 0.05 approximately and average pixel accuracy of 0.65 on the validation set). A lot of things contributed to these low metrics, some of it are only due to the model architecture, which we will discuss here and some of them are outside factors including size of dataset at hand. The intrinsic factors were mainly the loss function that was unweighted (regular multi class cross entropy) which incited the model to always predict 'background' (label: 0, which is the highest frequency label in the dataset) for the pixels since that was lowering that loss, and that was due to the class imbalance that we had in the data. This had a negative effect on the IoU which was visible later as this metric started to increase when we replaced the loss function with a weighted one. But even after improving the model (architecture and loss function) we couldn't achieve really higher results than what we got here in this very simple approach, so we wouldn't label this as a bad model considering it's a baseline one. I think one thing that made it better is the fact that we used a good gradient descent optimizer namely AdamW, I think that was a good point for the model and helped achieve these 'not so bad' results. As we progress with the subsequent sections, we will be comparing this model's accuracy and IoU metrics with the models that come, so it was good having this one as a benchmark for the models to come. The baseline model gave us a loss of 1.363, a pixel accuracy of 72.08% and an IoU of 0.055, so overall good numbers. The baseline model has a good loss curve, but when it comes to prediction it is falling in the trap of only predicting background, as we can see form the plots below, even though it has the highest pixel accuracy score, that it only due to predicting '0' for all pixels. That is why we introduced IoU as a better more generalized metric and a weighted loss.

5.2 Improvements over Baseline Model

In this section, we will discuss all the improvements that were made on the baseline model discussed above. We implemented three major changes, two of them were to the architecture of the model itself, or what we called, 'intrinsic' problems on the baseline model and one was to deal with extrinsic problems, mainly the dataset side, which we dealt with by doing data augmentation, that we had discussed the techniques of that earlier. In this paragraph, we will talk about how each one of these three improvements affected our metrics.
The first one is implementing a cosine annealing learning rate scheduler. The idea behind the cosine annealing learning rate scheduler is to gradually decrease the learning rate over time in a cyclical pattern. This is done by applying a cosine function to the learning rate, which starts at a maximum value and gradually decreases to a minimum value, before increasing back to the maximum value. The benefits of the cosine annealing learning rate scheduler are two-fold. First, it can help the model converge more quickly by starting with a high learning rate that enables it to make large weight updates early on. Second, it can help the model avoid getting stuck in local optima by periodically increasing the learning rate, which can help it explore different parts of the weight space. The second one was the data augmentation, which we did by using mirror flipping, rotating some images by arbitrary degrees and cropping some other images. By doing so, we increased our training data to 3 times the previous one which definitely contributed to better performance. Lastly, the weighted loss function tried to help in fixing the class imbalance issue that we had, and this also showed in the metrics. All of these improvements were only upside on what we had previously in the baseline model which actually shows in the metrics that we presented. We don't think there is necessarily anything that is bad about these models, especially in 4.c after combining these techniques together. As we have progressed throughout this part, our IoU was getting better, which means our model was getting better at predicting classes. Even though the pixel accuracy was overall decreasing, but that was also a good sign. With the help of our weighted loss function our model no longer only predicts a '0', which is a good thing. The data augmentation also helped the model by giving it more data to train on, and the rotations, cropping and flipping made the model more robust to noise and able to generalize better, which also showed in the results we had. Our IoUs went from 0.055 to 0.0613 then 0.0639 and finally 0.068. Each time we were adding a new feature we were keeping old ones, which helped the model become better incrementally each time for the reasons stated above.

We saw the impact of the improvements of the loss curve and the predictions for the segmentations of the images. Since the model started predicting somethings other than background, and here it predicted 'person' which was correct. The loss curve is nice, there still needs to be improvements on localization of the segmentation and having a better IoU, which will be seen in the following more complex networks.

5.3 Experimentation and Exploration

5.3.1 Our own model

This part was probably the most interesting part of the whole programming assignment, cause it involved a lot of trial and error, debugging, and imagination which we really enjoyed. We tried a bunch of different things, some worked, some didn't, and our target was actually to beat the 0.069 IoU achieved by part 4.c so this was our target. We first started by adding a single layer to the network, this didn't actually do much for the metric, we tried increasing the training size (after augmentation from x3 to x10 the original dataset) but still we couldn't break the 0.07 IoU threshold. As we were adding more layers or changing some kernel sizes, we were facing memory issues with the GPU. After all these trials and errors, we will give a rundown of the things we added/changed about the model in question 4.c. We changed the architecture to have, 2 convolutions layers followed by maxpooling layers (twice), and finally two convolution layers again. All the conv layer were activated using ReLU and LeakyReLU (we actually tried both and ReLU gave a better performance so we stuck with it.) We changed some kernel sizes in the encoder and decoder network for the number to stay consistent, for example the first layer had a kernel size of 11x11, and there were two deconv layers with 12x12 and 7x7 kernel sizes). All of the layers has a dilation of 1, stride of 2, some with output padding and some without. We also reduced the batch size to make it 8 instead of the 16 we kept throughout the PA and was also due to memory issues.

After doing these modifications, we tested our model again, using also the improvement added on the baseline model (weighted loss, data augmentation and the cosine annealing). The model with the ReLU activation functions got us a 0.0765 IoU, which was more than our set threshold and more than all the models we have seen previously in this assignment. Adding depth definitely helped the model, and also the kernel sizes helped the model focus on bigger objects in the image as well, and the max pooling was also helpful as well. All these key changed mentioned contributed to making the model better at generalizing and predicting classes.

Our model results were good, loss and metrics. There is just a weird spike in the loss function, early in the epochs but then it gets sorted out after many runs. We also improved on old models by changing kernel sizes, activation functions and adding layers. One thing we could have also added is skip connections, as this would help us make the model even deeper and give better results without falling into the vanishing gradient problem.

5.3.2 ResNet18 Transfer Learning

For the transfer learning part, we decided to use ResNet18, because of the balance it provides between good efficiency and fast runtime for training and inference. We will now present the architecture a bit, and then how we applied transfer learning on that. In short, ResNet-18 is a convolutional neural network that is 18 layers deep. The most important part of ResNet-18 architecture is its basic block, which contains a stacking of a few convolutional, batch normalization, and ReLU activation layers. The basic block also has a skip connection that allows the network to learn residual functions. ResNet-18 architecture consists of 17 convolutional layers, a fully-connected layer and a softmax layer. The convolutional layers are grouped into four stages, each with a different number of filters and spatial dimensions. The basic blocks are repeated within each stage with different strides. The skip connection in ResNet-18 allows the input to be added to the output of each basic block, which helps to avoid vanishing gradient problem and improve accuracy. These were definitely added values to the model and helped us achieved the best result in the whole PA with a 0.15 IoU metric.

Really good metrics as well. The model overfitted since it had too many epochs, but early stopping made things better for validation and testing, which can be seen from plots, metrics and visualization. Overall really good model, but could have been better if we had used a more recent, complex ResNet model but we had memory concerns so we balanced performance with train time.

5.3.3 U-Net Architecture Implementation

In this section we implemented the U-Net architecture provided for us in [1]. The main idea of U-Net is to supplement a usual contracting network by successive layers that increase the resolution of the output and capture both context and localization features. U-Net architecture is as follows: It consists of four blocks, each with two 3x3 convolutions followed by a ReLU activation and a 2x2 max pooling with stride of 2.
The expansive path also has four stages, each with a 2x2 up-convolution that halves the number of feature channels, followed by a concatenation with the corresponding feature map from the contracting path, and two 3x3 convolutions followed by a ReLU activation. The final layer is a 1x1 convolution that maps each feature vector to the desired number of classes. The skip connections between the contracting and expanding paths help to recover fine-grained details that are lost due to down sampling. They also allow for larger receptive fields without increasing the number of parameters. This architecture also performed really well, getting us a 0.083 IoU which was better than all of the baseline performance, the improved baseline and our own architecture, but fell short to ResNet18 that had a much better performance.

Plots and performance are okay, not that great with respect to ResNet model, but still better than our own model and previous baseline and improved baseline. The visualization are nice, as it is a bit more precise than other models who predicts generally big patches for classes.

6 Team contribution

We divided the project equally in all aspects. In fact, we are roommates so we did the whole project part together. For programming, it was pair programming with both of us on discord one guy sharing his screen and coding on a shared google colab notebook , alternating this role between different sections of this assignment.

Andrew normalized the dataset per image per channel (3-a), split the data into train validation and testing. In addition to that, he implemented backward propagation function and checked it by doing part 3-b. He also did part e, which is the experimenting with different activations.

Jay implemented the activation functions formulas, their derivatives, and the forward propagation function. In addition to that, he implemented momentum and regularization problems. Plus, he also did the experimenting with network topology and 100 classes.

For the project report, it was pretty straightforward to do, since we had done everything together, we split up the writing in half, each one of us took the lead in writing the sections that he took the lead in coding.

References

[1]. Ronneberger, O., Fischer, P., & Brox, T. (2015). U-Net: Convolutional Networks for Biomedical Image Segmentation. Lecture Notes in Computer Science, 234–241. https://doi.org/10.1007/978-3-319-24574-4_28

[2]. Shrivastava, I. (2021, December 24). Handling Class Imbalance by Introducing Sample Weighting in the Loss Function. Medium. https://medium.com/gumgum-tech/handling-class-imbalance-by-introducing-sample-weighting-in-the-loss-function-3bdebd8203b4

[3]. Ray, S. (2022, January 4). What Is Data Augmentation? - Lansaar. Medium. https://medium.com/lansaar/what-is-data-augmentation-3da1373e3fa1

[4]. resnet18 — Torchvision main documentation. (n.d.). https://pytorch.org/vision/main/models/generated/torchvision.models.resnet18.html