Image Segmentation Convolutional Neural Network

Jose Chan
Computer Science and Engineering Department
University of California, San Diego
La Jolla, CA 92093
`jchanpineda@ucsd.edu`

Diya Lakhani
Department of Mathematics
University of California, San Diego
La Jolla, CA 92093
`djlakhani@ucsd.edu`

Doanh Nguyen
Department of Mathematics
University of California, San Diego
La Jolla, CA 92093
`don012@ucsd.edu`

Brandon Chiou
Halıcıoğlu Data Science Institute
University of California, San Diego
La Jolla, CA 92093
`djlakhani@ucsd.edu`

Abstract

This report details the approach we took to performing semantic segmentation on the PASCAL VOC 2012 dataset. To begin, we implemented a fully convolutional network architecture, as this type of architecture has been proven to be incredibly successful at this type of task. Semantic segmentation aims to classify pixels into a predefined class. We then continued to try and improve our baseline model with several techniques to solve common problems, and then created a new architecture of our own, tried transfer learning, and implemented a U-Net to see how this would compare to our baseline. We achieved a pixel accuracy of 87% and an average IOU of 0.46.

1 Introduction

Semantic segmentation is an incredibly important topic as its range of possible applications is incredibly wide. It is used for everything from self-driving cars, to medical imaging. This report focuses semantic segmentation on the PASCAL VOC 2012 dataset, a dataset containing 21 total classes, including airplanes, dogs, and cars. Fully convolutional networks have been shown to be incredibly successful at these types of tasks. To solve common issues present in deep learning, we incorporated batch normalization into our model. Normally, when training model weights, the distribution of activations drifts as the data progresses through the model. This drifting can cause the model to converge slower, so therefore after each convolution layer we normalize the activations to stabilize them, ensuring faster convergence. We also incorporated Xavier weight initialization, which fixes two common problems in deep learning. When initializing the weights, we often face the problem of vanishing and exploding gradients. Xavier weight initialization helps with this by ensuring activation variance remains stable, once again speeding up convergence.

2 Related Work

Our work builds upon the foundational work of Long (2015) who introduced FCNs for semantic segmentation, demonstrating their effectiveness in pixel-wise prediction. We also draw inspiration from the U-Net architecture (Ronneberger et al., 2015), known for its skip connections that capture information that might otherwise be lost passing through multiple layers, making it particularly suitable for segmentation tasks. We also looked at the concept of transfer learning (Pan, Yang, 2009), which has been widely used in computer vision.

3 Methods

The Methods section details the implementation and architectural choices made. We begin by describing our baseline FCN model, outlining its architecture, loss function, and optimization strategy. Subsequently, we discuss the improvements implemented to enhance the baseline's performance, including techniques such as cosine annealing for learning rate scheduling, data augmentation, and a weighted loss function to handle class imbalance. Finally, we present the details of our experimental architectures, including a custom design, a transfer learning approach using a pre-trained ResNet, and a U-Net implementation, specifying their architectural configurations, regularization techniques, and training protocols.

3.1 Baseline

Our baseline model is a FCN with an encoder-decoder structure. The encoder consists of 5 convolutional layers, with a kernel of size 3, a stride of 2, a padding of 1, and a dilation of 1. After each convolutional layer, the weights are ReLu activated, and then there is a batch normalization layer. The input image, with 3 channels, is passed through the first convolutional block, which consists of a convolutional layer with 32 output channels. Every convolution layer after this doubles the amount of output channels. Each block (except the first, which takes in 3 inputs) takes the output of the previous block as input. Once we reach the final block, we go through the decoder, which mirrors the encoder. The decoder is made up of 5 transposed convolutional layers, also consisting of kernels of size 3, strides of 2, and padding of 1, which upsamples the feature maps. After each transposed convolution, there is also an activation function and batch normalization layers. The decoder starts with 512 inputs, and decreases the amount of inputs down to 3 in the same way that the encoder increases the inputs from 3 to 512. Finally, there is a 1x1 convolution layer at the very end that acts as a classifier. It takes the 32 channels from the last layer in the decoder to generate a segmentation map with the same dimensions as the input image. The amount of channels matches the number of classes, 20 and the background. We use CrossEntropyLoss as our criterion function, and the Adam optimizer with a learning rate of 0.001. We trained for a max of 30 epochs, and implemented early stopping, stopping early if we saw no improvement after 5 epochs.

3.2 improvements

To improve the performance of our baseline FCN, we incorporated several additions. First, we implemented a cosine annealing learning rate scheduler. This technique dynamically adjusts the learning rate during training by starting with a large learning rate, which decreases following the rate of a cosine graph. This allows the FCN to explore the learning rate space more effectively in early epochs and fine-tune the weights in later epochs, potentially leading to better convergence and improved performance. We used the amount of epochs as T_max, and $1e-6$ as the eta_min. Secondly, we augmented our training data to increase the variety of data that the model saw. We did this by applying random flips, rotations, and crops on the input data and its corresponding labels. This helped the model generalize better to unseen data, and allows it to handle variations in image orientation and scale. Finally, we addressed the class imbalance present in the dataset by implementing a weighted cross-entropy loss function. We calculated weights for each class proportional to its frequency in the training set. This approach was used to penalize the model more heavily for misclassifying less frequent classes, encouraging it to learn more balanced representations and improving segmentation performance for under represented classes.

3.3 Experimentation

3.3.1 Own architecture

For this part of the assignment, we decided to add an extra convolutional layer, integrate dilated filters, dropout, and switch from ReLU to Leaky ReLU as the activation function. The dilated convolution layer was added right after the encoder. We decided to add an extra convolutional layer, increasing the feature maps from 512 to 1024. By doing this, we hoped to capture more complex features of certain parts of an image, allowing it to perform better on parts of an image that has a lot of details. As for why we decided to use leaky ReLU over ReLU, we simply do not want to turn off neurons that output a 0 after ReLU, potentially losing a lot of neurons if ReLU outputs a 0, in this case, since adding another convolutional layer, we further increased the depth of our network, increasing the likelihood of dead neurons, thus leaky ReLU resolves this. We also added dropout since we increased our convolutional layer by 1, this is due to prevent overfitting as we just increased the depth of our network, thus also increasing the chances of overfitting. With these modifications, we experience a slight increase in our mean IoU and pixel accuracy, with a mean IoU of 0.055 and a mean pixel accuracy of 75%. We believe a slight increase in metrics was due to adding more depths within our layer, resulting in the model being able to capture more complex parts of the image. Adding leaky ReLU and dropout addresses some issues that arises from adding another layer.

Table 1: Network Architecture

#	Layer	in_channel	out_channel	Activation	Kernel	Stride	Padding	Dilation
1	Conv2d	3	32	LeakyReLU	3x3	2	1	1
2	Conv2d	32	64	LeakyReLU	3x3	2	1	1
3	Conv2d	64	128	LeakyReLU	3x3	2	1	1
4	Conv2d	128	256	LeakyReLU	3x3	2	1	1
5	Conv2d	256	512	LeakyReLU	3x3	2	1	1
6	Conv2d	512	1024	LeakyReLU	3x3	1	1	1
7	Conv2d	1024	1024	LeakyReLU	3x3	1	2	2
8	Conv2d	1024	1024	LeakyReLU	3x3	1	4	4
9	Conv2d	1024	1024	LeakyReLU	3x3	1	6	6
10	convTranspose2d	1024	1024	LeakyReLU	3x3	2	1	1
11	ConvTranspose2d	1024	512	LeakyReLU	3x3	2	1	1
12	ConvTranspose2d	512	256	LeakyReLU	3x3	2	1	1
13	ConvTranspose2d	256	128	LeakyReLU	3x3	2	1	1
14	ConvTranspose2d	128	64	LeakyReLU	3x3	2	1	1
15	Dropout(0.5)			LeakyReLU				

3.3.2 Transfer learning

For this part of the assignment we adopted a transfer learning approach for semantic segmentation. To begin we replace the encoding part of our model with the pretrained ResNet34 model. We remove the last two layers of this model to ensure the dimensions align with our first decoding layer. We also freeze the weights of the ResNet34 layer since we do not want to train that portion of the model again. Additionally, to avoid errors in the train.py file we make a copy of the train.py file and rename it as train_5b.py. In this file we add another condition to the function that initializes the weights of a layer (init_weights) to make sure that we are not initializing the weights of the pretrained model.

Below is a table detailing the model architecture:

Table 2: Network architecture

#	Layer	in_channel	out_channel	Activation	Kernel	Stride	Padding
1	ResNet34 (excluding last 2 layers)	3	512	ReLU	As specified in ResNet34	As specified in ResNet34	1
2	ConvTranspose2d	512	512	ReLU	3x3	2	1
3	ConvTranspose2d	512	256	ReLU	3x3	2	1
4	ConvTranspose2d	256	128	ReLU	3x3	2	1
5	ConvTranspose2d	128	64	ReLU	3x3	2	1
6	ConvTranspose2d	64	32	ReLU	3x3	2	1
7	ConvTranspose2d	32	21	No activation (classifier layer)	1x1	1	1

Following each layer we normalize the outputs using batch norm. For parameter initialization, as stated above, only weights of the convolutional layers were initialized as the ResNet34 model provided us with a pretrained model. Lastly, we implemented gradient descent with early stopping to obtain a model that performed the best on the validation set. The model converged after 25 epochs

3.3.3 UNet architecture

The UNet Architecture aims to build upon the baseline model's architecture using new layers like spatial pooling and performing two convolutions per encoding block. Our encoding block completed two successive convolutions with an activation function of ReLU. One encoding block is followed by a layer of max pooling. For the decoding, we utilize one layer of convolutional transpose and one similar to the encoding block mentioned. We also make the padding 1 to maintain dimensions as we have a 3x3 kernel.

Below is a table detailing the model architecture:

Table 3: Network Architecture with Encoding and Decoding Blocks

#	Layer	in_channel	out_channel	Activation	Kernel	Stride	Padding
x1	Encoding Block Conv2d Conv2d	3	64	ReLU (applied after each convolution)	3x3	1	1
1	MaxPool	64	64	N/A	2x2	1	0
x2	Encoding Block Conv2d Conv2d	64	128	ReLU (applied after each convolution)	3x3	1	1
2	MaxPool	128	128	N/A	2x2	1	0
x3	Encoding Block Conv2d Conv2d	128	256	ReLU (applied after each convolution)	3x3	1	1
3	MaxPool	256	256	N/A	2x2	1	0
x4	Encoding Block Conv2d Conv2d	256	512	ReLU (applied after each convolution)	3x3	1	1
4	MaxPool	512	512	N/A	2x2	1	0
x5	Encoding Block Conv2d Conv2d	512	1024	ReLU (applied after each convolution)	3x3	1	1
5	MaxPool	1024	1024	N/A	2x2	1	0
y1	**Decoding Block y1** = ConvTranspose2d + Concatenate y1 with x4 Conv2d	1024	512	conv-ReLU	2x2 (Transpose) 3x3 (Conv2d)	stride transpose = 2 Stride conv = 1	padding conv=1
y2	**Decoding Block y2** = ConvTranspose2d + Concatenate y2 with x3 Conv2d	512	256	conv-ReLU	2x2 (Transpose) 3x3 (Conv2d)	Stride transpose=2 Stride conv=1	Padding conv=1
y3	**Decoding Block y3** = ConvTranspose2d + Concatenate y3 with x2 Conv2d	256	128	conv=ReLU	2x2 (Transpose) 3x3 (Conv2d)	Stride$_transpose = 2Stride_conv = 1$	padding$_conv = 1$
y4	Decoding Block y4 = ConvTranspose2d Concatenate y4 with x1 Conv2d	128	64	conv=ReLU	2x2 (Transpose) 3x3 (Conv2d)	Stride$_transpose = 2Stride_conv = 1$	padding$_conv = 1$
y5	Classifier conv2d	64	21	ReLU	1x1	1	0

For regularization, we utilized data augmentation techniques which involved adding mirror reflection, rotated, and cropped images in the dataset. The weights of each layer were initialized as usual as no pretrained was being used. We implemented gradient descent with early stopping to obtain a model that performed the best on the validation set. The model converged after 6 epochs. To handle the case of class-imbalance, we utilized data augmentation for both parts. Class imbalance occurs when certain classes are less represented in the image which causes the model to classify most pixels as the majority class. Data augmentation techniques like cropping images can help mitigate the class-imbalance problem for all the experiments above.

4 Results

4.1 Baseline

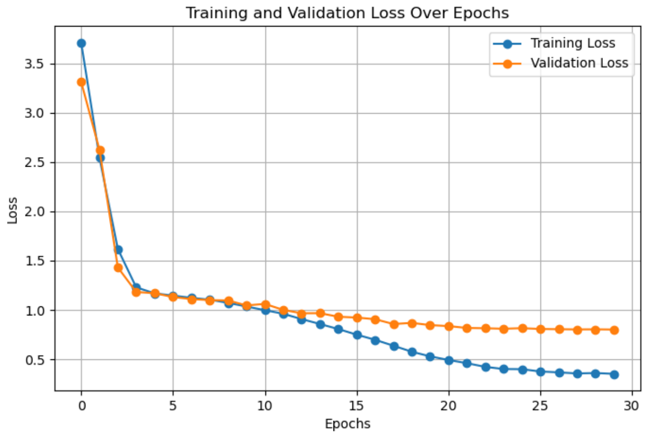

Figure 1: Loss plot for baseline

	Loss	IoU	Pixel Acc
Val epoch 29	0.7801366213547147	0.2314080683059341	0.8350322083656835
Test	0.7757258880269396	0.22857184466484728	0.8353981758933827

Table 4: Results baseline

4.1.1 Improvements

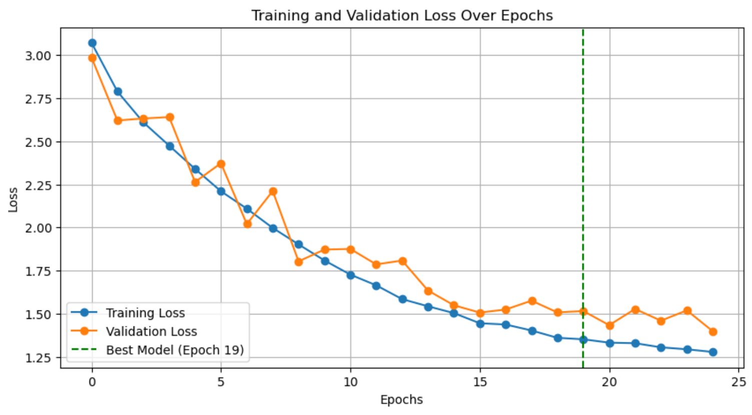

Figure 2: Loss plot for improvements

	Loss	IoU	Pixel Acc
Val epoch 25	1.421323659627334	0.05106221493474679	0.7343583737418201
Test	1.4644582114376865	0.04956124677716362	0.7261173940014422

Table 5: Results improvements

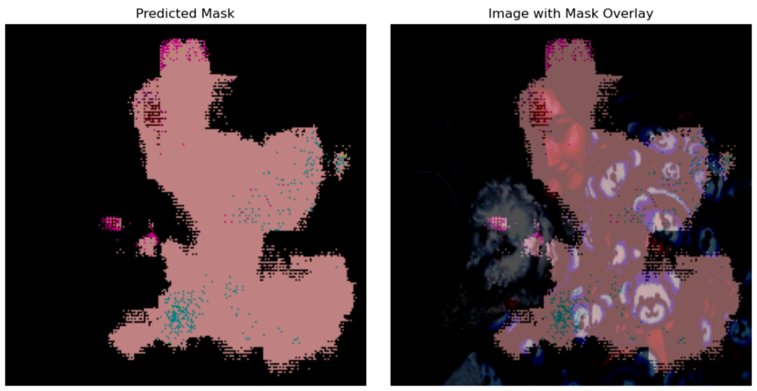

Figure 3: Predicted improvements mask and overlay

4.2 Experimentation

5 Discussion

5.1 Baseline

One great benefit of having our baseline model being somewhat limited is the speed at which the model trains on the data. Due to how bare-bones this model is, the training process is quick relative to how long it takes to train the other models. It also manages to achieve a respectable accuracy of 80-85The drawback is that it is a bare-bones network, using an autoencoder and no other additional features.

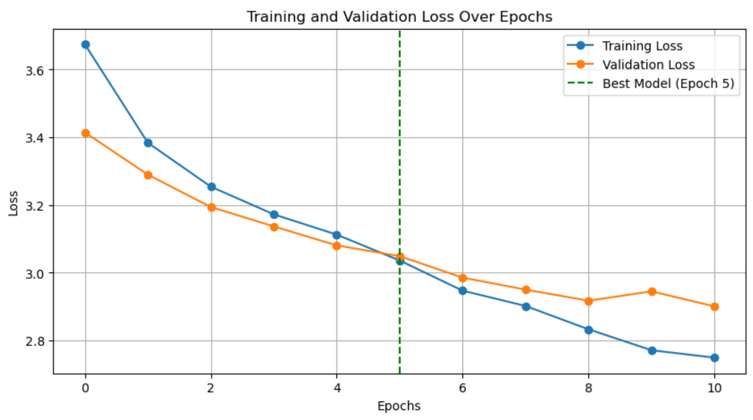

Figure 4: Loss plot for experimentation

	Loss	IoU	Pixel Acc
Val epoch 11	2.8623401496721352	0.03146557353963229	0.21812250768598887
Test	2.855935348259224	0.0307266694340395	0.2144307379903083

Table 6: Results improvements

5.2 Improvements

5.2.1 Cosine annealing learning rate

The cosine annealing learning rate scheduler dynamically adjusts the learning rate, starting high and slowly decreasing over time. This helps prevent the model from getting stuck in local minima and allows it to be more smooth. A drawback is that it can be too slow or fast, which might make the model overfit or underfit.

5.2.2 Data augmentation

Augmenting the dataset using transformations like mirror flips, rotations, and crops introduces more variability, which might prevent overfitting. It also reduces overfitting by ensuring that the network does not memorize specific patterns in the training data. A drawback might be that if augmentations make the original images too different, it might cut out crucial parts of the image, even though the classes in the mask are still there.

5.2.3 Class imbalance

Class imbalance is a problem because the background is very common. This means that it is much harder to train the model on images that contain rare classes, and training instead focuses on the background. Therefore, to combat this whenever we calculate loss, we also multiply each class by its own weight derived from the inverse frequency of the classes, ensuring that the model would get punished more for mistakes with rare classes, and punished less for mistakes with common ones. A big drawback is that since the background class is so common, penalizing it means we predict less, and we lose a lot of accuracy. We tried penalizing it less by manually multiplying the background weight by like 80, and the pixel accuracy did go back up to 73

5.3 Experimentation

5.3.1 Own architecture

For our own architecture, we used dilated filters, global context, and skip connections inspired by U-Net CNNs on top of our basic FCN. The dilated convolution layer was added after the encoder, and then followed by global context processing. We utilized the leaky ReLU activation function to improve feature extraction. During the forward pass, the decoder implemented skip connections to

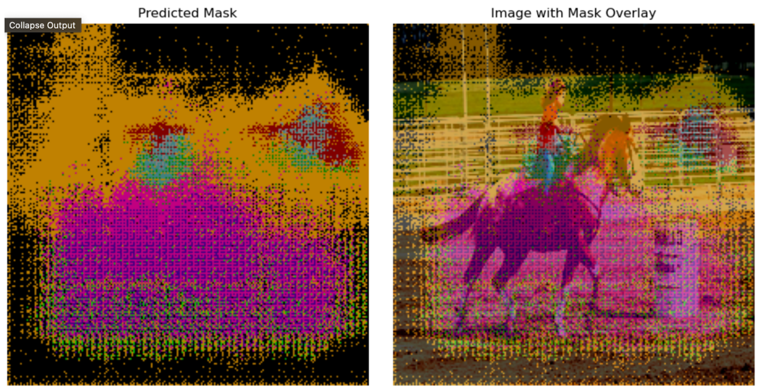

Figure 5: Predicted experimentation mask and overlay

preserve spatial information from the encoder. With this architecture, we achieved 0.06562 mean IOU and 0.7449 pixel accuracy.

5.3.2 Transfer learning

For this section, we employed a transfer learning approach by replacing the encoding portion of our model with a pretrained ResNet34 network. The last two layers of ResNet34 were removed to align the output dimensions with our first decoding layer. The pre-trained weights of ResNet34 were frozen to prevent redundant training. Additionally, we modified our training script (train_5b.py) to ensure that the initialization function (init_weights) did not override the pre-trained weights. This one performed the best with 88% pixel accuracy. The reason this model performed better than the others is due to the pretrained ResNet34 weights. These were created by training the ResNet34 model on Imagenet data which has over 100 categories. For this reason, the weights were better suited for our simpler model containing only 21 categories.

5.3.3 U-net

The U-Net architecture extended the baseline model by introducing spatial pooling and performing two convolutions per encoding block. Each encoding block applied a sequence of two convolutional layers followed by batch normalization and ReLU activation. The decoder mirrored the encoder structure, utilizing transposed convolutions to upsample feature maps. We also had Skip connections from the encoder to the decoder to preserve spatial information and to get more refined masks. The performance of the U-net model was quite similar to that of the baseline with an accuracy of 74% and IoU of 0.06. Initially we anticipated that the U-Net would improve the performance, however, the performance could have remained the same due to the changes we made to UNet. In particular, the changes to ensure the dimensions of the input and output remain consistent.

5.3.4 Comparison

The transfer learning model (5b) was the best because it used pretrained features from ResNet34. In contrast, 5a (dilated CNN with skip connections) performed well because dilated convolutions expanded the receptive field without losing resolution, while skip connections helped preserve spatial details.

Full U-Net focused on reconstructing detailed masks with transposed convolutions and batch normalization, which improved pixel accuracy but was not as good as 5a.

The baseline improvements (4b and 4c) addressed overfitting and class imbalance, but they lacked good architecture. While data augmentation (4b) helped generalization, it didn't really change the network's ability to extract features. Similarly, class rebalancing (4c) tried to correct class distribution issues but was bad at balancing segmentation accuracy and making good masks.

6 Author's Contributions

Much of the work was completed together, and we each reviewed each others work. The following paragraphs show which parts of the code we each focused on.

6.1 Doanh Nguyen

We were all equally involved in the programming section, and used each other's code to compare in order to improve the model so that it reaches the provided benchmark. Worked on ways to improve the baseline model and visualizing the masked images. Wrote out section 5 of the report.

6.2 Brandon

I did question 4 in the code portion. Also wrote 6 and 7 in the report

6.3 Jose Chan

We preferred to have several people preforming implementations to make sure we were on the same track, and see who's approach was better. I was involved in the util file, tho we ended up not using mine, creating a training loop, and experimenting with the improvements section of the assignment to try and fix some bugs we came across. I also the abstract, introduction, related work, methods intro, methods:baseline, methods:improvements, discussion:baseline, creating the latex tables for our data, references, and overall report maintenance,

6.4 Diya Lakhani

I completed the implementation of the baseline FCN model, U-Net model and the training loop. I also contributed to generating plots and creating the predicted mask and image overlay.

References

[1] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[2] O. Ronneberger, P. Fischer, and T. Brox. U-Net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, 2015.

[3] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 2009.