Final Project: Semantic Segmentation with Hypercolumns

Christopher Chaves

Resources

[1] Paper introducing hypercolumns: Bharath Harhiharan, Pablo Arbelaez, Ross Girshick, Jitendra Malik, "Hypercolumns for Object Segmentation and Fine-grained Localization", https://arxiv.org/pdf/1411.5752.pdf
[2] Paper introducing the VGG network: Karen Simonyan, Andrew Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition", https://arxiv.org/pdf/1409.1556.pdf
[3] Pascal Visual Object Classes data: http://host.robots.ox.ac.uk/pascal/VOC/voc2012/index.html#data
[4] VOC2012 Results: http://host.robots.ox.ac.uk/pascal/VOC/voc2012/results/index.html
[5] imagesegmentationhypercolumns.ipynb
[6] createtrainvalsets.ipynb

Task and Data

This project implements pixel-wise semantic segmentation on the Pascal VOC 2012 segmentation dataset. There are roughly 1500 training and 1500 validation images in this dataset. Each image contains any number of 20 possible objects. These 20 objects, along with a background class, form 21 classes, and pixels must be part of exactly one of these classes. There is also an "ambiguous" class, usually occurring around the borders of objects, which should not be used in training or evaluating. This is a pixel classification problem.

My goal is to build a model that is competitive with the winners of the Pascal VOC 2012 segmentation competition, none of which were neural networks. This is somewhat hard to judge because the only evaluation metric given by [4] is precision in each class and mean precision over the 21 classes. An important benchmark is the best mean precision, which was **47.3%** ([4]). I evaluate my model based on precision over classes, accuracy over classes, and total accuracy. Given that this is a harder segmentation problem than homework 3 where the benchmark was 70%, but also that I am using a powerful pretrained network, a reasonable benchmark for total pixel-wise accuracy is **70%**.

Model

My model is trained on hypercolumns extracted from the VGG16 network ([2]), which is pretrained for image classification on ImageNet. [1] defines a hypercolumn at a certain pixel to be a vector of activations "above" that pixel. In other words, an image is passed to a CNN, and then its activations are concatenated in some fashion to construct a hypercolumn for each pixel in the image. Then, I pair hypercolumns with pixel labels and learn a shallow classifier on them. To predict pixel-wise classification on an image, I pass the image through part of the VGG16 network, extract hypercolumns, pass these hypercolumns through my shallow classifier, and predict labels of the hypercolumns, which correspond to labels for the pixels in the original image. See the implementation section below for further details.

Preprocessing and Setup

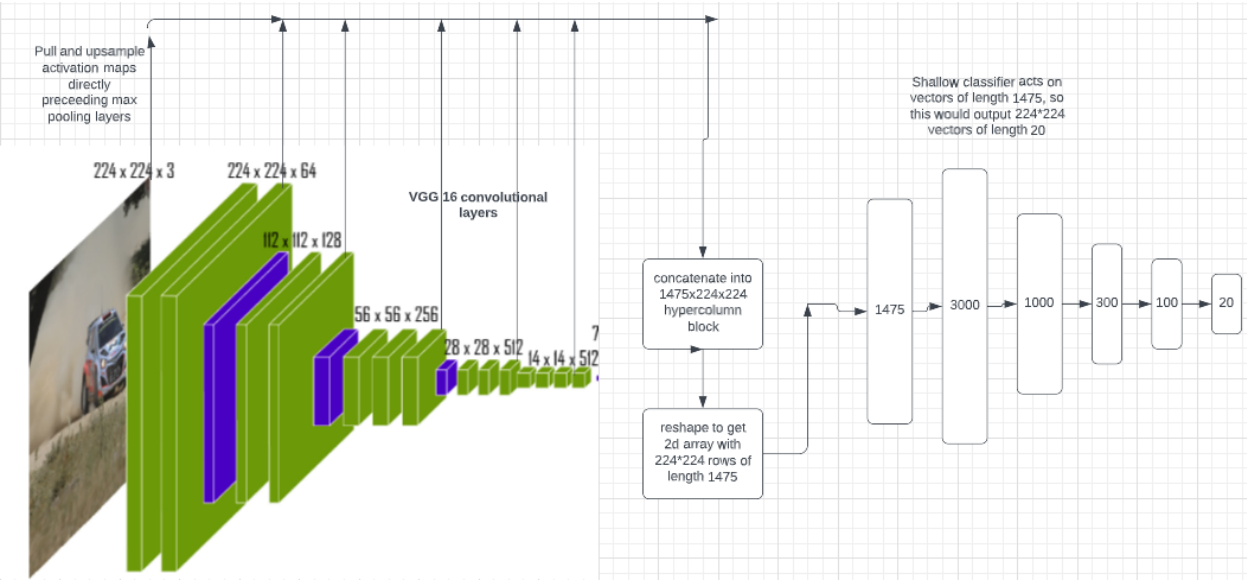
I use [6] to extract the images designated by [3] for the segmentation task. The training and validation sets for segmentation are named by [3] in a text file. I copy the images designated by this text file into training and validation folders, with an x and y folder for each. I then upload these 4 folders manually to my Google Drive main directory.

In Google Colab, I construct and save a sparse training dataset of hypercolumns. I do this by passing each training image through the VGG16 convolutions, generating hypercolumns. I then sample 6 hypercolumns per class in the image, and pair these with class labels to form a training set. This results in a training set of size around 21000. Because passing these images through requires cropping to 224x224, the masks containing the labels must also be upsampled and cropped. To do this, I upsampled the masks using the nearest neighbor method, as opposed to the bilinear method, so that all pixels in the upsampled version would still be class labels.

To run my code from start to finish, the user needs to download [3], run [6], and manually upload the folders under the names train_x, train_y, val_x, and val_y (15 minutes). They should also upload the text files naming the two sets. Then, they need to uncomment the code that constructs the hypercolumn dataset and run the notebook (15 minutes – 1 hour depending on GPU).

Implementation

As described above, I use the VGG16 model [2] in tandem with a shallow classifier learned on hypercolumns. The model passes images through the VGG16 model and uses the results to construct hypercolumns, which are then passed through the shallow classifier.



VGG16 architecture: I construct hypercolumns using only the convolutional layers of VGG16. [1] details how fully connected layers can be used to construct hypercolumns too, but I chose to focus on information that directly encodes features of a spatial location. I resize the image to 256x256 and center crop it to 224x224. I apply the necessary transforms and pass it through the

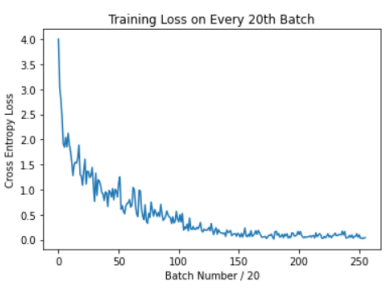
convolutional layers. I save only the activation maps occurring before max pooling layers. I upsample these activations to 224x224 and concatenate them all together. This results in hypercolumns of length 1475.

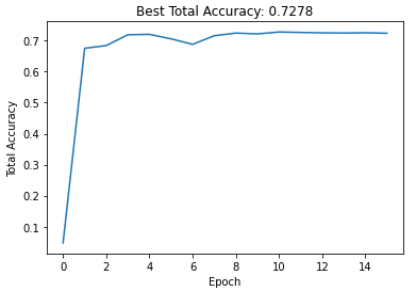
Shallow classifier: This classifier has 4 fully connected layers, with a 0.5 dropout layer after the first. The size progression is 1475 -> 3000 -> 1000 -> 300 -> 100 -> 21.

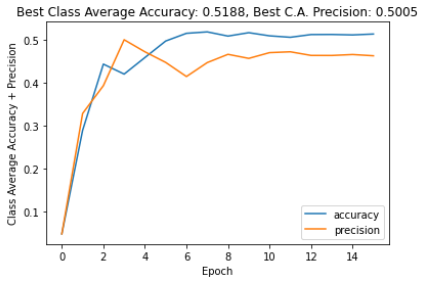
I train the shallow classifier using stochastic gradient descent with batch size of 64, learning rate of 0.01, and momentum of 0.9. I train over 15 epochs and decay the learning rate by a factor of 0.1 at epochs 6 and 12.

Results

I produce plots of total accuracy, mean class accuracy, mean class precision, and training loss over time.

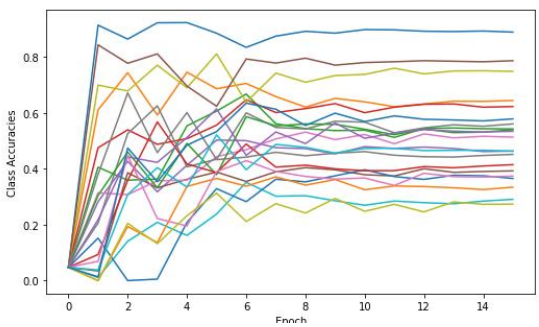


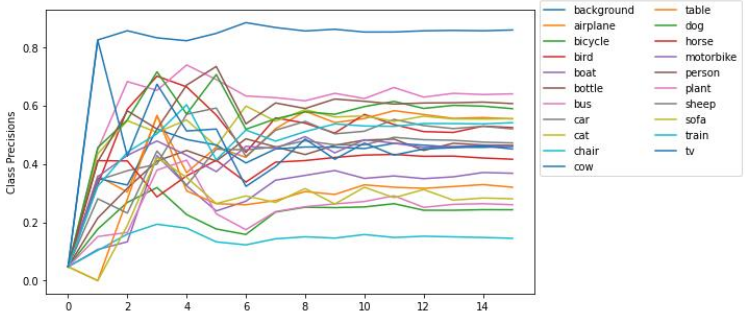




As shown, I passed both the 70% total accuracy benchmark and the 47.3% class mean precision benchmark with peaks of **72.78% and 50.05%** respectively. However, my peak class mean precision, occurring at the 3rd epoch, of 50.05% was somewhat fluky, and came at the cost of class mean accuracy of 42.04% and a total accuracy of 71.89%. In general, the right plot hints at the inverse relationship between precision and accuracy. If the competition was evaluating only on precision, I would submit this model. If the competition was evaluating more wholistically, I would probably submit my model after the 11th epoch, which had **class mean accuracy of 50.62%, class mean precision of 47.24%, and total accuracy of 72.63%.** This model provides essentially equal class mean precision, but with very strong accuracy.

I also show plots that break down accuracy and precision by class.





My model predicts some classes better than others, but the most glaring takeaway is that it predicts background pixels with far greater ability than other classes. This may be due to about a third of the training examples being background pixels, due to my sampling method. “Person” was the second most common class in the training set, and my model outperformed the class precision benchmark of 51.6%. However, my model underperformed in other classes. This suggests that a larger training set, especially in those less common classes, would have helped my model’s overall performance.

One could easily precision-hack their model by predicting an inordinate amount of background pixels. This would achieve high precision on the non-background classes, as the model would only predict them when very confident, and since every class is weighted equally in the class precision mean, this would raise this value. I did not do this and still got similar or better results to 2012 winners. Therefore, I conclude that my model would have been very competitive in the 2012 Pascal VOC segmentation competition.

As a further demonstration of a working system and an example of my best result, I present my results from epoch 11, or epoch 10 starting from 0.

```
vgg_model = VGG16()
shallow_classifier = ShallowModel()
optimizer = optim.SGD(shallow_classifier.parameters(), lr=0.01, momentum=0.9, weight_decay=0.0)
scheduler = MultiStepLR(optimizer, milestones=[6,12], gamma=0.1)
accs, class_avg_accs, class_accs, class_avg_precs, class_precs, best_acc, best_classavg_acc, best_classavg_prec = train(trainloader, vgg_model, shallow_classifier, optimizer, scheduler, epochs=15)
```

```
Epoch 10, Iteration 0, loss = 0.1440
Epoch 10, Iteration 20, loss = 0.0426
Epoch 10, Iteration 40, loss = 0.1752
Epoch 10, Iteration 60, loss = 0.0660
Epoch 10, Iteration 80, loss = 0.0658
Epoch 10, Iteration 100, loss = 0.1035
Epoch 10, Iteration 120, loss = 0.0456
Epoch 10, Iteration 140, loss = 0.1186
Epoch 10, Iteration 160, loss = 0.0587
Epoch 10, Iteration 180, loss = 0.0724
Epoch 10, Iteration 200, loss = 0.0840
Epoch 10, Iteration 220, loss = 0.0957
Epoch 10, Iteration 240, loss = 0.0218
Epoch 10, Iteration 260, loss = 0.0481
Epoch 10, Iteration 280, loss = 0.1272
Epoch 10, Iteration 300, loss = 0.0849
Epoch 10, Iteration 320, loss = 0.1682
100%|██████████| 1449/1449 [01:32<00:00, 15.59it/s]
Accuracy: 0.7263332605361938
Accuracy by class: tensor([0.8975, 0.6223, 0.5128, 0.6209, 0.4733, 0.3746, 0.4899, 0.5290, 0.7603,
        0.2848, 0.3724, 0.3389, 0.5224, 0.3937, 0.5285, 0.7834, 0.3407, 0.4488,
        0.2732, 0.4735, 0.5897])
Mean class accuracy: tensor(0.5062)
Precision by class: tensor([0.8517, 0.5822, 0.2636, 0.5346, 0.3586, 0.4849, 0.6615, 0.5522, 0.5457,
        0.1479, 0.4704, 0.3200, 0.6140, 0.4324, 0.4709, 0.6049, 0.2905, 0.4911,
        0.2842, 0.5285, 0.4300])
Mean class precision: tensor(0.4724)
```