Image Segmentation for Outfit Detection

Machine Learning Advanced Nanodegree Program
Capstone Project

Project Overview

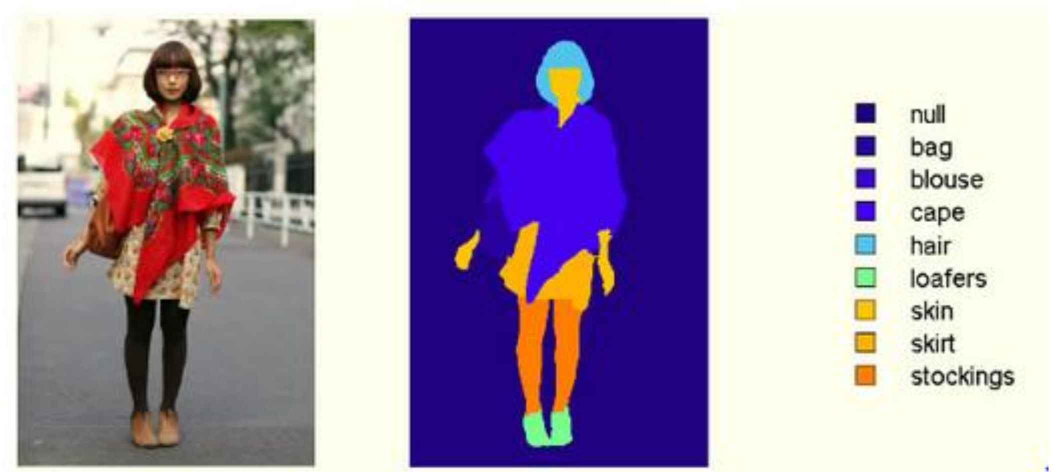
Introduction

Clothing parsing is a specific form of semantic segmentation, where the categories are one of the clothing items, such as t-shirt. Clothing parsing has been actively studied in the vision community, perhaps because of its unique and challenging problem setting, and also because of its tremendous value in the real-world application. Clothing is an essential part of our culture and life, and a significant research progress has been made with a specific application scenario in mind. In this project, we consider how we can utilize recent deep segmentation models in clothing parsing.

Clothing parsing distinguishes itself from general object or scene segmentation problems in that fine-grained clothing categories require higher-level judgment based on the semantics of clothing and the deforming structure within an image. What we refer the semantics here is the specific type of clothing combination people choose to wear in daily life. For example, people might wear dress or separate top and skirt, but not both of them together. However, from recognition point of view, both styles can look locally very similar and can result in false positives in segmentation. Such combinatorial preference at the semantics level introduces a unique challenge in clothing parsing where a bottom-up approach is insufficient to solve the problem. Here we approach the clothing parsing problem using fully-convolutional neural networks (FCN). FCN has been proposed for general object segmentation and shown impressive performance thanks to the rich representational ability of deep neural networks learned from a huge amount of data. To utilize FCN in clothing parsing, we need to take the above clothing-specific challenges into account, as well as a care to address the lack of training data for learning large neural networks.

Problem Statement

The problem statement can be posed as the detection and labeling of different categories of clothes within an image that consists of a single front-facing person with visible full body. Given below is a sample of input image and a corresponding output image (ground truth) that the model should be capable of learning and reproducing.



Metrics

Since we are dealing with convolutional neural networks, the following can be used to measure the performance of our models:

- Multinomial Cross Entropy Loss / Log Loss

$$L = \sum_{c=1}^{C} y_{o,c} \log(p_{o,c})$$

where, C is the number of classes,

y - binary indicator if the class label c is the correct classification for observation o

p - predicted probability observation o is of class c

- Accuracy

$$Accuracy = \frac{Number\ of\ pixels\ correctly\ classified}{Total\ number\ of\ pixels}$$

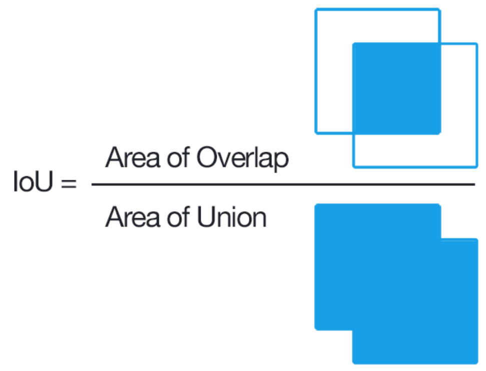
The mean accuracy over all the images in the test set can be used to compare different models.

- Intersection Over Union (IOU)

$$IOU = \frac{2 * Number\ of\ pixels\ correctly\ classified\ except\ background}{Sum\ of\ non\ background\ pixels\ in\ both\ label\ and\ prediction}$$

The iou serves as a more robust measure than accuracy in the sense that it does not inflates up over classifying background class as above 40% of pixels on average belong to that class. Other classes of interest like t-shirt, dress, skin individually would contribute to about 10% - 20% of the image.

The image below would clear up the concept of iou visually.



Analysis

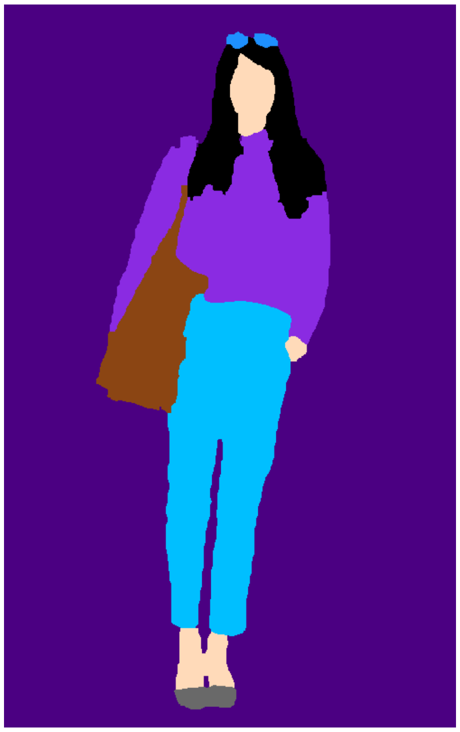
Dataset Exploration and Visualization

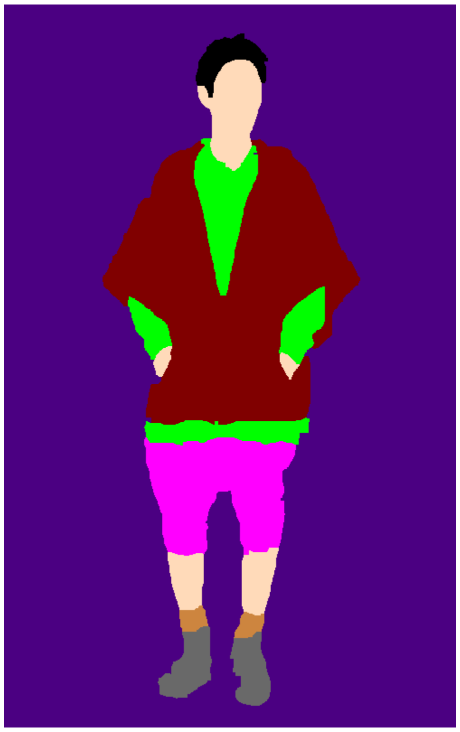
I have used two different datasets in this project. The first is the *clothing co-parsing* dataset hosted here - https://github.com/bearpaw/clothing-co-parsing. The other is the *colorful-fashion-parsing-dataset* hosted here: https://sites.google.com/site/fashionparsing/dataset.

The clothing co-parsing dataset contains 1004 images with pixel level annotations that could be used for training. The labels are divided between 59 categories including background. This data due to its high cardinality in the number of labels is difficult to learn. Also, not all classes are represented equally in the dataset. While the background takes most of the share, certain classes like *belts, earrings, necklace, tie, glasses, rings, accessories etc.* are not learned due to insufficient examples.

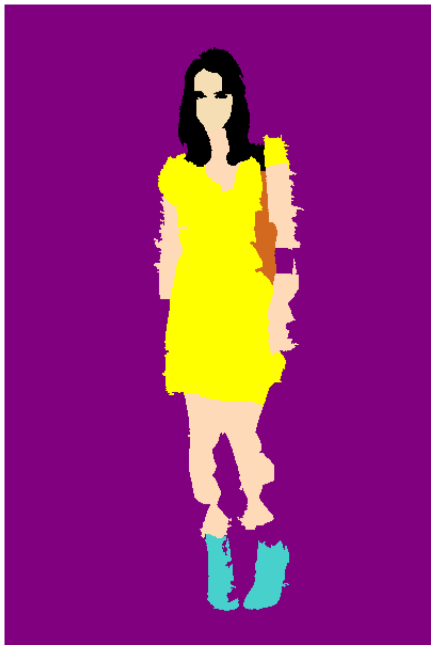
Apart from the problem of insufficient data, it also consists of labels that could be substituted or combined in one class, for example, *jeans, leggings, pants and tights* even though are different but might appear to be same and with not enough training data the classifier is bound to get confused among them. There are various examples of such labels, that could have been grouped under one category for simplicity. Another example : *flats, heels, loafers, sandals, shoes. sneakers, and wedges.* Some categories represent a broader aspect and covers other categories like - *suit, coat and blazer,* might be difficult for a classifier to learn within a very limited set of training images.

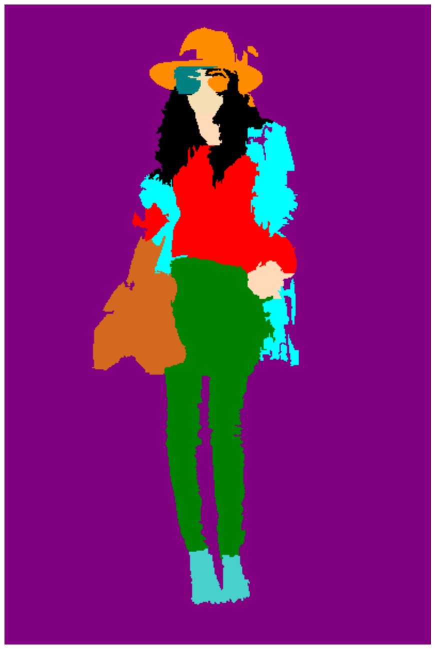
Some example from this dataset are shown below :





The colorful fashion parsing dataset contains 2682 images and just 23 different categories, making it comparatively easier to learn than the ccp dataset. Also, there is only much less overlap in their categories, thus the dataset is far less noisy, for example, it contains only one foot wear cateogy - *shoes*. The only short coming with this dataset is that its annotations are in superpixel notation with each feature image divided into 425 superpixels. So, the labels are nowhere as smooth as they are in the dataset above, and are quite noisy. A couple of examples from this dataset are below.

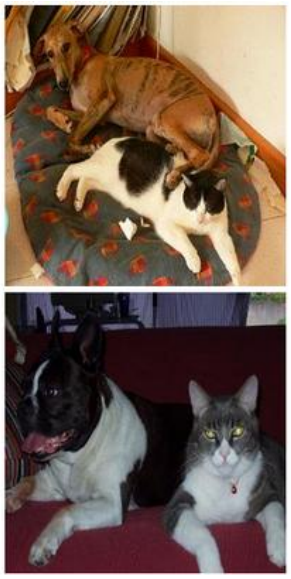


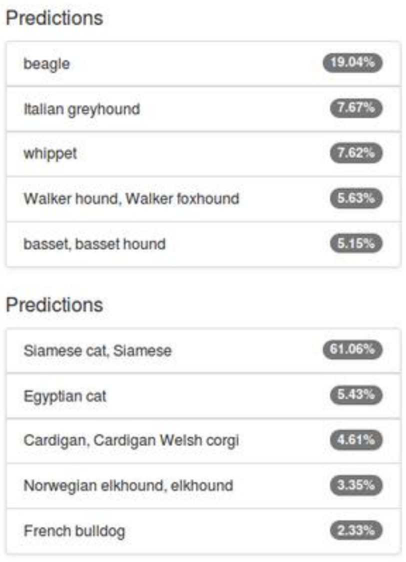


Algorithms and Techniques

Image classification is the problem of assigning an image a label from a fixed set of categories. This is one of the core problems in machine learning which even though appears simple on the surface are extremely difficult to tackle. This has a huge variety of practical applications in computer vision. Image classification is a technique by which a computer program can tell that a picture of a dog is the picture of a dog.

The output of an image classification model is a probability score between 0 and 1 for each the classes it was trained on. But what happens if we have both of the animals – cat and dog in a single image? The classier will might get confuse and might assign one of them as the label. This happened because the model was trained to deal with multiple objects to start with. Look at the sample outputs below from the PASCAL VOC dataset.





Predictions

beagle	19.04%
Italian greyhound	7.67%
whippet	7.62%
Walker hound, Walker foxhound	5.63%
basset, basset hound	5.15%

Predictions

Siamese cat, Siamese	61.06%
Egyptian cat	5.43%
Cardigan, Cardigan Welsh corgi	4.61%
Norwegian elkhound, elkhound	3.35%
French bulldog	2.33%

This is where the problem of image segmentation arises. Image segmentation is the task of dividing the input image into regions, where each region belongs to one of the fixed set of classes. Instead of predicting a single probability distribution for the whole image, the image is divided into patches and then each patch is classified. Therefore, we do not have a single label for the whole of image but label for each and every pixel of the image. Thus segmentation reduces to the problem of classification in just a slightly different manner. This allows the model to detect various objects within an image as well as their location. Look at the sample segmentation output below from the PASCAL VOC dataset.

CNNs work great for image classification tasks but however it is not possible to achieve image segmentation with them without some tweaks. Image classification models process one probability distribution function for the whole of image whereas image segmentation models need to predict one probability distribution function per pixel of the image. We have already discussed how image classification reduces down to image segmentation, here we will figure out how to redesign a classification network into a segmentation network.

Converting FC Layers to Convolutional Layers

Convolutional Neural Networks make use of fully connected layers that flatten the input feature maps into a vector, thus all spatial information is lost which is acceptable as spatial information is not needed for classification purposes. However, segmentation requires spatial information to predict multiple probability distributions across the image. So we go about converting fully converting fully connected layers into convolutional layers, hence the name fully convolutional network.

In a fully connected layer every neuron computes a weighted sum of its inputs. In comparison, in a convolutional layer every filter computes the weighted sum of its receptive field. When the input is larger than the receptive field, the filter slides over the image and computes again. This process repeats until the filter scans the whole of the image and generates a matrix of activations also called a feature map. Both layers are somewhat similar in nature. In order to replace the first fully connected layer with a convolutional layer, simply set the size of the filter to the size of the input and use as many filters as there were in the original fully connected layer. Now we can go ahead and covert all other fully connected layers by setting filter size to 1 and keeping the stride 1. We are done! This way we have added a spatial dimension to the output of a convolutional network.

The resulting FCN model has exactly the same number of learnable parameters, the same expressivity and the same computational complexity. Another advantage is that the resulting model is now independent of the size of the input and is no longer constrained to work on fixed size input. It can process images of larger sizes by scanning through the input as a sliding window and produce produces one probability distribution function for the number of sliding windows across the whole image.

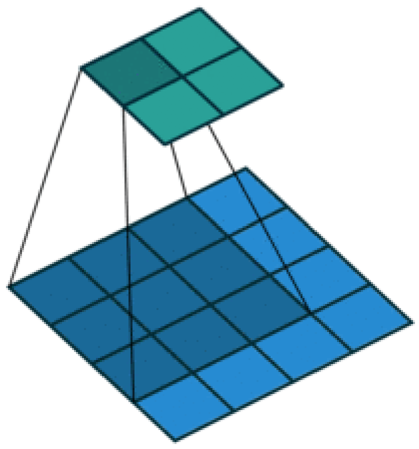
Deconvolution Layer

Ideally an image segmentation model should predict probability distribution per pixel of the input image but as we have seen above it predicts one per window. The number of windows depends upon the input image size, the window size and the stride. Thus, the final output of the model is not of the same size as the input image but it is of lesser size.

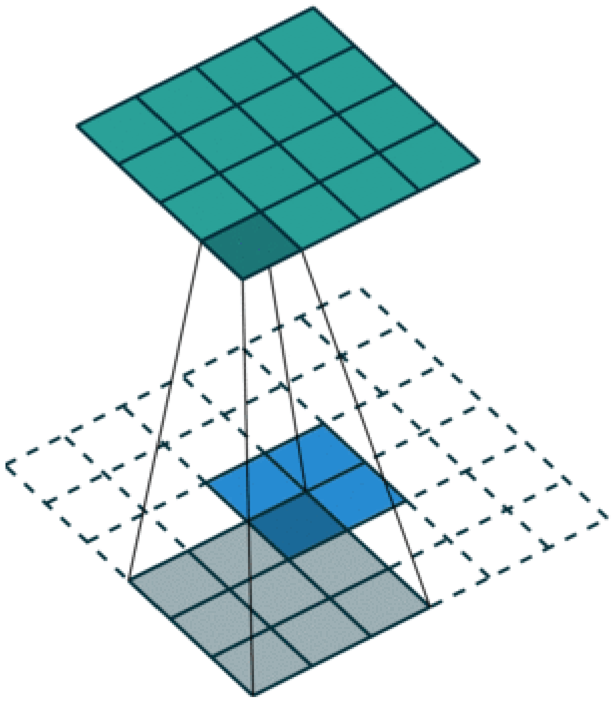
When the input image traverses the successive layers of the convolutionalized model, the pixel data at the input is effectively compressed into a set of coarser, higher-level feature representations. In image segmentation, the aim is to interpolate those coarse features to reconstruct a fine classification, for every pixel in the input. It turns out that this is easily done with deconvolutional layers. These layers perform the inverse operation of their convolutional counterparts: given the output of the convolution, a deconvolutional layer finds the input that would have generated the output, given the definition of the filter.

Deconvolution layers are bilinear interpolation kernels that project the input from convolution layers and map them to an image of the same size as the input. For this reason they are also referred as upsampling layer. Let us take an example to understand the operations in this layer. First we will compute the output of a direct convolution operation on an input, and then go about figuring out how we could have generated the input back given the output.

Let us consider a convolution of a kernel of size 3 over an input of size 4x4 with unit stride and no padding. This will generate an output feature map of size 2x2 as shown below.



Now, the transpose of this convolution (doconvolution) would be to output a shape of 4x4 when the same filter of size 3 is applied over the 2x2 input. A naïve way to think about it understanding it as a direct convolution operation with input of size 2x2, kernel size 3x3, unit stride and double zero padding on all sides, as shown below.



However, there exist efficient interpolation kernels that directly upsample the input, not involving a direct convolutional approach as it is computationally expensive.

Transfer Learning

Even though that we have now formally defined convolutional networks both for classification and segmentation, very few people train an entire convolutional from the scratch as it is rare to have datasets of sufficient size. The total number of learnable parameters in deep convolutional architectures are in order of 10^5 and to converge them with small dataset is infeasible. Instead of learning weights from the scratch, we use a sequence of transfer learning steps. The idea is to make use of already available huge datasets such as ImageNet or PascalVoc to extract features and representations, and then use these extracted features for the task of interest.

Transfer learning is applied in two major ways:

1. *Using convolutional network as a feature extractor:* In order to extract features, we remove the last fully connected layer of a pre-trained convolutional neural network, and now what we have is a network that outputs input data representation as a long fixed length vector. This can be used to generate vector representations directly from images. For our new image dataset once we have extracted vector representations of each image, they can be used to train any other classification algorithms like SVM, Random Forests, AdaBoost etc.

2. *Fine-tuning the convolutional network:* The idea is to take a pre-trained model and instead of removing the last year, modify it with respect to the new dataset and train the model again. In this way, all the pre-trained weights which were already learned in the original model will be fine-tuned for the purposes of the new dataset i.e. instead of using random weight initialization we are initializing the weights of our model with a set of beautiful pre-trained weights that are already excellent feature extractors. The idea is motivated by the observation that the pre-trained weights are not only useful for one dataset but are generic enough in nature to be useful for a variety of datasets. It is possible to fine-tune weights of all the layers, but due to overfitting concerns it is a common practice to retain the weights of the starting layers and only modify weights in the last few layers. This way the network extracts and make use of many generic features, and progressively becomes task specific in the later layers.

When we implement transfer learning, we get bound to some constrains in terms of the architecture we follow. For example, we cannot take out layers from a pre-trained network for the purpose of the new dataset or modify their hyper parameters as then the pre-learned weights won't fit into the new architecture. Further, we tend to keep lower learning rates when fine-tuning because we expect that the pre-trained weights are relatively good and we don't want to distort them too quickly and too much.

Conditional Random Fields

It is generally employed as a post-processing step to further refine the outputs from the above learnt networks. CRF helps to estimate the posterior distribution given predictions from our network and raw RGB features that are represented by our image. It does that by minimizing the energy function which are defined by the user. In our case it takes into account the spatial closeness of pixels and their similarity in RGB feature space (intensity space). On a very simple level, it uses RGB features to make prediction more localized – for example the border is usually represented as a big intensity change – this acts as a strong factor that objects that lie on different side of this border belong to different classes.

Benchmark

There is already a lot of advanced work done in this field using techniques like *outfit-encoders* and *conditional random fields* to optimize to further fine-tune the output of deep neural networks. Various strategies and their state-of-the-art outputs have been discussed in this paper here: https://arxiv.org/pdf/1703.01386.pdf

This paper can be used as a reference benchmark model against my outputs. This paper uses the colorful fashion parsing dataset and not the clothing co-parsing dataset. They claim an accuracy of 92.4% and IOU of 54.65% over the *cfpd* dataset.

Methodology

Data Preparation and Preprocessing

The data preparation steps were mostly common in both of the datasets. I needed to prepare label images from the *mat* files which contains annotations. For each input feature image, there needs to be a corresponding label image with pixel wise annotations.

CCP Data Preparation

- The *ccp* dataset contains 1004 mat files, each corresponding to the 1004 feature images. The mat file contains the ground truth array of the same size as the feature image.
- This array consists of integers in range 0 to 58, corresponding to the labels. The labels are given in a separate labels.mat file which contains a mapping for each label to category.
- I manually defined a color palette - RGB sequence for each class (*black for hair, violet for background, peach for skin and so on*) and created label images using this palette.
- The images were of variable size - approx 550 x 820 pixels. To keep training uniform, I resized the feature and label images to 500 x 800.
- I created a training and test split, with 200 images in the test set, and trained models on the remaining images.

CFPD Data Preparation

- All the annotations were given in a single mat file. It was a little trickier to extract data from this file as I couldn't find any documentation and the data was stored in an unintuitive structure.
- It contains four keys - '#refs#', 'all_category_name', 'all_colors_name', 'fashion_dataset'. 'fashion_dataset' is an array of length 2682 corresponding to all the feature images. It further contains annotations in supper pixel notation which could be used to generate label images.
- The *all_category_names* contained label texts for each of the 23 classes.
- I manually defined a color palette for all the 23 labels and generated label images.
- All these images were of the same size 400 x 600.
- I moved 15% of the data (405 images) into the validation set.

All the code for parsing *mat* files and label generation can be found in the accompanying *ipython notebooks.* They will bring more clarity into data preparation part for the *cfpd* dataset, the details of which are however unnecessary here.

Implementation

I have used BVLC's Caffe for training deep neural networks. I also used DIGITS (a web interface to Caffe), as it simplifies the process of training and produces training and validation accuracy curves for continuous evaluation. This saves a lot of time which otherwise would have gone in maintaining logs and other housekeeping stuff. I needed to train above 8 models over two different datasets and multiple architectures, and using DIGITS I could easily switch between them. Also, each training in DIGITS runs as a separate *job,* in a different directory with all the required files (architectures, mean images and solver hyper parameters, and learnt model weights), making it easier to among them. The following usual steps were followed for each training:

- We first prepare an *lmdb* dataset using DIGITS, which the caffe can use for training. Lmdb datasets were prepared for both training and validation sets. Caffe then uses this dataset to extract images and labels, and feed it to the neural network. It is a onetime step, after which we could simply use different architectures over these datasets.

- The neural network architectures in Caffe are defined in *.prototxt* files. I used the standard fcn-32s, fcn-16s, and fcn-8s prototxts and modified them as per my use.

- For training them from scratch, I simply had to update parametric layers (convolution and deconvolution layers) with weight fillers to initilize weights - for example, *gaussian distribution with 0 mean and standard deviation of 0.1, or xavier initialization method.* The deconvolution layer requires *bilinear* weight initialization.

- I also changed the number of outputs in the final convolution/deconvolution layers as per my datasets - 23 for *colorful fashion parsing* dataset and 59 for *clothing co-parsing* dataset.

- While training using pre-learnt weights I added a convolution layer, with a kernel size of 1 and stride 1, just before the loss layer to modify the number of outputs as per my requirement. The pre-learnt weights were trained on the PASCAL_VOC dataset which contained 21 classes, and thus the original architectures produced 23 outputs.

- I also experimented with different hyper-parameters, *learning rate, learning rate decay methods, optimizers and batch size.* I found that ADAM converges much more quickly than SGD, thus reducing training time significantly.

- I also implemented Conditional Random Fields, to further refine the outputs of the learnt models. I used hit and trial over few images from the validation to set the hyper-parameters required in conditional random fields.

Results & Justifications

I trained a number of models on both of the datasets to compare different architectures and measure their performance. I used pre-trained weights from the *pascal-voc* dataset, and further refined them as per my datasets. It was clear in a single training instance that models that were learnt using pre-trained weights were far superior in performance that models learnt from scratch.

It usually took me around 6 to 10 hrs for a training a single model over Nvidia GTX 1080 with 8GB of GPU memory. I learnt the following two models from scratch : *fcn-8s for the colorful fashion parsing dataset and fcn-16s for the clothing co-parsing dataset.* These models were significantly lower in accuracy when compared with their counterparts that were learnt using pre-trained weights. Learning other architectures from scratch would have produced similar results and so I avoided investing time training those. Fcn-8s achieved an accuracy of 78.8 % over the cfpd dataset, while fcn-16s achieved an accuracy of 76.95% over the ccp dataset. Below are their *loss* and *accuracy* curves.

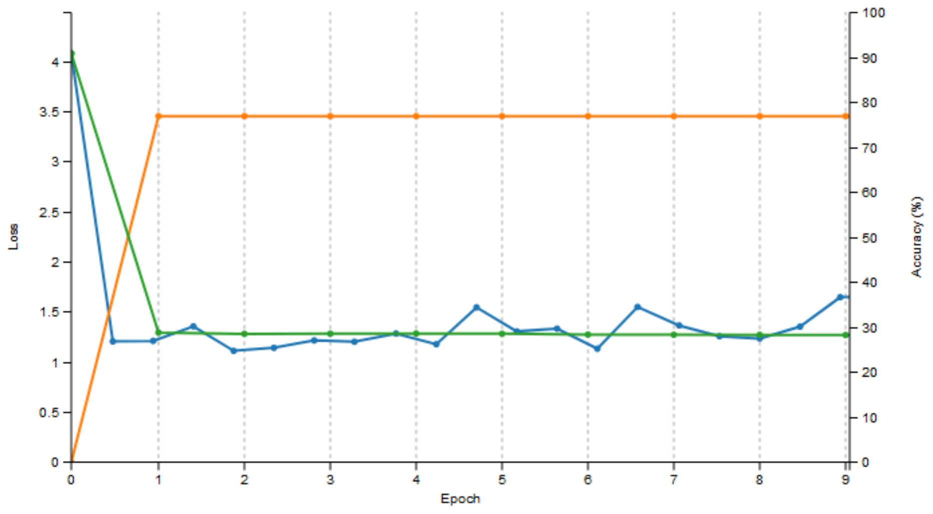


Figure 1:fcn-8s over cfpd dataset with random initialization

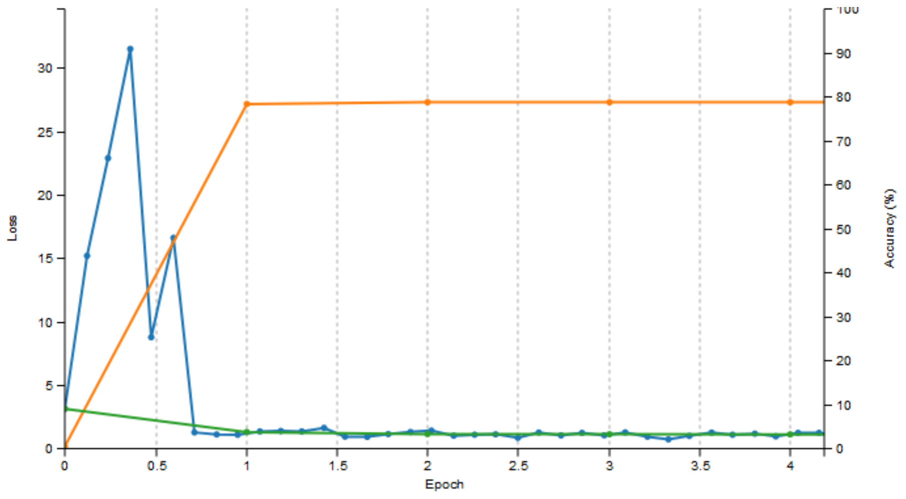


Figure 2: fcn-16s over ccp dataset with random initilization

When using pre-trained weights, the results were much decent. Below I have summarized the results over different architectures, over both of the datasets.

	CFPD Dataset	CCP Dataset
FCN-32s	90.72%	86.51%
FCN-16s	91.88%	**87.44%**
FCN-8s	**92.07%**	86.3%

The best performing architecture over the CFPD dataset was fcn-8s, closely followed by fcn-16s. In case of ccp dataset, fcn-16s was clear winner and out-performed fcn-8s.

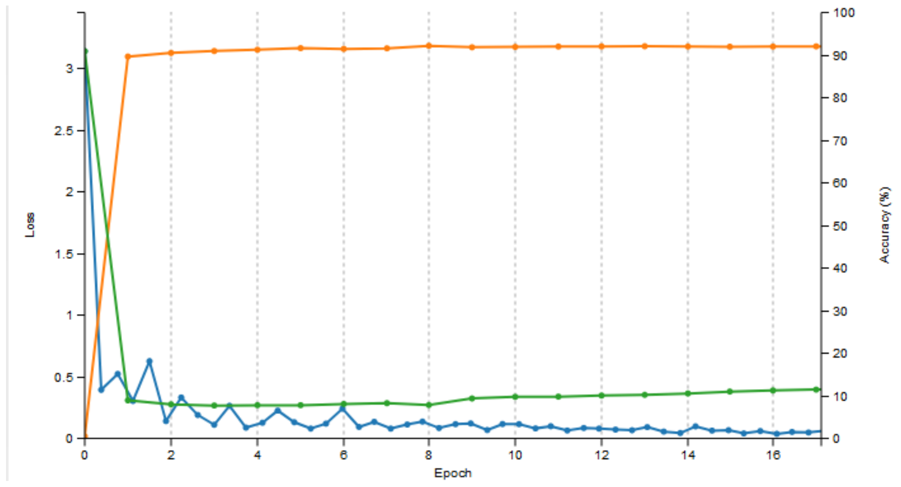


Figure 3:fcn-8s over cfpd dataset with pre-trained weights

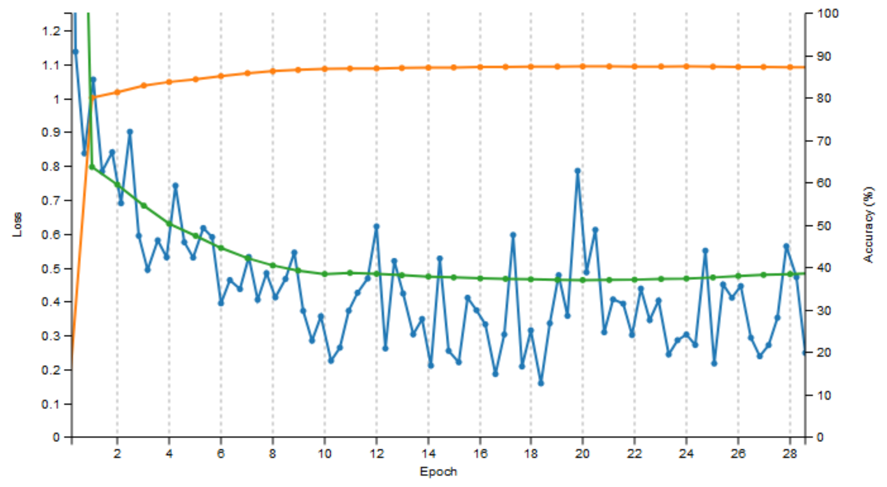


Figure 4: fcn-16s over ccp dataset with pre-trained weights

After identifying the best performing models, I calculated an IOU score for them by running the model over each of the images from the validation set. I obtained mean IOU of **67.82%** on fcn-8s over the cfpd dataset. This iou is decent as compared to our benchmark model that produced an accuracy of **91.58%** and an iou of **51.28%**.

Over the ccp dataset, mean iou of **49.78%** was obtained. Over this dataset iou was lower probably because of insufficient data size as well as large number of classes.

I then also applied *crf* post processing to further refine the output of the above models. The hyper parameters for *crf (position scaling, smoothness scaling and color scaling)* were tuned using hit and trial over images from the validation set. The *crf post processing* further improved the performance of our models and produced much smoother and visually appealing results. It fine-tuned the output from out models using color information directly from the feature images, thus reducing noise in the output.

With *crf* over the *cfpd* dataset, the accuracy of our model increased from 92.07% to **92.56%,** while *iou* increased from 67.82% to **70.02%**, which can be considered as a significant improvement. Our benchmark model obtained an iou of 54.6% only. In case of the *ccp* dataset, *crf* improved the accuracy of the model from 87.44% to **88.58%** and *iou* from 49.78% to **53.40%.**

Given below is an example from the cfpd dataset and its output from the model along with crf. With close observation one can easily notice that the ground truth was noisy near over the spectacles and footwear. The model's output is quite noisy, but after crf post processing the result is much smoother. Even the spectacles are now nicely shaped, and the skirt has taken its shape perfectly when compared with the feature image.

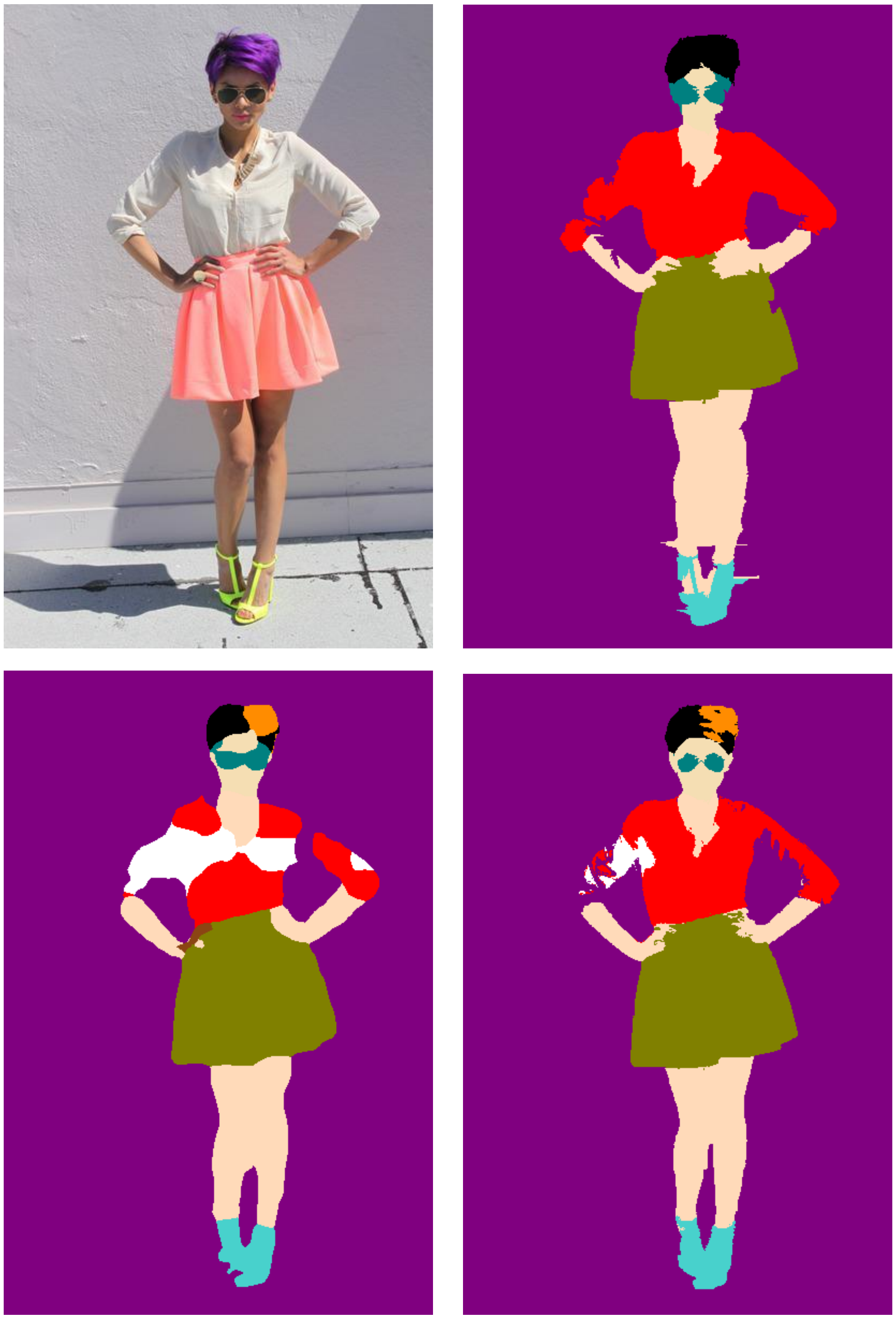


Figure 5: An example from cfpd dataset(top left) along with ground truth(top right), prediction(bottom left) and crf post processing(bottom right)

The following is an example from ccp dataset. We can clearly observe that the model has labeled khaki colored pants as skin over the leg. Also, we see significant change in the output on using *crf,* yet the output is not close to the ground truth. In terms of learning, one can argue that *ccp* with 59 different classes and just 1000 images is not a proper dataset for fashion parsing.

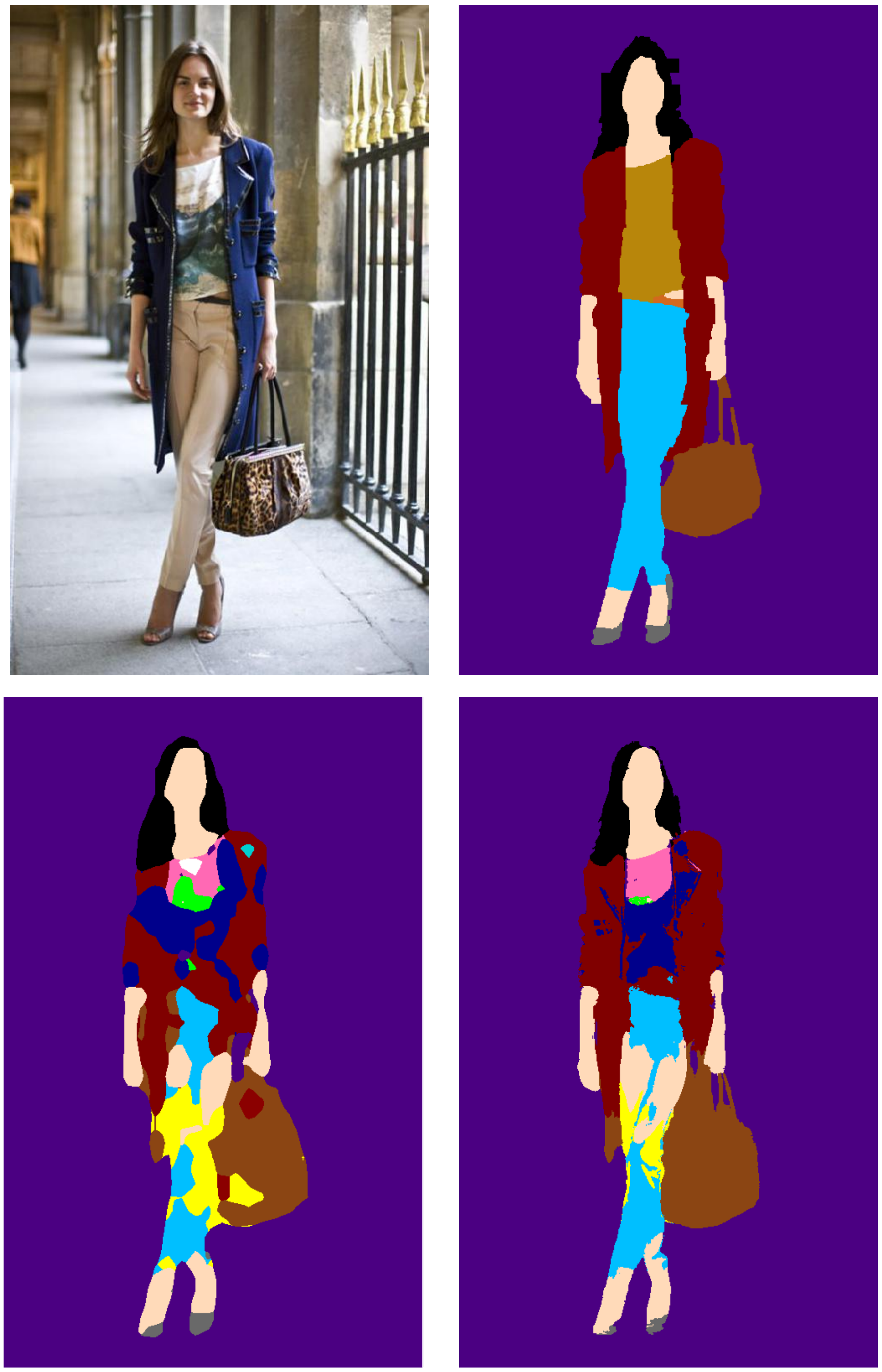


Figure 6: An example from ccp dataset(top left) along with ground truth(top right), prediction(bottom left) and crf post processing(bottom right)

Reflections and Improvement

The most important take away from this project is all the research and debugging I had to go through to understand segmentation networks and most importantly how fully convolutional networks are independent of the size of the input. Understanding the semantic differences between the networks fcn-8s, fcn-16s and fcn-32s and their relative complexity. I studied in detail about how an existing classification network can be converted into fully convolutional segmentation network. Simultaneously it is also important to understand that even classification networks can be directly used to segment an input, but that would be highly computationally expensive. I went through a lot of blogs, discussion forums to fully capture these ideas, and then implement them on my own.

The most difficult part in all this was definitely installing Caffe itself. It comes with a lot of dependencies and installing GPU supported caffe is in itself no less than a challenge. I was surprised to see the power of CRF and the significant improvement they gave to the models. I am still a little fuzzy over the mathematics involved in CRFs, but feeling proud to have it implemented. It is really difficult to find good resources over this technique.

The current state-of-the-art architectures in segmemtation include *residual networks,* and training them should definitely bring an improvement. Also, I simply tweaked the hyper-parameters in CRFs manually, while there are ways learn their optimal values. I would like to invest more time in understanding *residual networks* and their advantages over conventional fully convolutional networks. I would also like to bring clarity in my understanding of *crfs* and techniques required to learn their hyper parameters.