|  |  |
| --- | --- |
| Capstone Project Machine Learning Engineer Nanodegree | Peter James Bernante  July 4, 2016 |

# Definition

## Project Overview

Surgery oftentimes involves post-surgical pain. Managing pain involves the use of narcotics which have several unwanted side effects.

One way to manage pain with less dependency on narcotics is through the use of indwelling catheters that deliver anesthetic. Pain management catheters block or mitigate the pain at the source. These catheters are inserted to the area around the nerves. It is therefore imperative to accurately identify nerve structures in order to effectively insert the catheters.

The dataset for this project is taken from a [Kaggle](https://www.kaggle.com/c/ultrasound-nerve-segmentation) competition “Ultrasound Nerve Segmentation”. It consists of ultrasound images of the neck, from which nerve structures can be identified and would help improve the placement of a catheter.

## Problem Statement

The goal of this project is to identify a collection of nerve structures called Brachial Plexus (BP)[[1]](#endnote-1).

Given an ultrasound image, highlight or annotate the area in the image where BP is located.

We will apply deep learning in computer vision that will learn to recognize BPs in ultrasound images. In doing so, the following tasks will be involved:

1. Design a deep neural network
2. Train the neural network from dataset of ultrasound images
3. Create a script that accepts images as input, and outputs the same images with highlighted/annotated Brachial Plexus if present.

The generated trained model can be useful in several ways. For example, it can be integrated with ultrasound apparatus that will automatically show the area of interest.

## Metrics

This will be a classification problem, where we want to classify each pixel in an ultrasound image to whether belongs to BP (positive class) or not (negative class).

For this binary classification project, it is important to classify all positive classes, and it is also important that all positively identified classes are correct. F1 score[[2]](#endnote-2), where both precision and recall are considered, would be appropriate as the metric of choice to measure the performance of our model. The general formula for F measure in terms of Type I and Type II errors is given as:

where, for F1 score, .

Accuracy can also be used as the metric of choice, however, a quick look at the dataset shows that there is a huge imbalance between positive and negative classes, thereby would make accuracy not a very reliable metric for this case.

# Analysis

## Data Exploration

The dataset that will be used for training can be downloaded from Kaggle at <https://www.kaggle.com/c/ultrasound-nerve-segmentation/data>.

There are 5,635 images for training and 5,508 images for testing. The images are gray scale with dimensions 580 x 420 pixels and are noisy. Training images has masks to indicate where BP is present, while there is none for testing images. In the training images, only 2,323 images have positively identified BP.

There are inaccuracies in the annotations of BP. There is no prefect ground truth or gold standard. There are very similar images but with conflicting annotations; one image has positively identified BP, however the other has none (see Figure 3). There are also very similar images that have positive annotations, but the area where they are annotated differ in shape/area, although the annotations are located approximately in the same region (see Figure 4).

The classes are greatly imbalanced; consisting 1.2% positive class and 98.8% negative class. This class imbalance is addressed later in the Refinement section.

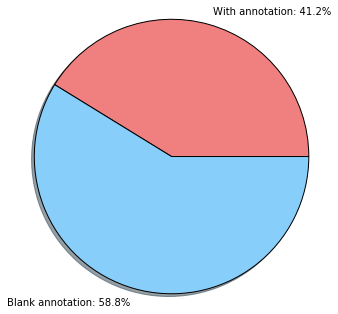
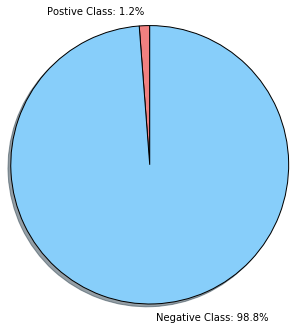
Where BP is present in an image, the BP annotations has the following characteristics:

|  |  |
| --- | --- |
| Minimum size | 2,684 pixels |
| Maximum size | 17,439 pixels |
| Average size | 7,125.74 pixels |

Groups of images are taken from the same patient. Images that come the same patient are highly correlated (see Figure 2).

## Exploratory Visualization

The pie charts below show the distribution of classes of the dataset. In terms of total number of pixels, it is very notable that negative class greatly outnumbers positive class. Majority of the images have blank annotation.



(a)

(b)

Figure : Distribution of classes. (a) shows the classes are severely imbalanced. (b) shows majority of the images has blank annotations (i.e. no annotated brachial plexus).

Images coming from the same patient have high correlation. Some images are very similar, but they are not exactly the same.

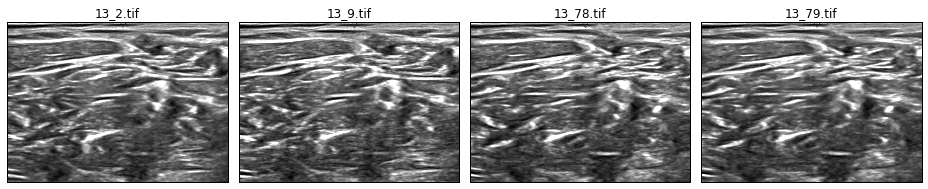
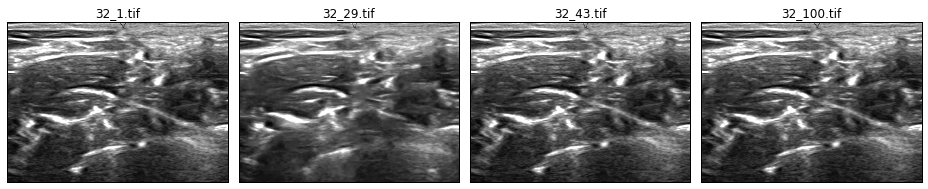
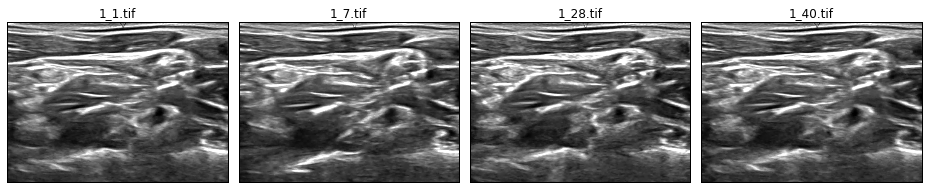


Figure : High correlation of images. The file names have the format <patient\_id>\_xxx.tif. Images coming from the same patient ID are highly correlated. (NOTE: The images are not exactly the same.)

Very similar images are expected to have similar annotations. However, due to human error during manual annotation of the dataset, there are very similar images that have conflicting annotations. One image has BP annotation, while another very similar image has none.

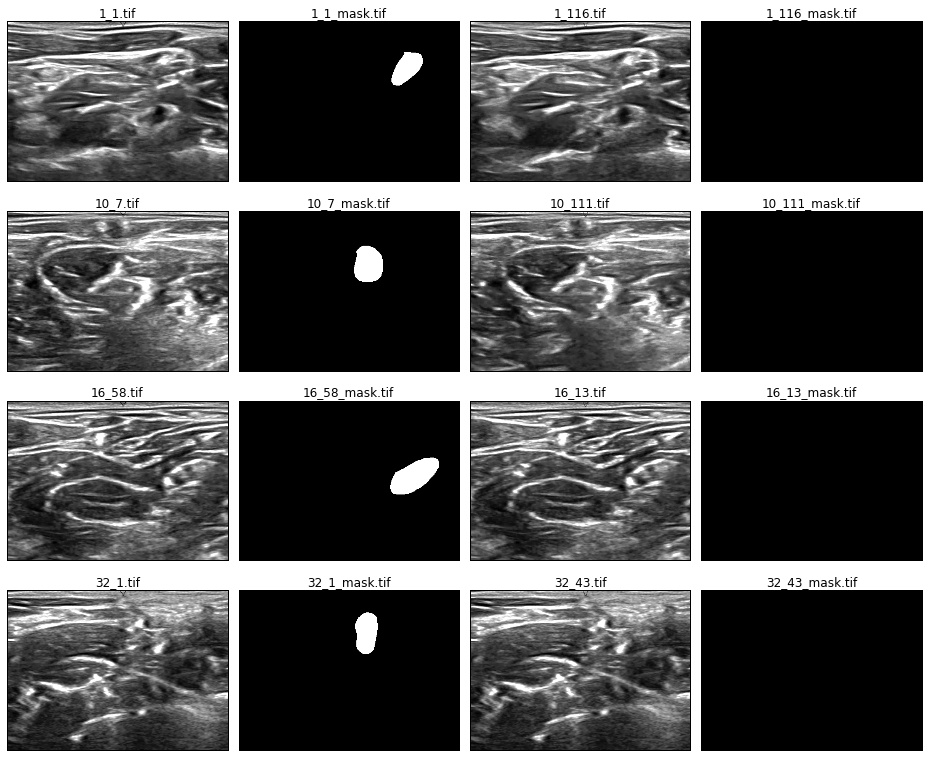


Figure : Conflicting annotations. Very similar images have conflicting annotations. One image has BP while the other has none. These are human errors during manual annotation of the dataset.

The annotations are not very accurate. There are similar images that have varying annotation; the shapes and sizes are different. However, they are located in the same general location. Some annotations of similar images vary dramatically as can be seen in images from patient 23 (see last row of Figure 4).

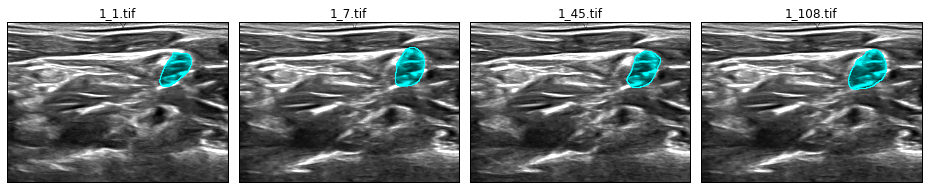
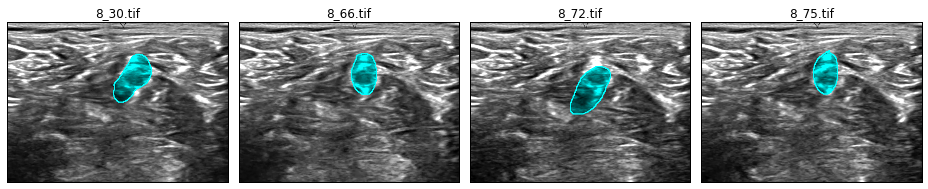
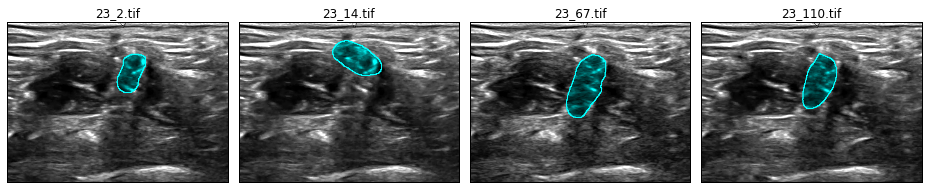


Figure : Similar images with varying annotations.

## Algorithms and Techniques

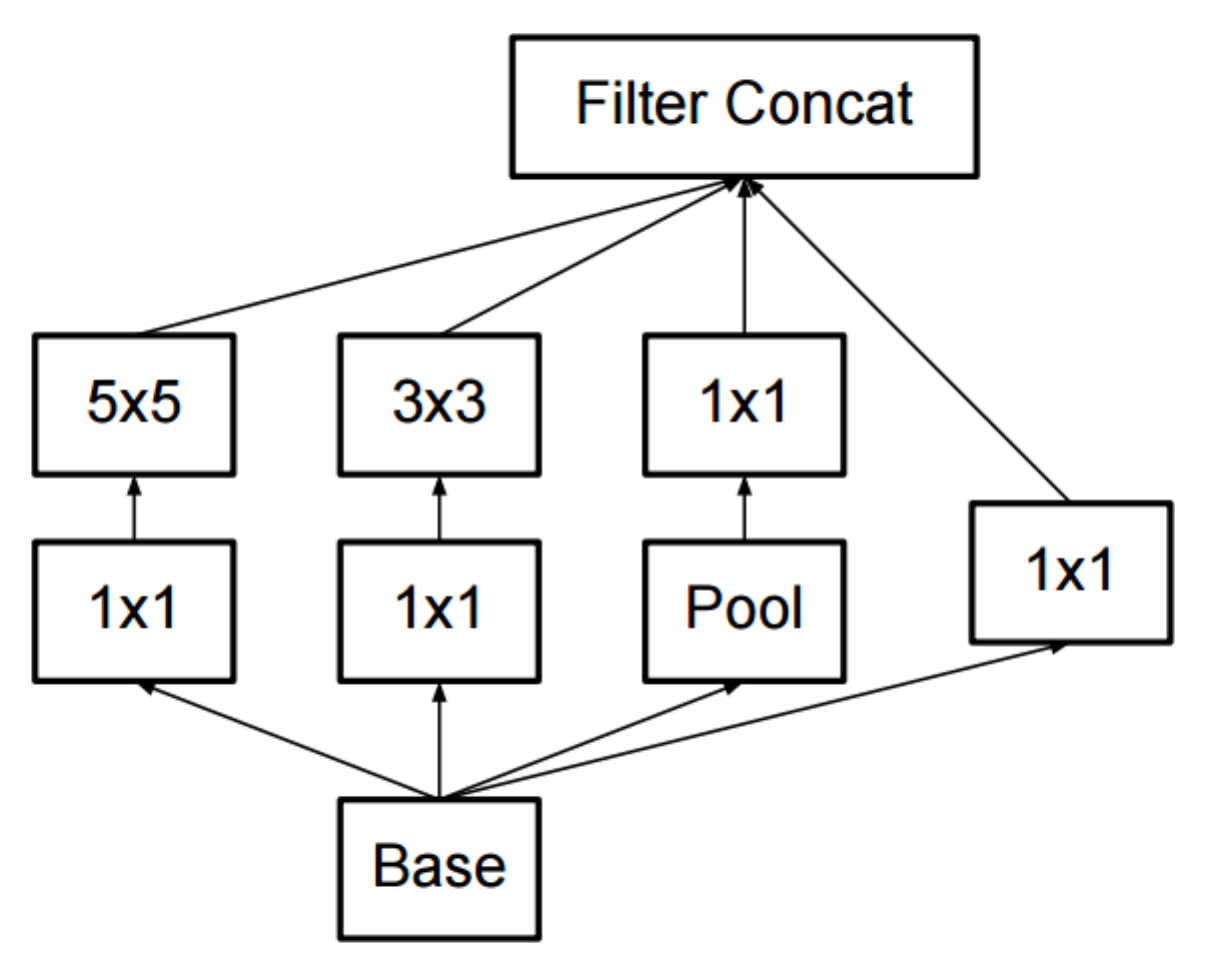
In this semantic image segmentation problem, deep neural network will be used to predict the labels for each pixel in the image. Yann Lecun introduced convolutional networks[[3]](#endnote-3) from which most state of the art image recognition techniques are derived.

In our dataset, the pixels are roughly similar all through out the image. There is no clear groupings or sharp boundaries that an untrained eye can clearly identify which part of the image is brachial plexus.

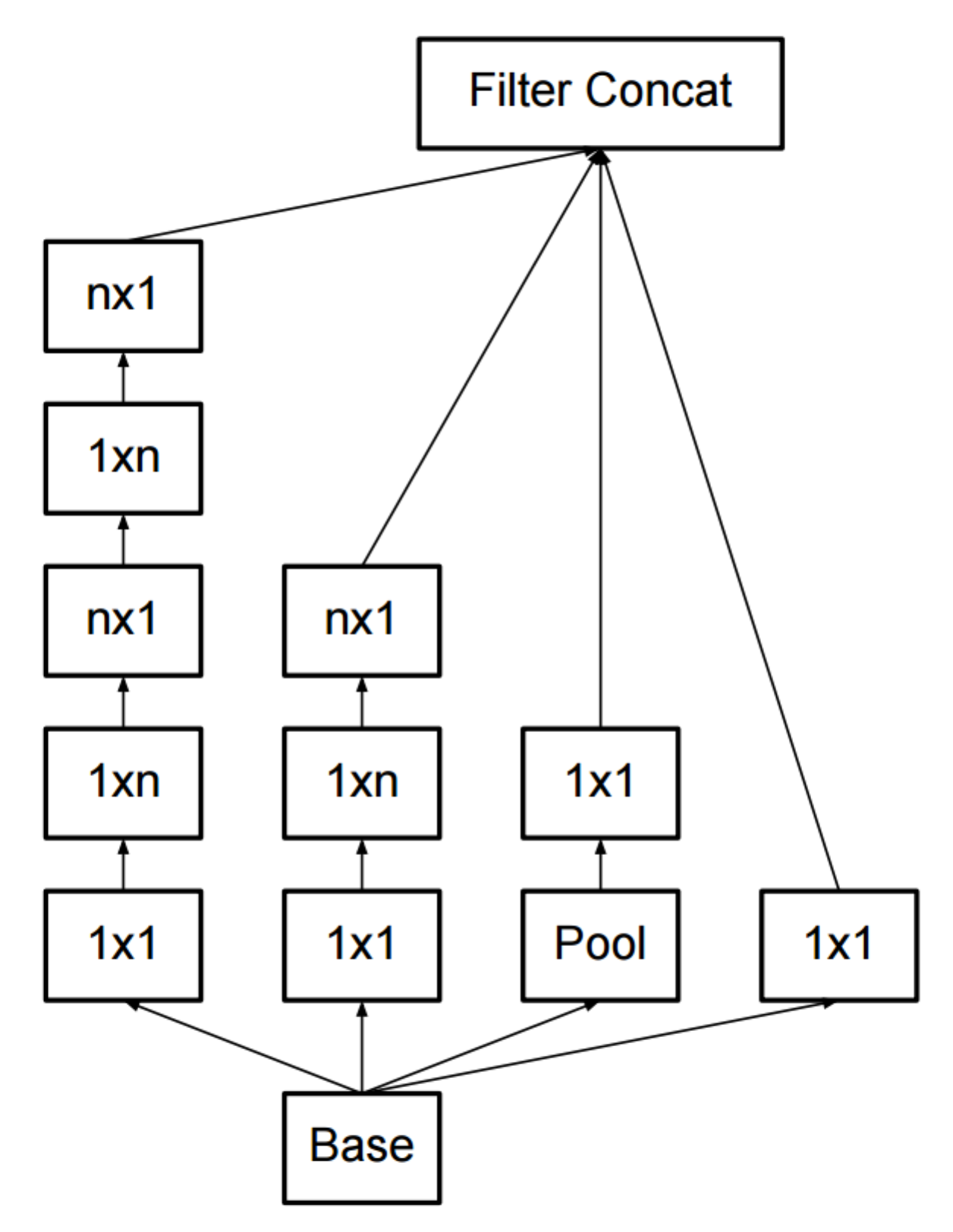
It can be theorized that, in order to classify a pixel to whether or not it belongs to a brachial plexus, the immediate surrounding pixels needs to be taken into account. The larger the patch around the target pixel, the better chance of classifying the pixel correctly. Translating this concept to convolutional network, it would mean larger patch size. However, performing convolutions with large patch is very computationally expensive.

Image pyramid[[4]](#endnote-4) can be used to generate a series of down scaled images. Using the same patch size across the images, the patch will create a contextual window around the pixel. The patch on the smaller scaled down image will have a blurred larger view version of the image, while the patch on the original full size image will have higher resolution view of the image. Image pyramid will allow the use smaller sized patch while having larger receptive field on the image.

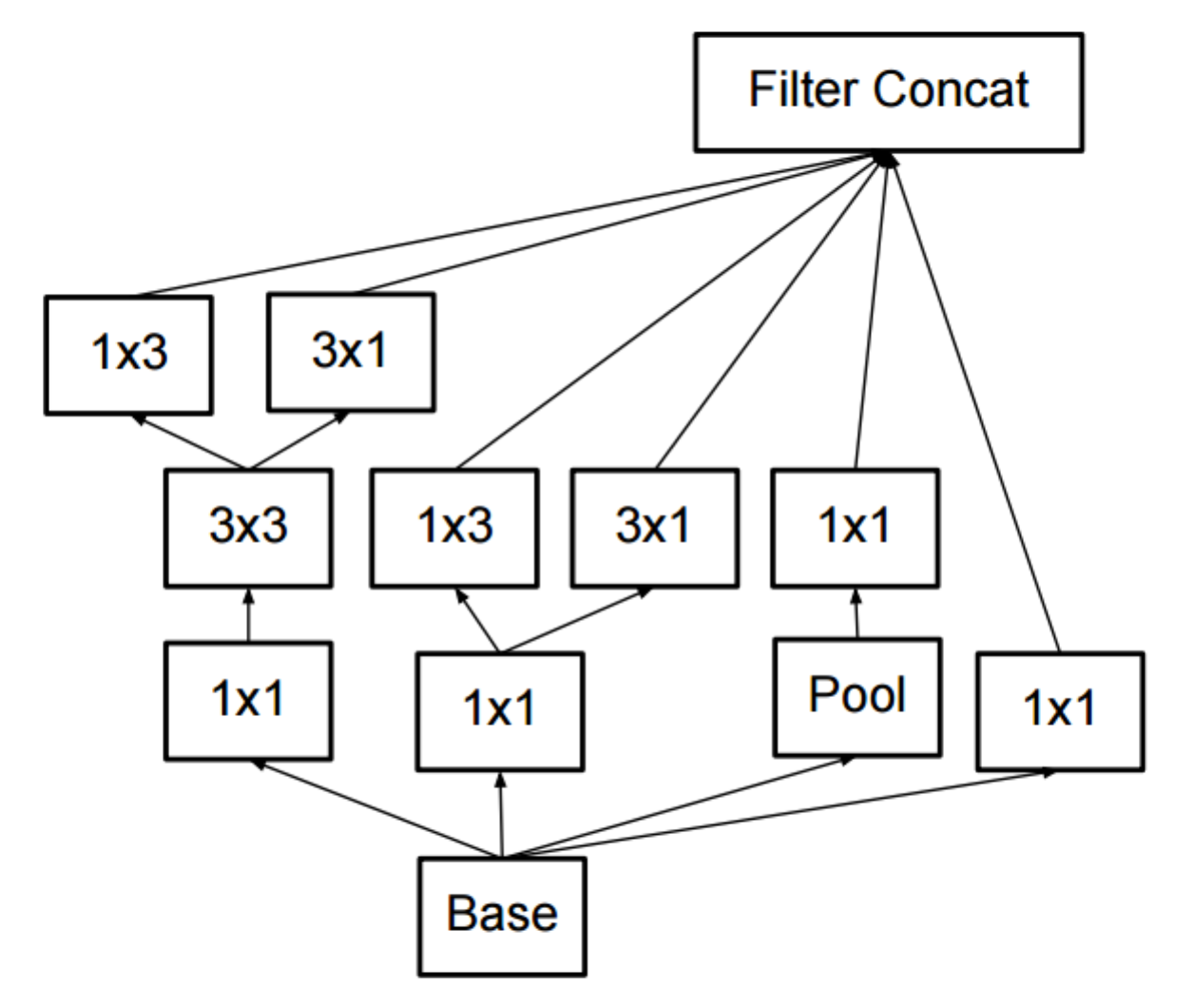
Applying inception architecture[[5]](#endnote-5),[[6]](#endnote-6) will increase the model size while utilizing computational efficiency and low parameter count. Three versions of inception modules will be used for this algorithm:



(a)



(b)



(c)

Figure : Inception modules. Version (a) is the original inception module as described in [5]. Versions (b) and (c) are newer versions of inception as described in [6]. For version (b), an n=7 is used for this project.

The overall neural network is shown in Figure 6. Image pyramid is generated, which are then fed to layers of convolutional network, followed by inception layers. The outputs of inception layers are gradually up-sampled, where needed, to the original input image size, and then concatenated. The concatenated layers are then fed to a series of convolutions. The final layer is a 1x1 convolution with depth of 2, one for each positive and negative class. The classification layer uses softmax to generate the prediction output mask.

Image

Convolutions

Inceptions

Up-samplings

7x7 convolution

7x7 convolution

3x3 convolution

3x3 convolution

3x3 convolution

3x3 convolution

32

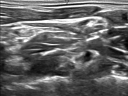
32

64

64

128

256



full size

7x7 convolution

7x7 convolution

3x3 convolution

3x3 convolution

3x3 convolution

3x3 convolution

32

32

64

64

128

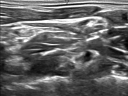
256

Inception version A

Inception version A

Upsize to full size

3x3 convolutions



½ size

Inception version B

Inception version B

Inception version B

7x7 convolution

7x7 convolution

3x3 convolution

3x3 convolution

3x3 convolution

3x3 convolution

32

32

64

64

128

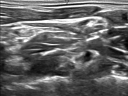
256

Upsize to ½ size

3x3 convolutions

Upsize to full size

3x3 convolutions



¼ size

7x7 convolution

7x7 convolution

3x3 convolution

3x3 convolution

3x3 convolution

3x3 convolution

32

32

64

64

128

256

Inception version C

Inception version C

Upsize to ¼ size

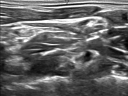
3x3 convolutions

Upsize to ½ size

3x3 convolutions

Upsize to full size

3x3 convolutions



1/8 size

downscale

downscale

downscale

input

Ful

Ful

Ful

Concatenate layers

Softmax

3x3 convolution

3x3 convolution

7x7 convolution

7x7 convolution

Inception version A

3x3 convolution

3x3 convolution

64

64

32

32

32

16

1x1 convolution

2

Output mask

Prediction

Figure : Overall architecture of the model. Image pyramid is generated from the input image, which are fed to layers of convolutions followed by inceptions layers. The outputs of the inception layers are upsized, where needed, to match the original input image size. The layers are then concatenated and then fed to another layers of convolutions to obtain the final output.

## Benchmark

To set a benchmark for measuring the performance of the model, all pixels are set to positive class and then the F1 score is computed against the ground truth of the validation set. The resulting F1 score is 0.0362.

Randomly setting the pixels to positive and negative classes resulted to a lower F1 score of ≈0.0356.

# Methodology

## Data Preprocessing

Data preprocessing involves 4 sub-processes:

1. Resizing images
2. Filtering out images without masks that are very similar to images with masks
3. Mean-center and normalize data to unit variance
4. Split stratify dataset to train-validation sets

The preprocessing code is in preprocess.py. This code handles all the needed preprocessing steps. Before running the script, download the train dataset first from <https://www.kaggle.com/c/ultrasound-nerve-segmentation/download/train.zip>. Extract the zip file and put the images (.tif files) in the data directory <root project DIR>/data/train/.

##### Resizing Images

The ultrasound images are big files and are noisy. To reduce the noise and have faster training, the images are downsized to 96x128 pixels using inter-area interpolation.

##### Filtering Images

As shown in Exploratory Visualization section, several images are very similar but have conflicting masks. This will greatly affect negatively the learning process. To mitigate the negative impact, images without masks but have very similar images that has masks are filtered out.

The similarity of images are measured using normalized cross-correlation with a threshold of 0.7 (similarity threshold can be configured in config.py).

##### Normalization

To have better training performance, the dataset is centered at zero mean and normalized to unit variance. This is done by subtracting the mean and then dividing by the standard deviation of the dataset.

##### Splitting

The dataset is split to 80% train and 20% validation sets. The split is stratified based on the presence of mask. The images with the same patient ID are highly correlated, so split is also stratified based on patient ID.

After running preprocess.py, the following directory and files should be generated:

|  |  |
| --- | --- |
| Files/directory | Description |
| data/train\_xs96/ | Folder containing the filtered and resized images |
| data/train\_set.npz | Train set in numpy readable format |
| data/validation\_set.npz | Validation set in numpy readable format |
| data/train\_stats.pkl | Pickle file that contains basic statistics about the train set |

## Implementation

[Tensorflow](https://www.tensorflow.org/) is used as the main deep learning library. The entry point of the code is train.py, where the model is also implemented. The train and validation dataset (output generated from preprocess.py) are fetched. The model function and the dataset are passed to run\_training() function which handles the execution of the training process.

The run\_training() function, which is located in engine.py, reads the latest checkpoint file. This allows the training to resume from previous runs in case it was interrupted. The script saves a checkpoint at the end of every epoch.

The \_compile\_model() function generates the tensor graph for training. It uses the supplied make\_model() function to create logits. It also handles generating of tensor graph for the loss function and the optimizer.

The loss\_and\_predict() function computes for the loss value using cross entropy[[7]](#endnote-7). It flattens the logits and target labels tensors before computing the softmax[[8]](#endnote-8) and cross entropy value.

Since both computations for prediction and cross entropy requires flattening of the logits, the same segment of the tensor graph is used for this operation. For efficiency, both loss and prediction is returned by a single call of this function.

Adam optimizer[[9]](#endnote-9) is used for learning with exponential learning rate decay. The initial learning rate, decay rate, and decay steps are set in config.py. Gradient clipping is applied to stabilize the loss function. The gradient clipping value is also set in config.py.

The do\_training() function executes the training loop. Each loop constitutes 1 epoch. Each epoch runs several iterations, where every iteration step constitutes 1 mini batch of training dataset. The training dataset is divided into mini batches. At the beginning of each epoch, the train dataset is shuffled so each mini batch contains different images every epoch. At the end of each epoch, validation is performed using similar process as training, except it is using validation dataset. The number of epochs to run mini batch sizes for training and validation are set in config.py.

In every iteration step, the number of correct and incorrect predictions (true/false positives, true/false negatives) is recorded for metrics gathering. At the end of every epoch, F1 scores for training and validation are computed using np\_f\_beta\_score() function which can be found in lib/stats\_tools.py. The scores are saved in output/training\_log.csv file.

Checkpoint files are saved for every epoch at output/checkpoints/. Since checkpoint files can easily take up disk space, a limit is set as to the number of check points to keep, which can be configured in config.py. Older checkpoints are automatically deleted when the limit is reached. The checkpoint point file that has the highest F1 validation score will not be deleted.

The training ends when the number of epochs to run is exhausted or when the user press CTRL+C in the command line. Running train.py again will resume the training from the last saved checkpoint.

Training takes approximately ≈17 minutes per epoch on Amazon AWS g2.2xlarge instance with GPU support. A pre-trained model can be downloaded at [TODO: link goes here] ran 45 epochs for 13 hours.

## Refinement

The model hyper parameters were adjusted iteratively. The initial model was trained using default parameters and learning rate of 0.01. The loss was dropping too fast within the first epoch. Several learning rate values were tried out, decreasing in order of magnitude, until a gradual descend of loss was obtained.

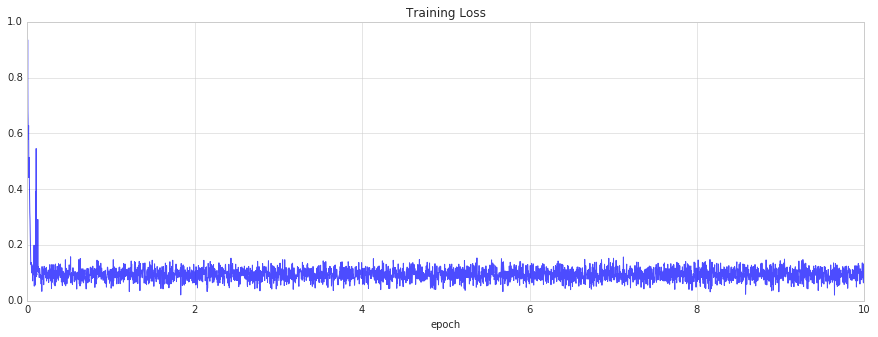


Figure : Training Loss with learning rate set to 0.01. The learning rate dropped too swiftly in just a few iterations.

Due to class imbalance, the model preferred to always predict negative class. To correct this, weighted class was applied using median frequency balancing[[10]](#endnote-10), where the weight of a class is the ratio of the median of class frequencies in the entire training set divided by the class frequency. The resulting weights are normalized so that it will be between 0.0 and 1.0. This implies that the positive class will have a weight of 1.0 (since the positive class has smaller frequency) while the negative class will have a weight << 1.0 (approximately ≈0.03). The class weights are multiplied to the logits before the loss is computed.

The model overfitted very easily. Dropout layers were then applied at the end of the inception layers and at the concatenated layer. Dropout keep rate was gradually adjusted from 0.8 down to 0.5.

To further improve the overfitting issue, L2 regularization was applied. Several regularization strengths were tried out starting from a weak value of 0.000001, gradually increasing in order of magnitude until the model underfits.

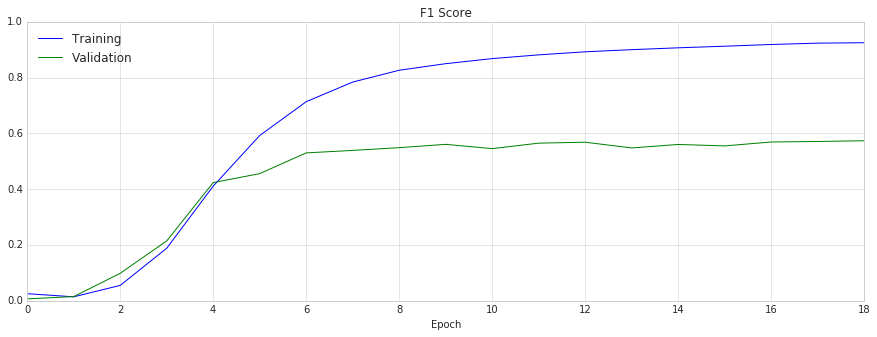


Figure : Overfitting model. There is a big gap between training and validation loss, which indicates that the model is overfitting.

# Results

## Model Evaluation and Validation

The learning rate that produced reasonable result is around 0.00001. Higher learning rate converged to higher loss value while with lower learning the training was not able to converge after several epochs. Exponential learning rate decay of 0.98, decayed at every epoch, helped stabilized the loss convergence at the later stage of training.

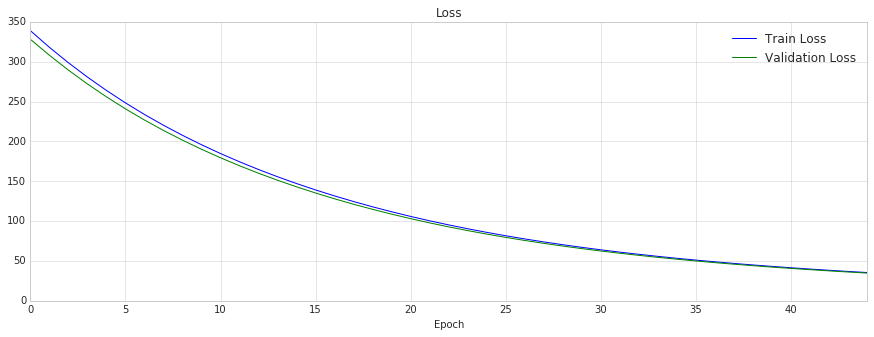


Figure : Loss graph of the final model

With limited training data, the model very easily overfits. The model had high F1 score on training set, but performed badly on validation set. Applying dropout and L2 regularization minimized overfitting. Dropout keep rate of 0.5 and L2 regularization strength of 0.005 yielded reasonable result, which reduced the gap between training and validation F1 scores.

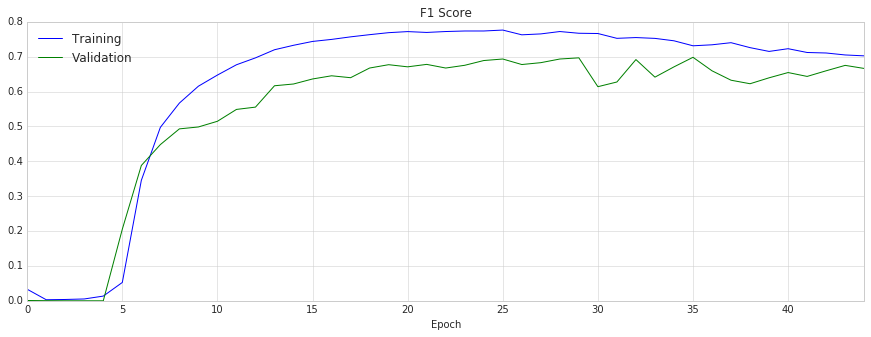


Figure : F1 score graph of the final model. The highest peak of the validation set is at epoch 35.

The final model was chosen using early stopping. The checkpoint with highest validation score was chosen as the final model. The validation set uses images that were not included during training. It generally has lower score compared to training, which reflects more accurate performance metric of the model.

To verify the robustness of the model, inference was run using the [test dataset](https://www.kaggle.com/c/ultrasound-nerve-segmentation/data). This dataset does not have accompanying ground truth that can be compared with to produce a quantitative measure. It is rather difficult for untrained eyes to make a judgment as to whether or not the predicted annotations are accurate. As a non-medical professional, we can look over and again the training set and pick up general pattern of nerve structures. However, this would still be not very reliable and consulting with a trained medical professional would be best to verify the result. Some samples predictions using the test dataset are shown in Figure 13 for qualitative assessment.

## Justification

The final model yielded an F1 score of 0.698 on validation set which is significantly higher than the benchmark 0.0362.

Given the training data have a good portion of inaccuracies, the predictions of the model is sufficient enough to point the location of nerve ending structure, or the absence thereof. The predicted area of the image shows the general location of brachial plexus.

# Conclusion

## Free-form Visualization

Some images from validation and test set that were not used in training were tried out. It can be seen that the model performs well on most images. However, there are also cases where it misses. The area highlighted in cyan is the ground truth while the area outlined in yellow is the prediction produced by the model.

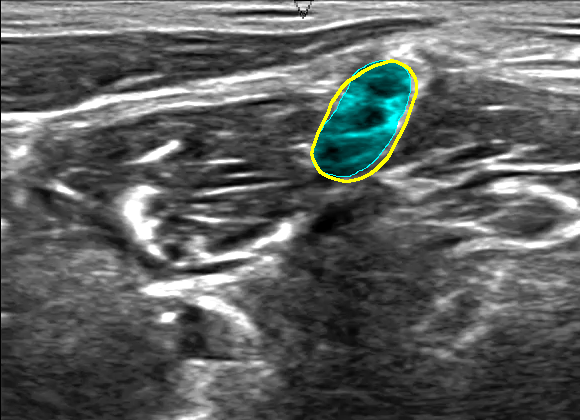
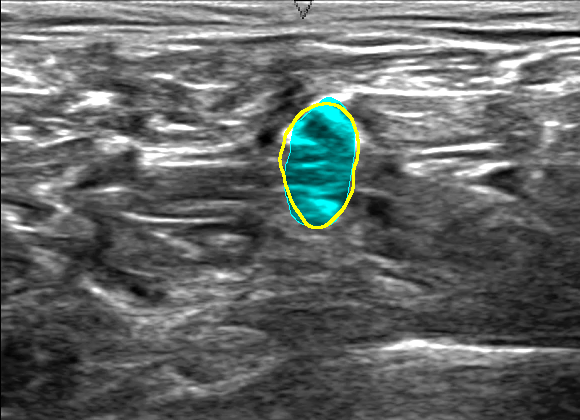
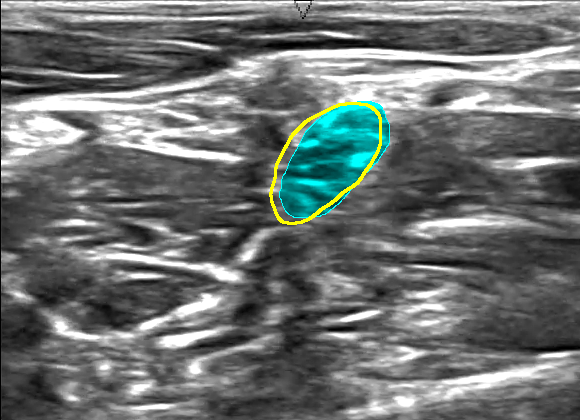
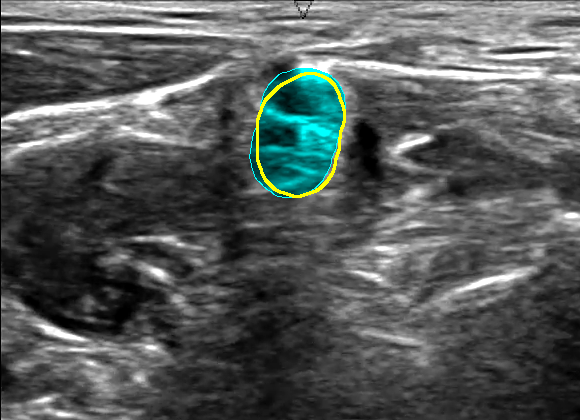


Figure : Good performance of the model. The area highlighted in cyan is the ground truth while the area outlined in yellow is the prediction of the model. The model predictions are almost identical to the ground truth.

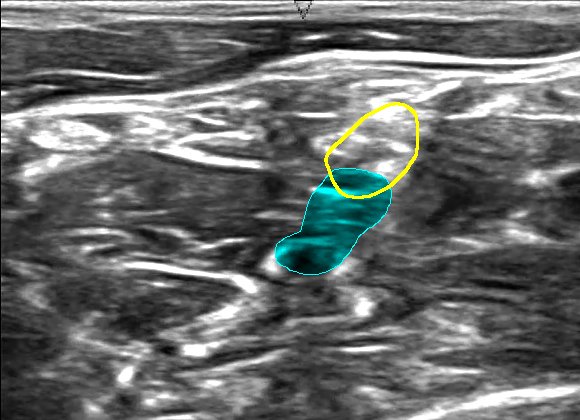
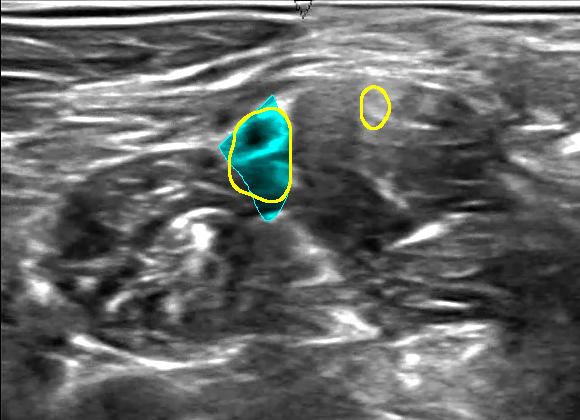
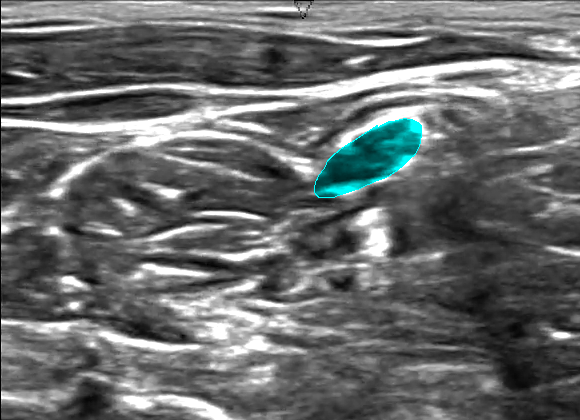
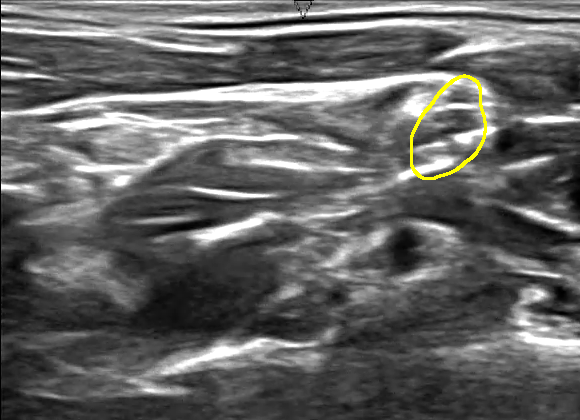


Figure : Missed out predictions. From left to right: (a) the model prediction mostly missed the ground truth, (b) the model hit the ground truth but also annotated an extra area, (c) the model was not able to detect brachial plexus but there actually exists, and (d) the model was able to find brachial plexus where human annotator missed out.

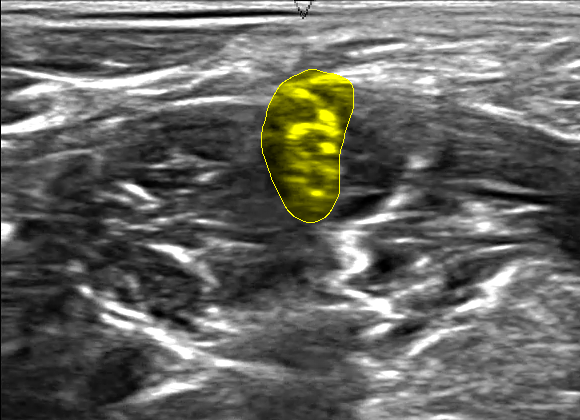
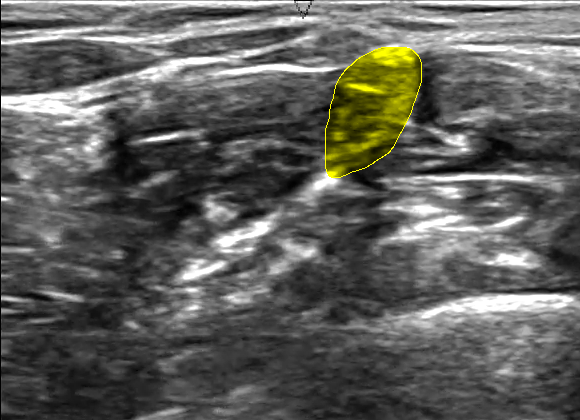
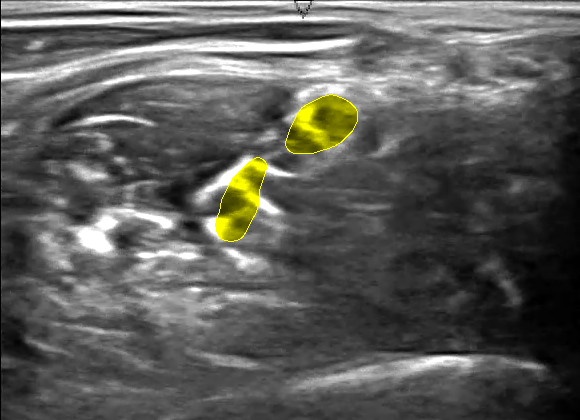
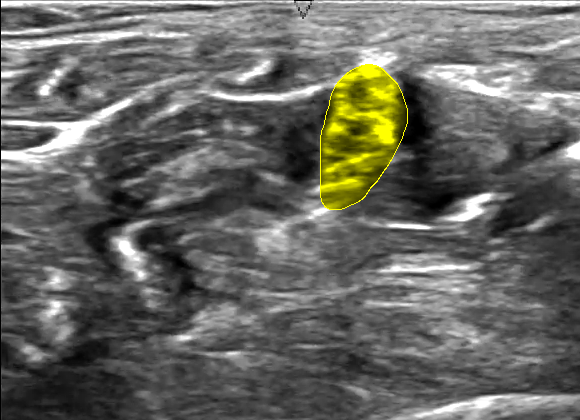


Figure : Inference on test dataset. The yellow highlights are the prediction of the model. The test dataset has no accompanying ground truth to compare the results with. Consultation from trained ultrasound professional is needed.

## Reflection

The entire project revolves around semantic image segmentation. Reaching to a final solution involved the following:

1. Finding relevant dataset
2. Preprocessing the images
3. Implement a neural network
4. Train the neural network using the dataset, fine-tuning the parameters until a good result is obtained
5. Implement a script that utilizes the trained model to annotate brachial plexus in ultrasound images

In completing the final model, I have learned several exciting things that previously not known to me. These include deep learning in general and more specific aspects of it including convolutions, inceptions, Tensorflow, image processing, among other things.

As a non-medical person, the most challenging part of the project is not having knowledge on what brachial plexus exactly looks like. Even after going through all the sample images, it is still difficult to identify the nerve structures in the ultrasound images without medical training.

It is interesting to note that machine learning model was able to identify a nerve structures after training from set of images within hours, while a typical human eyes won’t still be able to figure out the pattern from the same set of images.

## Improvement

The accuracy of the result can be greatly improved with more training data that has higher accuracy. In addition, data augmentation may also he improve the result.

Using a different neural architecture, such as U-Net[[11]](#endnote-11), may have faster inference time. Using other deep learning library, such as Theano or Torch, may improve training time.

Significant improvement on training time would be very useful to make the model run on real time application in video input, or run natively in less powerful devices such as on mobile platform.

1. <https://en.wikipedia.org/wiki/Brachial_plexus> [↑](#endnote-ref-1)
2. <https://en.wikipedia.org/wiki/F1_score> [↑](#endnote-ref-2)
3. Y. LeCun et al. “Gradient-based Learning Applied to Document Recognition.” *Proc. Of the IEEE,* <<http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf>> (1998). [↑](#endnote-ref-3)
4. C. Farabet et al. “Learning Hierarchical Features for Scene Labeling.” <<http://yann.lecun.com/exdb/publis/pdf/farabet-pami-13.pdf>> (2013). [↑](#endnote-ref-4)
5. C. Szegedy et al. “Going deeper with convolutions.” *arXiv preprint arXiv:1409.4842v1* (2014). [↑](#endnote-ref-5)
6. C. Szegedy et al. “Rethinking the Inception Architecture for Computer Vision.” *arXiv preprint arXiv:1512.00567v3* (2015). [↑](#endnote-ref-6)
7. <https://en.wikipedia.org/wiki/Cross_entropy> [↑](#endnote-ref-7)
8. <https://en.wikipedia.org/wiki/Softmax_function> [↑](#endnote-ref-8)
9. D. Kingma, J. Ba. “Adam: A Method for Stochastic Optimization.” *arXiv preprint arXiv:1412.6980v8* (2015). [↑](#endnote-ref-9)
10. D. Eigen, R. Fergus. “Predicting Depth, Surface Normals and Semantic Labels with a Common Multi-Scale Convolutional Architecture.” *arXiv preprint arXiv:1411.4734v4* (2015). [↑](#endnote-ref-10)
11. O. Ronneberger, P. Fischer, T. Brox. “U-Net: Convolutional Network for Biomedical Image Segmentation.” *arXiv preprint arXiv:1505.04597v1* (2015). [↑](#endnote-ref-11)