|  |  |
| --- | --- |
| 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”[1]. 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) from ultrasound images. The training dataset is manually annotated to show areas with BP.

Each pixel in the ultrasound images can either belong to BP or not. A computer vision process called semantic image segmentation [2] can be applied to assign each pixel to which class it should belong. A deep learning model can be trained from the dataset to learn to identify each pixel.

This project therefore aims to classify each pixel of ultrasound images to either positive class, which represents it belongs to BP, or negative class otherwise. A collection of positive class pixels would show the area in ultrasound images where BP is present.

## Metrics

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 [3], 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 score in terms of Type I and Type II errors is given as:

where, for F1 measure, .

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

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 two very similar images; one image has positively identified BP, however the other one has none (see figure xx). 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 xx).

The classes are greatly imbalanced; consisting 1.2% positive class and 98.8% negative class.

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 xx)

Images are noisy

Annotations are inaccurate:

- Similar images, conflicting annotations

- Similar images, varying annotations

Imbalance classes; - >> +

Number of images with annotations

Number of images without annotations

Minimum annotation size image-wise, where annotation is present

Maximum annotation size image-wise, where annotation is present

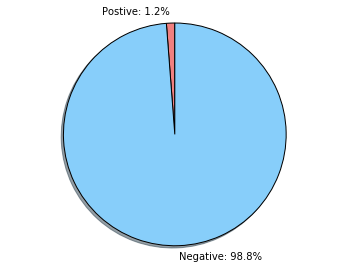
Average annotation size image-wise, where annotation is present

Images from the same patient ID are highly correlated

Patches are similar all throughout

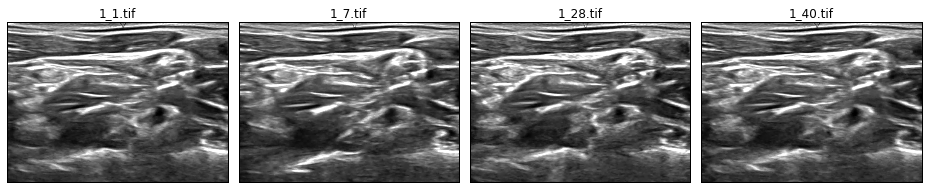
## Exploratory Visualization

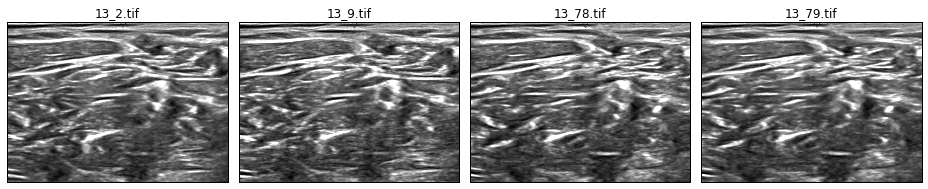
Pie chart for +/- classes, to show imbalance

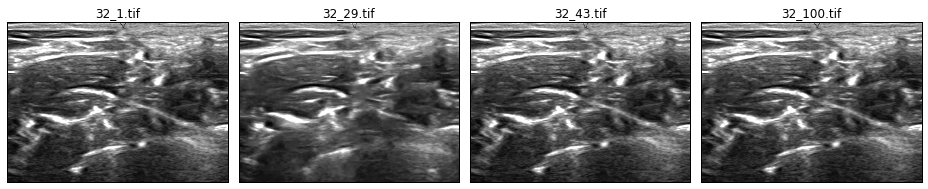


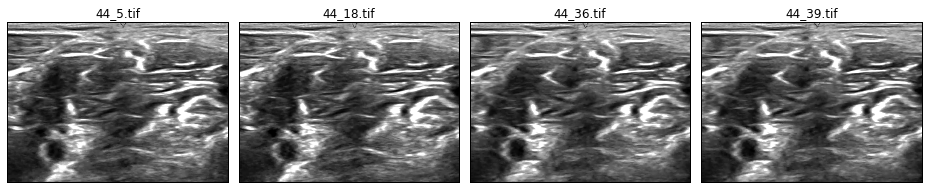
TODO: Number of images that has no mask

Correlated images per patient

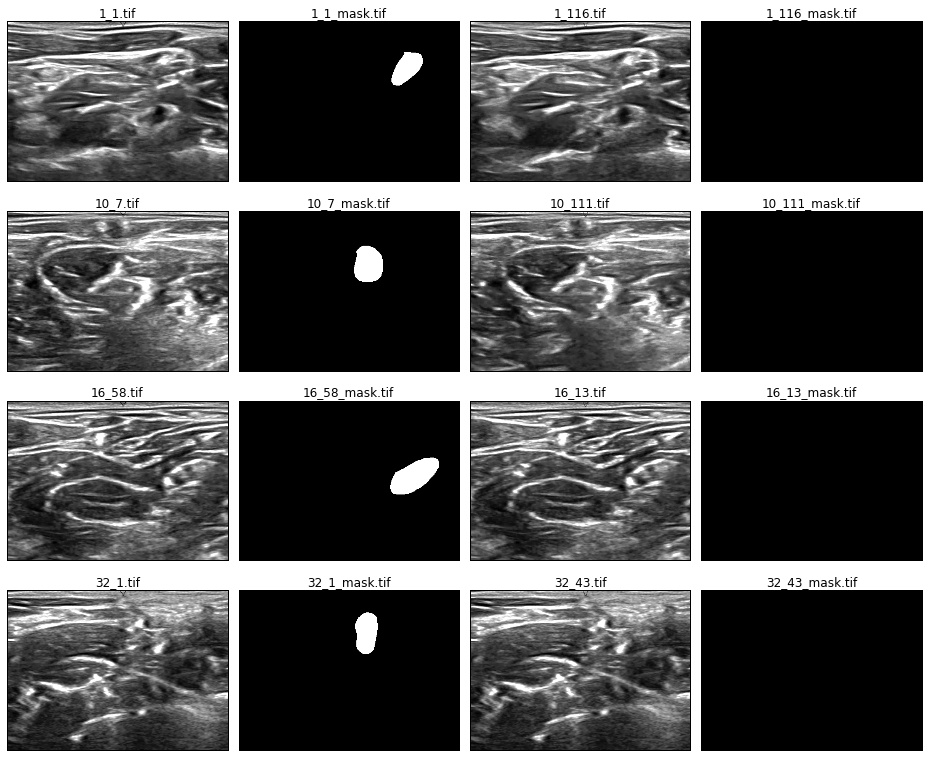




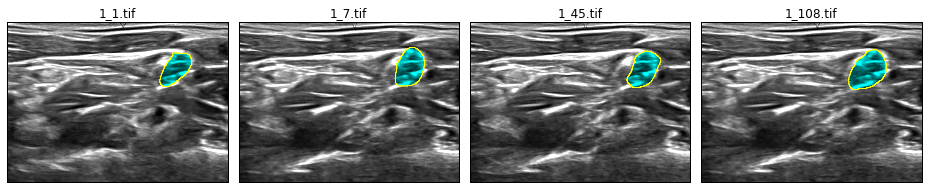


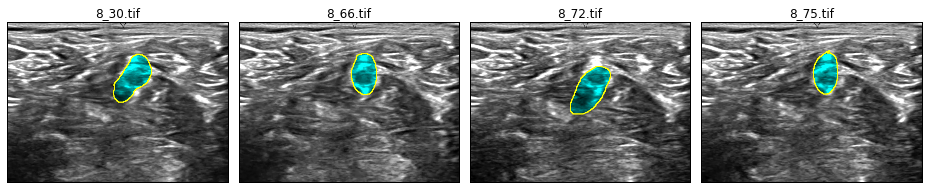


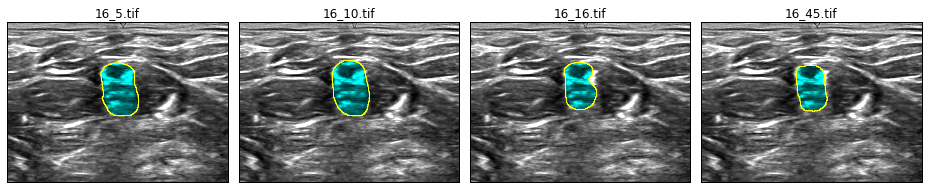
Similar images, conflicting annotations

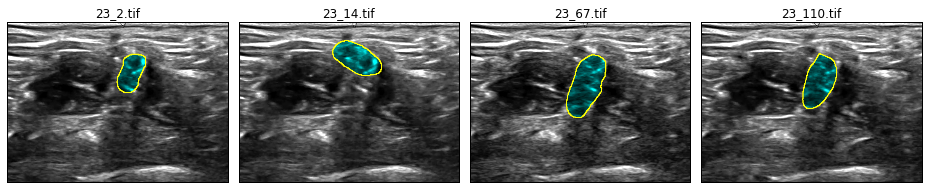


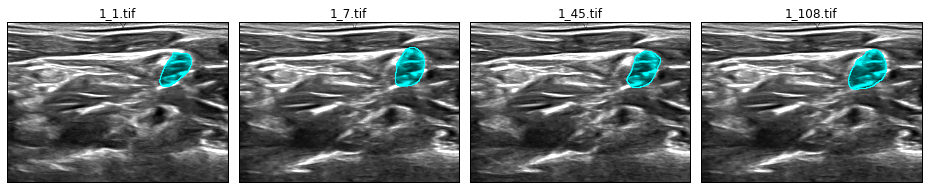
Similar images, varying annotations

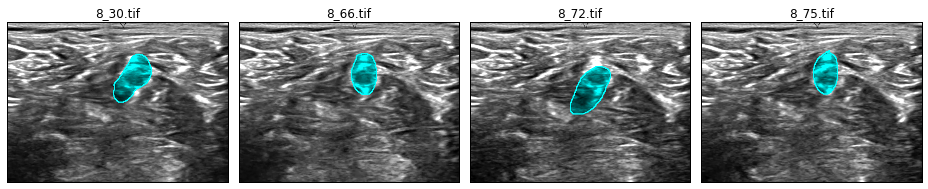


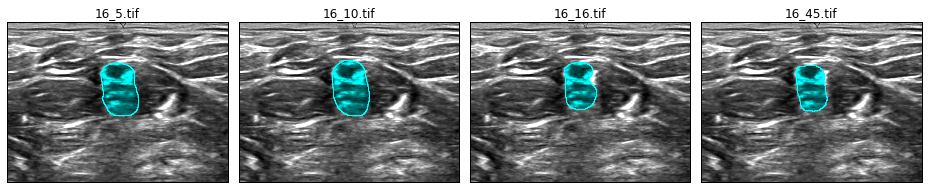


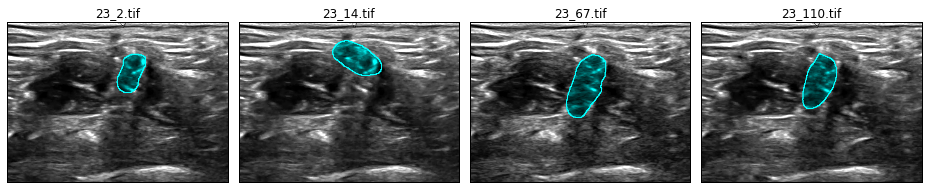












Average annotation location

## Algorithms and Techniques

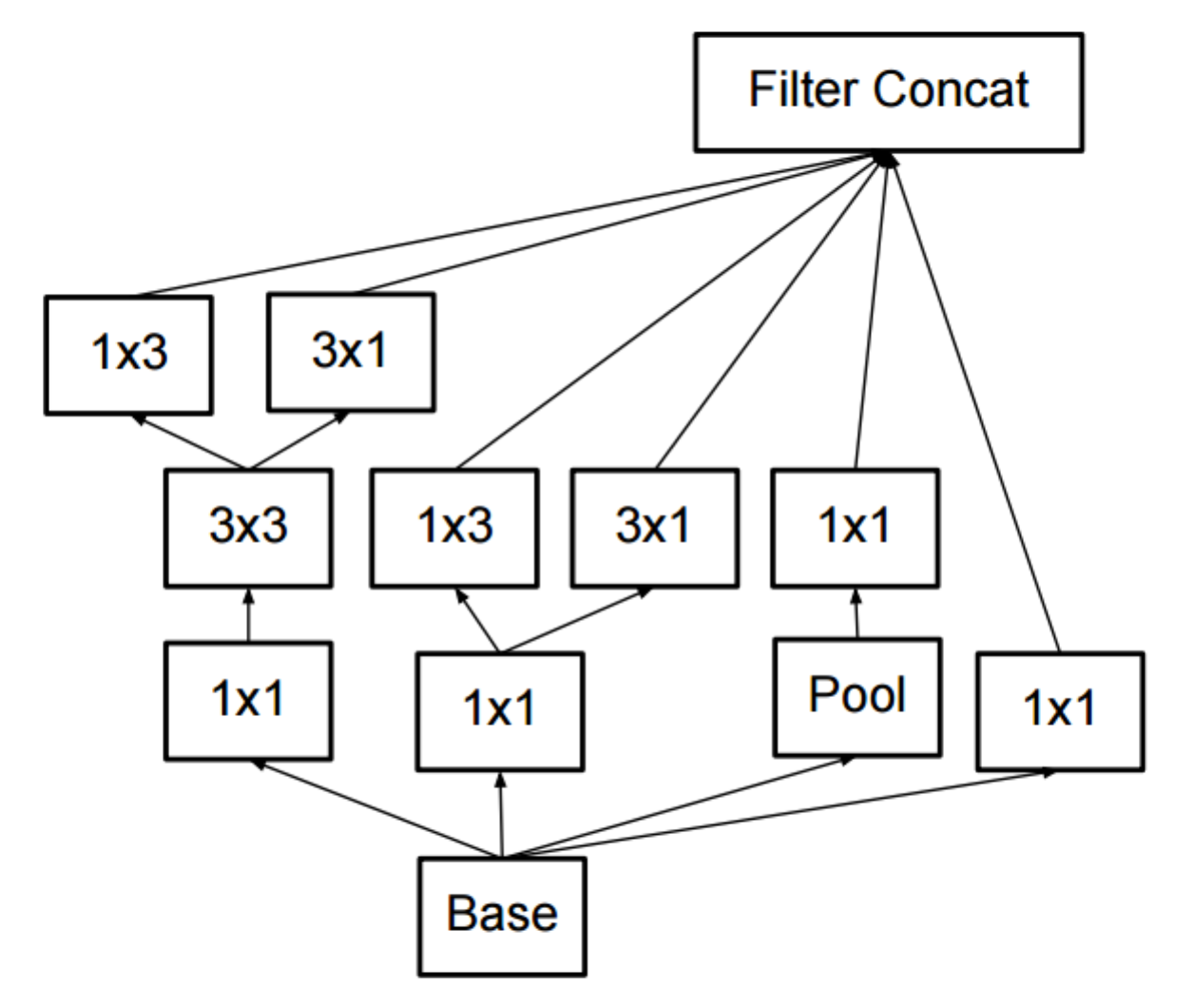
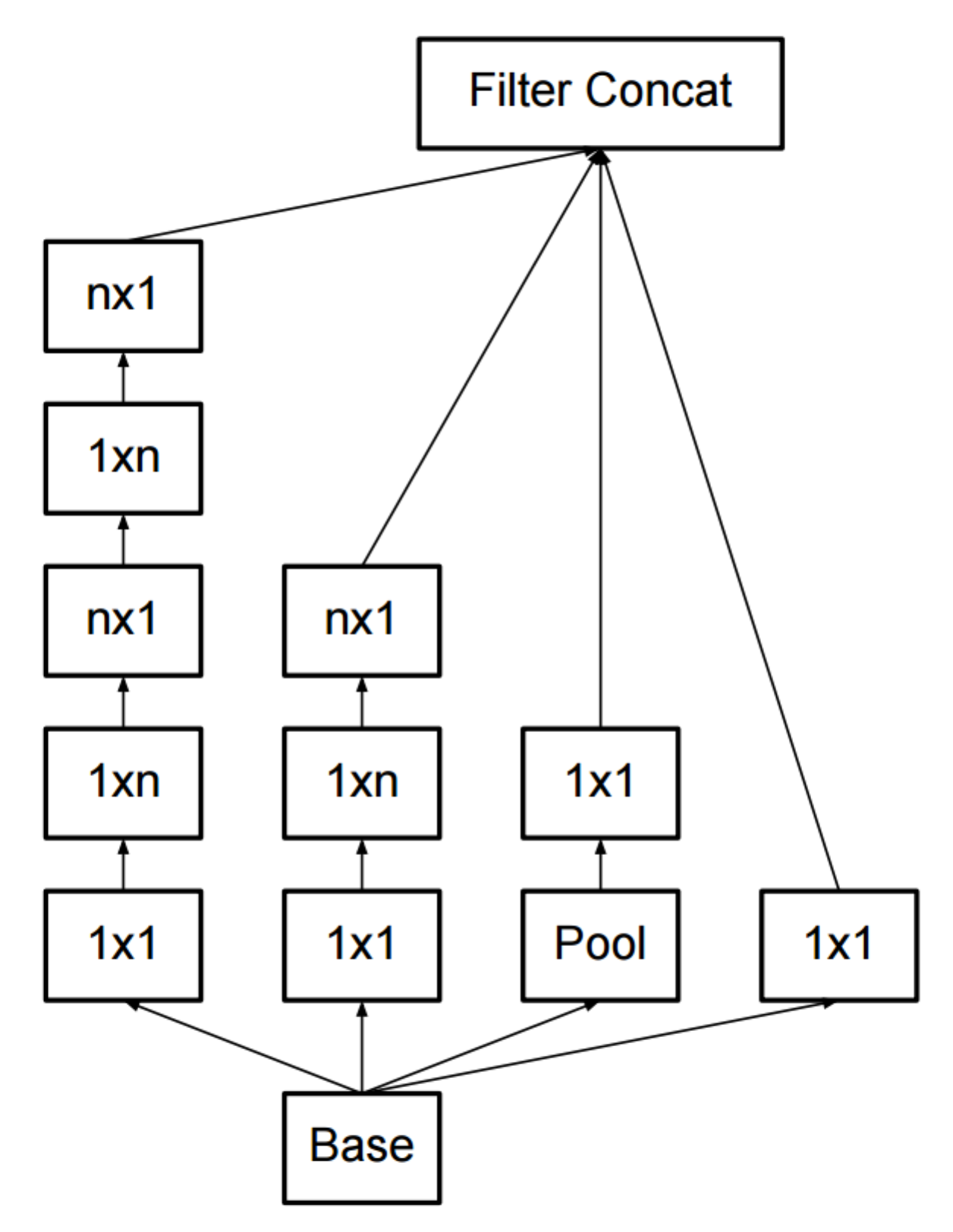
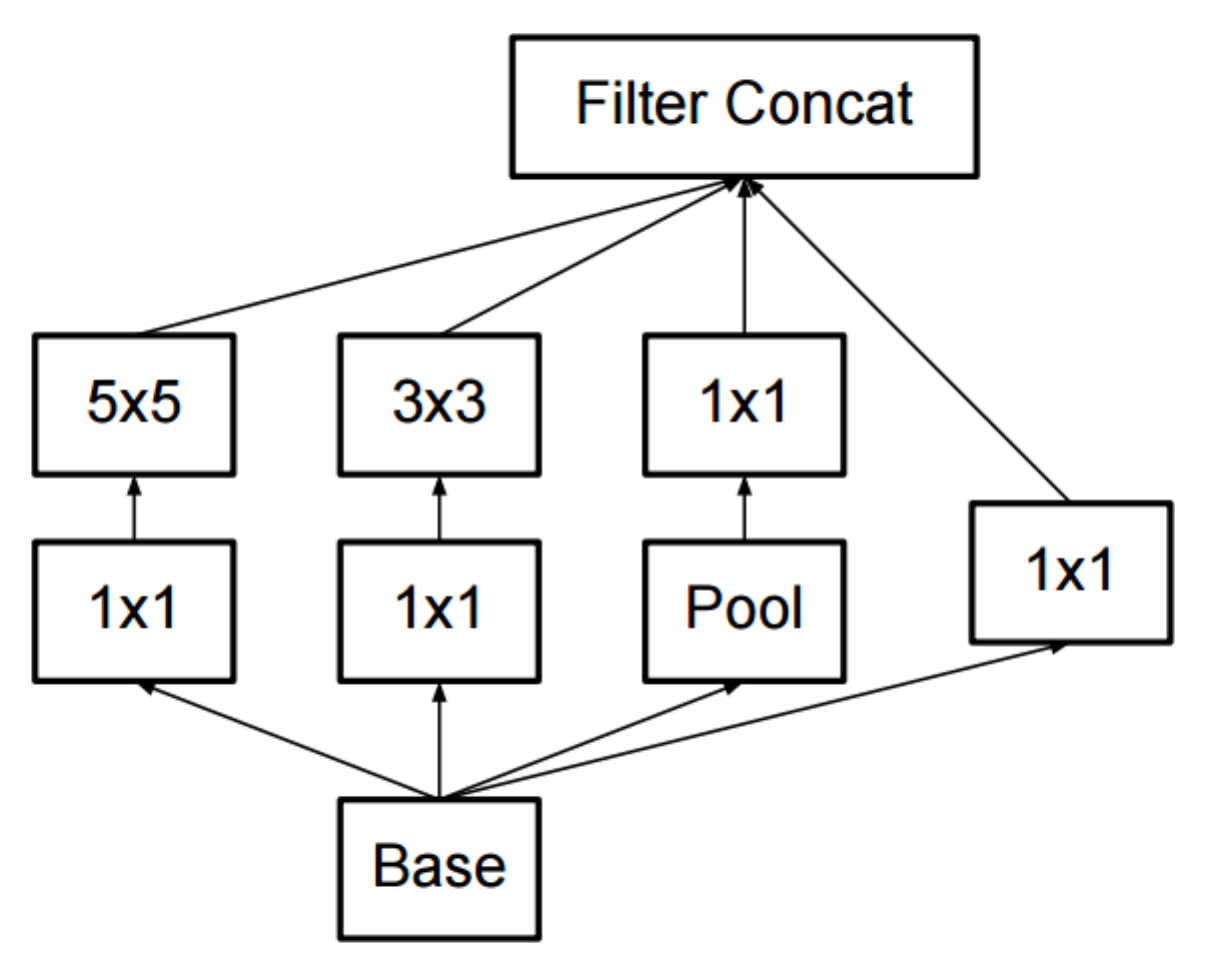
In this semantic image segmentation problem, deep neural network will be used to predict the labels for each pixel in the image. Y. Lecun introduced convolutional networks[] from which most state of the art image recognition techniques are derived.

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[] can be used to generate a series of scaled down 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 [] will increase the model size while utilizing computational efficiency and low parameter count. Three versions of inception modules will be used for this algorithm:



The overall neural network is shown in figure (). 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 it 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.

Segmentation:

* ??? Only use images with annotation present
* Convolution
* U-net

Image

Convolutions

Inceptions

Up-samplings

7x7 convolution

7x7 convolution

3x3 convolution

3x3 convolution

3x3 convolution

3x3 convolution

32

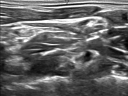
32

64

64

128

256



full size

7x7 convolution

7x7 convolution

3x3 convolution

3x3 convolution

3x3 convolution

3x3 convolution

32

32

64

64

128

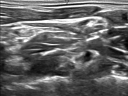
256

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

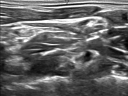
256

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

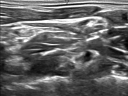
3x3 convolutions

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

???

Classification:

* Classify whether or not BP is present
* Use images with or without annotation present
* Convolution
* Inception

Cross-validation???

Ensemble???

## 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.

Train-validation F1 graph

TODO: Use Kaggle empty bench mark

# 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 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.

##### Normalization

As with many deep learning problems, centering the dataset to zero mean and normalizing to unit variance are important in order for the learning model to more likely converge to the minima faster.

##### 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 |

Resize images to 96 x 128

Filter out very similar images without annotations per patient

Mean-center and normalize by standard deviation

Stratify split data set to train-validation set per patient per BP presence (80-20)

## Implementation

Tensorflow 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, the output of 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 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[]. It flattens the logits and target labels tensors before computing the softmax 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 [] 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 can take approximately ≈17 minutes per epoch on Amazon AWS g2.2xlarge instance with GPU support. A pre-trained model can be downloaded at [link goes here] ran xxx epochs for xxx hours.

Exponential learning rate decay

Adam optimizer

Gradient clipping

Loss function uses cross entropy

Metrics

## 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 was tried out, decreasing in order of magnitude, until a gradual descent of loss was obtained.

TODO: Show graph with high learning rate

Due to class imbalance, the model preferred to always predict negative class. To correct this, weighted class was applied using median frequency balancing [], 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.

TODO: Show graph with always predicting negative class

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.

TODO: Show graph with severe case of overfitting

Adjust learning rate

Adjust class weights

Apply dropout

Apply L2 regularization

Exponential learning rate decay

# 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.

TODO: Show final loss graph

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.

TODO: Show final F1 graph

The final model was chosen using early stopping. The checkpoint with highest validation score 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] from Kaggle. 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 () for qualitative assessment.

LEARNING\_RATE = 1e-5

LEARNING\_RATE\_DECAY = 0.98

DECAY\_STEP = 1

DROPOUT\_KEEP\_RATE = 0.5

GRADIENT\_CLIPPING = 2.0

L2\_LAMBDA = 1e-2

Train-validation loss graph

Train-validation F1 graph

## Justification

The final model yielded an F1 score of 0.xxx on validation set which is significantly higher than the benchmark 0.0362.

TODO: Kaggle empty benchmark

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.

Train-validation F1 graph

# Conclusion

## Free-form Visualization

Ground truth vs prediction images

* Sample good performance
* Sample poor performance

Images from validation set

* Good performance (presence and absence of BP)
* No ground truth annotation but BP is found
* Sample poor performance

Images from test set

First layer weights

## Reflection

## Improvement

More accurate training data

Use ensemble???

References???