Machine Learning Engineer Nanodegree

Capstone Project

Eddy Shyu August 23, 2017

I. Definition

Project Overview

This project is a computer vision and object detection task to help identify dangerous objects during airport security screening. The data is provided by the Transportation Security Administration (TSA) of the United States, and includes 3-dimensional images of body scans, some which include objects that we want to detect automatically. In addition to identifying these objects, the goal is to identify where on the body these objects are located. The TSA defines 17 body regions on which objects can be detected. Examples of these regions include the left forearm, the right waist, left ankle, etc.

The goal is to develop a system that views a body scan, then correctly predicts the probability that a threat is hidden on each of the 17 body regions.

This project is a Kaggle competition that is hosted by the U.S Department of Homeland Security. The data is available through the Kaggle website in multiple formats. Two of the data formats are 2D slices of the 3D image, at various angles, as if a camera is moving around the person and taking x-ray images at equally spaced intervals. The smaller data set (.aps) has 16 angles. The larger data set has 64 angles. A third data format consists of 3 coordinates, and looks like several 2D slices stacked on top of each other. Instructions for getting permission and accessing the data on a google cloud platform "bucket" are found here.

Problem Statement

My goal is to train a convolutional neural network that can view a new body scan and output a prediction between 0 and 1 for each of 17 body regions. A prediction close to '1' should be output when there is a dangerous object present in the image, for that body region. An prediction close to '0' should be output when that body region has no dangerous object present.

Since we have 3D data that is provided as multiple 2D slices, I will train a multi-view convolutional neural network. The multi-view CNN reuses the same filter windows (reuses the weights of the convolutions) for all 2D slices. Each convolutional layer has an output for each 2D slice, so for instance, if the data has 16 2D slices at different angles around the body scan, each convolutional layer will have 16 different outputs.
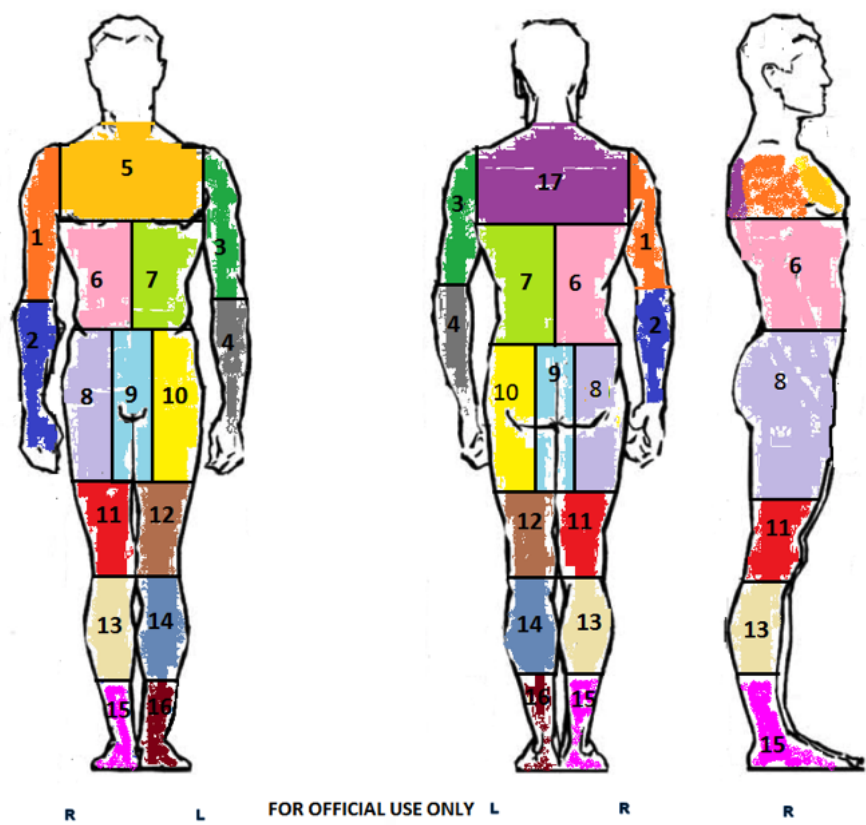
Figure 1: body zones

The multiple outputs are then combined with an aggregate function (in this case, a maximum), so that the 16 outputs are combined into one output with the same dimension as a single 2D slice.

I will use transfer learning to improve my prediction results and reduce training time. I will use a pre-trained VGG network, and add a few trainable convolutional and fully connected layers to the end.

Metrics

I will measure the model performance using a the average log loss function. The log loss function is minimized when the prediction is close to 1 and the actual value should be 1. It is also minimized when the prediction is close to zero and the actual value is 0.

The log loss function is also the metric that is used for the Kaggle competition.

Here is the loss function as described on the Kaggle competition page.

If there are N images, you will be making 17N predictions. Submissions are scored on the log loss:

$$-\frac{1}{N} \sum_{i=1}^{N} \left[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \right],$$

where:

- N is the 17 * the number of scans in the test set
- $\hat{y}_i$ is the predicted probability of the scan having a threat in the given body zone
- y_i is 1 if a threat is present, 0 otherwise
- $log()$ is the natural (base e) logarithm

Figure 2: loss function

II. Analysis

Data Exploration

There are 1147 training images. Since these are provided by the TSA, the resolution and size of images are identical across samples, and the volunteers being scanned are of various sizes and genders, but are consistently placed in the center of the image. The data format of the smallest size (.aps) has images in one channel, a height of 660, width of 512, and 16 angles.

I display 16 sample images and the locations of the threats.
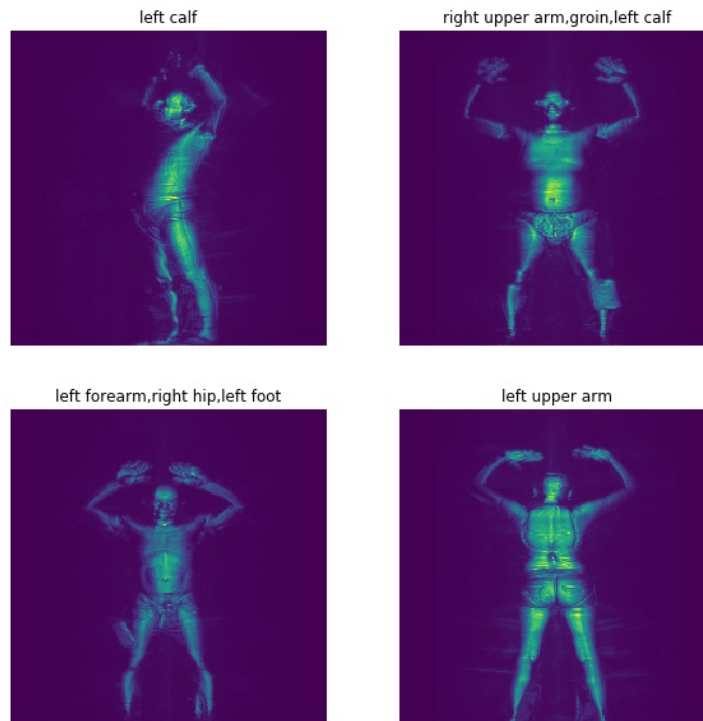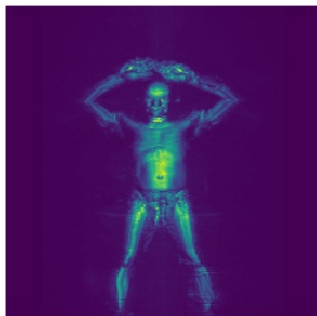
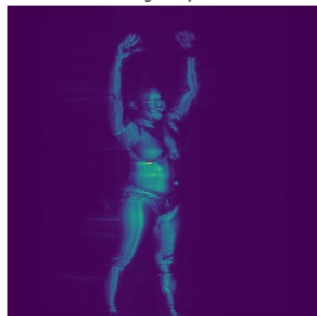right foot

right forearm

groin,right thigh,right calf

right upper arm,left calf
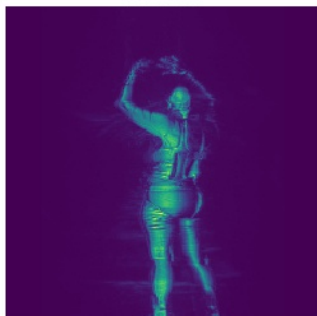
left forearm,left foot
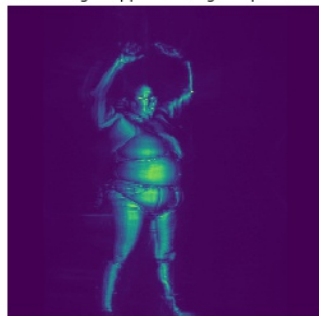

right hip

left waist

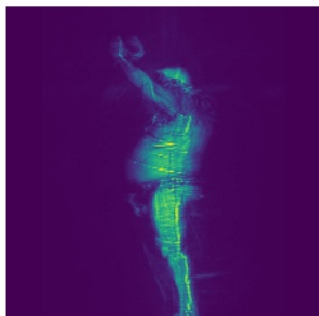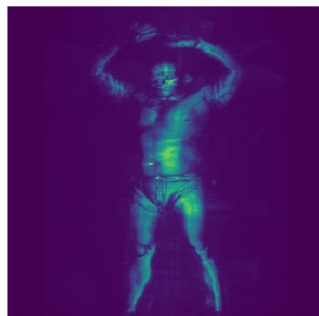right upper arm,left waist,left thigh

right calf

right upper arm,right hip

left waist

chest,left foot

Exploratory Visualization

I aggregate the label data to show the frequency that a dangerous object is hidden in each of the 17 body regions. The frequency that a threat occurs on a particular body zone ranges from 7.8% to 11.6%, with an average of 9.6% across all training samples.

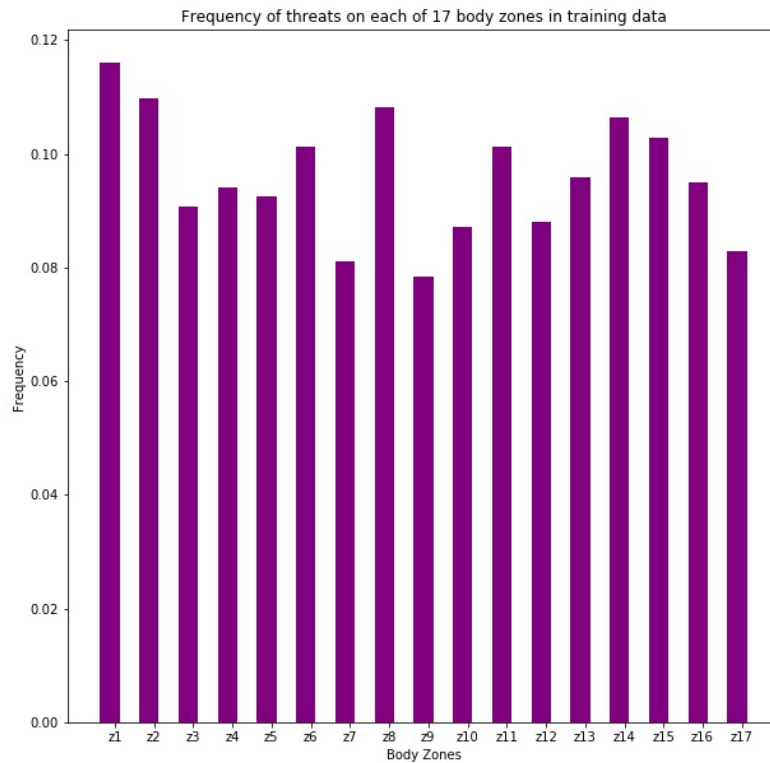
Figure 3: frequency of threats

Algorithms and Techniques

Data Pre-processing

In order to fit the images into the VGG network, I size the images to be 224 by 224, with values that range from 0 to 1. To avoid distorting the images, which

are originally 660 high by 512 wide, I pad the image with zeros first, so that the shorter side (width) first equals the longer side (the height). Once the image is square, I transform it from a 612 x 612 square to a 224 x 224 square.

Also, since the VGG network expects the inputs to be 3 channel RGB images, I copy the single channel image into three channels before feeding the data into the network.

Neural Network

I am using a pre-trained VGG convolutional neural network to generate inputs into a smaller multi-view convolutional neural network. I used the vgg16 network. For vgg16, I took the output of conv5_3 (the 13th and last convolutional layer).

I am using transfer learning and the pre-trained VGG network because it provides more insight into lower level features, since it was trained on more images for a longer period of time.

I added 2 convolutional layers to process the output of the pre-trained network. convolution 1 has a 3x3 kernel, stride of 2, and depth 512. convolution 2 has a 3x3 kernel, stride of 2, and depth of 1024

By condensing the height and width I attempt to aggregate lower level features into higher level features. By increasing depth, I leave more room for different combinations of the lower level features.

Since the inputs are multiple 2D images for a single observation, I reuse the same weights for the trainable convolutional layers, for all angles. It still generates outputs for each of the angles, so if we have 16 angles, the convolutional layers also output 16 sets of activations.

Next, an aggregation step flattens the last convolutional layer and aggregates them by taking the maximum for each element position, across all 16 sets of tensors. So it's in effect collapsing the set of 16 tensors from 16 angles into one tensor of the same shape. From this point onward, the rest of the neural network is the same as a convolutional neural network that works with 2D images.

The next layers are fully connected (dense) layers of decreasing size. The first dense layer is 1024 units, the second is 256. The fully connected layers get progressively smaller, so that each successive layer aggregates the information from the previous layer.

The output layer has 17 units, to represent the 17 body region zones. We pass the 17 logits through a sigmoid activation so that the output can be treated as a probability that ranges between 0 and 1.

Benchmark

The Kaggle leaderboard calculates an average log loss function on a test data set, and serves as a good source for a benchmark. My goal is to reach a loss score of 0.29 or lower, as this is where a significant number of competitors rank. The loss score for a completely naive 0.5 prediction is 0.69, so I definitely want to score below that.

III. Methodology

Data Preprocessing

Since the data was generated by the TSA, there aren't any abnormalities or outliers that should be removed from the training or testing data. I needed to re-scale and resize the data in order for it to match the input format expected by the pre-trained vgg network. To do this, I first scaled the values between 0 and 1, padded the shorter side (the width) to make the image a square, then resized the square to 224 by 224 pixels. Then I copied the image into 3 channels, so the final format looks like a 224 by 224 RGB image that the network expects.

Implementation

Pre-trained network

For the VGG pre-trained network, I downloaded the version of tensorflow_vgg provided by Udacity, since the one directly from the original github had a bug when I ran it. So I used https://github.com/udacity/deep-learning.git instead of using https://github.com/machrisaa/tensorflow-vgg.git

From there, I copied the deep-learning/transfer-learning/tensorflow_vgg folder. Udacity has a convenient place on Amazon Web Services that stored the pre-trained weights for vgg16, vgg16.npy

In the get_codes_vgg function, I'm working with 3D data. The data dimensions are: batch, angle, height, width, depth. When I feed each 2D slice into the VGG network, I'm saving the outputs of the network into a list of size 16 (1 for each angle), so that each numpy array stored in the list has dimensions batch, height, width, depth.

When I'm done collecting all the outputs of the pre-trained network, I convert the list of size 16 into one big numpy array, which has dimensions angle, batch, height, width, depth. Since I want to keep the order of the dimensions the same as the input data, I use a transpose to reorder the dimensions into batch, angle, height, width, depth. I save these "codes" to disk so that I can use them later.

I saved the training data codes that are output from the vgg16 network. I did the same for the smaller test data set.

Trainable network

I built a small multi-view convolutional neural network that takes the outputs of the pre-trained network as its inputs. I add some convolutional layers, and each layer includes a convolution, batch normalization, a leaky relu activation, then a dropout.

Since this is 3D data, I feed each of the 16 2D slices into the network in a loop. I initialize weights for the first angle, but all subsequent angles reuse the weights. I use a variable scope to reuse the weights. I flatten the output of the final convolutional layer, and save these in a list of 16 elements, one for each angle.

Next, I use a view pooling layer that combines all 16 angles into a single tensor. I use a reduce_max function, which lines up all 16 tensors, and for each position within those tensors, takes the maximum out of the 16, and saves that into a new tensor. The output of this pooling layer has the same dimensions as one single tensor within the list of 16.

I pass this pooled layer into some fully connected layers. The final output layer has 17 logits that represent the 17 body regions. I pass these logits through a sigma activation so that each represents a probability between 0 and 1.

Training

I use a log loss function to measure the error between prediction and actual targets, as this is the same metric used for the kaggle competition. I use an Adam Optimizer to perform back propagation, which uses a learning rate of 0.0001, and a beta1 of 0.5.

Refinement

I chose some hyper-parameters and mostly modified the number of epochs, and the design of the trainable network layers. I used a low learning rate of 0.0001, beta1 of 0.5, batch size of 64, and dropout keep probability of 0.5

For the network design, I incrementally adjusted the number of layers, the number of filters (a.k.a. channels, or depth), and the stride size.
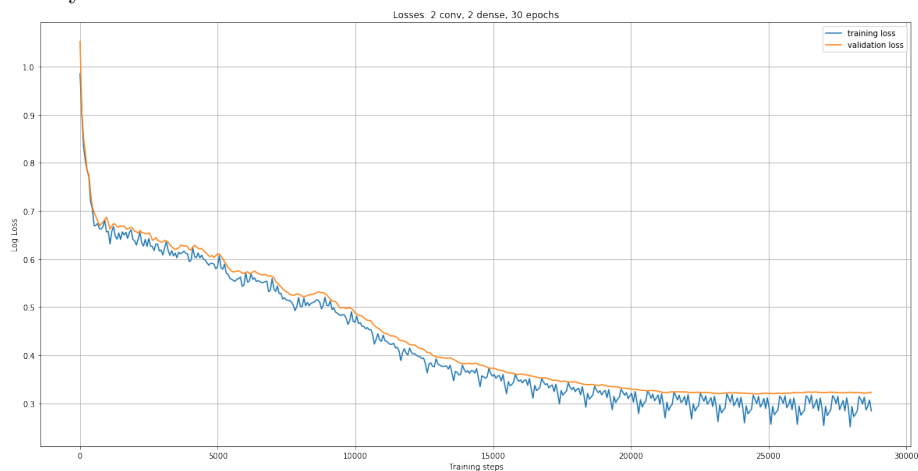
Trial 3

I first started with two convolutional layers and two dense layers. Note that trial 1 and 2 are the same design, but with 10 and 20 epochs instead of 30 epochs for trial 3.

Layer Name	Kernel/Filter Size	Stride	Padding	Depth / Channels
convolution 1	3x3	2x2	valid	512
convolution 2	3x3	2x2	valid	1024

Layer Name	Size
fully connected 1	1024
fully connected 2	256

Layer Name	Size
output logits	17

With 30 epochs, I get a validation loss of 0.3227, which follows the training loss closely.

Losses 2 conv, 2 dense, 30 epochs

Trial 4

Next, I change the stride of convolution 1 from 2x2 to 1x1. The training and validation loss jump up and down more during training. I trained for 20 epochs, it reach the best validation loss at epoch 16, at 0.3274.

Layer Name	Kernel/Filter Size	Stride	Padding	Depth / Channels
convolution 1	3x3	**1x1**	valid	512
convolution 2	3x3	2x2	valid	1024

Layer Name	Size
fully connected 1	1024
fully connected 2	256
output logits	17

Trial 5

I went back to using 2x2 strides for each convolutional layer, and added a third layer. In order to keep the height and width of the output from shrinking as
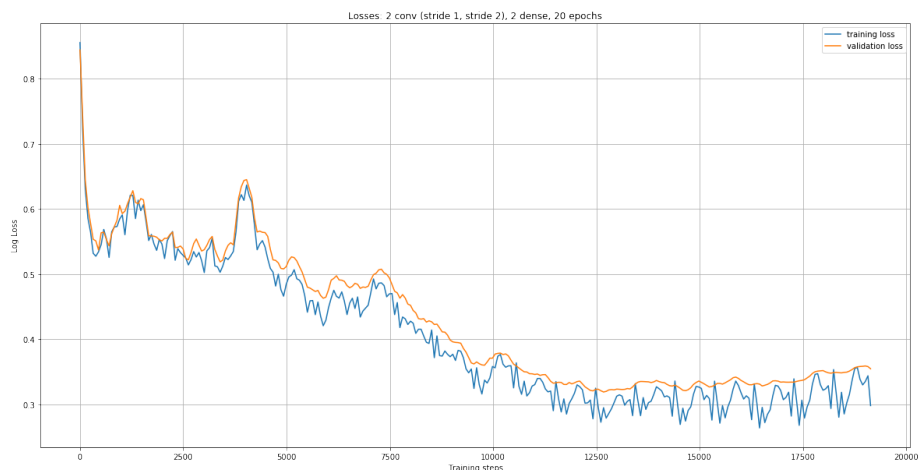
Figure 4: loss chart 4

quickly, to allow for a third layer, I change padding from valid to same. I also make the second convolution have a depth of 750 instead of 1024, and the third layer has a depth of 1024. I trained for 20 epochs; it gets valid loss .3206 at epoch 17, and gets worse (increases after that).

Layer Name	Kernel/Filter Size	Stride	Padding	Depth / Channels
convolution 1	3x3	**2x2**	**same**	512
convolution 2	3x3	2x2	**same**	**750**
convolution 3	**3x3**	**2x2**	**same**	**1024**

Layer Name	Size
fully connected 1	1024
fully connected 2	256
output logits	17

14
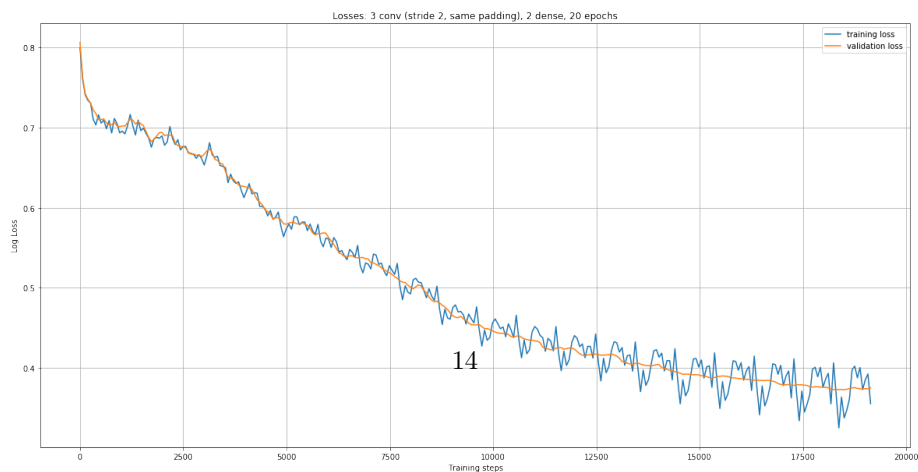
Figure 5: loss chart 5

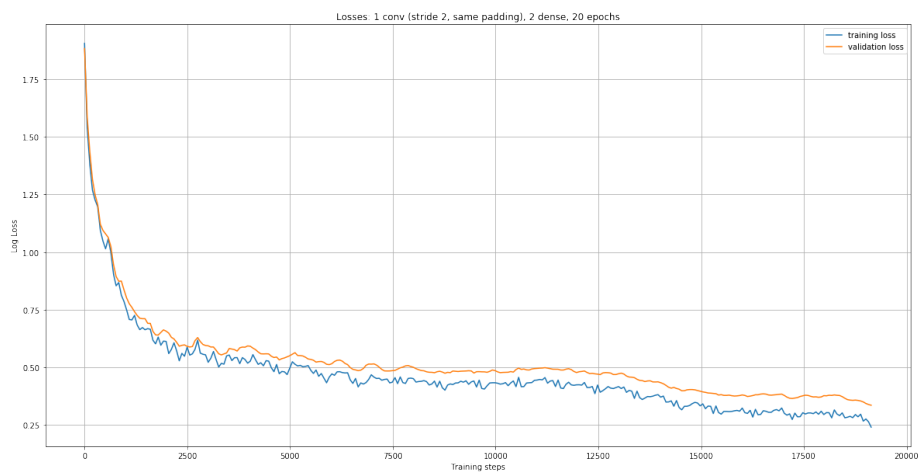| Layer Name	Size
fully connected 1	1024
fully connected 2	256
output logits	17

Figure 6: loss chart 6

Trial 7

I try increasing the size of the convolutional layer from 512 to 1024, to give more room for learning features. I trained for 20 epochs, and got a validation loss of 0.3489.

Layer Name	Kernel/Filter Size	Stride	Padding	Depth / Channels
convolution 1	3x3	2x2	same	**1024**

Layer Name	Size
fully connected 1	1024
fully connected 2	256
output logits	17

Trial 8
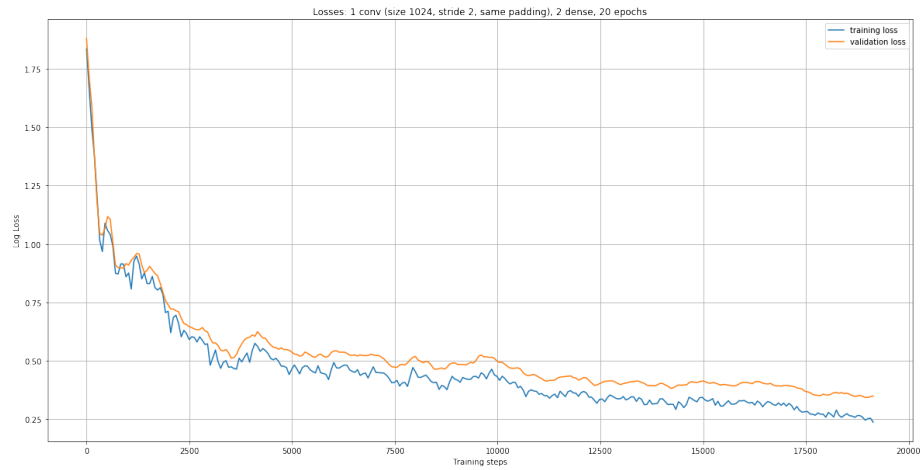
Figure 7: loss chart 7

I try doubling the size of the fully connected dense layers. I trained for 20 epochs, and the best validation loss at epoch 19 was 0.2898.

Layer Name	Kernel/Filter Size	Stride	Padding	Depth / Channels
convolution 1	3x3	2x2	same	1024

Layer Name	Size
fully connected 1	**2048**
fully connected 2	**512**
output logits	17
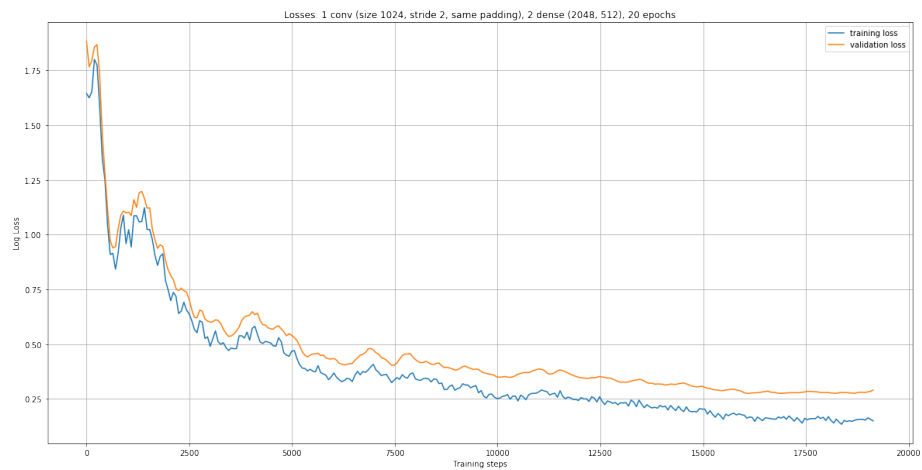
Figure 8: loss chart 8
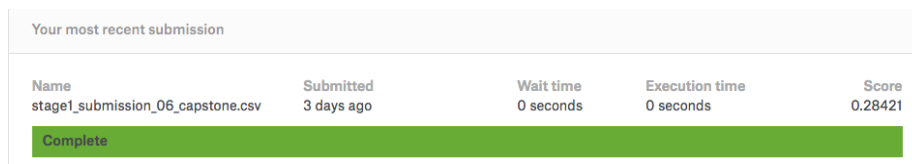
IV. Results

Model Evaluation and Validation

In order to evaluate the model, I use test data that I did not use during training. This consisted of 100 samples from the original 1147 training samples, that I did not use in the training or in validation. This resulted in a test loss of 0.269204, which is lower than the validation loss of 0.2865.

I also modify the test train and validation split, so that I reserve 300 samples for testing, 10 for validation, and 847 for training. The test loss is 0.281498, which is slightly higher, but this is expected because I used fewer training samples.

Another way I evaluated the model is by predicting on the test sample selected by Kaggle, which is a separate 100 samples that are not part of the 1147 training samples. Since I don't have the labels for this test sample, I make the predictions and then upload them to Kaggle. Kaggle reports a loss score of 0.28421, which is below my goal of 0.29.

Your most recent submission				
Name	Submitted	Wait time	Execution time	Score
stage1_submission_06_capstone.csv	3 days ago	0 seconds	0 seconds	0.28421
Complete				

Figure 9: kaggle test loss

Justification

Kaggle provides a benchmark loss value for when all predictions have a 50% chance that every body part of every sample has a hidden threat; this average log loss is 0.69315. I definitely wanted to get below this.

A benchmark that I set for myself was to reach a loss below 0.2900. This is because there were many people on the Kaggle leaderboard scoring between 0.2900 and 0.30, ranging from a rank of 57 to 132 (as of August 22, 2017).

Since I reached a score of 0.28421, I reached my goal of reaching a loss lower than 0.2900. This score puts me in the top 22% of the contestants as of August 22, 2017.

V. Conclusion

Free-Form Visualization

I plot the accuracy, precision, recall, and F1 score of a test sample of 300 observations. The main point is that precision is high, but recall is low. Precision is high, meaning that when the model predicts a threat, it's very likely that the threat is there (it's very unlikely to incorrectly predict a threat). However, it's also very likely not to predict a threat when a threat exists (low recall).

The reason for this is that in most instances, a body zone's observation does not include a threat. In order for the model to minimize the average log loss, it has a bias for predicting a lower probability of a threat. Hence, there are few false positives, but there are more false negatives.
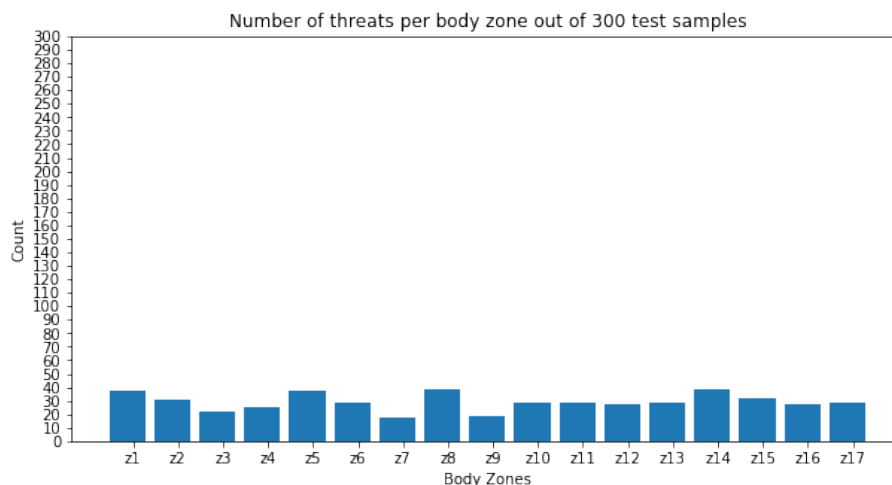

Figure 10: threat count

Reflection

The goal of this project was to predict whether each part of the body contains a threat or not, using 3D scans of volunteers who posed in airport security screening machines. I used transfer learning, with a pre-trained vgg convolutional network, and then added an additional trainable layer in a multi-view convolutional neural network. This multi-view cnn reused the same weights for all angles of the 3D data, and then pooled those multiple angles together into a single tensor using a maximum function.

The new aspect for me was in handling 3D image data for the first time. This required some care in reading and processing the data, as well as using a single
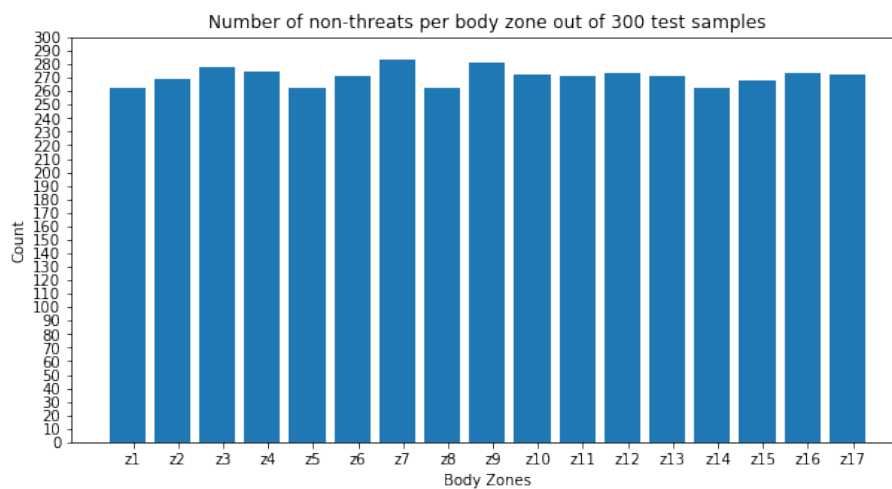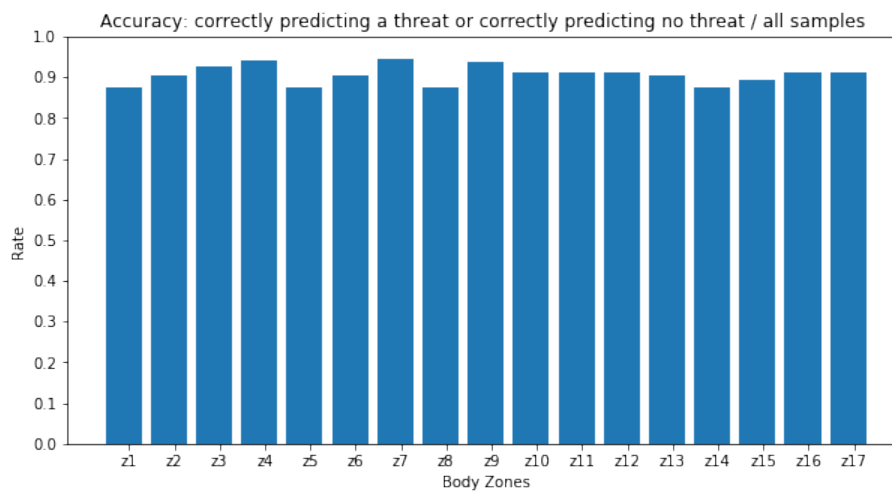
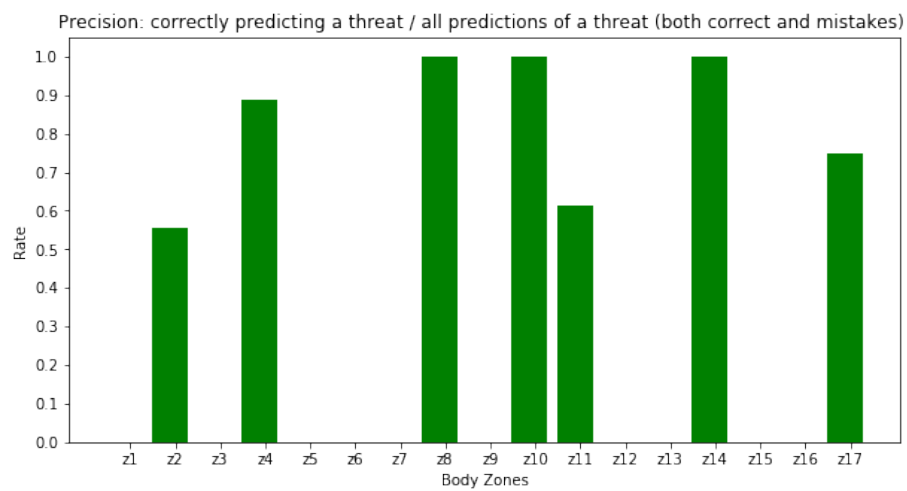Figure 11: non threat count

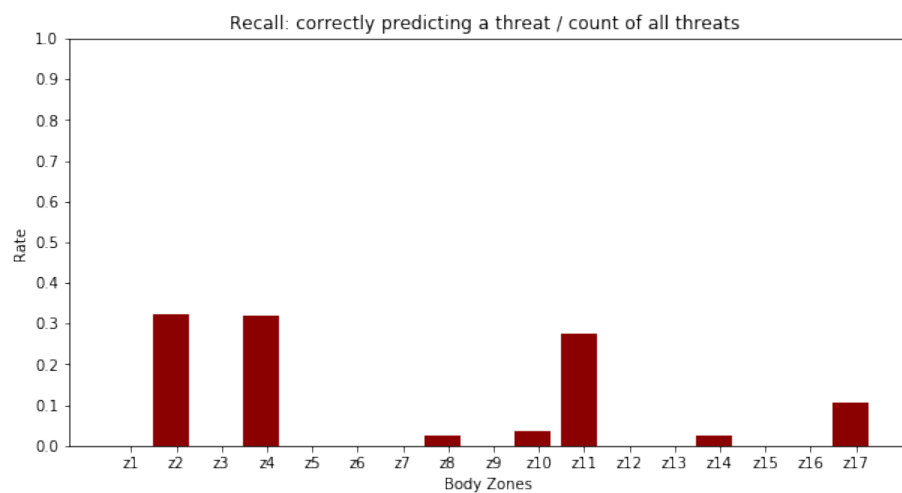Figure 12: accuracy

19

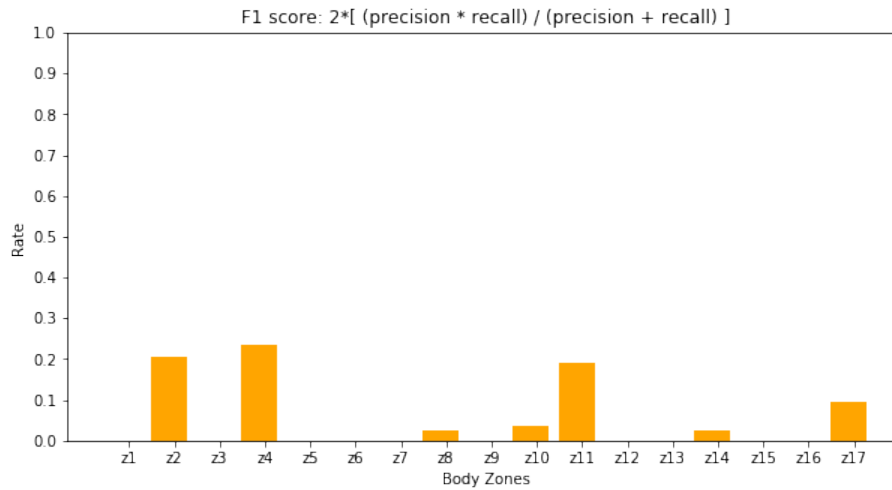Figure 13: precision

Figure 14: recall

20

Figure 15: f1

variable scope in TensorFlow that re-uses the trainable weights. In order to fit the data into the VGG network, this also required reshaping and resizing the data to fit the pre-trained model's expected input.

The final model works as intended in reaching a log loss function on the kaggle leaderboard that is under 0.2900. In order to do this, it has a bias towards predicting that a body zone does not have a threat, which means that it misses actual threats more frequently. This is not ideal, but even the Kaggle competition page states that one goal of the TSA is to reduce the amount of false positives that occur during airport screening, because false positives result is a manual screening, which increases wait times and increases costs.

Improvement

I would like to use the more detailed data sets, such as the .a3daps, which has 64 angles instead of 16 for .aps. This requires more compute time, so it made more sense for me to refine the model using the smallest data set first.

I am also interested in using the .a3d data set, which saves the data as x,y,z volumetric form, instead of as 2D slices at varying angles. I wasn't sure how to work with this data, and I'm not sure if 3D convolutions can actually do better than 2D convolutions. I also wanted to use pre-trained networks, which all train on 2D images, so it would be too costly and time consuming to train a network from scratch without the help of transfer learning.

I also wanted to look into other pre-trained networks, such as inception or resnet networks. I haven't yet found other pre-trained networks that I felt comfortable

with implementing, so I stayed with the vgg network that I knew would work.

Since there are many people above me on the Kaggle leaderboard, with losses as small as 0.01, there are better solutions to this problem.

References

- Kaggle competition site: https://www.kaggle.com/c/passenger-screening-algorithm-challenge
- Multi-View Convolutional Neural Networks for 3D Shape Recognition, by Hang Su, Subhransu Maji, Evangelos Kalogerakis, and Erik Learned-Miller. http://vis-www.cs.umass.edu/mvcnn/
- TensorFlow implementation of a Multi-View CNN: https://github.com/WeiTang114/MVCNN-TensorFlow
- VGG pre-trained network: git clone https://github.com/udacity/deep-learning.git; then copy the folder ./transfer-learning/tensorflow_vgg
- VGG pre-trained weights are found here: https://s3.amazonaws.com/content.udacity-data.com/nd101/vgg16.npy