Report for the Deep Learning Course Assignment 3

Georgios Methenitis
georgios.methenitis@cwi.nl

Abstract

In this assignment I implemented a convolutional neural network (CNN) with two different architectures on the CIFAR10 dataset. The first was a straightforward architecture where the input image was fed into a single neural network, while the goal of the neural network was to identify the class of the image. In the second architecture, two images were the input of two copies of the same neural network, while the goal of the learning was to distinguish between two different images' classes. In all experiments I used a desktop computer equipped with an NVIDIA GTX770 GPU with 2GB of ram.

1 Task 1

In this section I explain the implementation and present the results of the first task of the assignment. I implemented the architecture for the CNN as this was given in the assignment. For the initialization and regularization of the weights I used:

```
initializer = tf.random_normal_initializer( mean=0.0, stddev=0.001 )
regularizer = regularizers.l2_regularizer( 0.001 )
```

but also experimented with xavier initializer without however reporting the results here, since there was no improvement over the normal distributed initialization of the weights. Furthermore all biases were initialized at zero:

```
initializer=tf.constant_initializer(0.0))
```

Figure 1 illustrates the accuracy and the loss during training and test without regularization. As shown

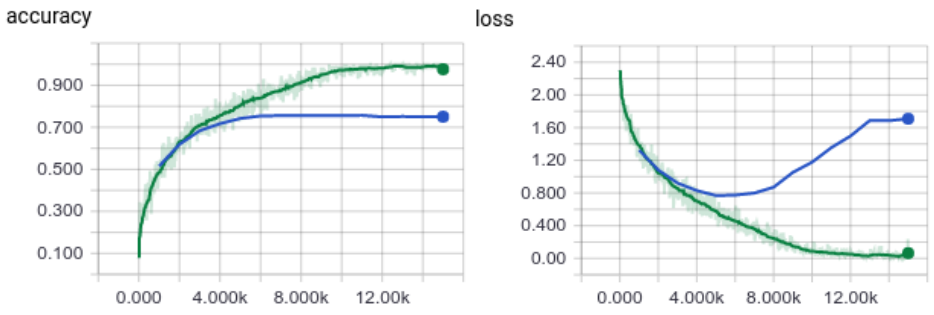

Figure 1: Accuracy (left) and loss (right) for the CNN with batch size of 128 and no regularization. Green lines illustrate performance on the training set and blue lines on the test set.

in the Figures 1, the model overfits the training data. The accuracy in the training set approximates 1.0 in the last 5000 steps of the training where the loss in the test set is increasing while accuracy in the test set is not affected.

30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

1.1 Regularization

In this section I present the results of the CNN with regularization losses in the weights. More specifically I added regularization losses in the fully connected layers $fc1, fc2$

```
if self.wd is not None:
    weight_decay = tf.mul(tf.nn.l2_loss(w_fc1), self.wd, name='weight_loss')
    tf.add_to_collection('losses', weight_decay)
```

making sure the losses were added to the loss function in the loss function of convNet class. Figure 2 illustrates the accuracy and the loss during training and test with regularization $wd = 0.005$. There is

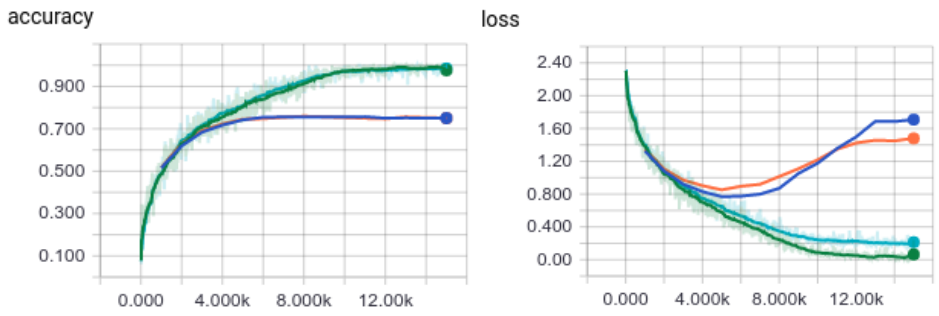

Figure 2: Accuracy (left) and loss (right) for the CNN with batch size of 128 and no regularization. Cyan lines illustrate performance on the training set and orange lines on the test set.

no performance gain by the regularization apart from a decrease in the loss in the test set.

1.2 Dropout

In the next experiment I implemented dropout in the fully connected layers $fc1, fc2$.

```
h_fc1 = tf.cond(tf.cast(self.isTrain, tf.bool),
lambda: tf.nn.dropout(h_fc1, 0.8), lambda: h_fc1)
```

Figure 3 illustrates the accuracy and the loss during training and test with dropout $0.5, 0.2$. The

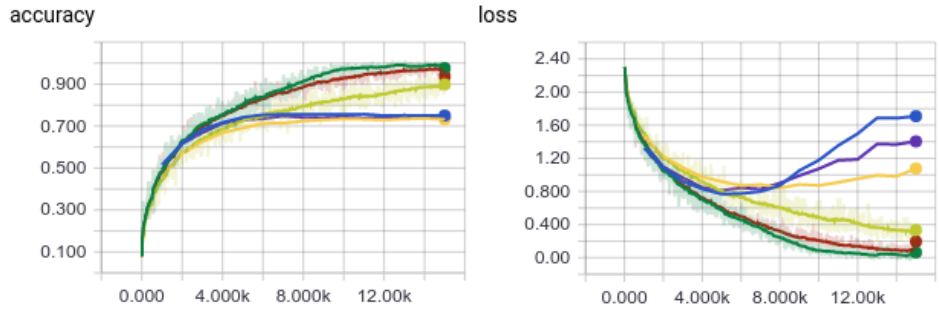

Figure 3: Accuracy (left) and loss (right) for the CNN with batch size of 128 and dropout. Red lines illustrate performance on the training set and purple lines on the test set for dropout 0.2, while light green lines illustrate performance on the training set and yellow lines on the test set for dropout 0.5.

performance on the test set in the end of the training was:

- For no dropout: **0.7503**
- For dropout 0.5: 0.7324
- For dropout 0.2: 0.7433

There was no performance gain, but only a decrease in overfitting with dropout 0.5.

1.3 Smaller Batch Size

Here, I used half the size of the batch to complete training using batch size of 64. Figure 4 illustrates the accuracy and the loss during training and test with the given batch size of 128 and the half batch size of 64.

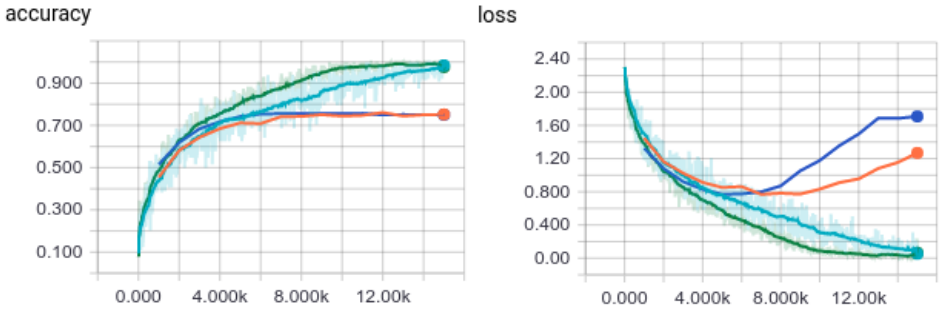

Figure 4: Accuracy (left) and loss (right) for the CNN with batch size of 64. Cyan lines illustrate performance on the training set and orange lines on the test set for dropout 0.2. Green and blue lines illustrate the performance of batch size 128 with default configuration CNN as in all previous figures.

The performance on the test set in the end of the training was:

- For batch size 128: 0.7503
- For batch size 64: **0.7520**

There was a performance gain by using smaller batch size and a limited overfitting.

1.3.1 Regularization

As in Section 1.1 I used regularization for the weights in the fully connected layers $fc1, fc2$. Figure 5 illustrates the accuracy and the loss during training and test with the given batch size of 64 two regularization values for the weight decays.

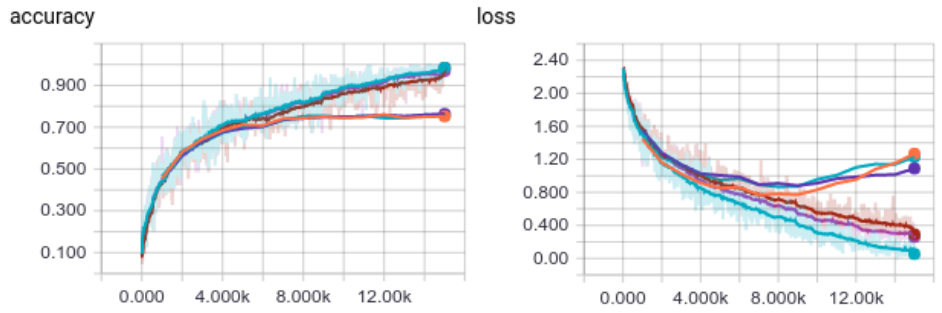

Figure 5: Accuracy (left) and loss (right) for the CNN with batch size of 64. Cyan lines illustrate performance on the training set and orange lines on the test set without weight regularization. Red lines illustrate performance on the training set and purple lines on the test set with weight regularization 0.01. Light purple lines illustrate performance on the training set and cyan lines on the test set with weight regularization 0.005.

The performance on the test set in the end of the training was:

- No regularization: 0.7520
- For regularization 0.01: **0.7634**
- For regularization 0.005: 0.7569

The maximum reported accuracy on the test set was achieved by using mini batches of 64 size and regularization values of 0.01.

1.3.2 Dropout

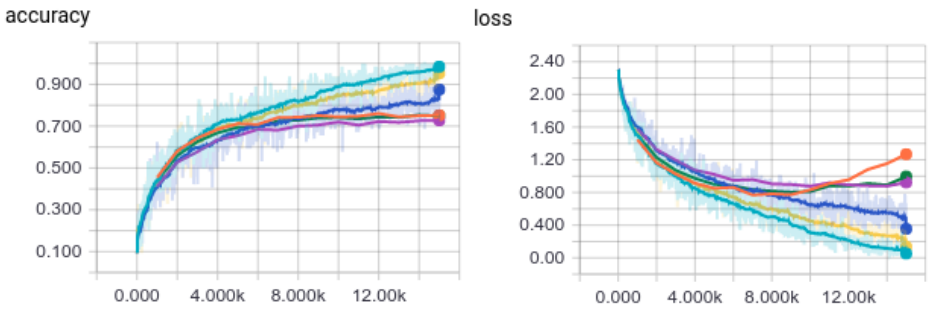

Figure 6: Accuracy (left) and loss (right) for the CNN with batch size of 64. Cyan lines illustrate performance on the training set and orange lines on the test set without weight regularization. Red lines illustrate performance on the training set and purple lines on the test set with weight regularization 0.01. Light purple lines illustrate performance on the training set and cyan lines on the test set with weight regularization 0.005.

The performance on the test set in the end of the training was:

- No dropout: 0.7520
- For dropout 0.5: 0.7279
- For dropout 0.2: 0.7493

The was no performance gain by using dropout when using smaller batches of 64 size.

1.4 Feature Visualization

In this section I show how the features learned during training of the CNN can be illustrated and how they can be used to train an one-vs-all classifier. I load the learned weights during training and converting them to 2-d points using TSNE.

```
tsne = TSNE(n_components=2, verbose=1, perplexity=40, n_iter=300)
tsne.fit_transform(features)
```

Figure 7 presents the visualization of the features in the two dimensional space. You can observe that features obtained in $fc2$ layer can distinguish better features of different classes. On the contrary when see the flatten features all labels are located close to points of other classes.

1.4.1 One-vs-Rest Classifier

To test the accuracy of the one-vs-all classifier I use the features retained by the learned model to train the classifiers. Using the 80 percent of the features I train a linear kernel OneVsRest classifier, and test the accuracy over the rest 20 percent.

```
train_set = range(int(features.shape[0]*0.8))
test_set = range(train_set[-1] + 1, features.shape[0])

classif = OneVsRestClassifier(SVC(kernel='linear'))
classif.fit(features[train_set], batch_y[train_set]
classif.score(features[test_set], batch_y[test_set]
```

Here, the reported accuracy[1] of the one-vs-classifiers on the test set:

- Accuracy with flatten features: 0.59.

[1]The accuracy is reported to half of the test set (5000 random samples without replacement due to slow training of the OneVsRestClassifier)

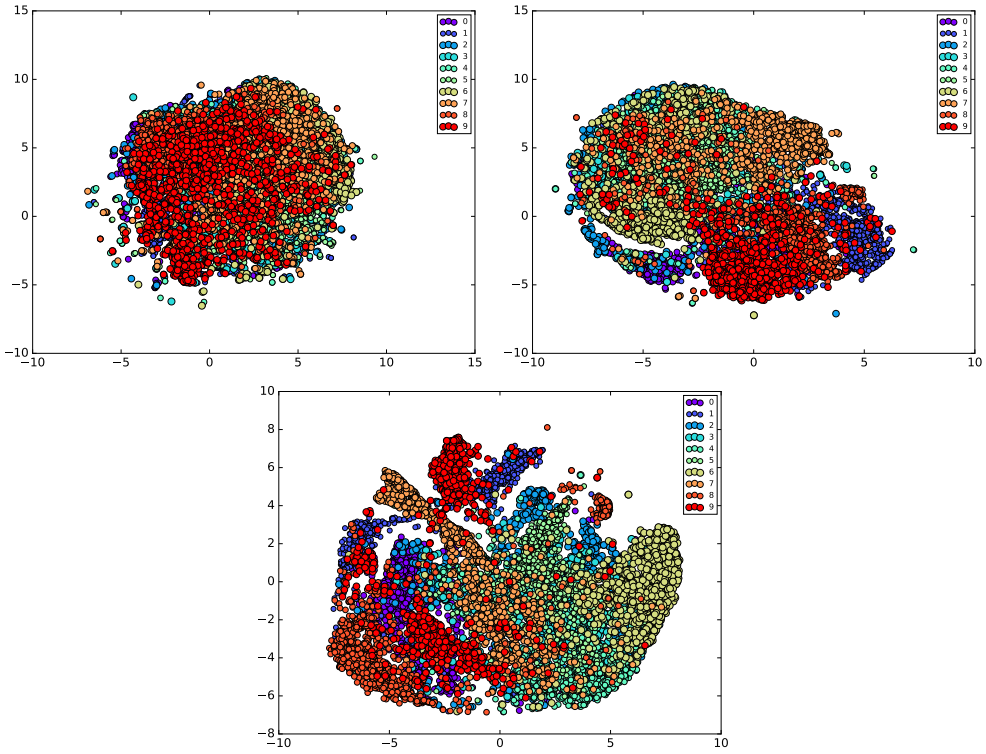

Figure 7: Visualization of the features in the 2-d space of the `flatten` layer (top), `fc1` (bottom-left), `fc2` (bottom-right).

- Accuracy with fc1 features: 0.59.
- Accuracy with fc2 features: 0.703.

It is natural that the accuracy is higher for later layers. As it was earlier $fc2$ layer can distinguish better features of different classes.

2 Task 2

In this section I present my implementation and the results achieved by the siamese architecture of the CNN. First, I implemented the functions `create_dataset` and `next_batch` in the `cifar10_siamese_utils.py`.

2.1 Dataset

The create_dataset function:

```
dset = []
    for sample_tuple in range(num_tuples):

        n_correct = int(fraction_same * batch_size)

        val_set_indexes = range(int(source_data.train._images.shape[0] * 0.8), source_data.train._images.shape[0])

        index_x1 = np.random.choice(val_set_indexes)
        label_x1 = source_data.train._labels[index_x1]

        index_x2_similar = np.random.choice(val_set_indexes[0] +
                                            np.where(source_data.train._labels[val_set_indexes] == label_x1)[0],
                                            replace=False, size=n_correct)

        index_x2_opposite = np.random.choice(val_set_indexes[0] +
                                             np.where(source_data.train._labels[val_set_indexes] != label_x1)[0],
                                             replace=False, size=(batch_size - n_correct))

        x1 = np.array([source_data.train._images[index_x1] for i in range(batch_size)])
```

```
        x2 = source_data.train._images[index_x2_similar]
        x2 = np.vstack((x2, source_data.train._images[index_x2_opposite]))
        labels = np.hstack((np.ones((n_correct)), np.zeros((batch_size - n_correct))))

        dset.append((x1, x2, labels))

    return dset
```

which returns `num_tuples` tuples of the following batches:

```
      X_1             X_2                Y
| image_cl1_1, image_cl1_10  | -->   | 1 |
| image_cl1_1, image_cl1_4   | -->   | 1 |
| image_cl1_1, image_cl1_163 | -->   | 1 |
| image_cl1_1, image_cl1_145 | -->   | 1 |
| image_cl1_1, image_cl3_8   | -->   | 0 |
|      .             .       | -->   | 0 |
|      .             .       | -->   | 0 |
|      .             .       | -->   | 0 |
| image_cl1_1, image_cl5_8   | -->   | 0 |
| image_cl1_1, image_cl2_    | -->   | 0 |
| image_cl1_1, image_cl10_8  | -->   | 0 |
```

and used for the evaluation of the training set.

The `next_batch` function, which include the following piece of code was given to us in `cifar10_utils.py` to sample as uniformly the dataset as possible.

```
n_correct = int(fraction_same * batch_size)
        val_set_indexes = range(int(self._images.shape[0] * 0.8))

        start = self._index_in_epoch
        self._index_in_epoch += batch_size
        if self._index_in_epoch > self._num_examples * 0.8:
            self._epochs_completed += 1
            np.random.shuffle(val_set_indexes)
            start = 0
            self._index_in_epoch = batch_size
            assert batch_size <= self._num_examples

        end = self._index_in_epoch

        val_set_indexes = val_set_indexes[start:end]
\end{tiny}
```

The `next_batch` function then creates each training batch in two following ways.

- In the *default* implementation the structure of each batch is the same as the test set following the same idea. All images in $x1$ are the same, while $x2$ contains `fraction` percentage of similar label images to $x1$ and the rest random images of different classes.
- In the *alternative* implementation the following piece of code

  ```
  random_index = np.array(np.random.choice(val_set_indexes, replace=False, size=batch_size))

  matches = [np.random.choice(np.array(np.where(self._labels[range(int(self._images.shape[0] * 0.8))] ==
  self._labels[x]))[0]) for x in random_index]

  mismatches = [np.random.choice(np.array(np.where(self._labels[range(int(self._images.shape[0] * 0.8))] !=
  self._labels[x]))[0]) for x in random_index]

  x1 = self._images[random_index]
  x2 = self._images[matches[:n_correct]]
  x2 = np.vstack((x2, self._images[mismatches[n_correct:]]))
  labels = np.hstack((np.ones((n_correct)), np.zeros((batch_size-n_correct))))
  ```

  generates batches of random pairs of images respecting the `fraction` of same label images. The resulted batch

  ```
        X_1             X_2                Y
  | image_cl1_10, image_cl1_50   | -->   | 1 |
  | image_cl4_17, image_cl4_234  | -->   | 1 |
  | image_cl8_654, image_cl8_14  | -->   | 1 |
  | image_cl8_66, image_cl6_234  | -->   | 0 |
  |      .             .         | -->   | 0 |
  ```

```
|      .              .           | -->   | 0 |
|      .              .           | -->   | 0 |
| image_cl2_327, image_cl6_234 | -->   | 0 |
```

Is is expected that classes are uniformly picked in expectation given the random sampling. Given the default way of selecting training batches training among classes is not guarranteed to be uniform. Later we see that the alternative way of creating batches achieves the highest accuracy in the end of training using the one-vs-rest classifier.

2.2 Loss function

I used the following two loss functions:

1. $\mathcal{L} = Y * \frac{1}{2} * d^2 + (1 - Y) * max(margin - d^2, 0)$
2. $\mathcal{L} = Y * d^2 + (1 - Y) * max(margin - d^2, 0)$

2.3 Training

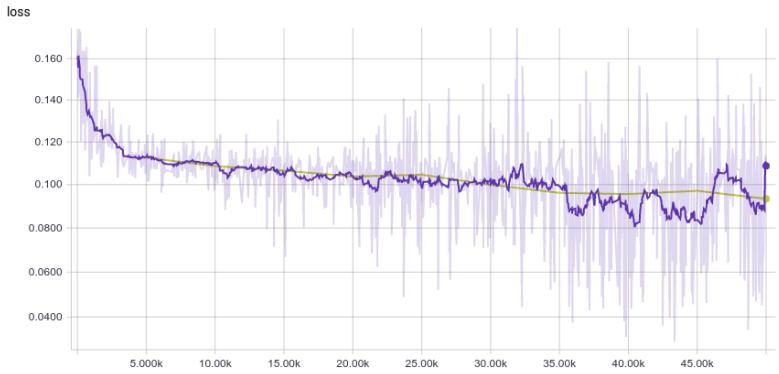

Figure 8: **Experiment 1** Loss in the training (purple line) and the test set (yellow line) for 50000 training steps, $margin = 1.0$, using the default `next_batch` method and the loss function (1).

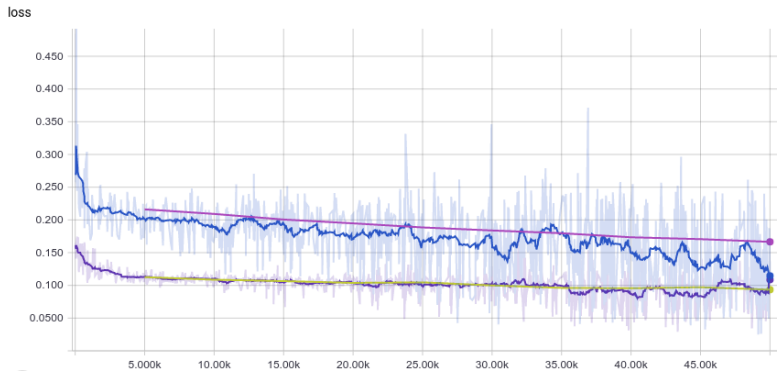

Figure 9: **Experiment 2** Loss in the training (blue line) and the test set (light purple line) for 50000 training steps, $margin = 1.0$, using the default `next_batch` method and the loss function (2). Loss in the training (purple line) and the test set (yellow line) for 50000 training steps, $margin = 1.0$, using the default `next_batch` method and the loss function (1).

2.4 Feature Visualization

2.5 One-vs-Rest Classifier

Here, I present the accuracy of the one-vs-rest classifier on the features learned from in the final layer of the siamese network for all experiments

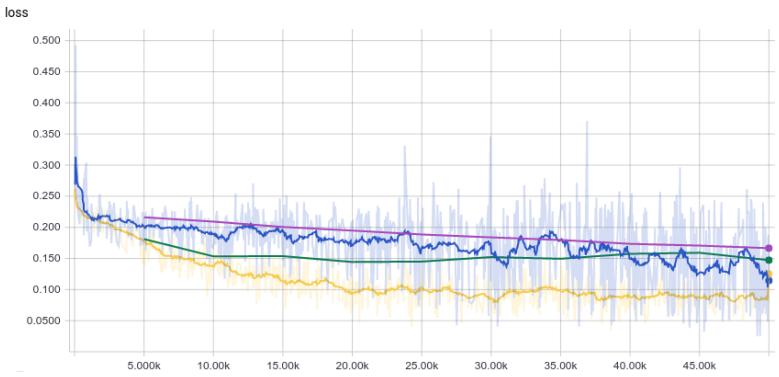

Figure 10: **Experiment 3** Loss in the training (blue line) and the test set (light purple line) for 50000 training steps, $margin = 1.0$, using the default `next_batch` method and the loss function (2). Loss in the training (yellow line) and the test set (light purple line) for 50000 training steps, $margin = 1.0$, using the default `next_batch` method and the loss function (2).

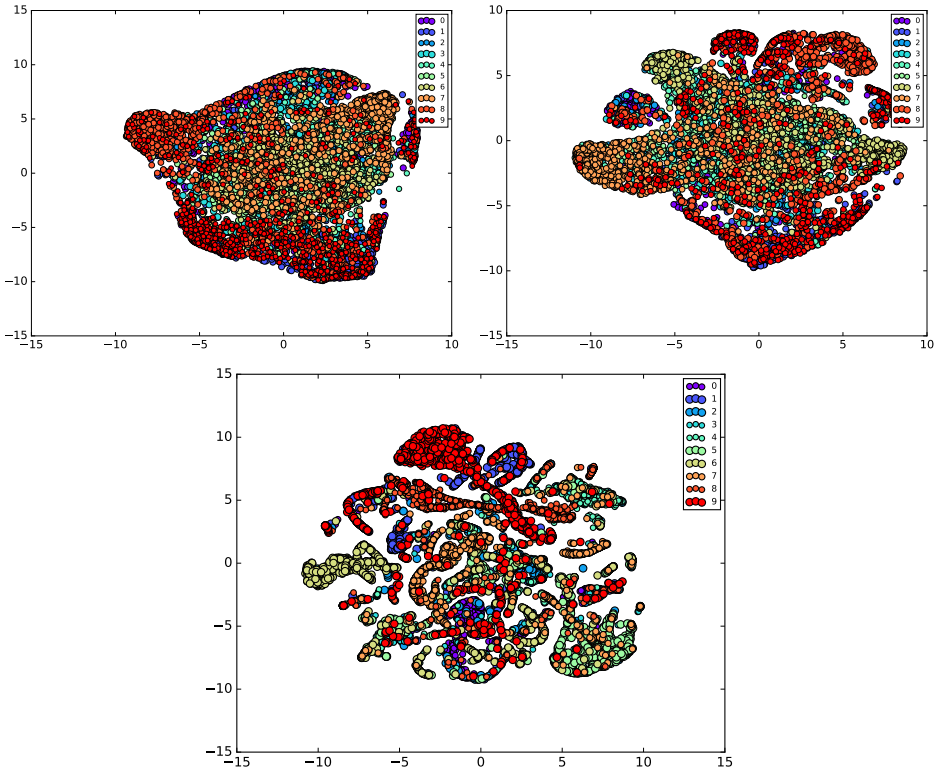

Figure 11:

- **Experiment 1**: 0.476
- **Experiment 2**: 0.473
- **Experiment 3**: $\mathbf{0.682}$

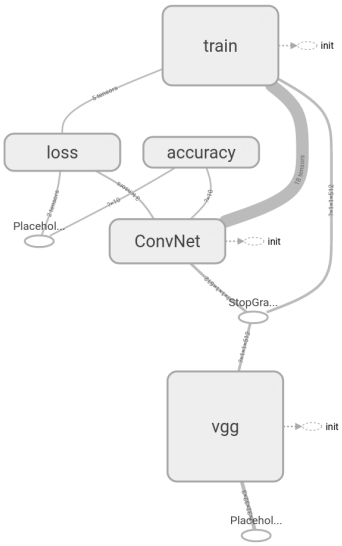

Figure 12: Graph for the task 3 of the assignment, the final three layers of convnet were used. Note the stop gradient operation, where we don't want to update the pre-trained weights.

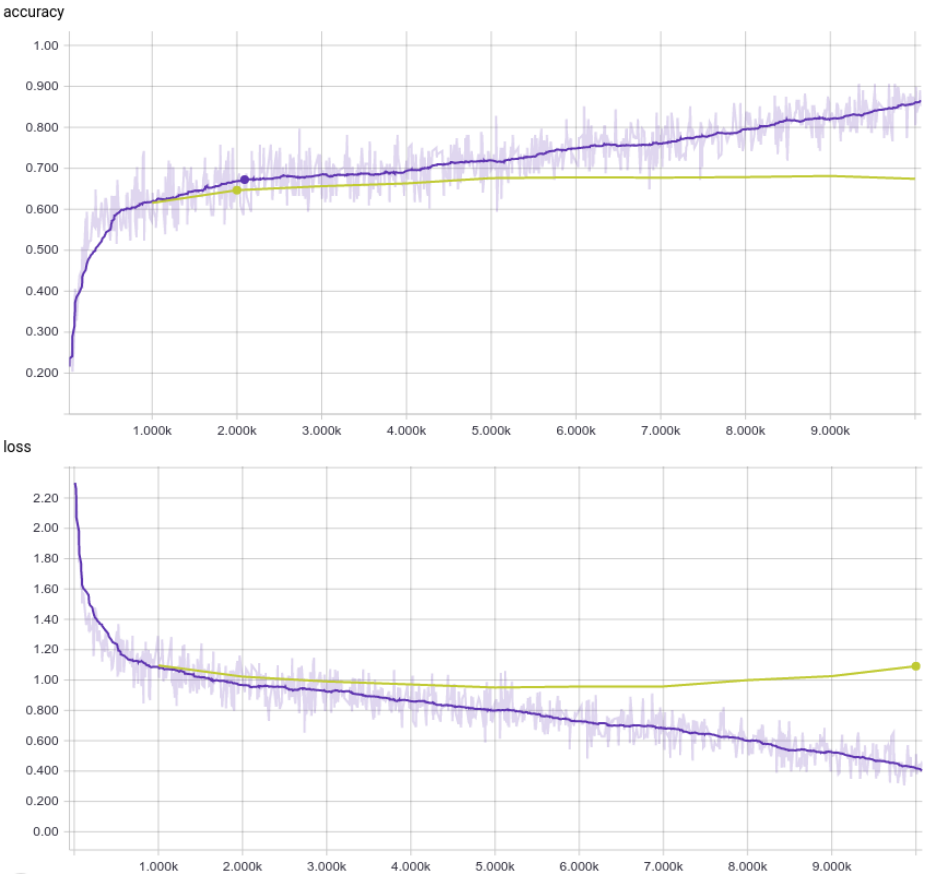

Figure 13:

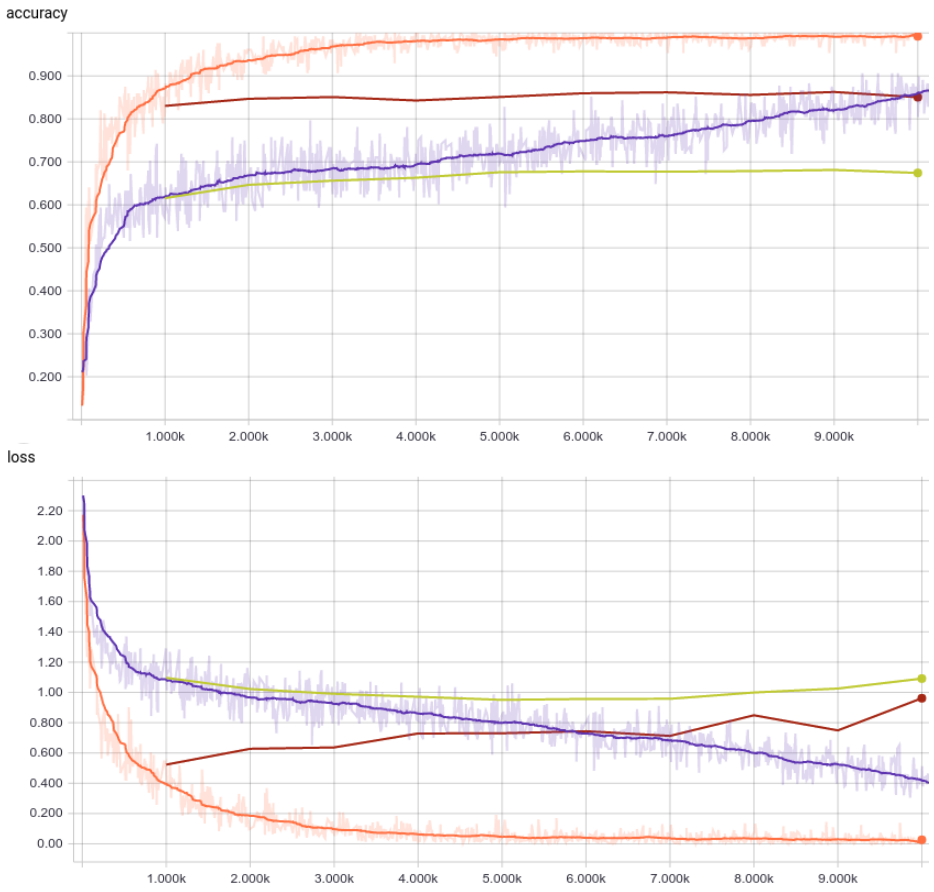

Figure 14:

3 Transfer Learning

3.1 Feature Extraction

3.2 Retraining

3.2.1 Refining