

NC State University ECE

ECE 763

Computer Vision

Dr. Tianfu Wu

Project 2

John Jefferson

04/06/2018

1. Introduction

This project focuses on an implementation of applying the algorithm used by Google's tensorflow MNIST CNN for classifying pictures into select groups. We are using the same input pictures from Project 1 to identify whether a 52x52 pixel image contained a face or was just a background image with the implementation of preprocessing, regularization/normalization, and the actual CNN algorithm.

2. Method

2.1. Image Preparation

We were to pull 10000 training images and 1000 test images from a bank of face images provided in the links given to us. I decided to gather normalized grayscale pictures where the faces were all centered around the spot between the eyes. Below are two source pictures for understanding:



Fig. 1 – Two source images of famous people

For this project, the images were cropped and scaled down to 52x52 pixel images for computing intensity and time issues. Below are examples of two images after the cropping and scaling was applied to them. In my case, 52x52 images were used to really save on computation time due to laptop restrictions, staying divisible by 4 so that we can interface with Google's tensorflow algorithm



Fig. 2 – Two scaled 52 x 52 images within the datasets

The image datasets were then split into two different datasets: training set and test set. For computing purposes again, we could reduce the number of pictures contained in each set. The pictures were then flattened into one-dimensional arrays for ease of indexing and overall computation abilities.

2.2. No Pre-processing Test

The first test that was tried was the implementation of the tensorflow algorithm on the raw images without any preprocessing done on the images beforehand. Building on the implementation of the given tutorial code, we can see that the important inputs of the model are the learning rate, the iterations/steps over the batches, and the use of preprocessing on the input data. For the first run of the model, I had the learning rate set to 0.001, an iteration size of 2500:

```
if mode == tf.estimator.ModeKeys.PREDICT:
    return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)

# Calculate Loss (for both TRAIN and EVAL modes)
loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)

# Configure the Training Op (for TRAIN mode)
if mode == tf.estimator.ModeKeys.TRAIN:
    optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
    train_op = optimizer.minimize(
        loss=loss,
        global_step=tf.train.get_global_step())
    return tf.estimator.EstimatorSpec(mode=mode, loss=loss, train_op=train_op)

# Add evaluation metrics (for EVAL mode)
eval_metric_ops = {
    "accuracy": tf.metrics.accuracy(
        labels=labels, predictions=predictions["classes"])}
return tf.estimator.EstimatorSpec(
    mode=mode, loss=loss, eval_metric_ops=eval_metric_ops)
```

Code Example 1. – Learning Rate 0.001

```
# Train the model
train_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"x": train_data},
    y=train_labels,
    batch_size=100,
    num_epochs=None,
    shuffle=True)
mnist_classifier.train(
    input_fn=train_input_fn,
    steps=2500,
    hooks=[logging_hook])

# Evaluate the model and print results
eval_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"x": eval_data},
    y=eval_labels,
    num_epochs=1,
    shuffle=False)
eval_results = mnist_classifier.evaluate(input_fn=eval_input_fn)
print(eval_results)
```

Code Example 2. – Iteration Number

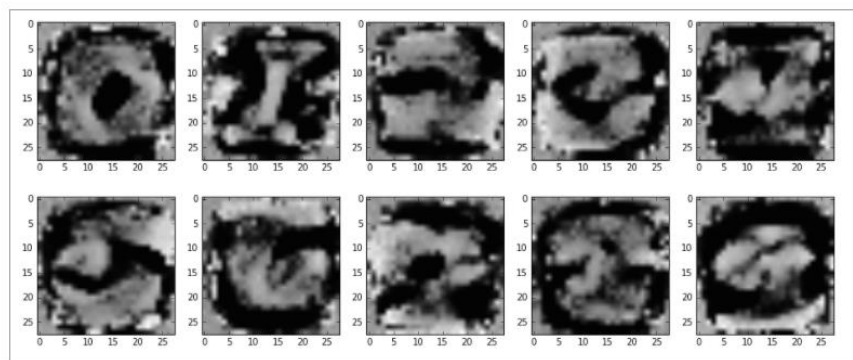
As shown below in the screenshot, the accuracy of the model was very high at 98.6% being trained with 10000 facial images and 10000 background images. The loss was relatively respectable with no preprocessing being done at 0.0489. It is intuitive to me that with preprocessing of some sort, without overtraining the model, should yield in an even higher accuracy. As for the percentage now, I believe that the grayscale images could have been preprocessed somehow before being put into the dataset, seeing as I grabbed from the image bank of images with aligned features.

```
Run cnn_mnist_face
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Starting evaluation at 2018-04-06-19:58:24
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Restoring parameters from /tempor/proj2\model.ckpt-2500
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Finished evaluation at 2018-04-06-19:58:30
INFO:tensorflow:Saving dict for global step 2500: accuracy = 0.986, global_step = 2500, loss = 0.048966587
{'accuracy': 0.986, 'loss': 0.048966587, 'global_step': 2500}
Process finished with exit code 0
PEP 8: block comment should start with '#'
```

Result Output 1. – No preprocessing result of test

2.3 Pre-processing (Subtracting mean)

The next test that was tried was the implementation of the tensorflow algorithm on the raw images with preprocessing done on the images beforehand. Building on the implementation of the given tutorial code, we can see that the important inputs of the model are the learning rate, the iterations/steps over the batches, and the use of preprocessing on the input data. For the first run of the model, I had the learning rate set to 0.001, an iteration size of 2500. The major difference with this method was the subtraction of the mean. I noticed that subtracting the means of the images from the original gave a facial map that greatly resembled a paper on regularization. The map was similar to Gaussian Weight Regularization much seen in the example of written numbers below:



Gaussian weight regularization: these 'weight images' are really different but the model approaches the same training accuracy as the unregularized version.

Fig. 3 – Gaussian Weight Reg - <https://greydanus.github.io/2016/09/05/regularization/>

Regularizing/normalizing the facial images extracted features like the mouth and eyes with I believed would increase the accuracy of the model. It seems that just subtracting the mean was not enough though as some of the backgrounds could have been optimized to look like pictures without dividing some of the variance out:

```
# Calculate Loss (for both TRAIN and EVAL modes)
loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)

# Configure the Training Op (for TRAIN mode)
if mode == tf.estimator.ModeKeys.TRAIN:
    optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
    train_op = optimizer.minimize(
        loss=loss,
        global_step=tf.train.get_global_step())
    return tf.estimator.EstimatorSpec(mode=mode, loss=loss, train_op=train_op)

# Add evaluation metrics (for EVAL mode)
eval_metric_ops = {
    "accuracy": tf.metrics.accuracy(
        labels=labels, predictions=predictions["classes"])}
return tf.estimator.EstimatorSpec(
    mode=mode, loss=loss, eval_metric_ops=eval_metric_ops)
```

Code Example 3. – Learning rate of preprocessing run

```
# Train the model
train_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"x": train_data},
    y=train_labels,
    batch_size=100,
    num_epochs=None,
    shuffle=True)
mnist_classifier.train(
    input_fn=train_input_fn,
    steps=2500,
    hooks=[logging_hook])

# Evaluate the model and print results
eval_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"x": eval_data},
    y=eval_labels,
    num_epochs=1,
    shuffle=False)
eval_results = mnist_classifier.evaluate(input_fn=eval_input_fn)
print(eval_results)
```

Code Example 4. – Iteration size of preprocessing run

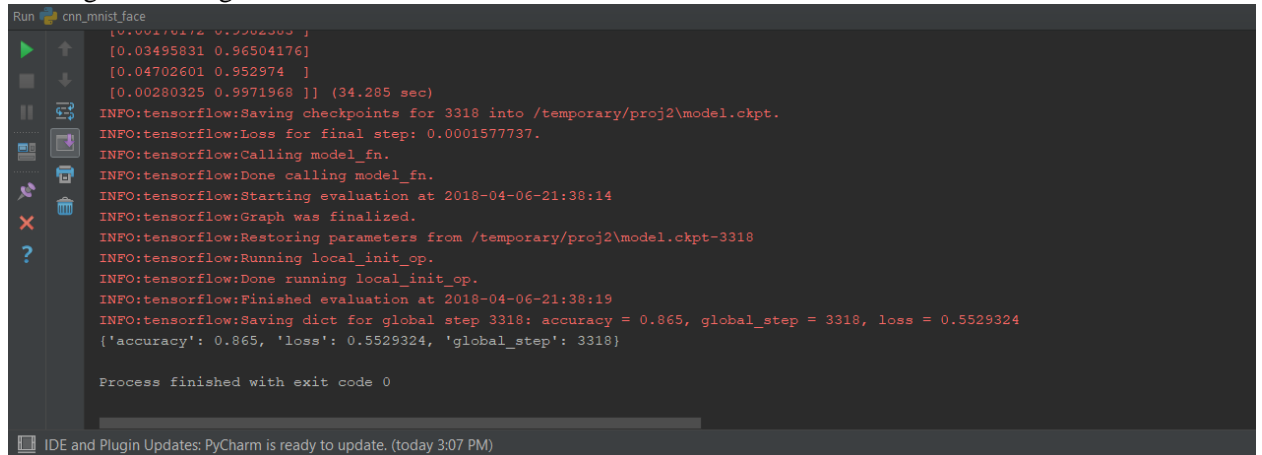
Here is an example of the code that I used to subtract the mean from all of the flattened images before passing them through the tensorflow algorithm:

```
#####
#Flattens and concatenates the image set passed in
#####
def standardizeImages(flatMatrix):
    meanOfData = 0 #np.mean(flatMatrix.T,axis = 0)
    #stdOfData = 1 #np.mean(flatMatrix.T,axis=0)
    standardMatrix = flatMatrix.T - meanOfData

    return standardMatrix.T
```

Code Example 5. – Standardization by subtracting the mean

As shown below in the screenshot, the accuracy of the model was very high at 86.5% being trained with 10000 facial images and 10000 background images. The loss was relatively respectable with no preprocessing being done at 0.5529. It is intuitive to me that with preprocessing of some sort, without overtraining the model, should yield in an even higher accuracy. As for the percentage now, I believe that the grayscale images could have been preprocessed somehow before being put into the dataset, seeing as I grabbed from the image bank of images with aligned features.



```
Run cnn_mnist_face
[0.00170172 0.99829828]
[0.03495831 0.96504169]
[0.04702601 0.952974 ]
[0.00280325 0.9971968 ]] (34.285 sec)
INFO:tensorflow:Saving checkpoints for 3318 into /temporary/proj2\model.ckpt.
INFO:tensorflow:Loss for final step: 0.0001577737.
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Starting evaluation at 2018-04-06-21:38:14
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Restoring parameters from /temporary/proj2\model.ckpt-3318
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Finished evaluation at 2018-04-06-21:38:19
INFO:tensorflow:Saving dict for global step 3318: accuracy = 0.865, global_step = 3318, loss = 0.5529324
{'accuracy': 0.865, 'loss': 0.5529324, 'global_step': 3318}

Process finished with exit code 0

IDE and Plugin Updates: PyCharm is ready to update. (today 3:07 PM)
```

Result Output 2. – Preprocessing but subtracting mean result of test

2.4 Changing the Learning Rate

The next test that was tried was the implementation of the tensorflow algorithm on the raw images with the same preprocessing done on the images beforehand. Building on the implementation of the given tutorial code, we can see that the important inputs of the model are the learning rate, the iterations/steps over the batches, and the use of preprocessing on the input data. For the first run of the model, I had the learning rate set to 0.01, an iteration size of 2500. I changed the learning rate of the model 0.01, a higher rate, with the intuition that the model would learning faster and yield a higher accuracy as a result. I believe that the preprocessing that dropped the overall accuracy rate is to blame for the result I got but regardless below is the slide from where the intuition stems:

Babysitting the learning process

Lets try to train now...

Start with small regularization and find learning rate that makes the loss go down.

loss not going down:
learning rate too low

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
```

Now let's try learning rate 1e6.

Fig 4. – Slide 71 from notes recommending change in learning rate

Below shows the code that was used to change the learning rate and the results associated with that change. As shown, the accuracy dramatically dropped with the increase of the learning rate. The intuition was not correct at this point in time because of my misunderstanding of the variable. With the increased learning rate, the model was overfit and the accuracy suffered due to the faces not fitting the model correctly. As shown below in the screenshot, the accuracy of the model was very high at 50.25% being trained with 10000 facial images and 10000 background images. The loss was relatively respectable with no preprocessing being done at 1.812. It is intuitive to me that with preprocessing of some sort, without overtraining the model, should yield in an even higher accuracy. As for the percentage now, I believe that the grayscale images could have been preprocessed somehow before being put into the dataset, seeing as I grabbed from the image bank of images with aligned features.

```
if mode == tf.estimator.ModeKeys.PREDICT:
    return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)

# Calculate Loss (for both TRAIN and EVAL modes)
loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)

# Configure the Training Op (for TRAIN mode)
if mode == tf.estimator.ModeKeys.TRAIN:
    optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01)
    train_op = optimizer.minimize(
        loss=loss,
        global_step=tf.train.get_global_step())
    return tf.estimator.EstimatorSpec(mode=mode, loss=loss, train_op=train_op)

# Add evaluation metrics (for EVAL mode)
eval_metric_ops = {
    "accuracy": tf.metrics.accuracy(
        labels=labels, predictions=predictions["classes"])}
return tf.estimator.EstimatorSpec(
    mode=mode, loss=loss, eval_metric_ops=eval_metric_ops)
```

Code Example 6. – Learning Rate changed to 0.01


```
Run cnn_mnist_face
[0.00814362 0.99185646]
[0.0088408 0.99115914]
[0.00749049 0.9925095 ]
[0.99999726 0.00000272]] (33.945 sec)
INFO:tensorflow:Saving checkpoints for 2501 into /ty/proj2\model.ckpt.
INFO:tensorflow:Loss for final step: 0.0671269.
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Starting evaluation at 2018-04-06-22:17:01
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Restoring parameters from /ty/proj2\model.ckpt-2501
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Finished evaluation at 2018-04-06-22:17:06
INFO:tensorflow:Saving dict for global step 2501: accuracy = 0.5025, global_step = 2501, loss = 1.8127041
{'accuracy': 0.5025, 'loss': 1.8127041, 'global_step': 2501}

Process finished with exit code 0

IDE and Plugin Updates: PyCharm is ready to update. (today 3:07 PM)
```

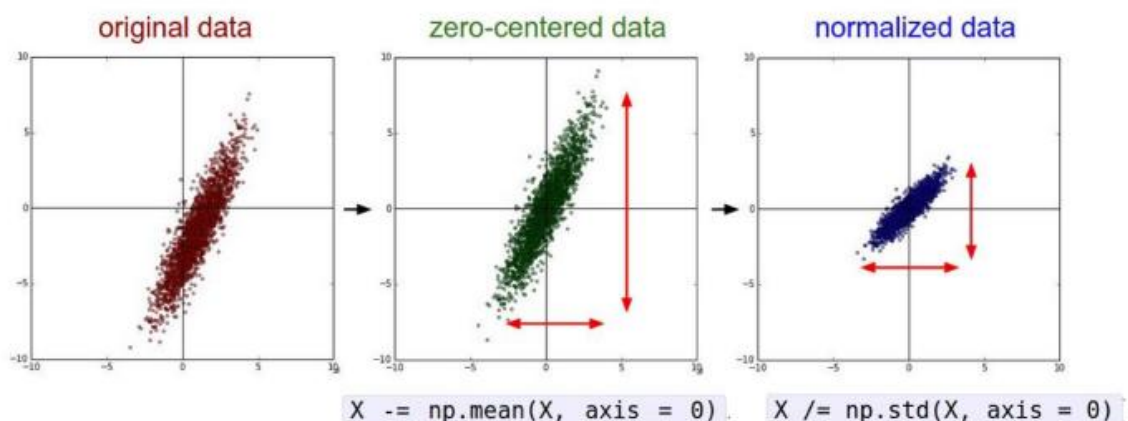
Result Output 3. – Result from changing learning rate

2.5 Preprocessing (Normalizing Data fully)

The next test that was tried was the implementation of the tensorflow algorithm on the raw images with preprocessing done on the images beforehand. Building on the implementation of the given tutorial code, we can see that the important inputs of the model are the learning rate, the iterations/steps over the batches, and the use of preprocessing on the input data. For the first run of the model, I had the learning rate set to 0.001, changed back from the previous test, an iteration size of 2500. From the notes, normalizing the data by subtracting the mean and dividing by the standard deviation. This is very much like standardizing a gaussian distribution to fit the data around a zero-mean plot.

Babysitting the learning process

Step 1: Preprocess the data



(Assume X [NxD] is data matrix,
each example in a row)

Fig 5. – Normalizing data from method in notes


```

if mode == tf.estimator.ModeKeys.PREDICT:
    return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)

# Calculate Loss (for both TRAIN and EVAL modes)
loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)

# Configure the Training Op (for TRAIN mode)
if mode == tf.estimator.ModeKeys.TRAIN:
    optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
    train_op = optimizer.minimize(
        loss=loss,
        global_step=tf.train.get_global_step())
    return tf.estimator.EstimatorSpec(mode=mode, loss=loss, train_op=train_op)

# Add evaluation metrics (for EVAL mode)
eval_metric_ops = {
    "accuracy": tf.metrics.accuracy(
        labels=labels, predictions=predictions["classes"])}
return tf.estimator.EstimatorSpec(
    mode=mode, loss=loss, eval_metric_ops=eval_metric_ops)

```

Code Example 7. – Learning Rate returned to previous state

```

#####
#Flattens and concatenates the image set passed in
#####
def standardizeImages(flatMatrix):
    meanOfData = 0 #np.mean(flatMatrix.T,axis = 0)
    stdOfData = 1 #np.mean(flatMatrix.T,axis=0)
    standardMatrix = (flatMatrix.T - meanOfData)/stdOfData

    return standardMatrix.T

```

Code Example 8. – Standardization function

As shown below, taking advantage of fully normalizing the data greatly improved the accuracy of the model. With the variance of the images weighted less in the overall image, a more robust model can be formed to depict a wider variety of facial images. As shown below in the screenshot, the accuracy of the model was very high at 99.55% being trained with 10000 facial images and 10000 background images. The loss was relatively respectable with no preprocessing being done at 0.0209. It is intuitive to me that with preprocessing of some sort, without overtraining the model, should yield in an even higher accuracy. As for the percentage now, I believe that the grayscale images could have been preprocessed somehow before being put into the dataset, seeing as I grabbed from the image bank of images with aligned features.

```
Run cnn_minist_face
[1. 0. ]
[0.9999995 0.0000005 ] (36.296 sec)
INFO:tensorflow:Saving checkpoints for 2500 into /std/proj2\model.ckpt.
INFO:tensorflow:Loss for final step: 0.008039942.
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Starting evaluation at 2018-04-06-23:24:50
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Restoring parameters from /std/proj2\model.ckpt-2500
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Finished evaluation at 2018-04-06-23:24:55
INFO:tensorflow:Saving dict for global step 2500: accuracy = 0.9955, global_step = 2500, loss = 0.020998495
{'accuracy': 0.9955, 'loss': 0.020998495, 'global_step': 2500}

Process finished with exit code 0

IDE and Plugin Updates: PyCharm is ready to update. (today 3:07 PM)
```

Result Output 4. – Full standardization of data in training set

2.6 Epoch and Learning Rate Decrease

The next test that was tried was the implementation of the tensorflow algorithm on the raw images with preprocessing done on the images beforehand. Building on the implementation of the given tutorial code, we can see that the important inputs of the model are the learning rate, the iterations/steps over the batches, and the use of preprocessing on the input data. For the first run of the model, I had the learning rate set to 0.0001, an iteration size of 2500. From the notes, I noticed that the number of epochs were changed to be a higher amount to make more passes through the batches to increase learning between sets of data. I also implemented the learning rate method shown in the notes that lowers the rate per batch. I felt that decreasing the learning rate but spending more time processing each batch of data would produce a similar, if not better result.

Babysitting the learning process

Lets try to train now...

Start with small regularization and find learning rate that makes the loss go down.

loss not going down:
learning rate too low
loss exploding:
learning rate too high

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, lr=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = 100,
                                  learning_rate=3e-3, verbose=True)

Finished epoch 1 / 10: cost 2.186654, train: 0.330000, val 0.306000, lr 3.000000e-03
Finished epoch 2 / 10: cost 2.176230, train: 0.330000, val 0.350000, lr 3.000000e-03
Finished epoch 3 / 10: cost 1.942257, train: 0.376000, val 0.352000, lr 3.000000e-03
Finished epoch 4 / 10: cost 1.827860, train: 0.329000, val 0.310000, lr 3.000000e-03
Finished epoch 5 / 10: cost inf, train: 0.128000, val 0.128000, lr 3.000000e-03
Finished epoch 6 / 10: cost inf, train: 0.144000, val 0.147000, lr 3.000000e-03
```

3e-3 is still too high. Cost explodes....

=> Rough range for learning rate we should be cross-validating is somewhere [1e-3 ... 1e-5]

Ref: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture6.pdf

Fig 6. – Changing learning rate and epoch number

```

if mode == tf.estimator.ModeKeys.PREDICT:
    return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)

# Calculate Loss (for both TRAIN and EVAL modes)
loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)

# Configure the Training Op (for TRAIN mode)
if mode == tf.estimator.ModeKeys.TRAIN:
    optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.0001)
    train_op = optimizer.minimize(
        loss=loss,
        global_step=tf.train.get_global_step())
    return tf.estimator.EstimatorSpec(mode=mode, loss=loss, train_op=train_op)

# Add evaluation metrics (for EVAL mode)
eval_metric_ops = {
    "accuracy": tf.metrics.accuracy(
        labels=labels, predictions=predictions["classes"])}
return tf.estimator.EstimatorSpec(
    mode=mode, loss=loss, eval_metric_ops=eval_metric_ops)

```

Code Example 9. – Lowered learning rate

```

# Train the model
train_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"x": train_data},
    y=train_labels,
    batch_size=50,
    num_epochs=10,
    shuffle=True)
mnist_classifier.train(
    input_fn=train_input_fn,
    steps=2500,
    hooks=[logging_hook])

# Evaluate the model and print results
eval_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"x": eval_data},
    y=eval_labels,
    num_epochs=1,
    shuffle=False)
eval_results = mnist_classifier.evaluate(input_fn=eval_input_fn)
print(eval_results)

```

Code Example 10. – Increasing epoch per batch

As shown below in the screenshot, the accuracy of the model was very high at 99.45% being trained with 10000 facial images and 10000 background images. The loss was relatively respectable with no preprocessing being done at 0.0294. It is intuitive to me that with preprocessing of some sort, without overtraining the model, should yield in an even higher accuracy. As for the percentage now, I believe that the grayscale images could have been preprocessed somehow before being put into the dataset, seeing as I grabbed from the image bank of images with aligned features. The overall test was not that much different from the previous one. I believe that an even high epoch rate would yield better results but this close to 100% is very good for a model. The loss is very high though.

```

Run cnn_mnist_face
[1. 0. ] (20.780 sec)
INFO:tensorflow:Saving checkpoints for 2500 into /epoch/proj2\model.ckpt.
INFO:tensorflow:Loss for final step: 0.055444993.
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Starting evaluation at 2018-04-06-23:46:28
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Restoring parameters from /epoch/proj2\model.ckpt-2500
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Finished evaluation at 2018-04-06-23:46:33
INFO:tensorflow:Saving dict for global step 2500: accuracy = 0.9945, global_step = 2500, loss = 0.029459117
{'accuracy': 0.9945, 'loss': 0.029459117, 'global_step': 2500}

Process finished with exit code 0
IDE and Plugin Updates: PyCharm is ready to update. (today 3:07 PM)

```

Result Output 5. – Epoch and learning rate change

3. Architecture

In this section, I would like to discuss the architecture of the CNN that was used for all of the testing above. From a neuron approach the CNN looked very much like that picture below:

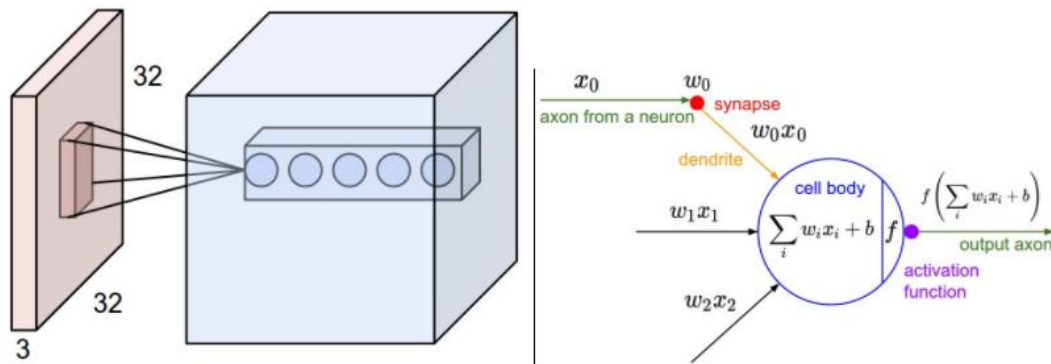


Fig 7. – Neuron analysis of neural network

The convolution layers and pooling layers can be described by the picture below best. This is a very common architecture for a classifier:

Building the CNN MNIST Classifier

Let's build a model to classify the images in the MNIST dataset using the following CNN architecture:

1. **Convolutional Layer #1:** Applies 32 5x5 filters (extracting 5x5-pixel subregions), with ReLU activation function
2. **Pooling Layer #1:** Performs max pooling with a 2x2 filter and stride of 2 (which specifies that pooled regions do not overlap)
3. **Convolutional Layer #2:** Applies 64 5x5 filters, with ReLU activation function
4. **Pooling Layer #2:** Again, performs max pooling with a 2x2 filter and stride of 2
5. **Dense Layer #1:** 1,024 neurons, with dropout regularization rate of 0.4 (probability of 0.4 that any given element will be dropped during training)
6. **Dense Layer #2 (Logits Layer):** 10 neurons, one for each digit target class (0–9).

Fig 8. – Classifier architecture used in this project

I would also like to point out some of the variable that I held constant throughout the testing process to see exactly how these variables explained above actually affect the data. The iterations steps were held constant at 2500 for speed purposes. A run with 10000 iterations was tried and though the model was very accurate the learning time was not worth it. Obviously from above, one can tell that the architecture of the CNN was held constant across all tests. This was to see which aspects of the classifier really yield higher results as opposed to using a completely different design. Within the neuron representation shown above, the activation function used was the ReLU function since it seems to yield the best results as long as the learning rate is treated with care, which I got a hands-on experience seeing the learning rate of the ReLU deplete the accuracy of the model.

4. Conclusion

In the end, I see that the use of Google's tensorflow and other machine learning APIs can be very useful in building reliable and robust models for classifiers, estimators, and CNNs in general. I have learned that training these networks can be very tedious and yield unexpected results generally. Modeling the networks like the human brain can bring about some intuition towards training the models. Such as the epoch, batch sizes, and iteration are much like a human brain looking over the same information over and over to gain a better understanding of the information. I look forward to implementing these methods in project 3 of the class to further progress my knowledge in the field.