Don Baker
Class ID 6

2017 Fall: COMP-SCI 5590/490 - Special Topics

Deep Learning

Lab Assignment

I. Introduction

This lab assignment has the following three parts:

- Implement a text classification code using the CNN model
- Illustrate the graph in TensorBoard
- Change the hyperparameter or learning rate and compare the results of the run

For this exercise, two text files rt-polarity.neg and rt-polarity.pos were used.

II. Objectives

The Objectives of this lab exercise were to write a program to classify text using the CNN model, determine the effect of changing the learning rate and then illustrating the results in TensorBoard.

III. Approaches/Methods

In order to complete this exercise, the Python programing was broken into three programs that manages the data files, trained the data, converted the text file into a useable format and then completes the CNN model.

IV. Workflow

The following code was used for each of the three files.

Training

```
#Don Baker
#Comp-Sci 5590
#Deep Learning Lab 2

import tensorflow
import numpy
import os
import time
import datetime
import FileFunction
import TextConvNet
from tensorflow.contrib import learn

#%%Parameters%%#

# Data loading parameters
#Percent of data being split for validation set
dev_percent = 0.1
#Data file for positive examples
pos_dat_file = 'rt-polarity.pos'
#Data file for negative examples
neg_dat_file = 'rt-polarity.neg'

#-->Hyperparameters
#Embedding size
embed_size = 128
#String for filter size of 3, 4, and 5
temp = "3,4,5"
#Filter size
```

```
fil_sz = list(map(int, temp.split(",")))
#Number of filters
fil_num = 128
#Original dropout keep probability
drop_keep = 0.5
#Initialize l2_lambda_reg as 0.0
l2 = 0.0
#Learning Rate
learning_rate = 1e-2

#-->Training parameters
#Size of each batch
batch_sz = 64
#Total number of epochs
epoch_num = 200
#Validation evaluations
eval_on_dev = 100
#Checkpoint evaluations
ckpt = 100
#Total number of checkpoints
ckpt_num = 5

#-->Misc Parameters
soft_pl = True
log_pl = False

#%%Prep Data for Text Classification%%#

#-->Load data
print("\nLoad data from input files...")
x_text, y = FileFunction.import_data(pos_dat_file, neg_dat_file)

#-->Build vocabulary
print("\nBuilding Vocabulary...")
#Determine the maximum length of the document from the combined data of positive and negative samples
max_document_length = max([len(x.split(" ")) for x in x_text])
#Process the vocabulary using VocabularyProcessory from tensorflow.contrib learn
process_voc = learn.preprocessing.VocabularyProcessor(max_document_length)
#Convert data into numpy array and use fit_transform from the vocabulary above
x = numpy.array(list(process_voc.fit_transform(x_text)))

#-->Randomly shuffle data
#Random number generator through numpy
numpy.random.seed(10)
#Create index for start and end for randomly shuffled data
shuffle_indices = numpy.random.permutation(numpy.arange(len(y)))
#Shuffled value of x
x_shuffled = x[shuffle_indices]
#Shuffled value of y
y_shuffled = y[shuffle_indices]

#-->Split data into training and testing data.
#Testing/Validation data is dev_percent of data. In this case it is 10% of the data
index_dev = -1 * int(dev_percent * float(len(y)))
x_train, x_dev = x_shuffled[:index_dev], x_shuffled[index_dev:]
y_train, y_dev = y_shuffled[:index_dev], y_shuffled[index_dev:]

#-->Training
#Create the tensorflow graph and session
with tensorflow.Graph().as_default():
    #Define session configuration
    configuration = tensorflow.ConfigProto(allow_soft_placement=soft_pl,log_device_placement=log_pl)
    #Create the session using the configuration created above
    sess = tensorflow.Session(config=configuration)
    with sess.as_default():
        #Classify text using TextCNN from file TextConvNet. THis is the convolution neural network for text
        #classification
        conv_neural_net = TextConvNet.TextCNN(len_seq=x_train.shape[1],class_num=y_train.shape[1],
                                              vocab=len(process_voc.vocabulary_),sz_emb=embed_size,
                                              sz_fil=fil_sz,fil_num=fil_num,l2_reg_lambda=l2)

        # Define Training procedure
        #Define the global step and create variable tensor
```

```
stp_glo = tensorflow.Variable(0, name="stp_glo", trainable=False)
#Create optimizer with a learning rate of learning_rate
opt = tensorflow.train.AdamOptimizer(learning_rate)
#Determine gradients and variables
gradient = opt.compute_gradients(conv_neural_net.loss)
#Create training optimizer
train_op = opt.apply_gradients(gradient, global_step=stp_glo)
#Determine the values of gradient values and sparsity
grad_summaries = []
for g, v in gradient:
    if g is not None:
        #Write gradient history summary
        grad_hist_summary = tensorflow.summary.histogram("{}/grad/hist".format(v.name), g)
        #Write sparsity summary
        sparsity_summary = tensorflow.summary.scalar("{}/grad/sparsity".format(v.name),
                                                     tensorflow.nn.zero_fraction(g))
        #Add to grad_Summaries from gradient history and sparsity history
        grad_summaries.append(grad_hist_summary)
        grad_summaries.append(sparsity_summary)
#Merge summary for grad_summaries
grad_summaries_merged = tensorflow.summary.merge(grad_summaries)

#Determine directory for output of the models and summary
#timestamp
timestamp = str(int(time.time()))
#Output directory
out_dir = os.path.abspath(os.path.join(os.path.curdir, "runs", timestamp))

#-->Loss Summary
loss_summary = tensorflow.summary.scalar("loss", conv_neural_net.loss)
#-->Accuracy Summary
acc_summary = tensorflow.summary.scalar("accuracy", conv_neural_net.accuracy)

#-->Train Summaries
#Train summary optimizer
train_summary_op = tensorflow.summary.merge([loss_summary, acc_summary, grad_summaries_merged])
#Train summary directory
train_summary_dir = os.path.join(out_dir, "summaries", "train")
#Train summary writer
train_summary_writer = tensorflow.summary.FileWriter(train_summary_dir, sess.graph)

#-->Validation Set Summaries
#Validation summary optimizer
dev_summary_op = tensorflow.summary.merge([loss_summary, acc_summary])
#Validation summary directory
dev_summary_dir = os.path.join(out_dir, "summaries", "dev")
#Validation summary writer
dev_summary_writer = tensorflow.summary.FileWriter(dev_summary_dir, sess.graph)

#Checkpoint directory within current directory
#Directory for created checkpoints
checkpoint_dir = os.path.abspath(os.path.join(out_dir, "checkpoints"))
checkpoint_prefix = os.path.join(checkpoint_dir, "model")
#Check if directory exists. If yes, create a new one
if not os.path.exists(checkpoint_dir):
    os.makedirs(checkpoint_dir)
#Save to tensorboard
saver = tensorflow.train.Saver(tensorflow.global_variables(), max_to_keep=ckpt_num)

#-->Write vocabulary
process_voc.save(os.path.join(out_dir, "vocab"))

#-->Initialize variables
sess.run(tensorflow.global_variables_initializer())

#-->Generate batches inside FileFuntion
batches = FileFunction.iteration_bat(
    list(zip(x_train, y_train)), batch_sz, epoch_num)
#-->Training loop.
for batch in batches:
    #Separate batch into x_batch and y_
    x_batch, y_batch = zip(*batch)
    #Define the feed_dict to load into tensorflow placeholders for conv_neural_net for training
```

```
            feed_dict = {
                conv_neural_net.input_x: x_batch,
                conv_neural_net.input_y: y_batch,
                conv_neural_net.drop_keep: drop_keep
            }
            #Run the training iteration inside each batch. Returns the step, summaries, loss, and accuracy
            _, step, summaries, loss, accuracy = sess.run(
                [train_op, stp_glo, train_summary_op, conv_neural_net.loss, conv_neural_net.accuracy],
                feed_dict)
            #Create a timestamp for each training iteration
            time_str = datetime.datetime.now().isoformat()
            print("{}: step {}, loss {:g}, acc {:g}".format(time_str, step, loss, accuracy))
            train_summary_writer.add_summary(summaries, step)

            current_step = tensorflow.train.global_step(sess, stp_glo)

            #Validation part of training
            if current_step % eval_on_dev == 0:
                print("\nEvaluation:")
                # Define the feed_dict to load into tensorflow placeholders for conv_neural_net for validation
                feed_dict = {
                    conv_neural_net.input_x: x_batch,
                    conv_neural_net.input_y: y_batch,
                    conv_neural_net.drop_keep: 1.0
                }
                #Run the validation iteration inside each batch. Returns the step, summaries, loss, and accuracy
                step, summaries, loss, accuracy = sess.run(
                    [stp_glo, dev_summary_op, conv_neural_net.loss, conv_neural_net.accuracy],
                    feed_dict)
                # Create a timestamp for each training iteration
                time_str = datetime.datetime.now().isoformat()
                print("{}: step {}, loss {:g}, acc {:g}".format(time_str, step, loss, accuracy))
                #Write validation summary
                if dev_summary_writer:
                    dev_summary_writer.add_summary(summaries, step)
                print("")
            #Save model after each checkpoint
            if current_step % ckpt == 0:
                path = saver.save(sess, checkpoint_prefix, global_step=current_step)
                print("Saved model checkpoint to {}\n".format(path))
```

This code completes functions with the data file.

The following is the code for the text conversion and CNN model.

```
import tensorflow

tensorflow.reset_default_graph()

class TextCNN(object):
    # len_seq = length of the sentences,classes = number of classes in output layer, vocab = size of vocabulary
(needed
    # to define size of embedding layer), embed_sz = dimensionality of embeddings, fil_sz = number of words
    # convolutional filters cover, filters = number of filters per filter size
    def __init__(
      self, len_seq, class_num, vocab,sz_emb, sz_fil, fil_num, l2_reg_lambda=0.0):

        # Placeholders for input, output and dropout
        self.input_x = tensorflow.placeholder(tensorflow.int32, [None, len_seq], name="input_x")
        self.input_y = tensorflow.placeholder(tensorflow.float32, [None, class_num], name="input_y")
        self.drop_keep = tensorflow.placeholder(tensorflow.float32, name="drop_keep")

        # Keeping track of l2 regularization loss
        l2_loss = tensorflow.constant(0.0)

        # Define embedding layer that maps vocabulary word indices into low-dim vectors
        # This is a lookup table learned from the data
```

```
        # Forces an operation to be executed on CPU instead of the default GPU if available
        # Embeddings does not support GPU execution so CPU is needed to execute
        with tensorflow.device('/cpu:0'), tensorflow.name_scope("embedding"):
            # Embedding matrix learned at training. tensorflow.embedding_lookup creates the embedding operation
            # Result will be a 3-D tensor with shape[None, len_seq, sz_emb]
            self.W = tensorflow.Variable(tensorflow.random_uniform([vocab, sz_emb], -1.0, 1.0),name="W")
            self.emb_ch = tensorflow.nn.embedding_lookup(self.W, self.input_x)
            self.emb_ch_exp = tensorflow.expand_dims(self.emb_ch, -1)

        # Build convolutional layers followed by max-pooling
        # Each convolution produces tensors with different shapes. Layers will be created for each of them
        #  then combined into a feature vector

        outs = []
        for i, filter_size in enumerate(sz_fil):
            with tensorflow.name_scope("conv-maxpool-%s" % filter_size):
                # Layer for Convolution
                shape_fil = [filter_size, sz_emb, 1, fil_num]
                W = tensorflow.Variable(tensorflow.truncated_normal(shape_fil, stddev=0.1), name="W")
                b = tensorflow.Variable(tensorflow.constant(0.1, shape=[fil_num]), name="b")
                convolution = tensorflow.nn.conv2d(self.emb_ch_exp,W,strides=[1, 1, 1, 1],padding="VALID",
                                            name="conv")
                # Nonlinearity Application
                h = tensorflow.nn.relu(tensorflow.nn.bias_add(convolution, b), name="relu")
                # Maxpooling over the outputs
                pd = tensorflow.nn.max_pool(h,ksize=[1, len_seq - filter_size + 1, 1, 1],strides=[1, 1, 1, 1],
                                                padding='VALID',name="pool")
                outs.append(pd)

        #Combine into feature vector
        num_filters_total = fil_num * len(sz_fil)
        self.h_pool = tensorflow.concat(outs, 3)
        self.h_pool_flat = tensorflow.reshape(self.h_pool, [-1, num_filters_total])

        # Dropout Layer
        # Dropout later desables a fraction of its neurons. This prevents neurons from co-adapting
        # and forces them to learn individually useful features

        # Add the dropout layer
        with tensorflow.name_scope("dropout"):
            self.h_drop = tensorflow.nn.dropout(self.h_pool_flat, self.drop_keep)

        # Determine prediction and score from feature vector from max pooling with the dropout applied
        # Complete matrix multiplication and pick the class with highest score
        with tensorflow.name_scope("output"):
            W = tensorflow.get_variable("W",shape=[num_filters_total, class_num],
                                        initializer=tensorflow.contrib.layers.xavier_initializer())
            b = tensorflow.Variable(tensorflow.constant(0.1, shape=[class_num]), name="b")
            l2_loss += tensorflow.nn.l2_loss(W)
            l2_loss += tensorflow.nn.l2_loss(b)
            self.scores = tensorflow.nn.xw_plus_b(self.h_drop, W, b, name="scores")
            self.predictions = tensorflow.argmax(self.scores, 1, name="predictions")

        # Calculate the loss
        with tensorflow.name_scope("loss"):
            losses = tensorflow.nn.softmax_cross_entropy_with_logits_v2(logits=self.scores, labels=self.input_y)
            self.loss = tensorflow.reduce_mean(losses) + l2_reg_lambda * l2_loss

        # Calculate the accuracy
        with tensorflow.name_scope("accuracy"):
            correct_predictions = tensorflow.equal(self.predictions, tensorflow.argmax(self.input_y, 1))
            self.accuracy = tensorflow.reduce_mean(tensorflow.cast(correct_predictions, "float"),
name="accuracy")


import numpy
import re

#This function takes the input text (string) and then line by line, cleans up the data by removing the special
#characters shown below. It will a return a string that is completely stripped of special characters and
unwanted
#characters,as well as all lowercase characters
```

```
def data_cleanup(string):
    string = re.sub(r"[^A-Za-z0-9(),!?\'\`]", " ", string)
    string = re.sub(r"\'s", " \'s", string)
    string = re.sub(r"\'ve", " \'ve", string)
    string = re.sub(r"n\'t", " n\'t", string)
    string = re.sub(r"\'re", " \'re", string)
    string = re.sub(r"\'d", " \'d", string)
    string = re.sub(r"\'ll", " \'ll", string)
    string = re.sub(r",", " , ", string)
    string = re.sub(r"!", " ! ", string)
    string = re.sub(r"\(", " \( ", string)
    string = re.sub(r"\)", " \) ", string)
    string = re.sub(r"\?", " \? ", string)
    string = re.sub(r"\s{2,}", " ", string)
    return string.strip().lower()

#This function pulls data from a positive data file and negative data file. It will pull in the data (x) and
then
#clean the data by sending it to the function data_cleanup(sent) sentence by sentence. This function will return
x_text
#which is the clean data set, and it will also return y, which is a collection of the positive and negative
labels
#in the dataset. Y will be returned as a numpy array
def import_data(positive_data_file, negative_data_file):
    # Load data from files pos and neg datasets
    #Open pos dataset and read it line by line
    positive_examples = list(open(positive_data_file, "r").readlines())
    #Strip the data contained in the pos dataset
    positive_examples = [s.strip() for s in positive_examples]
    #Open neg dataset adn read it line by line
    negative_examples = list(open(negative_data_file, "r").readlines())
    #Strip the dadta contained in the pos dataset
    negative_examples = [s.strip() for s in negative_examples]
    #Split data contained in x_text by words (tokenization)
    #Compbine the positive and negative samples and store result in x_text
    x_text = positive_examples + negative_examples
    #Clean the data in x_text by sending it to data_cleanup() sentence by sentence in the data of x_text
    x_text = [data_cleanup(sent) for sent in x_text]
    # Generate labels
    #Create positive labels from the positive dataset
    positive_labels = [[0, 1] for _ in positive_examples]
    #Create negative labels from the negative dataset
    negative_labels = [[1, 0] for _ in negative_examples]
    #Concat results in a numpy array stored in y. This is a collection of both positive and negative labels
    y = numpy.concatenate([positive_labels, negative_labels], 0)
    #Returns both x_text and y
    return [x_text, y]

#This function returns batches for the training procedure to traverse through. It will date the data collection
of
#both x and y, the size of each batch, the number of epochs, and shuffle which by default True.
def iteration_bat(data, sz_bat, num_epochs, shuffle=True):
    #Convert the received data into a numpy array
    data = numpy.array(data)
    #Determine the size of the received data
    data_size = len(data)
    #Determine the number of batches per epoch from the length of data divided by the size of each batch, then
add 1
    epoch_bat = int((len(data)-1)/sz_bat) + 1
    #Travers through each epoch a total of num_epochs times
    for epoch in range(num_epochs):
        #This if/else statement shuffles the data if the value of shuffle is True. By default, it is true
        if shuffle:
            #Determine the index for the shuffled data
            index = numpy.random.permutation(numpy.arange(data_size))
            dat_shuf = data[index]
        #If value of shuffle is false, the data will not be shuffled
        else:
            dat_shuf = data
        #for each batch in the range number of epochs per batch, determine the start and end index for the
shuffled
        #data
        for z in range(epoch_bat):
```

```
            start = z * sz_bat
            end = min((z + 1) * sz_bat, data_size)
            yield dat_shuf[start:end]
```

V. Datasets

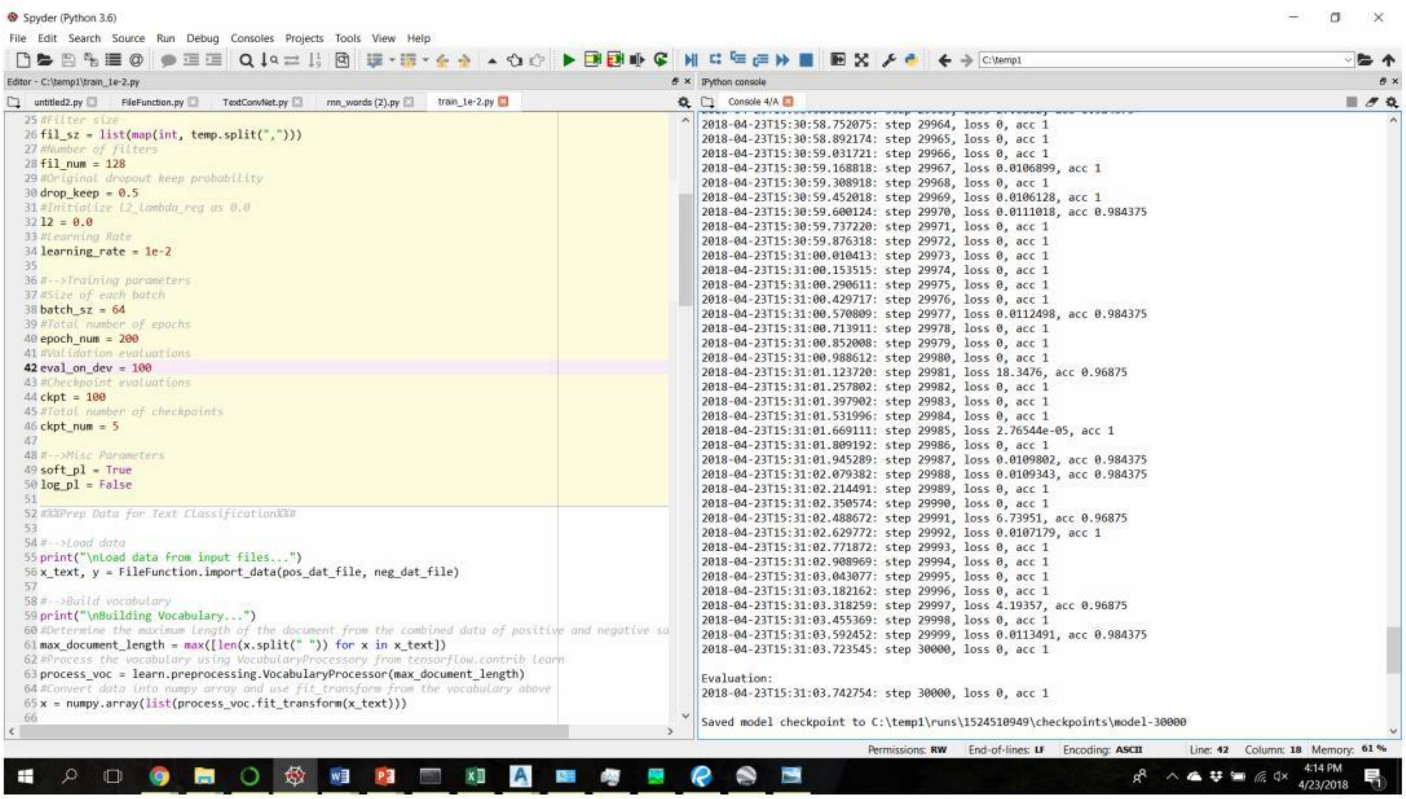
The datasets used were rt-polarity.neg and rt-polarity.pos.

VI. Parameters

The main parameters explored in this lab were the learning rate or hyperparameter and the number of iteration. The two learning rates used were 1e-2 and 9e-5. The number of iterations was 30,000 for each run.

VII. Evaluation and Discussion

The evaluation of the data was completed using the above referenced models. The following screenshot illustrates the model.



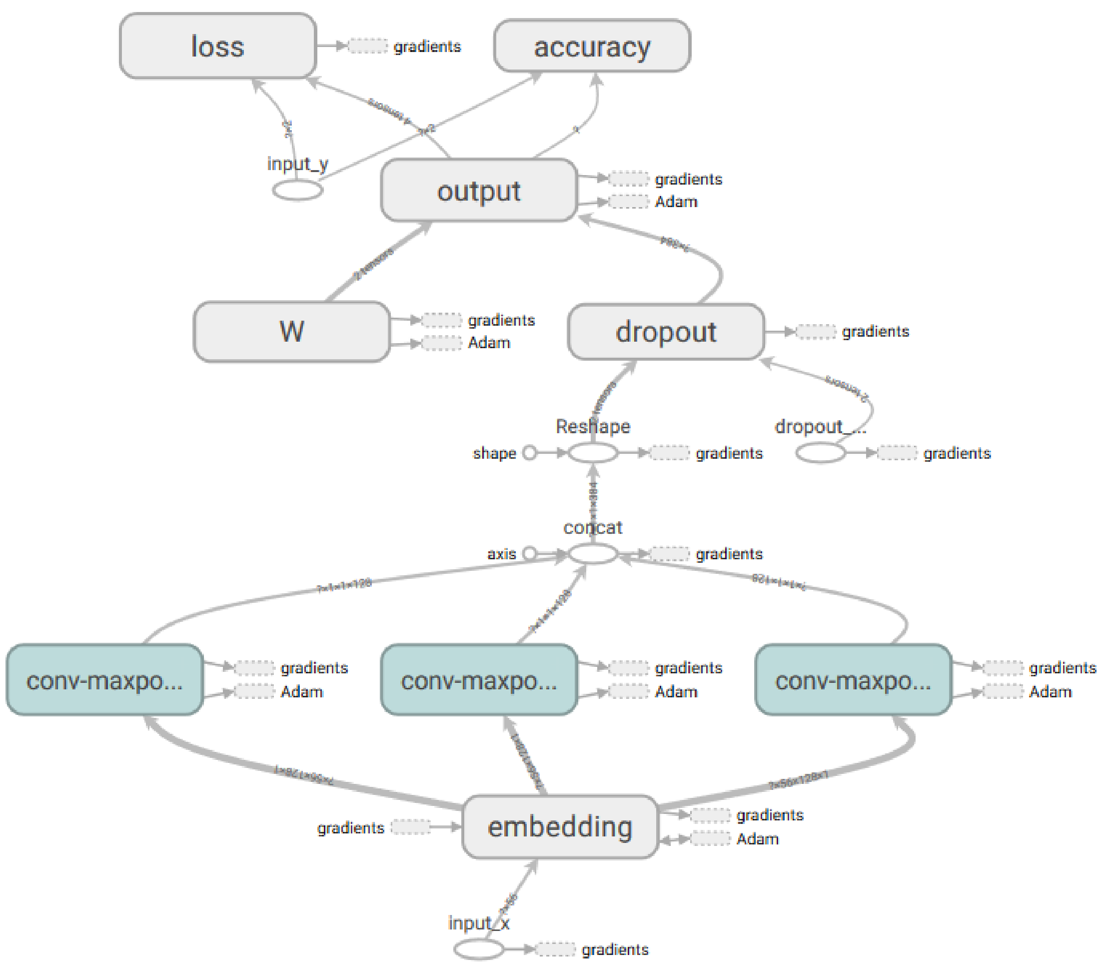
The results for the run with a learning rate of 1e-2 resulted in a loss of 0 and an accuracy of 1. The run with a learning rate of 9e-5 resulted in a loss of 0.000197424 and an accuracy of 1.

VIII. Conclusion

Both learning rates used resulted in an accuracy of 1. However the faster learning rate resulted in a great loss.

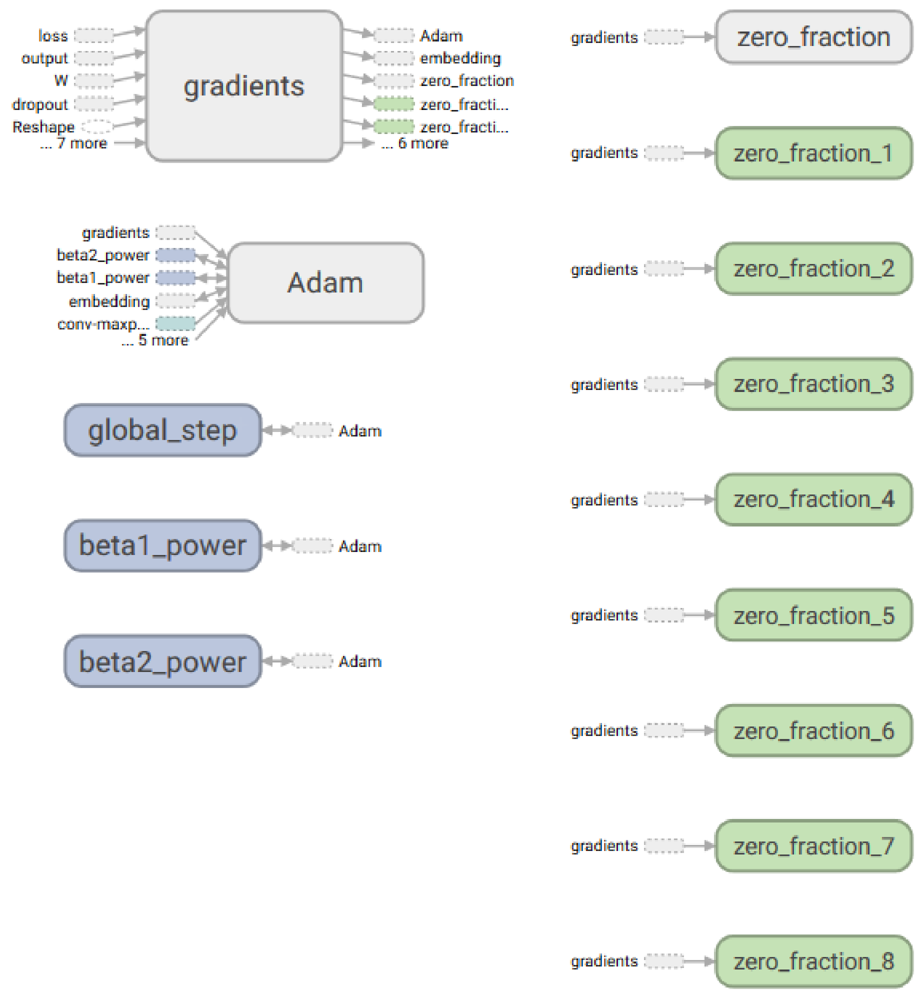
The following graphs were obtained from TensorBoard.

Main Graph



Don Baker
Class ID 6
Page 9

Auxiliary Nodes



Don Baker
Class ID 6
Page 10