DL-Lab 2 Report

VINAYCHANDRA VASAMSETTI, Class ID: 52

Introduction

In class, we learned about the properties of convolutional neural networks (CNN) and how one can be used to create a text classifier that can determine which of two classes a given line of text belongs to. This binary classification of text can be very useful in the real world, such as when companies wish to classify tweets regarding the company as being either positive or negative. Knowing how useful binary text classification can be, the next question must be whether classifying text into one of many categories can be just as useful. The purpose of this lab is to explore the idea of multi-class classification using a CNN and its implementation in Tensorflow.

Objectives

The objective of this lab is to explore multi-class classification using convolutional nerual networks in Tensorflow to determine which of several categories a given string of text belongs to. In order to meet our objective we will use a data consisting of text files grouped together by subject to train our network. Once the network has been trained we will see whether it is able to correctly predict which subject a never before seen text file belongs to, based on its features.

Approaches

To achieve our objective, the in-class CNN example will be improved upon to accommodate not only multiple classes, but a corpus of files for each class as well. Allowing more than one file to be used for training a class will let us increase the data set size more easily. The design of the CNN used will remain the same as that of the in-class example so that we may compare the results of the two approaches. In future attempts we may wish to tune some parameters to achieve higher accuracy.

Workflow

Process Data

A data set consisting of 5 or more classes needs to be obtained.
The in-class example must be altered to accept more than two classes and more than one file per class.
Each file must be cleaned and labeled before being fed to the CNN.

Train Network

The text input and labels are to be fed to the network in batches.
The words from the input are embedded to create feature vectors which will get applied to weights in the network and the weights will be updated over many iterations to determine the most important features.

Evaluate

A portion of the data set is used to test the accuracy of the network after a specified number of steps.
The network predicts the correct class of given feature vectors which is then compared to the correct label.

Data Set

20 newsgroups was used as the data set and originally contained 20 categories with 1000 articles within each category. Related categories were condensed down to 7 total categories and only approximately 10 articles were kept for each category to ensure the model could be trained in a reasonable amount of time.

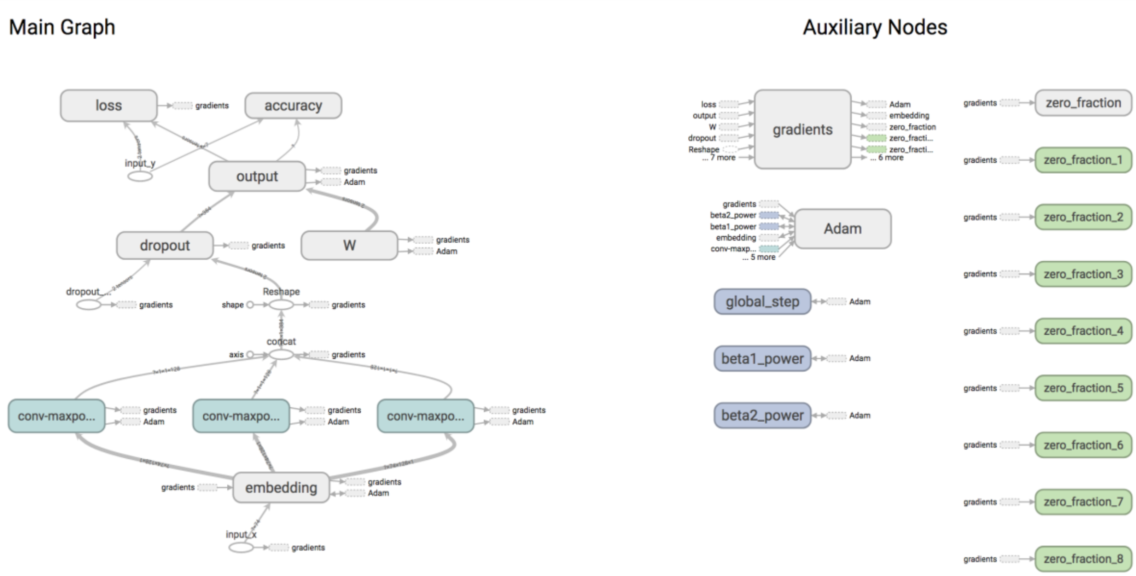
Parameters

Parameters were kept the same as the in-class example:

- Batch Size: 64 lines of text processed at a time
- Epochs: 200 iterations over the training data
- Steps Before Evaluation: 100 batches processed before calculating accuracy and loss
- Steps Before Checkpoint: 100 batches processed before writing to file
- Stored Checkpoints: 5 of the last checkpoints retained

Evaluation & Discussion

Graph



The structure of the CNN is reliable with the in-class case since our goal was to watch how well the same CNN would perform while ordering between different unmistakable classes.

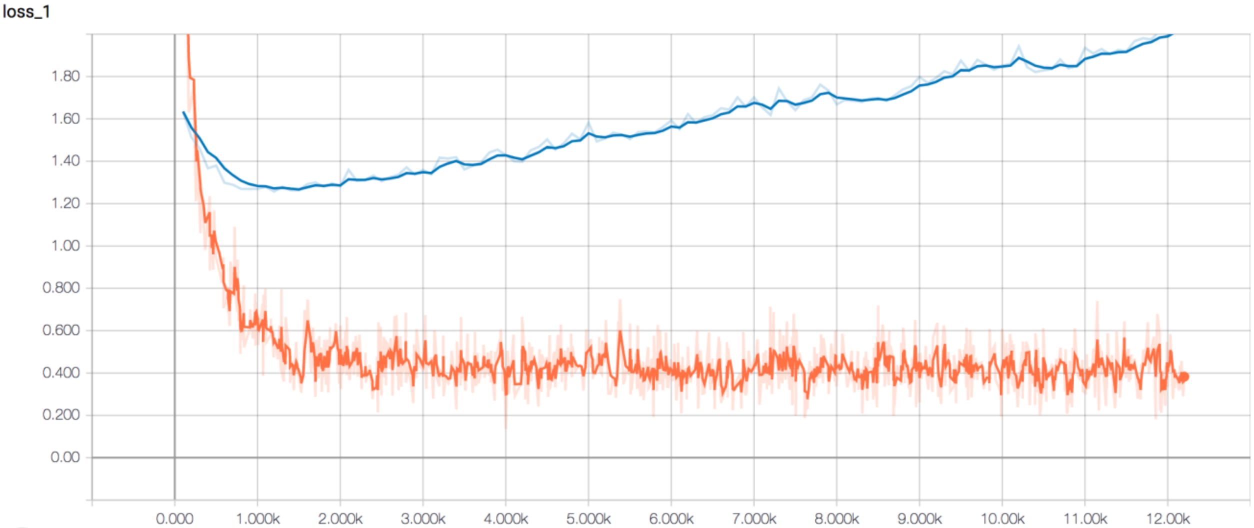
Accuracy

accuracy_1

It is normal that the test accuracy will be much lower than the training precision however our outcomes demonstrate a critical contrast between the two. The training accuracy accomplishes satisfactory rates while the test accuracy is no place close worthy.

Loss

loss_1



The training loss seems to demonstrate an alluring impact by beginning high, of course, and rapidly diminishing before leveling off to a relatively static rate. Nonetheless, the test loss demonstrates a significantly less attractive outcome. While the test loss decreases at the outset, it rapidly achieves its least point, which is still high, before starting to increment. This demonstrates our system isn't enhancing with an ever increasing number of emphasis and that a few parameters should be tuned to accomplish better outcomes.

Conclusion

Here we provide a detailed summary /conclusion for the assignment theory uploaded till now.

Train.py DataHelper.py TextCNN.py

```
        self.h_pool_flat = tf.reshape(self.h_pool, [-1, num_filters_total])

        # Add dropout
        with tf.name_scope("dropout"):
            self.h_drop = tf.nn.dropout(self.h_pool_flat, self.dropout_keep_prob)

        # Final (unnormalized) scores and predictions
        with tf.name_scope("output"):
            W = tf.get_variable(
                "W",
                shape=[num_filters_total, num_classes],
                initializer=tf.contrib.layers.xavier_initializer())
            b = tf.Variable(tf.constant(0.1, shape=[num_classes]), name="b")
            l2_loss += tf.nn.l2_loss(W)
            l2_loss += tf.nn.l2_loss(b)
            self.scores = tf.nn.xw_plus_b(self.h_drop, W, b, name="scores")
            self.predictions = tf.argmax(self.scores, 1, name="predictions")

        # CalculateMean cross-entropy loss
        with tf.name_scope("loss"):
            losses = tf.nn.softmax_cross_entropy_with_logits(logits=self.scores, labels=self.input_y)
            self.loss = tf.reduce_mean(losses) + l2_reg_lambda * l2_loss

        # Accuracy
        with tf.name_scope("accuracy"):
            correct_predictions = tf.equal(self.predictions, tf.argmax(self.input_y, 1))
            self.accuracy = tf.reduce_mean(tf.cast(correct_predictions, "float"), name="accuracy")
```

While the in-class example was successfully altered to accommodate multi-class classification, using the previous parameter and hyper-parameter values, and a relatively small data set, seems to result in a text classifier which is only slightly better than random guessing (~ 60% accuracy). Because there are more files available from the original data set that were not used in order to keep training time to a minimum, the easiest way to improve accuracy would be to add those files back into the data set used. However, before this is done, it would be wise to run several more experiments with different variations of parameter values on the reduced data set, to conclude the optimal tuning without wasting effort training the larger data set. In sum, our objective was met by showing that a binary text classifier could be used as a base for a multi-class text classifier. In future work, the focus will shift to reducing the loss and improving the accuracy of the classifier.

CODE FOR IMPLENTATION :

Train.py

```python
import datetime
import os
import time

import numpy as np
import tensorflow as tf
from tensorflow.contrib import learn

import DataHelper
from TextCNN import TextCNN

# Parameters
# ==================================================

# Data loading params
tf.flags.DEFINE_float("dev_sample_percentage", .1, "Percentage of the training data to use for validation")
tf.flags.DEFINE_string("alt_data", "./newsgroups/alt", "Data source for articles about atheism.")
tf.flags.DEFINE_string("comp_data", "./newsgroups/comp", "Data source for computer articles.")
tf.flags.DEFINE_string("misc_data", "./newsgroups/misc", "Data source for sales ads.")
tf.flags.DEFINE_string("rec_data", "./newsgroups/rec", "Data source for sports articles.")
tf.flags.DEFINE_string("sci_data", "./newsgroups/sci", "Data source for scientific articles.")
tf.flags.DEFINE_string("soc_data", "./newsgroups/soc", "Data source for religious articles.")
tf.flags.DEFINE_string("talk_data", "./newsgroups/talk", "Data source for open forums.")

# Model Hyperparameters
tf.flags.DEFINE_integer("embedding_dim", 128, "Dimensionality of character embedding (default: 128)")
tf.flags.DEFINE_string("filter_sizes", "3,4,5", "Comma-separated filter sizes (default: '3,4,5')")
tf.flags.DEFINE_integer("num_filters", 128, "Number of filters per filter size (default: 128)")
tf.flags.DEFINE_float("dropout_keep_prob", 0.5, "Dropout keep probability (default: 0.5)")
tf.flags.DEFINE_float("l2_reg_lambda", 0.0, "L2 regularization lambda (default: 0.0)")

# Training parameters
```

Train.py

```python
# Training parameters
tf.flags.DEFINE_integer("batch_size", 64, "Batch Size (default: 64)")
tf.flags.DEFINE_integer("num_epochs", 200, "Number of training epochs (default: 200)")
tf.flags.DEFINE_integer("evaluate_every", 100, "Evaluate model on dev set after this many steps (default: 100)")
tf.flags.DEFINE_integer("checkpoint_every", 100, "Save model after this many steps (default: 100)")
tf.flags.DEFINE_integer("num_checkpoints", 5, "Number of checkpoints to store (default: 5)")
# Misc Parameters
tf.flags.DEFINE_boolean("allow_soft_placement", True, "Allow device soft device placement")
tf.flags.DEFINE_boolean("log_device_placement", False, "Log placement of ops on devices")

FLAGS = tf.flags.FLAGS
FLAGS._parse_flags()
print("\nParameters:")
for attr, value in sorted(FLAGS.__flags.items()):
    print("{}={}".format(attr.upper(), value))
print("")


# Data Preparation
# ==================================================

# Load data
print("Loading data...")
data_flags = [FLAGS.alt_data, FLAGS.comp_data, FLAGS.misc_data, FLAGS.rec_data,
              FLAGS.sci_data, FLAGS.soc_data, FLAGS.talk_data]
x_text, y = DataHelper.load_data_and_labels(data_flags)


# Build vocabulary
max_document_length = max([len(x.split(" ")) for x in x_text])
vocab_processor = learn.preprocessing.VocabularyProcessor(max_document_length)
x = np.array(list(vocab_processor.fit_transform(x_text)))
```

Train.py

```python
# Randomly shuffle data
np.random.seed(10)
shuffle_indices = np.random.permutation(np.arange(len(y)))
x_shuffled = x[shuffle_indices]
y_shuffled = y[shuffle_indices]

# Split train/test set
# TODO: This is very crude, should use cross-validation
dev_sample_index = -1 * int(FLAGS.dev_sample_percentage * float(len(y)))
x_train, x_dev = x_shuffled[:dev_sample_index], x_shuffled[dev_sample_index:]
y_train, y_dev = y_shuffled[:dev_sample_index], y_shuffled[dev_sample_index:]
print(x_train.shape)
print(y_train.shape)
print("Vocabulary Size: {:d}".format(len(vocab_processor.vocabulary_)))
print("Train/Dev split: {:d}/{:d}".format(len(y_train), len(y_dev)))

# Training
# ==================================================

with tf.Graph().as_default():
    session_conf = tf.ConfigProto(
      allow_soft_placement=FLAGS.allow_soft_placement,
      log_device_placement=FLAGS.log_device_placement)
    sess = tf.Session(config=session_conf)
    with sess.as_default():
        cnn = TextCNN(
            sequence_length=x_train.shape[1],
            num_classes=y_train.shape[1],
            vocab_size=len(vocab_processor.vocabulary_),
            embedding_size=FLAGS.embedding_dim,
            filter_sizes=list(map(int, FLAGS.filter_sizes.split(","))),
            num_filters=FLAGS.num_filters,
```

Train.py

```python
        # Define Training procedure
        global_step = tf.Variable(0, name="global_step", trainable=False)
        optimizer = tf.train.AdamOptimizer(1e-1)
        grads_and_vars = optimizer.compute_gradients(cnn.loss)
        train_op = optimizer.apply_gradients(grads_and_vars, global_step=global_step)

        # Keep track of gradient values and sparsity (optional)
        grad_summaries = []
        for g, v in grads_and_vars:
            if g is not None:
                grad_hist_summary = tf.summary.histogram("{}/grad/hist".format(v.name), g)
                sparsity_summary = tf.summary.scalar("{}/grad/sparsity".format(v.name), tf.nn.zero_fraction(g))
                grad_summaries.append(grad_hist_summary)
                grad_summaries.append(sparsity_summary)
        grad_summaries_merged = tf.summary.merge(grad_summaries)

        # Output directory for models and summaries
        timestamp = str(int(time.time()))
        out_dir = os.path.abspath(os.path.join(os.path.curdir, "runs", timestamp))
        print("Writing to {}\n".format(out_dir))

        # Summaries for loss and accuracy
        loss_summary = tf.summary.scalar("loss", cnn.loss)
        acc_summary = tf.summary.scalar("accuracy", cnn.accuracy)

        # Train Summaries
        train_summary_op = tf.summary.merge([loss_summary, acc_summary, grad_summaries_merged])
        train_summary_dir = os.path.join(out_dir, "summaries", "train")
        train_summary_writer = tf.summary.FileWriter(train_summary_dir, sess.graph)
```

Train.py

```python
# Dev summaries
dev_summary_op = tf.summary.merge([loss_summary, acc_summary])
dev_summary_dir = os.path.join(out_dir, "summaries", "dev")
dev_summary_writer = tf.summary.FileWriter(dev_summary_dir, sess.graph)

# Checkpoint directory. Tensorflow assumes this directory already exists so we need to create it
checkpoint_dir = os.path.abspath(os.path.join(out_dir, "checkpoints"))
checkpoint_prefix = os.path.join(checkpoint_dir, "model")
if not os.path.exists(checkpoint_dir):
    os.makedirs(checkpoint_dir)
saver = tf.train.Saver(tf.global_variables(), max_to_keep=FLAGS.num_checkpoints)

# Write vocabulary
vocab_processor.save(os.path.join(out_dir, "vocab"))

# Initialize all variables
sess.run(tf.global_variables_initializer())

def train_step(x_batch, y_batch):
    """
    A single training step
    """
    feed_dict = {
      cnn.input_x: x_batch,
      cnn.input_y: y_batch,
      cnn.dropout_keep_prob: FLAGS.dropout_keep_prob
    }
    _, step, summaries, loss, accuracy = sess.run(
        [train_op, global_step, train_summary_op, cnn.loss, cnn.accuracy],
        feed_dict)
    time_str = datetime.datetime.now().isoformat()
```

Train.py

```python
    print("{}: step {}, loss {:g}, acc {:g}".format(time_str, step, loss, accuracy))
    train_summary_writer.add_summary(summaries, step)

def dev_step(x_batch, y_batch, writer=None):
    """
    Evaluates model on a dev set
    """
    feed_dict = {
      cnn.input_x: x_batch,
      cnn.input_y: y_batch,
      cnn.dropout_keep_prob: 1.0
    }
    step, summaries, loss, accuracy = sess.run(
        [global_step, dev_summary_op, cnn.loss, cnn.accuracy],
        feed_dict)
    time_str = datetime.datetime.now().isoformat()
    print("{}: step {}, loss {:g}, acc {:g}".format(time_str, step, loss, accuracy))
    if writer:
        writer.add_summary(summaries, step)

# Generate batches
batches = DataHelper.batch_iter(
    list(zip(x_train, y_train)), FLAGS.batch_size, FLAGS.num_epochs)
```

TEXT CNN CODE :

```python
import tensorflow as tf


class TextCNN(object):
    """
    A CNN for text classification.
    Uses an embedding layer, followed by a convolutional, max-pooling and softmax layer.
    """
    def __init__(
      self, sequence_length, num_classes, vocab_size,
      embedding_size, filter_sizes, num_filters, l2_reg_lambda=0.0):

        # Placeholders for input, output and dropout
        self.input_x = tf.placeholder(tf.int32, [None, sequence_length], name="input_x")
        self.input_y = tf.placeholder(tf.float32, [None, num_classes], name="input_y")
        self.dropout_keep_prob = tf.placeholder(tf.float32, name="dropout_keep_prob")

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

        # Embedding layer
        with tf.device('/cpu:0'), tf.name_scope("embedding"):
            self.W = tf.Variable(
                tf.random_uniform([vocab_size, embedding_size], -1.0, 1.0),
                name="W")
            self.embedded_chars = tf.nn.embedding_lookup(self.W, self.input_x)
            self.embedded_chars_expanded = tf.expand_dims(self.embedded_chars, -1)

        # Create a convolution + maxpool layer for each filter size
        pooled_outputs = []
        for i, filter_size in enumerate(filter_sizes):
            with tf.name_scope("conv-maxpool-%s" % filter_size):
```

Train.py × DataHelper.py × TextCNN.py ×

```python
                # Convolution Layer
                filter_shape = [filter_size, embedding_size, 1, num_filters]
                W = tf.Variable(tf.truncated_normal(filter_shape, stddev=0.1), name="W")
                b = tf.Variable(tf.constant(0.1, shape=[num_filters]), name="b")
                conv = tf.nn.conv2d(
                    self.embedded_chars_expanded,
                    W,
                    strides=[1, 1, 1, 1],
                    padding="VALID",
                    name="conv")
                # Apply nonlinearity
                h = tf.nn.relu(tf.nn.bias_add(conv, b), name="relu")
                # Max pooling over the outputs
                pooled = tf.nn.max_pool(
                    h,
                    ksize=[1, sequence_length - filter_size + 1, 1, 1],
                    strides=[1, 1, 1, 1],
                    padding='VALID',
                    name="pool")
                pooled_outputs.append(pooled)

        # Combine all the pooled features
        num_filters_total = num_filters * len(filter_sizes)
        self.h_pool = tf.concat(pooled_outputs, 3)
        self.h_pool_flat = tf.reshape(self.h_pool, [-1, num_filters_total])

        # Add dropout
        with tf.name_scope("dropout"):
            self.h_drop = tf.nn.dropout(self.h_pool_flat, self.dropout_keep_prob)
```

Train.py × DataHelper.py × TextCNN.py ×

```python
        self.h_pool_flat = tf.reshape(self.h_pool, [-1, num_filters_total])

        # Add dropout
        with tf.name_scope("dropout"):
            self.h_drop = tf.nn.dropout(self.h_pool_flat, self.dropout_keep_prob)

        # Final (unnormalized) scores and predictions
        with tf.name_scope("output"):
            W = tf.get_variable(
                "W",
                shape=[num_filters_total, num_classes],
                initializer=tf.contrib.layers.xavier_initializer())
            b = tf.Variable(tf.constant(0.1, shape=[num_classes]), name="b")
            l2_loss += tf.nn.l2_loss(W)
            l2_loss += tf.nn.l2_loss(b)
            self.scores = tf.nn.xw_plus_b(self.h_drop, W, b, name="scores")
            self.predictions = tf.argmax(self.scores, 1, name="predictions")

        # CalculateMean cross-entropy loss
        with tf.name_scope("loss"):
            losses = tf.nn.softmax_cross_entropy_with_logits(logits=self.scores, labels=self.input_y)
            self.loss = tf.reduce_mean(losses) + l2_reg_lambda * l2_loss

        # Accuracy
        with tf.name_scope("accuracy"):
            correct_predictions = tf.equal(self.predictions, tf.argmax(self.input_y, 1))
            self.accuracy = tf.reduce_mean(tf.cast(correct_predictions, "float"), name="accuracy")
```