ECE1504 Statistical Learning
University of Toronto

Assignment 2:
Neural Networks

Send Email for Q&A
Due date: 23 November 2018
Electronic submission to: ece1504.fall2018@gmail.com

General Note:

- The purpose of this assignment is to investigate the classification performance of neural networks. In this assignment, you will gain some experience in training a neural network and will use an effective way to avoid overfitting. All the implementations need to be done using Python and TensorFlow. You are encouraged to look up TensorFlow APIs for useful utility functions, at: `https://www.tensorflow.org/api_docs/python/`.
- Full points are given for complete solutions, including justifying the choices or assumptions you made to solve each question. Both a written report and your complete source code (as an appendix and separate files) should be included in the final submission.
- Homework assignments are to be solved by yourself or in groups of two. You are encouraged to discuss the assignment with other students, but you must solve it within your own group. Make sure to be closely involved in all aspects of the assignment. If you work in a group, please indicate the contribution percentage from each group member at the beginning of your report.
- Please call the following function before building your computational graph to make your implementation outputs stable across runs.

```
def reset_graph(seed=42):
    tf.reset_default_graph()
    tf.set_random_seed(seed)
    np.random.seed(seed)
```

1 Deep Learning [60 pt.]

In Section 1.1, some techniques for training deep neural netowrks (DNN) are briefly discussed. You are encouraged to check out other sources for more discussion. Then, in Section 1.2, you are asked to design a DNN for MNIST data set.

1.1 Background

- **Initialization:** Xavier initialization can significantly speed up the training of deep neural networks. In the recent paper, He et al. [1] proposed similar initialization strategies for different activation functions. For instance for ReLU activation function and its variants, He initialization uses the normal distribution with variance $\sigma^2 = \frac{4}{n_{\text{inputs}}+n_{\text{outputs}}}$. In TensorFlow you can use `tf.contrib.layers.variance_scaling_initializer()` for He initialization.

- **Activation Function:** The problem of vanishing gradient[2] in part is due to a poor choice of activation function. One of the widely used activation function that behaves much better in deep neural networks is ReLU. However, it suffers from a problem of dying ReLUs: during training if the neurons weights updated such that the input to the neuron is negative it will start outputting 0. To solve this problem, some recent papers propose variants of the ReLU function, such as the leaky ReLU, randomized ReLU (RReLU), parametric leaking ReLU (PReLU), and experiential linear unit (ELU). TensorFlow provides functions such as `tf.nn.elu` and `tf.nn.relu` to implement different activation functions.

- **Batch Normalization:** He and Xavier initialization along with the good choice of activation function can reduce the vanishing gradient problems at the beginning of the training. However, these methods do not guarantee that the vanishing gradient problem won't come back during training. Batch Normalization (BN) which is proposed by Sergey Ioffe ad Christian Szegedy [3] addresses the problem that the distribution of each layer's input changes during training. TensorFlow provides `tf.layers.batch.normalization()` function which handles BN method for you.

- **Dropout:** Arguably the most popular regularization technique is dropout. At every training step, every neuron has a probability p of being temporarily ignored during training but it may be active in the next training step. To implement dropout using TensorFlow, you can simply apply the `tf.layers.dropout()` function to the input layer and/or the output of any hidden layer you want.

- **Early Stopping:** To avoid overfitting, a great solution is to stop training when the performance on the validation set starts dropping which is known as early stopping. To implement this method using TensorFlow you can evaluate the model on a validation set at regular intervals and save a "winner" snapshot if it outperforms the previous "winner". Also, in

[1] Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification.
[2] https://en.wikipedia.org/wiki/Vanishing_gradient_problem
[3] Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

the cases that the number of steps since the last "winner" model reaches the limit you can simply stop training. Then, the last "winner" is the final model.

- **Saving and Restoring Models:** Once you have trained a model, you can save the learned parameters and weights so that you can use it in another program. Moreover, since the training of models could take a long time your computer might crash during the training, so you should save your model at regular intervals during training in order to continue from the last checkpoint if your computer crashes. In TensorFlow you can create a saver node by `tf.train.Saver()`, then in the execution phase just call `save()` whenever you want. For restoring a model, after creating `Saver` node you can call the `restore()` method of saver object.

1.2 Exercise

In this exercise, you need to design a deep neural network with five hidden layers. Also, you examine different methods such as initialization, batch normalization, dropout, and early stopping. The dataset that we will use in this assignment is MNIST, which contains 28-by-28 images of 10 digits (0 to 9) in different fonts which can be divided into different sets for training, validation and testing. You can load this file as follows.

```
(X_train, y_train), (X_test, y_test) = tf.keras.datasets.mnist.load_data()
X_train = X_train.astype(np.float32).reshape(-1, 28*28) / 255.0
X_test = X_test.astype(np.float32).reshape(-1, 28*28) / 255.0
y_train = y_train.astype(np.int32)
y_test = y_test.astype(np.int32)
X_valid, X_train = X_train[:5000], X_train[5000:]
y_valid, y_train = y_train[:5000], y_train[5000:]
```

1.2.1 Neural Network Architecture [20 pt.]

Build a DNN with five hidden layers of 100 neurons each, He initialization, and the ELU activation function. For the output layer, implement softmax with five neurons. You can use `tf.layers.dense()` to create a fully connected layer, where all inputs are connected to all the neurons in the next layer. Also, `tf.nn.sparse_softmax_cross_entropy_with_logits()` and `tf.nn.softmax_cross_entropy_with_logits()` are useful for computing cross entropy.

1.2.2 Training [15 pt.]

Using Adam optimization and early stopping, train the designed neural network in Part 1.2.1 on reduced version of MNIST which consists of digits 0 to 4. For the cost function consider the cross entropy between the output of the softmax layer and the labels. You can use the following script to remove digits 5 to 9 from the dataset.

```
X_train1 = X_train[y_train < 5]
y_train1 = y_train[y_train < 5]
X_valid1 = X_valid[y_valid < 5]
y_valid1 = y_valid[y_valid < 5]
X_test1 = X_test[y_test < 5]
y_test1 = y_test[y_test < 5]
```

Make sure to save check points at regular intervals and save the final model so you can reuse it later. For this part use the the following hyperparameters:

- Batch size = 20
- Number of epochs = 1000
- Learning rate = 0.005
- Maximum number of epochs without progress = 20

On the training set and validation set record your classification errors and cross-entropy losses after each epoch. Plot the training and validation classification error and loss function on them vs. the number of epochs. What precision can you achieve over the test set? How many iteration does it take to converge? Finally, visualize one sample from the test set for which the estimated label is not the same as its true label. **Note: Since the stochastic gradient method is used for training, each epoch consists of one pass of all the training examples.**

1.2.3 Tuning Hyperparameters [5 pt.]

Train the neural network in part 1.2.1 with the following set of hyperparameters. Plot the training and validation classification error and loss vs. the number of epochs. What precision can you achieve over the test set? How many iteration does it take to converge?

1. (a) Batch size = 20
   (b) Number of epochs = 1000
   (c) number of neurons per layer = 50
   (d) Learning rate = 0.005
   (e) Activation function `ReLU`
   (f) Maximum number of epochs without progress = 20
2. (a) Batch size = 500
   (b) Number of epochs = 1000
   (c) number of neurons per layer = 140
   (d) Learning rate = 0.1
   (e) Activation function `Leaky ReLU` with `alpha`=0.1
   (f) Maximum number of epochs without progress = 30

1.2.4 Batch Normalization [10 pt.]

Now apply Batch Normalization to the neural network designed in Part 1.2.2. Momentum is one of the inputs for `tf.layers.batch.normalization()` function in TensorFlow. Explain what is momentum, and how does the momentum is used to updates mean and variance? Train different neural networks with momentum of $\{0.85, 0.9, 0.95, 0.99\}$, and compare their performance in terms of cross entropy and precision on the training set and validation set? Also, report the accuracy you can achieve on the test set.

1.2.5 Dropout [5 pt.]

Try adding dropout to every layer for the model in Parts 1.2.2. Also, add dropout to the model in 1.2.4 with the best test accuracy, the momentum whose test accuracy is the best. Train different neural networks with dropout rates $\{0.1, 0.3\}$. Compare their performance in terms of cross entropy and precision on the training set and validation set? Also, report the accuracy you can achieve on the test set. (**Note that you need to scale the neuron's output by the keep probability while testing.**)

1.2.6 Impact of Regularization Methods [5 pt.]

In Parts 1.2.5 and 1.2.4, you examined three different approaches for regularization, namely, 1) batch normalization, 2) dropout, 3)batch normalization+dropout. Please comment on your observation on the impact of these approaches on the performance. Also, compare the results with part 1.2.2 where you did not apply any regularization.

2 Transfer Learning [40 pt.]

In Section 2.1 Transfer Learning and its implementation using TensorFlow are discussed. Then, you are asked to use and modify the model you designed in Section 1 to perform the detection of digits 5 to 9 on MNIST data set.

2.1 Background

It is generally not a good idea to train a very large DNN from scratch. The better option is to try to find an existing DNN that perform a similar task to the one you need to solve. Then, just reuse the lower layers of the pretrained network. This approach is called *transfer learning*. However, the upper hidden layers of the original model are less likely to be as useful as the lower layers, since the high-level features of the new task may significantly differ from the original task. For this reason, the designer of the new DNN should find the right number of layers to reuse. In this exercise, you examine the implementation of transfer learning using TensorFlow. To import the operational graph of the pretrained model, you may find the `tf.train.import_meta_graph` useful in this exercise. Also, the following script gives you access to the operation names of the graph.

```
for op in tf.get_default_graph().get_operations():
    print(op.name)
```

2.2 Exercise

In Section 1, you implemented a DNN on MNIST data set to predict digits 0 to 4. In this exercise, you will use the transfer learning for detection of digits 5 to 9. You can download the training set, validation set, and test set by using the following script.

```
(X_train, y_train), (X_test, y_test) = tf.keras.datasets.mnist.load_data()
X_train = X_train.astype(np.float32).reshape(-1, 28*28) / 255.0
X_test = X_test.astype(np.float32).reshape(-1, 28*28) / 255.0
y_train = y_train.astype(np.int32)
y_test = y_test.astype(np.int32)
X_valid, X_train = X_train[:5000], X_train[5000:]
y_valid, y_train = y_train[:5000], y_train[5000:]
X_train2_full = X_train[y_train >= 5]
y_train2_full = y_train[y_train >= 5] - 5
X_valid2_full = X_valid[y_valid >= 5]
y_valid2_full = y_valid[y_valid >= 5] - 5
X_test2 = X_test[y_test >= 5]
y_test2 = y_test[y_test >= 5] - 5
```

2.2.1 Reusing the model [10 pt.]

Create a new DNN that reuses all the pretrained hidden layers of the architecture in Part 1.2.1, freezes them, and replaces the softmax output layer with a new one. (Hint: To freeze the lower layers, you can exclude their variables from the optimizer's list of trainable variables, keeping only the output layer's trainable variables. For instance, you can give the optimizer the list of variables to train as follows. `training_op = optimizer.minimize(loss, var_list=output_layer_vars)`)

2.2.2 Training [10 pt.]

Train this new DNN on digits 5 to 9. Plot the training and validation classification error and loss vs. the number of epochs. What precision can you achieve over the test set? How many iteration does it take to converge? Train again the DNN using 100 images per digit and compare the performance with the case that the DNN is trained over full training set. You can use the following script to reduce the number of samples per digits.

```
def sample_n_instances_per_class(X, y, n):
    Xs, ys = [], []
    for label in np.unique(y):
        idx = (y == label)
        Xc = X[idx][:n]
        yc = y[idx][:n]
        Xs.append(Xc)
        ys.append(yc)
    return np.concatenate(Xs), np.concatenate(ys)
```

2.2.3 Removing the last hidden layer[10 pt.]

Remove the fifth hidden layer, and train this new DNN. Plot the training and validation classification error and loss vs. the number of epochs. What precision can you achieve over the test set? Compare the precision over the test set with that of Part 2.2.2. Comment on your observations.

2.2.4 Unfreezing hidden layers 3 and 4 [10 pt.]

The DNN designed in Part 2.2.3 consists of 4 hidden layers and a softmax output layer. In Part 2.2.3, you used the pretrained weights for layers 1 to 4. In this part, unfreeze weights of layers 3, 4, and the output and train them with the training dataset. What precision can you achieve over the test set? Compare the precision over test set with that of Part 2.2.2 and 2.2.3. Comment on your observations. (Hint: One solution is to give the optimizer the list of variables to train, excluding the variables from the layers 1 and 2)