ELEC 576 / COMP 576 – Fall 2020
Assignment 1

Due: Wednesday Oct. 7, 11AM

Submission Instructions

Every student must submit their work in a zip file in the following format: netid-assignment1.zip. You should also provide intermediate and final results as well as any necessary code. Submit your zip file on Canvas.

GPU Resource from AWS

To accelerate the training using GPU, you can optionally use Amazon Web Services(AWS) GPU instance using AWS Education credits. You can also get additional AWS credits from Github Student Developer Pack.
After having an AWS account, You can either create a fresh ubuntu instance and install software dependencies by yourself or use off-the-shelf TensorFlow ready image from AWS Marketplace.

1 Backpropagation in a Simple Neural Network

In this problem, you will learn how to implement the backpropagation algorithm for a simple neural network. To make your job easier, we provide you with starter code in `three_layer_neural_network.py`. You will fill in this starter code to build a 3-layer neural network (see Fig. 1) and train it using backpropagation.

a) Dataset

We will use the `Make-Moons` dataset available in Scikit-learn. Data points in this dataset form two interleaving half circles corresponding to two classes (e.g. "female" and "male"). In the `main()` function of `three_layer_neural_network.py`, uncomment the "generate and

visualize Make-Moons dataset" section (see below) and run the code. Include the generated figure in your report.

```
# generate and visualize Make-Moons dataset
X, y = generate_data()
plt.scatter(X[:, 0], X[:, 1], s=40, c=y, cmap=plt.cm.Spectral)
```

b) Activation Function

Tanh, Sigmoid and ReLU are popular activation functions used in neural networks. You will implement them and their derivatives.

1. Implement function `actFun(self, z, type)` in three_layer_neural_network.py. This function computes the activation function where `z` is the net input and `type` $\in$ {'Tanh', 'Sigmoid', 'ReLU'}.
2. Derive the derivatives of Tanh, Sigmoid and ReLU
3. Implement function `diff_actFun(self, z, type)` in three_layer_neural_network.py. This function computes the derivatives of Tanh, Sigmoid and ReLU.

c) Build the Neural Network

Lets now build a 3-layer neural network of one input layer, one hidden layer, and one output layer. The number of nodes in the input layer is determined by the dimensionality of our data, 2. The number of nodes in the output layer is determined by the number of classes we have, also 2. The input to the network will be x- and y- coordinates and its output will be two probabilities, one for class 0 ("female") and one for class 1 ("male"). The network looks like the following.

Mathematically, the network is defined as follows.

$$z_1 = W_1 x + b_1 \tag{1}$$
$$a_1 = \text{actFun}(z_1) \tag{2}$$
$$z_2 = W_2 a_1 + b_2 \tag{3}$$
$$a_2 = \hat{y} = \text{softmax}(z_2) \tag{4}$$

where z_i is the input of layer i and a_i is the output of layer i after applying the activation function. $\theta \equiv \{W_1, b_1, W_2, b_2\}$ are the parameters of this network, which we need to learn from the training data.

If we have N training examples and C classes then the loss for the prediction $\hat{y}$ with respect to the true labels y is given by:

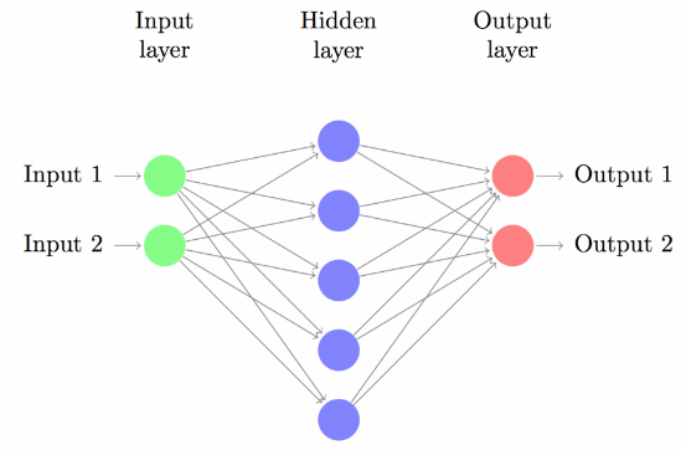

Figure 1: A three-layer neural network

$$L(y, \hat{y}) = -\frac{1}{N} \sum_{n \in N} \sum_{i \in C} y_{n,i} \log \hat{y}_{n,i} \tag{5}$$

Note that y are one-hot-encoding vectors and $\hat{y}$ are vectors of probabilities.

1. In three_layer_neural_network.py, implement the function `feedforward(self, X, actFun)`. This function builds a 3-layer neural network and computes the two probabilities (`self.probs` in the code or a_2 in Eq. 4), one for class 0 and one for class 1. `X` is the input data, and `actFun` is the activation function. You will pass the function `actFun` you implemented in part b into `feedforward(self, X, actFun)`.

2. In three_layer_neural_network.py, fill in the function `calculate_loss(self, X, y)`. This function computes the loss for prediction of the network. Here `X` is the input data, and `y` is the given labels.

d) Backward Pass - Backpropagation

It's time to implement backpropagation, finally!

1. Derive the following gradients: $\frac{\partial L}{\partial W_2}$, $\frac{\partial L}{\partial b_2}$, $\frac{\partial L}{\partial W_1}$, $\frac{\partial L}{\partial b_1}$ mathematically

2. In three_layer_neural_network.py, implement the function `backprop(self, X, y)`. Again, X is the input data, and y is the given labels. This function implements backpropagation (i.e., computing the gradients above).

e) Time to Have Fun - Training!

You already have all components needed to run the training. In three_layer_neural_network.py, we also provide you function `visualize_decision_boundary(self, X, y)` to visualize the decision boundary. Let's have fun with your network now.

1. Train the network using different activation functions (Tanh, Sigmoid and ReLU). Describe and explain the differences that you observe. Include the figures generated in your report. In order to train the network, uncomment the `main()` function in three_layer_neural_network.py, take out the following lines, and run three_layer_neural_network.py.

```
plt.scatter(X[:, 0], X[:, 1], s=40, c=y, cmap=plt.cm.Spectral)
plt.show()
```

2. Increase the number of hidden units (`nn_hidden_dim`) and retrain the network using Tanh as the activation function. Describe and explain the differences that you observe. Include the figures generated in your report.

f) Even More Fun - Training a Deeper Network!!!:

Let's have some more fun and be more creative now. Write your own `n_layer_neural_network.py` that builds and trains a neural network of n layers. Your code must be able to accept as parameters (1) the number of layers and (2) layer size. We provide you hints below to help you organize and implement the code, but if you have better ideas, please feel free to implement them and ignore our hints. In your report, please tell us why you made the choice(s) you did.

Hints:

1. Create a new class, e.g `DeepNeuralNetwork`, that inherits `NeuralNetwork` in three_layer_neural_network.py
2. In `DeepNeuralNetwork`, change function `feedforward`, `backprop`, `calculate_loss` and `fit_model`
3. Create a new class, e.g. `Layer()`, that implements the feedforward and backprop steps for a single layer in the network
4. Use `Layer.feedforward` to implement `DeepNeuralNetwork.feedforward`
5. Use `Layer.backprop` to implement `DeepNeuralNetwork.backprop`

6. Notice that we have L2 weight regularizations in the final loss function in addition to the cross entropy. Make sure you add those regularization terms in `DeepNeuralNetwork.calculate_loss` and their derivatives in `DeepNeuralNetwork.fit_model`.

Train your network on the `Make_Moons` dataset using different number of layers, different layer sizes, different activation functions and, in general, different network configurations. In your report, include generated images and describe what you observe and what you find interesting (e.g. decision boundary of deep vs shallow neural networks).

Next, train your network on another dataset different from `Make_Moons`. You can choose datasets provided by Scikit-learn (more details here) or any dataset of your interest. Make sure that you have the correct number of input and output nodes. Again, play with different network configurations. In your report, describe the dataset you choose and tell us what you find interesting.

Be curious and creative!!! You are exploring Deep Learning. :)

2 Training a Simple Deep Convolutional Network on MNIST

Deep Convolutional Networks (DCN) have been state-of-the-art in many perceptual tasks including object recognition, image segmentation, and speech recognition. In this problem, you will build and train a simple 4-layer DCN on MNIST Dataset. We provide you with starter code in dcn_mnist.py. You will fill in this starter code to complete task (a), (b), and (c) below. Also, since one of the purposes of this assignment is to get you familiar with Tensorflow, dcn_mnist.py is purposely written in a very simple form, a.k.a no object-oriented structure, no fancy tool in both Tensorflow and Python. That says, you are encouraged (but not required) to re-organize dcn_mnist.py as you like, but be sure to explain your code in the report. For acknowledgement, dcn_mnist.py inherits from this tutorial Deep MNIST for Expert.

MNIST is a dataset of handwritten digits (from 0 to 9). This dataset is one of the most popular benchmarks in machine learning and deep learning. If you develop an algorithm to learn from static images for tasks such as object recognition, most likely, you will want to debug your algorithm on MNIST first before testing it on more complicated datasets such as CIFAR10 and SVHN. There are also modified versions of MNIST, such as permutation invariant MNIST, which will come in handy for benchmarking at times.

More details, the MNIST data is split into three parts: 55,000 data points of training data (mnist.train), 10,000 points of test data (mnist.test), and 5,000 points of validation data (mnist.validation). The digits have been size-normalized and centered in a fixed-size image. MNIST images are of size 28 x 28. When loaded in Tensorflow, each image is flattened into a vector of 28x28=784 numbers. Each MNIST image will have a corresponding label which is a number between 0 and 9 corresponding to the digit that is drawn in that image. In Tensorflow, the following code will load the MNIST dataset.

```
#input_data file is provided to load the mnist
import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```

The tutorial on how to install Tensorflow can be found at TensorFlow Download and Setup.

a) Build and Train a 4-layer DCN

The architecture of the DCN that you will implement is as follows.

`conv1(5-5-1-32) - ReLU - maxpool(2-2) - conv2(5-5-32-64) - ReLU - maxpool(2-2) - fc(1024) - ReLU - DropOut(0.5) - Softmax(10)`

More details on the architecture can be found in this tutorial Deep MNIST for Expert.

Follow the tutorial Deep MNIST for Expert to fill in dcn_mnist.py. Particularly,

1. Read the tutorial Deep MNIST for Expert to learn how to use Tensorflow.
2. Complete functions `weight_variable(shape)`, `bias_variable(shape)`, `conv2d(x, W)`, `max_pool_2x2(x)` in dcn_mnist.py. The first two functions initialize the weights and biases in the network, and the last two functions will implement convolution and max-pooling operators, respectively.
3. **Build your network:** In dcn_mnist.py, you will see "FILL IN THE CODE BELOW TO BUILD YOUR NETWORK". Complete the following sections in dcn_mnist.py: `placeholders for input data and input labeles`, `first convolutional layer`, `convolutional layer`, `densely connected layer`, `dropout`, `softmax`.

4. **Set up Training:** In dcn_mnist.py, you will see "FILL IN THE FOLLOWING CODE TO SET UP THE TRAINING". Complete section `setup training` in dcn_mnist.py.

5. **Run Training:** Study the rest of dcn_mnist.py. Notice that, different from the tutorial Deep MNIST for Expert, I use the summary operation (e.g. `summary_op`, `summary_writer`, ...) to monitor the training. Here, I only monitor the training loss value. Now, run dcn_mnist.py. What is the final test accuracy of your network? Note that I set the batch size to 50, and to save time, I set the max_step to only 5500. Batch size is the number of MNIST images that are sent to the DCN at each iteration, and max_step is the maximum number of training iterations. max_step = 5500 means the training will stop after 5500 iterations no matter what. When batch size is 50, 5500 iterations is equivalent to 5 epochs. Remind that, in each epoch, the DCN will see the whole training set once. In this case, since there are 55K training images, each epoch is consisted of 55K/50 = 1100 iterations.

6. **Visualize Training:** In your terminal, type tensorboard --logdir=path/to/results where path/to/results is result_dir in dcn_mnist.py. Follow the instruction in your terminal to visualize the training loss in the training. You will be asked to navigate to a website to see the results, e.g. http://172.28.29.81:6006. Include the figures generated by TensorBoard in your report.

b) More on Visualizing Your Training

In part (a) of this problem, you only monitor the training loss during the training. Now, let's visualize your training more! Study dcn_mnist.py and this tutorial TensorBoard: Visualizing Learning to learn how to monitor a set of variables during the training. Then, modify dcn_mnist.py so that you can monitor the statistics (min, max, mean, standard deviation, histogram) of the following terms after each 100 iterations: weights, biases, net inputs at each layer , activations after ReLU at each layer, activations after Max-Pooling at each layer. Also monitor the test and validation error after each 1100 iterations (equivalently, after each epoch). Run the training again and visualize the monitored terms in TensorBoard. Include the resultant figures in your report.

c) Time for More Fun!!!

As you have noticed, I use ReLU non-linearity, random initialization, and Adam training algorithm in dcn_mnist.py. In this section, run the network training with different non-linearities (tanh, sigmoid, leaky-ReLU, MaxOut,...), initialization techniques (Xavier...) and training algorithms (SGD, Momentum-based Methods, Adagrad..). Make sure you

still monitor the terms specified in part (b). Include the figures generated by TensorBoard and describe what you observe. Again, be curious and creative! You are encouraged to work in groups, but you need to submit separate reports.

Collaboration Policy

Collaboration both inside and outside class is encouraged. You may talk to other students for general ideas and concepts, but individual write-ups must be done independently.

Plagiarism

Plagiarism of any form will not be tolerated. You are expected to credit all sources explicitly.