Homework 3: Neural Network

Part1: Deep Learning: a minimal case study (5 pts)

In Part 1 of this homework, you will implement forward and backward propagation in neural networks. Most of the necessary logic is already provided as starter code---your task is to write key lines that complete the machine learning algorithms.

Start by getting the starter code from part1_code.zip, or download it from git using command:

git clone https://github.com/yu71941628/minimal-nn.git

"fnn.py" contains a minimal implementation of multi-layer feedforward neural network. The main class is FNN, which holds a list of *layers*, and defines the high level iterative process for forward and backward propagation. The class Layer implements each layer in the neural network. The class GradientDescentOptimizer implements an optimizer for training the neural network. The utility functions at the end implement different activation functions, loss functions and their gradients. Read through "fnn.py" to get an overview of the implementation. Like most efficient implementations of neural network, we are using minibatch gradient descent instead of stochastic gradient descent.

Complete the following steps to finish part 1:

First, read this note on intuition and implementation tips for Backpropagation. The **Backprop in practice: Staged computation** and **Gradients for vectorized operations** sections are especially helpful, with good examples and practical tips.

Then, complete the code in the following two parts,

a) Forward propagation: the prediction algorithm for neural nets. In this part, you need to complete the forward method in class Layer in “fnn.py” (search for “Question 1” to see the instructions). (2 lines of code, 1 pt)
b) Backpropagation: the training algorithm for neural nets. In this part, you need to complete the backward method in class Layer in “fnn.py” (search for “Question 2” to see the instructions). (4 lines of code, 3 pts)

To test your implementation, first download the MNIST dataset by running:

```
python get_mnist_data.py
```

then run

```
python test_fnn.py
```

There are two tests (test_forwardprop and test_backprop). When your implementation passes both of them, run

```
python mnist_experiment.py
```

to train a small deep neural network with 2 hidden layers (containing 128 and 32 RELU units each) for handwritten digit recognition on the MNIST dataset. The accuracy should be around 99% on the training set and around 97% on the validation and test set.

To demonstrate the effect of learning, 100 randomly selected test images will be shown with true labels (black on top left corner), predictions before training (red on bottom right corner), and predictions after training (blue on bottom left corner). See Figure 1 for an example. You can see that the predictions improve from random guesses before training to almost perfect after training.

Report your final test loss and accuracy, and include a screenshot of the example images like Figure 1. (1 pt)

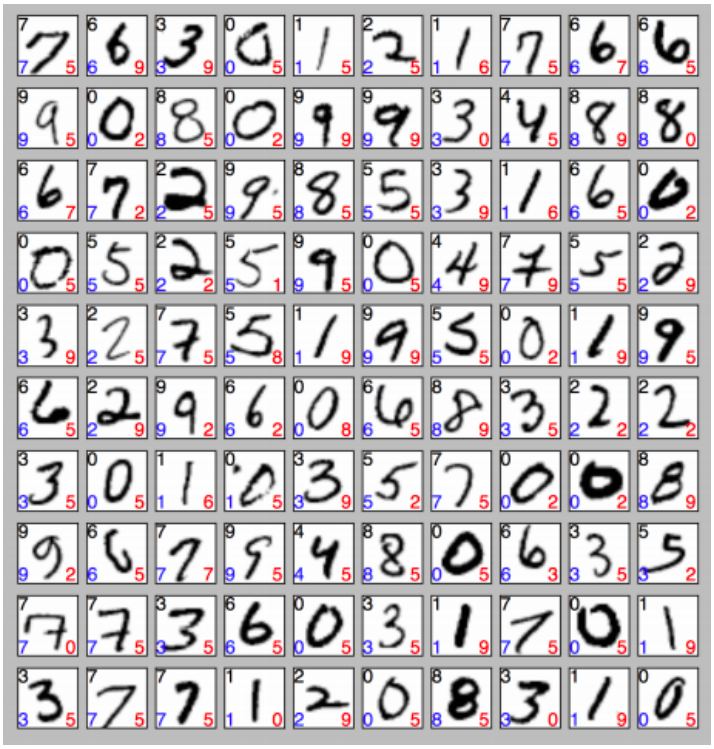

Figure 1: Example images with labels and predictions

Part2: Char-RNN in TensorFlow (5 pts)

In Part 2 of this homework, you will get experience working with an existing deep learning package, on the task of language modeling. *Char-RNNs* are Recurrent Neural Networks for Character-level language modeling. Read this fun blog to learn more about it. We will play with a TensorFlow implementation of this model for training and sampling.

Setup: in this part, instead of writing your own code and running experiments on your own laptop, you will use existing open source code from Github on a cloud computing platform (Amazon Web Service). Follow the instructions in AWS_Char_RNN_setup.pdf to setup your AWS machine and github repo.

a) **Model complexity and regularization** (2 pts) A key to successful applications of neural networks, especially deep neural networks, is *regularization* which allows very large neural networks to be trained effectively.

To see the utility of regularization, use screen command to run:

screen -S small ./scripts/eecs-349-experiment-small.sh and detach by "Crtl-a d".

Then run

screen -S large ./scripts/eecs-349-experiment-large.sh

and detach.

These two scripts will train two recurrent neural networks with 8 and 256 hidden units respectively on data/eecs349-data.txt, which is a small subset of Shakespeare scripts.

Navigate your browser to 'http://your-ec2-public-DNS:6006' and 'http://your-ec2-public-DNS:6007' to see what we'll call a *learning trace,* which plots how the training and validation loss/perplexity (lower is better) change as training proceeds. When done, remember to get back to the screen sessions using screen -r and terminate them to free port 6006 and 6007.

Include screenshots of the learning traces like those in Figure 2. Answer the questions: **what** is the difference between the curves of the two recurrent neural networks, and **why** does this difference make sense? (1 pt)

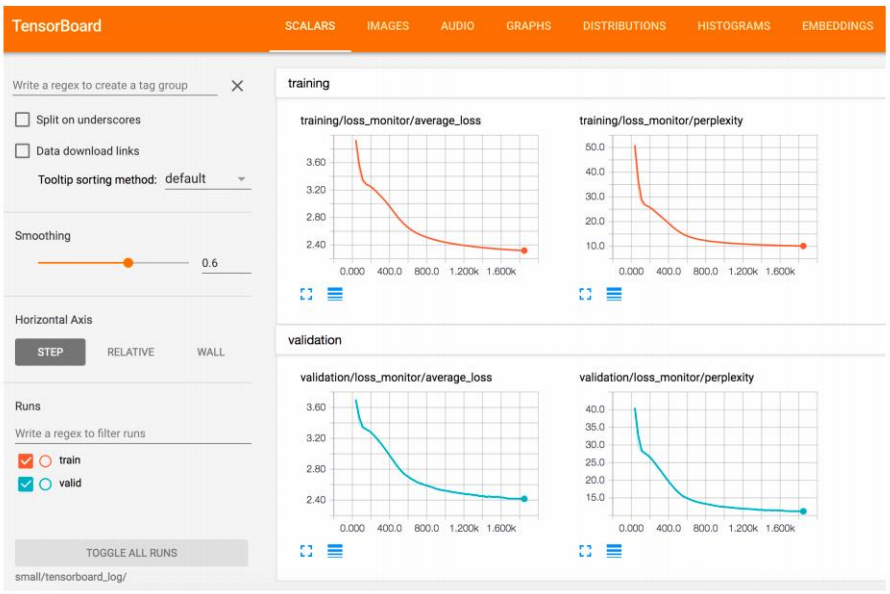

Figure 2: An example screenshot of the learning curve

Dropout is an effective way to regularize deep neural network.

Make a copy of eecs-349-experiment-large.sh and modify it to use dropout=0.1, 0.3, 0.5. Include screenshots of each run's learning trace. Report the final validation and test perplexities (saved in the best_valid_ppl and test_ppl fields in result.json in your output folder, you may find cat command handy). What is the difference between their learning traces, and why? (1 pt)

Note: this example is just for illustration purposes. The dataset is too small to train a good model. Typically, dropout has a much larger effect in real applications of very large neural networks.

b) **Sampling** (2 pts) A fun aspect of language modeling is that, after training a model, you can use it to generate samples. A model pretrained on shakespeare scripts is included in pretrained_shakespeare folder in the repo.
To get an example sample, run:

```
python3 sample.py --init_dir=pretrained_shakespeare
                  --length=1000
```

--temperature=0.5

--start_text="TRUMP:"

(You may need to combine these lines into one line in your terminal.)

The secret sauce for generating good samples is a hyperparameter called temperature. It changes the shape of the output probability distribution in the Char-RNN's Softmax output function. The original softmax distribution is

$$p(c_i) = \frac{e^{s_i}}{\sum_{i=1}^{n} e^{s_i}}$$

After adding temperature, it becomes

$$p(c_i) = \frac{e^{\frac{s_i}{t}}}{\sum_{i=1}^{n} e^{\frac{s_i}{t}}}$$

in which $p(c_i)$ is the probability to output the i-th character, and s_i is the model's score for the i-th character.

The temperature has a default value of 1.0. Usually values smaller than 1.0, such as 0.5, will yield more reasonable samples. To get a feeling of the effect of low and high temperature, try sampling with temperature=0.01 and 5.0. How are the samples different from the previous one (with temperature=0.5) and why? (think about how the temperature would change the shape of the distribution, and perhaps try some simple mathematical examples.)

c) **Have fun** (1 pt) Now the fun part. Collect your own dataset (a file of characters, usually TXT files, for example: your favorite novel, Taylor Swift's lyrics, etc) and train a Char-RNN on it and get some fun samples. Note that you need to know the encoding your file and use the "encoding" argument (default to "utf-8") of "train.py". Here's a list of encodings you can try if you are not sure.

To get a good result, the dataset should be large enough (at least more than 10KB, and a good file should be more than 1MB). Typically it helps to use a larger model, and to train

for longer time (around 3 hrs using the default settings on 1MB). Use tensorboard to monitor your training.

Start with the default settings like:

```
python3 train.py --data_file=path/to/your-own-data
                 --encoding="your-file-encoding"
                 --batch_size=100
                 --output_dir=your-output-dir
```

(You may need to combine these lines into one line in your terminal.)

Then, tune the hyperparameters like hidden size, number of layers, number of unrollings and dropout rate to get lower perplexity.

Tune the temperature parameter to generate some fun samples from your trained model. Use `start_text` to warm up your Char-RNN, and use `seed` to make your samples replicable.

```
python3 sample.py --init_dir=your-output-dir
                  --length=1000
                  --seed=your-integer-random-seed
                  --start_text="your-start-text"
                  --temperature=0.5
```

(You may need to combine these lines into one line in your terminal.)

Describe the dataset you used for training. Include screenshots (Figure 2) of your learning curves, the result.json file in your output folder, and some of your favorite samples. We will share some of the funniest samples in the class :)

Working in a group

- For part 1, you can discuss in groups, but the coding should be done individually.
- For part 2, you can work in groups of 2-3, but each needs to submit a write-up separately. It is recommended that at least one of the group members should be familiar with ssh and the Linux commandline.

Submission

A zip file containing the following:

1. The original folder “part1_code” with your modified “fnn.py”.
2. A PDF file including the answers to your questions, screenshots, the content of the “result.json” files and your favorite samples.
3. Include a list of your group members and who did what in the PDF