Homework 3: Neural Network

Chen Liang

May 18, 2016

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

Like any machine learning methods, neural network has two main parts: inference/prediction and learning/training. They are implemented as forward propagation and backward propagation (backpropagation for short).

The folder "part-1" contains all the code for this part. "fnn.py" contains a minimal implementation of multi-layer feedforward neural network. The main class is `FNN` that holds a list of layers, and defines the high level iterative process for forward and backward propagation. Class `Layer` implements each layer in the neural network. Class `GradientDescentOptimizer` implements an optimizer for training the neural network. The utility functions at the end implements different activation functions, loss functions and their gradients. Read through "fnn.py" to get an overview of the implementation. Like most effcient implementations of neural network, we are using minibatch gradient descent instead of stochastic gradient descent, see this video to learn more.

1. **Forward propagation**: the core of inference/prediction in NN.

   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)

2. **Backpropagation**: the core of learning/training in NN.

   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)

Read this notes on intuition and implementation tips for Backpropagation. *Backprop in practice: Staged computation* and *Gradients for vectorized operations* sections are especially helpful with good examples and practical tips.

To test your implementation, 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 digits recognition using MNIST dataset. The accuracy should be around 99% on training set and around 97% on 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 guess to almost perfect. Yes, it LEARNS!

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

2 Char-RNN in TensorFlow (5 pts)

Char-RNN is a Recurrent Neural Network 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 (some guides here) on a cloud computing platform (Amazon Web Service). Follow the instructions to setup your AWS machine and github repo.

1. **Model complexity and regularization** (2 pts) A key to successful applications of neural network, especially deep neural network, is regularization so that very large neural networks can be applied.

   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/#events` and `http://your-ec2-public-DNS:6007/#events` to see their learning curves of training and validation loss/perplexity (the lower the better). When done, remember to get back to the screen sessions using `screen -r` and terminate them to free port 6006 and 6007.

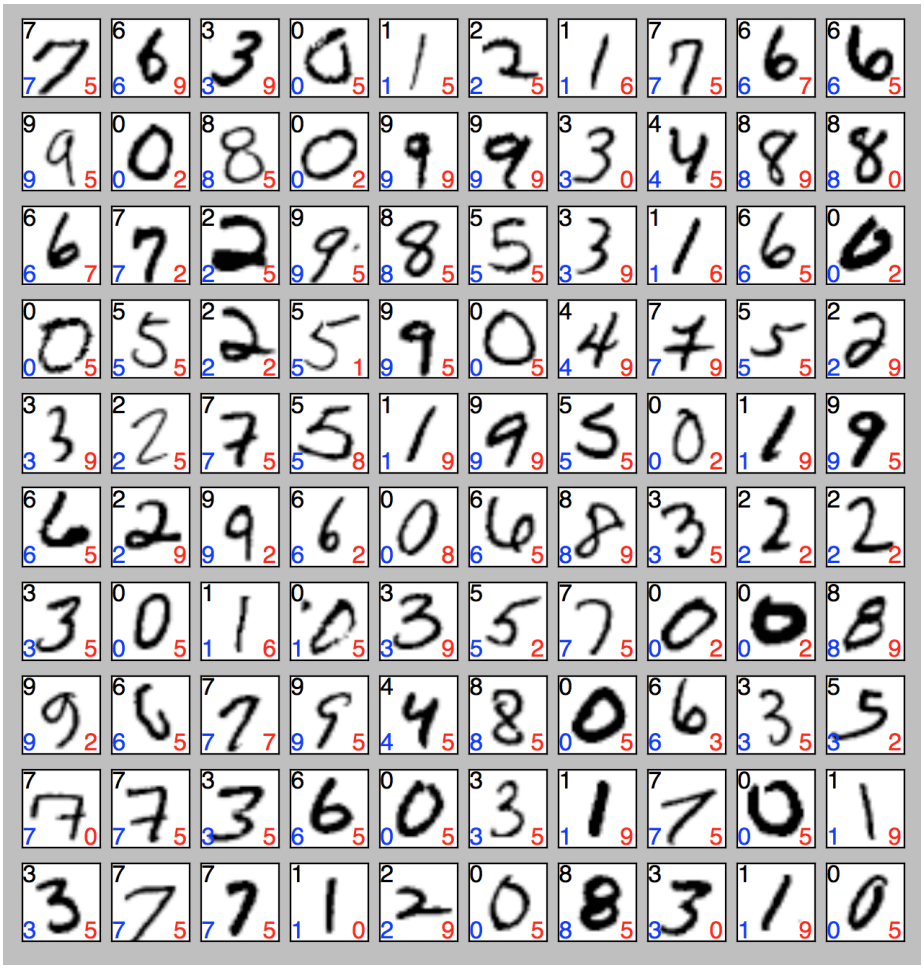

Figure 1: Example images with labels and predictions

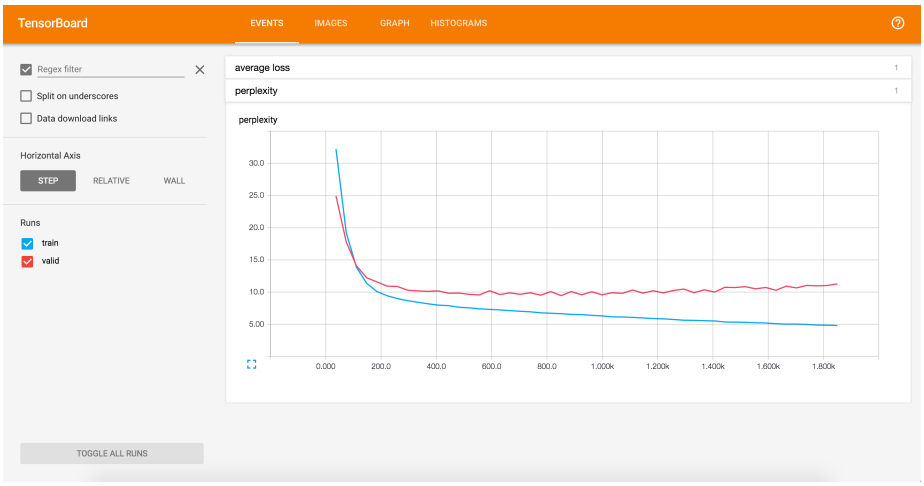

Figure 2: An example screenshot of the learning curve

Include screenshots of the learning curves (perplexity) like Figure 2. And answer the questions: what is the difference between the learning curves of the two recurrent neural network and why? (1 pt)

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 their learning curves. Report the final validation and test perplexities (saved in `result.json` in your output folder, you may find `cat` command handy). What is the difference between their learning curves and why? (1 pt)

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

2. **Sampling** (2 pts) The fun 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_models` folder of the homework handout. Use scp (Linux and Mac) or PuTTY/WinSCP (Windows) to copy this folder to your tensorflow-char-rnn folder.

   To get an example sample, run

```
python sample.py --init_dir=pretrained_models/shakespeare \
                 --length=1000 \
                 --temperature=0.5 \
                 --start_text="TRUMP:"
```

   The secret sauce to good samples is a hyperparameter called temperature. It would change the shape of the output probability distribution from

Softmax function. The original distribution is

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

After adding temperature, it becomes

$$p(c_i) = \frac{e^{\frac{s_i}{t}}}{\sum_{i=1}^{n} e^{\frac{s_i}{t}}} \tag{2}$$

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

The temperature is defaulted to be 1.0. Usually value smaller than 1.0, for example 0.5, gives you more reasonable samples. But 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 (temperature=0.5) and why? (think about how the temperature would change the shape of the distribution and try some simple examples.)

3. **Have fun** (1 pt) Now is 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 Char-RNN on it and get some fun samples. Note that you need to know the encoding (default to "utf-8") of your file and use the "encoding" argument 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 one should be more than 1MB). You usually need larger model, and train for longer time (around 3 hrs using default settings on 1MB). Use tensorboard to monitor your training.

   Start with the default settings like

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

   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.

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

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

3 Groups

1. For part 1, you can discuss in groups, but the coding should be done individually.
2. 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 Linux commandline.

4 What to submit

A zip file containing the following:

1. The original folder "part-1" 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