ECE 544NA HW2

Jiaqi Mu jiaqimu2
collaborating with Hongyu Gong hgong6
Department of Electrical and Computer Engineering

September 30, 2016

1 Pencil-and-Paper

In this part of the assignment, you will compute the derivatives/gradients and backpropped error for the following common modules in a neural network.

- **Derivative of Softmax:** Recall the softmax function:

$$\vec{y}[k] = \frac{e^{\vec{z}[k]}}{\sum_{l=1}^{C} e^{\vec{z}[l]}},$$

what is $\frac{\partial \vec{y}[k]}{\partial z[j]}$.

Proof. The derivative is given as follows,

$$\begin{aligned}
\frac{\partial \vec{y}[k]}{\partial \vec{z}[j]} &= \frac{\partial}{\partial \vec{z}[j]} \frac{e^{\vec{z}[k]}}{\sum_{l=1}^{C} e^{\vec{z}[l]}} \\
&= \frac{\sum_{l=1}^{C} e^{\vec{z}[l]} \frac{\partial}{\partial \vec{z}[j]} e^{\vec{z}[k]} - e^{\vec{z}[k]} \frac{\partial}{\partial \vec{z}[j]} \sum_{l=1}^{C} e^{\vec{z}[l]}}{\left(\sum_{l=1}^{C} e^{\vec{z}[l]}\right)^2} \\
&= \frac{e^{\vec{z}[j]} \mathbf{1}_{j=k} \sum_{l=1}^{C} e^{\vec{z}[l]} - e^{\vec{z}[k]} e^{\vec{z}[j]}}{\left(\sum_{l=1}^{C} e^{\vec{z}[l]}\right)^2} \\
&= \frac{e^{\vec{z}[j]} \mathbf{1}_{j=k}}{\sum_{l=1}^{C} e^{\vec{z}[l]}} - \frac{e^{\vec{z}[k]+\vec{z}[j]}}{\left(\sum_{l=1}^{C} e^{\vec{z}[l]}\right)^2}.
\end{aligned}$$

□

- **Negative Log Likelihood loss for Multi-class:** Recall the negative log likelihood,

$$L = -\sum_{i}^{N} \sum_{k}^{K} \mathbf{1}_{y_i=k} \log(\hat{\vec{y}}_i[k]).$$

What is $\frac{\partial L}{\partial \hat{\vec{y}}_i[j]}$.

Proof. The derivative is given as follows (here we take the $\log$ as natural logarithm),

$$\begin{aligned}\frac{\partial L}{\partial \hat{\vec{y}}_i[j]} &= -\sum_i^N \sum_k^K \mathbf{1}_{y_i=k} \frac{\partial}{\partial \hat{\vec{y}}_i[j]} \log(\hat{\vec{y}}_i[k]) \\ &= -\sum_i^N \sum_k^K \mathbf{1}_{y_i=k} \mathbf{1}_{k=j} \frac{1}{\hat{\vec{y}}_i[j]} \\ &= -\sum_i^N \mathbf{1}_{y_i=j} \frac{1}{\hat{\vec{y}}_i[j]}\end{aligned}$$

□

- **Avg-pooling (1D):** Revall Avg-pooling operation with window size W:

$$\vec{y}[i] = \frac{1}{W} \sum_{k=0}^{W-1} \vec{x}[i+k].$$

What is $\frac{\partial \vec{y}[i]}{\partial \vec{x}[j]}$.

Proof. The derivative is given as follows,

$$\begin{aligned}\frac{\partial \vec{y}[i]}{\partial \vec{x}[j]} &= \frac{1}{W} \sum_{k=0}^{W-1} \frac{\partial}{\partial \vec{x}[j]} \vec{x}[i+k] \\ &= \frac{1}{W} \mathbf{1}_{i \le j < i+W}\end{aligned}$$

□

- **Max-pooling (1D):** Recall max-pooling (1D) operation with window size W:

$$\vec{y}[i] = \max_{k=0}^{W-1} \vec{x}[i+k].$$

What is $\frac{\partial \vec{y}[i]}{\partial \vec{x}[j]}$.

Proof. The derivative is given as follows,

$$\frac{\partial \vec{y}[i]}{\partial \vec{x}[j]} = \mathbf{1}_{j=i+\arg\max_{k=0}^{W-1} \vec{x}[i+k]}$$

□

- **Convolutional layer (1D):** Recall Convolution (1D) operation, assume $\vec{w}$ is length 3, and zero index at the center:

$$\vec{y}[i] = (\vec{w} \star \vec{x})[i] = \sum_{k=-1}^{1} \vec{x}[i-k] \vec{w}[k].$$

What is $\frac{\partial \vec{y}[i]}{\partial \vec{x}[j]}$? What is $\frac{\partial \vec{y}[i]}{\partial \vec{w}[j]}$?

Proof. The derivatives are given as follow,

$$\begin{aligned}\frac{\partial \vec{y}[i]}{\partial \vec{x}[j]} &= \sum_{k=-1}^{1} \frac{\partial}{\partial \vec{x}[j]} \vec{x}[i-k]\vec{w}[k] \\ &= \sum_{k=-1}^{1} \vec{w}[k]\mathbf{1}_{i-k=j} \\ \frac{\partial \vec{y}[i]}{\partial \vec{w}[j]} &= \sum_{k=-1}^{1} \frac{\partial}{\partial \vec{w}[j]} \vec{x}[i-k]\vec{w}[k] \\ &= \vec{x}[i-j]\mathbf{1}_{-1\leq j\leq 1}.\end{aligned}$$

□

2 Code-From-Scratch

Perform 9-way classification among the 9 letters in the ee-set using a fully-connected neural network with 2 hidden layers. Use the first 70 frames of each example and drop any example that is shorter than 70 frames. The use mini-batch gradient descent for optimization, you can pick the batch size. The test set accuracy is in the range of 35% to 50%.

- Experiment with different number of hidden-nodes, $\{10, 50\}$, assume the two hidden layers have the same size.
- Experiment with different type of nonlinearities [sigmoid, tanh, relu]

2.1 Methods

- Describe the functions you wrote, and the overall structure of your code.

  Proof. The feedforward neural network is implemented as a class FNN in `fnn.py`. This contains five functions:

  - `__init__`: initialize all hyperparameters, all weight matrices and biases.
  - `gradient`: compute the gradient of activation nonlinear functions.
  - `activation`: compute the activation nonlinear functions.
  - `train`: train the model using training data. This function can be divided into three parts: (a) feedforward, (b) back-propagation and (c) gradient update.
  - `test`: test new instances.

  The overall structure of my code is,

  - First read data from training/dev/testing files.
  - Initialize FNN model.
  - Train FNN model.
  - Test FNN model.

  □

- Describe the model architecture and specific hyperparameter you have chosen.

  Proof. The model is compatible with hidden layers of size $N_{h_1}, N_{h_2}, ..., N_{h_K}$, where we choose $K = 2$, and $N_{h_1} = N_{h_2} \in \{10, 50\}$ as a special case. The input dimension is $N_i = 16 \times 70$, the output dimension is $N_o = 9$.

  In our case we choose step size to be `1e-2`, the number of iterations to be 100,000, batch size to be 50. □

- Report the total number of weights in the models.

  Proof. Thus the total number of weights in this model is,

$$\text{weightmatrix}: \quad N_i \times N_{h_1} + \sum_{k=1}^{K-1} N_{h_k} \times N_{h_{k+1}} + N_K \times N_o$$

$$\text{bias}: \quad \sum_{k=1}^{K} N_{h_k} + N_o$$

  In our case, the number of parameters are listed in Table 1. □

hidden layer	weight matrix	bias
10	11390	29
50	58950	109

Table 1: The total number of parameters.

2.2 Results

- Report training and testing accuracy for your best model.

  Proof. The best model in terms of development set is `tanh` + hidden layer = 50. The accuracies are 100.00% and 42.79% for training and testing respectively. This is because of overfitting. □

- Report training and testing accuracy for all the models. (All combinations between nonlinearities and number of hidden nodes.)

  Proof. The accuracies are given in Table 2. □

Accuracy	hidden layer size = 10			hidden layer size = 50		
	sigmoid	tanh	relu	sigmoid	tanh	relu
Training	80.01	78.74	71.66	100.0	100.0	100.0
Development	44.46	39.81	43.50	46.79	**48.42***	48.29
Testing	39.60	37.62	38.94	38.28	42.79	45.87

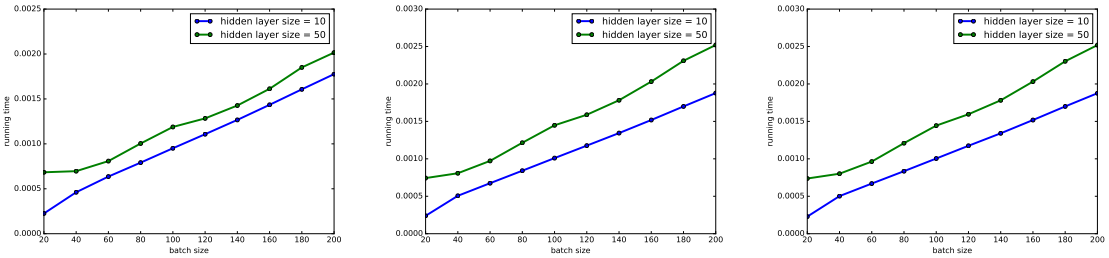
Table 2: Training and testing Accuracy (x100)

- Report the running time (in seconds) for one iteration of backpropagation on model with 10 and 50 hidden-nodes. Also describe how the runng time varies with batch size.

  Proof. The running time is given in Figure 2. We can observe the running time almost grows linearly with the batch size. □

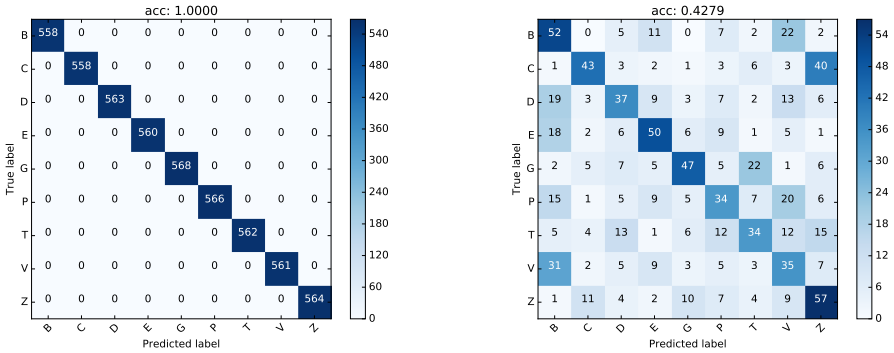
- Plot training and testing classification confusion matrix for your best model.

  Proof. The training and testing confusion matrix for the best model described in part 2 is given in Figure **??**. □



(a) `reLu` (b) `tanh` (c) `sigmoid`

Figure 1: Running time versus batch sizes.



(a) train (b) test

Figure 2: Confusion matrix.

3 TensorFlow

Perform 9-way classification among the 9 letters in the ee-set using the TDNN architecture of Waibel, Hanazawa, Hinton, Shikano and Lang[1]. Use the first 70 frames of each example and drop any example that is shorter than 70 frames. The use mini-batch gradient descent for optimization, you can pick the batch size. The test set accuracy is in the range of 35% to 50%.
Use the following network architecture: Input layer: (J=16, N=2), Hidden Layer 1: (J=8, N=4), Hidden Layer 2: (J=3, N=1), Final Layer: multi-class logistic regression.
Next, try changing the TDNN architecture or even CNN architecture and see if you can improve the accuracy (The accuracy for this part will not affect your grade) Here is a TensorFlow tutorial [2] with CNN on the MNIST datase.

3.1 Metods

- Describe the TDNN architecture, and discuss how you used TensorFlow functions to create such architecture.

  Proof. For simplicity, we ignore the index for sample. We denote the input layer as h^0, the first hidden layer as h^1, the second hidden layer as h^2, and the output multi-class logistic regression as $\hat{y}$. Here we know that $h^0 \in \mathbb{R}^{16\times 70}$, $h^1 \in \mathbb{R}^{8\times 68}$, $h^2 \in \mathbb{R}^{3\times 64}$, and $\hat{y} \in \mathbb{R}^9$. The dependencies are as

[1] `http://ieeexplore.ieee.org/document/21701/?arnumber=21701&tag=1`
[2] `https://www.tensorflow.org/versions/r0.10/tutorials/mnist/pros/index.html`

follow,

$$h_t^1 = f_1\left(\sum_{\tau=0}^{2} W_\tau^1 h_{t+\tau}^0 + b^1\right), t = 1, 2, ..., 68 \tag{1}$$

$$h_t^2 = f_2\left(\sum_{\tau=0}^{4} W_\tau^2 h_{t+\tau}^1 + b^2\right), t = 1, 2, ..., 64 \tag{2}$$

$$\tag{3}$$

where the subscript t indicates the time frame, $W_\tau^1 \in \mathbb{R}^{8\times 16}$, $b^1 \in \mathbb{R}^8$, $W_\tau^2 \in \mathbb{R}^{3\times 8}$, $b^2 \in \mathbb{R}^3$ and f_1 and f_2 are non-linear kernels. The output $\hat{y}$ is computed via a multi-class softmax, where $\vec{y}$ is computed via,

$$\vec{y} = \text{softmax}\left(W^3 \sum_{t=1}^{64} h_t^2 + b^3\right), \tag{4}$$

where $W^3 \in \mathbb{R}^{9\times 3}$ and $b^3 \in \mathbb{R}^9$.
We can achieve this architecture using the built-in convolutional network module from TensorFlow. Specific functions are as follow:

- Initialization of weight matrices through `weight_variable(shape)`, which uses `tf.truncated_normal` to randomly generated weight matrices.
- Initialization of bias vectors through `bias_variable(shape)`, which uses `tf.constant` to generate bias vectors.
- Convolution layer through `tf.nn.conv2d(z, W, stride=[1,1,1,1], padding='VALID')` where we set the stride size to be 1, and the padding size to be 'VALID' meaning we do not pad zeros on the boundary. Note here:
  - `z` denotes for h^0, h^1, and h^2, and has to be reshaped into $[-1, T_l, 1, J_l]$, where T_l is the number of total frames, and J_l is the number of units/channels in l-th layer.
  - `W` denotes the weight matrix associated with `z`, and should be shaped as $[N_l + 1, 1, J_l, J_{l+1}]$.
- Activation layer though ReLU/Sigmoid by calling `tf.nn.relu`/`tf.sigmoid`.
- Softmax layer though `tf.nn.softmax` as seen in HW1.

□

- Describe the variations of the TDNN architecture you have tried and anything interesting.

  Proof. Apart from this architecture, we use a different output layer. We first reshape h^2 into a vector, and directly apply a weighted softmax on that. Note this is a generalization to (4). □

- Report the total number of weights in the original TDNN architecture.

  Proof. The number of weights in the first layer is 384(=8x16x3), the number of weights in the second layer is 120(=5x3x8), and the number of weights in the output layer is 27(=9x3). The number of biases in the first layer is 8, that in the second layer is 3, and that in the output layer is 9. □

- Report the activation dimensions at each layer.

  Proof. The activation dimension at the first hidden layer is 8, that at the second hidden layer is 3. □

- Discuss which Tensorflow functions you used, and how. Additionally, explain the overall organization/structure of your code, you should refer to specific part of your code.

Proof. The functions are listed in part 1. Details of the code organization is as follows:

- `data.py` is a class storing training samples, development samples, and test samples. Two functions are involved in this class:
  - `readFile(filename)` is to read data from files.
  - `nextBatch(batchSize)` is to get the next batch for training, specially when `batchSize=-1` return all samples.
- `part3.py` is the main script for TDNN.
  - Line 36-38: set up train/dev/test data;
  - Line 48-51: preprocess samples (i.e., reshape them into proper dimensions);
  - Line 56-61: compute the first hidden layer;
  - Line 66-71: compute the second hidden layer;
  - Line 76-85: compute the output;
  - Line 90: specify the objective loss function;
  - Line 93: set up the gradient descent;
  - Line 94-95: set up the evaluation metric;
  - Line 99-106: execute the training model.

□

3.2 Results

The hyperparameters are as follow: step size: 1e-3 for `AdamOptimizer`; batch size: 50; iterations: 100,000 for softmax and 10,000 for reLu.

- Report training and testing accuracy for the TDNN model.

  Proof. The accuracy for the original TDNN structure with two different nonlinear filters is reported in Table 4. □

	train	test	dev
`sigmoid`	50.19	38.72	47.20
`reLu`	43.18	37.07	40.77

Table 3: Training and testing accuracy of original TDNN (x100).

- Report training and testing accuracy for your best model.

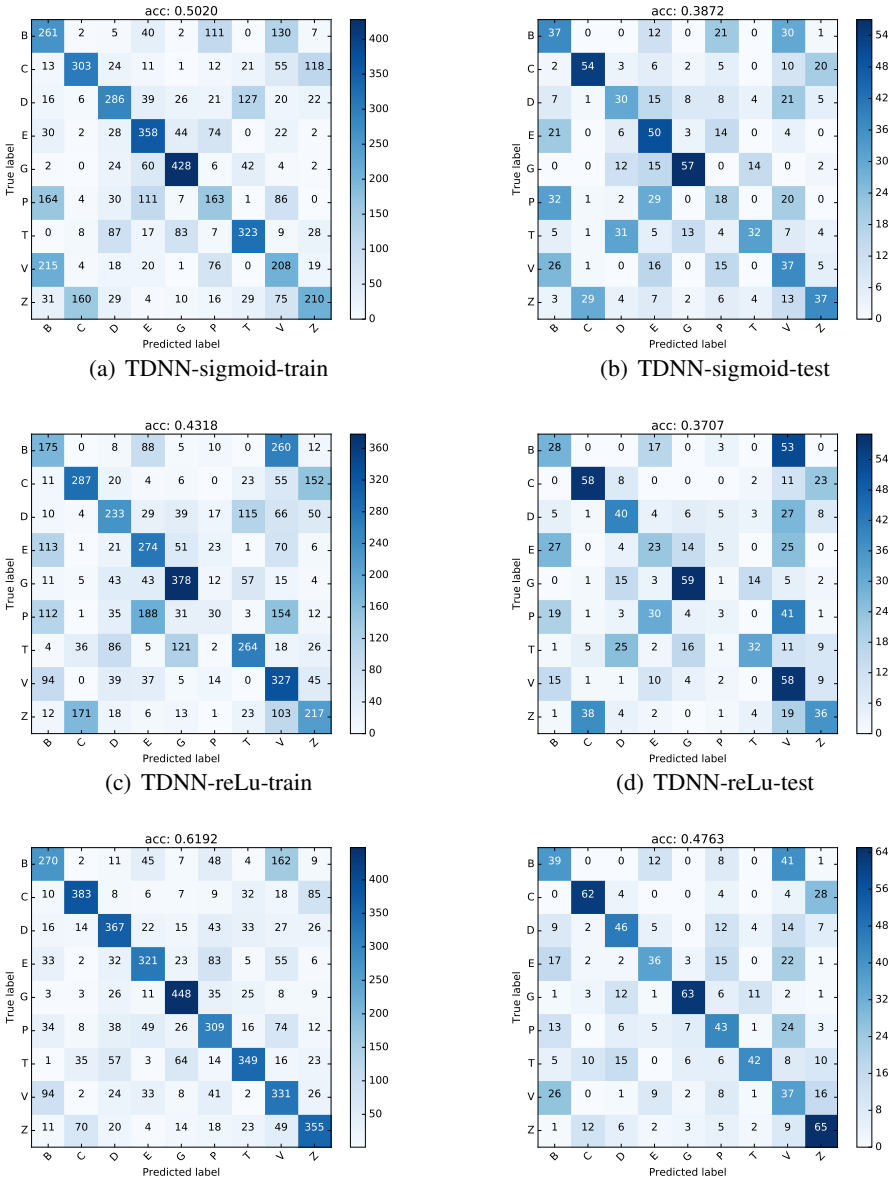
  Proof. We flatten the hidden layer to feed in a fully-connected feedforward neural network. The accuracies are in Table 4. The best one is a FNN+reLu. □

	train	test	dev
`sigmoid`	68.34	44.22	**52.12***
`reLu`	61.91	47.63	49.79

Table 4: Training and testing accuracy of modified TDNN (x100).

- Plot training and testing classification confusion matrix for the TDNN model, and your best model.

  Proof. The confusion matrices are reported in Figure 3, where (a) and (b) are for TDNN with sigmoid activation, (c) and (d) are for TDNN with relu activation, and (e) and (f) are for the best model. □



(a) TDNN-sigmoid-train (b) TDNN-sigmoid-test

(c) TDNN-reLu-train (d) TDNN-reLu-test

(e) best model-train (f) best model-test

Figure 3: Confusion matrix.