IE 534: Deep Learning, Fall 2018

Homework 2: Convolution Neural Network on MNIST

Lecturer: Justin Sirignano *Submitted by: Rachneet Kaur, Sept. 14th, 2018*

Implementation

The objective is to **classify** the input 28*28 dimensional image of a digit into the corresponding one of the 10 distinct digit classes namely $\{0, 1, ..., 9\}$. The MNIST dataset has 60000 training images and 10000 images in the test set of dimensions 28*28.
Hence, we have:

- In my code, I store training set in x_train and y_train. Each image in x_train is a matrix of dimensions $d \times d = 28 \times 28$.
- Dimensions of training set input X = (60000, 28, 28) and labels Y = (60000, 1)
- In my code, I store test set in x_test and y_test. Each image in x_test is a matrix of dimensions $d \times d = 28 \times 28$.
- Dimensions of test set input X = (10000, 28, 28) and labels Y = (10000, 1)

The system in equation (1) represents a convolution neural network with multiple no. of channels C, filter K of dimension $d_y \times d_x \times C$ and a single hidden layer H of dimension $(d - d_y + 1) \times (d - d_x + 1) \times C$.

$$\begin{aligned} Z_{:,:,p} &= X_{:,:} * K_{:,:,p} \\ Z_{i,j,p} &= \sum_{m=0}^{d_y-1} \sum_{n=0}^{d_x-1} K_{m,n,p} X_{i+m,j+n} \\ H_{:,:,p} &= \sigma(Z_{:,:,p}) \\ U_k &= W_{k,:,:,:} H + b_k \\ f(x, \theta) &= F_{softmax}(U) \end{aligned} \tag{1}$$

Note that in order to perform 10 class classification, a softmax layer is added as the last layer of the neural network. Equation (1) is called as the **forward propagation or forward pass** in the training of a convolution neural network.
The convolution $Z_{:,:,p} = X_{:,:} * K_{:,:,p}$ is $(d - d_y + 1) \times (d - d_x + 1) \times C$ dimensional, where X

is $d \times d$ dimensional and K is $d_y \times d_x \times C$ dimensional.
To implement the convolution for forward step in Python, I used np.tensordot to compute the sum of the element wise multiplication of $K_{:,:,p}$ and $X_{i:i+d_y,j:j+d_x}$ where i iterates through $0 : d - d_y + 1$, j through $0 : d - d_x + 1$ and p through the no. of channels $0 : C$. I used itertools.product function in Python to compute these iterates, to avoid for loops in my code.

Consider the steps implemented for the Stochastic Gradient Descent algorithm:

Step 1: Random Initialization of model parameters

The model parameters for a single layer convolution neural network with multiple channels are $\{K, W, b\}$.

- For implementation, I used the C = 5 channels.
- I used filter K with dimensions $d_y \times d_x \times C = 5 \times 5 \times 5$.
- Since this is a $k = 10$ class classification problem, I have 10 units in the output layer.
- Since filter K has dimensions $d_y \times d_x \times C = 5 \times 5 \times 5$, I have $(d - d_y + 1) \times (d - d_x + 1) \times C = 24 \times 24 \times 5$ dimensional hidden layer.

First, I randomly initialize the following:

- K = $d_y \times d_x \times C$ tensor = $5 \times 5 \times 5$ tensor
- W = $k \times (d - d_y + 1) \times (d - d_x + 1) \times C$ tensor = $10 \times 24 \times 24 \times 5$ tensor
- $b = k \times 1$ matrix = 10×1 matrix

The forward, backward propagation and parameter update steps in SGD algorithm (Steps 1, 2 and 3 below) are repeated 60000 (number of rows in training set) times in each epoch. Using these 60000 iterations, the training accuracy is calculated in each epoch as mentioned in Step 2.

- For implementation, I trained the model for 10 epochs.

Step 2: Forward Step

Next, we randomly select a data sample (X, Y) from the training set and compute Z, H, U and $f(x, \theta)$ using the equations in 1. The dimensions of Z is $(d-d_y+1) \times (d-d_x+1) \times C = 24 \times 24 \times 5$,

H is $(d - d_y + 1) \times (d - d_x + 1) \times C = 24 \times 24 \times 5$ and U is $k \times 1 = 10 \times 1$.

To compute Z in my code, I use my own written $convolution(x, model['K'], iteratable\_forward, d\_y, d\_x)$ function. To compute H, I use the $activation(Z, derivative = 0)$ function.
For U, I use numpy.tensordot between $model['W']$ and H and for probability distribution, I use $softmax\_function(U)$ that I wrote.

- For implementation in Python, I used the activation function as $\sigma(z) = tanh(z)$. The activation function is element wise applied to each element of the hidden layer.

Next, we calculate the predicted label as argmax$(f(x, \theta))$ for the training sample which is compared with true label y to compute the accuracy of the training set.

- I wrote a function $forward(x, y, model)$ in my code to compute the forward step of the model. It returns the calculated Z, H and probability distribution of the 10 classes.
- $softmax\_function(z)$ function in my code computes the softmax function for the output layer.
- $activation(z, derivative = 0)$ function in my code computes the $tanh(z)$ and it's derivative $1 - tanh^2(z)$ if derivative flag = 1) activation for the hidden layers.
- $convolution(X, K, iteratable, d\_y, d\_x)$ function in my code computes the convolution function between X and K. To implement the convolution for forward step in Python, I used np.tensordot to compute the sum of the element wise multiplication of $K_{:,:,p}$ and $X_{i:i+d_y, j:j+d_x}$ where i iterates through $0 : d - d_y + 1$, j through $0 : d - d_x + 1$ and p through the no. of channels $0 : C$. I used itertools.product function in Python to compute these iterates, to avoid for loops in my code.
- model = {} is the dictionary in my code containing the model parameters, namely, model['K'], model['W']and model['b'].

Step 3: Gradient Calculation

Equation (2) defines the negative log likelihood objective function to be minimized.

$$\begin{aligned} L(\theta) &= E_{(X,Y)}(\rho(f(x, \theta), y)) \\ \varrho(v, y) &= -\sum_{k=0}^{K-1} (\mathbb{1}_{\{y-k\}}\ log\ v_k) \end{aligned} \tag{2}$$

We use Stochastic Gradient Descent to minimize the $\bigtriangledown_\theta \rho$ = - $\bigtriangledown_\theta log(F_{softmax})$.
To update the parameter values using SGD, we compute the derivatives of the parameters

namely $\frac{\partial \rho}{\partial b}$, $\frac{\partial \rho}{\partial K}$ and $\frac{\partial \rho}{\partial W}$ using the equations in 3.

$$\begin{aligned} \frac{\partial \rho}{\partial U} &= -(e(y) - f(x, \theta)) \\ \frac{\partial \rho}{\partial b} &= \frac{\partial \rho}{\partial U} \\ \frac{\partial \rho}{\partial W_{k,:,:,:}} &= \frac{\partial \rho}{\partial U_k} H \\ \delta_{i,j,p} &= \frac{\partial \rho}{\partial U} W_{:,i,j,p} \\ \frac{\partial \rho}{\partial K_{:,:,p}} &= X * (\sigma'(Z_{:,:,p}) \cdot \delta_{:,:,p}) \end{aligned} \tag{3}$$

where e(y) = $\{\mathbb{1}_{\{y=0\}}, ..., \mathbb{1}_{\{y=k=10\}}\}$
Equation (3) defines the gradients of the model parameters.
The dimensions of $\frac{\partial \rho}{\partial b}$, $\frac{\partial \rho}{\partial W}$ and $\frac{\partial \rho}{\partial K}$ are same as dimensions of b, W, K in Step 1.
In my back propagation code, following is my implementation:

- $model\_grads = \{\}$ is the dictionary in my code containing the gradients of the model parameters, namely, model_grads['b'], model_grads['W'], and model_grads['K'].
- I wrote a function $backward(x, y, Z, H, prob\_dist, model, model\_grads)$ in my code to compute the gradients of the model parameters. It returns the dictionary model_grads. I used numpy.tensordot to compute model_grads['W'] and δ, avoiding the use of for loops for iterating while computing the sum of element wise multiplication.
  To compute model_grads['K'], I used my convolution() function. I pass the element wise multiplication of δ and $\sigma'(z)$ as K and then use numpy.tensordot to con-volute X and $\delta \cdot \sigma'(z)$. The convolution window is of size $d - d_y + 1 \times d - d_x + 1$ and we iterate through $0 : d_y$, $0 : d_x$ and $0 : C$ using itertools.product.
- For implementation, the derivative of activation function is $\sigma'(z) = 1 - tanh^2(z)$. The derivative of activation function is applied element wise.

Step 4: Parameter Update

- For implementation, the learning rate $\alpha^{(l)}$ is defined based on no. of epoch (l) we are training at: $\alpha^{(l)} = \begin{cases} 0.01, & \text{if epoch } \ l \leq 5 \\ 0.001, & \text{if epoch } l > 5 \\ 0.0001, & \text{if epoch } l > 10 \\ 0.00001, & \text{if epoch } l > 15 \end{cases}$

```
    for n in range(len(x_train)):
        n_random = randint(0,len(x_train)-1)
        y = y_train[n_random]
        x = x_train[n_random][:]
        Z, H, prob_dist = forward(x, y, model)
        prediction = np.argmax(prob_dist)
        if (prediction == y):
            total_correct += 1
        model_grads = backward(x, y, Z, H, prob_dist, model, model_grads)
        model['W'] = model['W'] - LR*model_grads['W'] # Updating the parameters W, b_1, C and b_2 via the SGD step
        model['b'] = model['b'] - LR*model_grads['b']
        model['K'] = model['K'] - LR*model_grads['K']
    print('In epoch ', epochs, ', accuracy in training set = ', total_correct/np.float(len(x_train)))
test_accuracy = compute_accuracy(x_test, y_test, model)
print('Accuracy in testing set =', test_accuracy)
```

```
In epoch  0 , accuracy in training set =  0.9292833333333334
In epoch  1 , accuracy in training set =  0.9618166666666667
In epoch  2 , accuracy in training set =  0.96655
In epoch  3 , accuracy in training set =  0.9711
In epoch  4 , accuracy in training set =  0.9726333333333333
In epoch  5 , accuracy in training set =  0.9764333333333334
In epoch  6 , accuracy in training set =  0.98625
In epoch  7 , accuracy in training set =  0.9881666666666666
In epoch  8 , accuracy in training set =  0.9894333333333334
In epoch  9 , accuracy in training set =  0.9901
Accuracy in testing set = 0.9735
```

Figure 1: Results

For each iteration t out of 60000 iterations, in each epoch l, we update the parameters of the neural network model $\{K, W, b\}$ using the stochastic gradient descent update as below:

$$\begin{aligned} K^{(t)} &= K^{(t)} - \alpha^{(l)} \frac{\partial \rho}{\partial K^{(t)}} \\ W^{(t)} &= W^{(t)} - \alpha^{(l)} \frac{\partial \rho}{\partial W^{(t)}} \\ b^{(t)} &= b^{(t)} - \alpha^{(l)} \frac{\partial \rho}{\partial b^{(t)}} \end{aligned} \tag{4}$$

where the parameter values are calculated in forward step in Step 1 and derivatives are calculates in backward step in Step 2.
A snapshot of the training and test accuracy's is given in Figure 1.
Note that the accuracy on the training set after 10 epochs reaches approx. 0.9901, but clearly since the test accuracy is 0.9735, we can notice that the model starts to **overfit** the training set.

Accuracy

- The accuracy of the model on the test set = 97.35%
- The estimated time to train the model for 10 epochs = approx. 4 hours.

I wrote a function *compute_accuracy*() in my code to compute the accuracy of the test set. I predict the label based on each element in test set and check if the true label is same as the predicted label to calculate the accuracy.