Deep Learning

Y.Chen

November 2023

1 Assignment 1

All source code can be accessed here [GitHub]

Q1: Work out the local derivatives of both, in scalar terms. Show the derivation. Assume that the target class is given as an integer value.

A:

Use the softmax function $y_i = \frac{\exp o_j}{\sum_k \exp o_k}$ to find its local derivative with respect to o_j

- When $i = j$

$$\frac{\partial y_i}{\partial o_j} = \frac{\partial}{\partial o_j}\frac{\exp o_j}{\sum_k \exp o_k} = \frac{\exp o_j \sum_k \exp o_k - \exp o_j \exp o_j}{(\sum_k \exp o_k)^2} = \frac{\exp o_j}{\sum_k \exp o_k}\left(1 - \frac{\exp o_j}{\sum_k \exp o_k}\right) = y_j(1-y_j)$$

- When $i \neq j$

$$\frac{\partial y_i}{\partial o_j} = \frac{\partial}{\partial o_j}\frac{\exp o_i}{\sum_k \exp o_k} = \frac{0 - \exp o_i \exp o_j}{(\sum_k \exp o_k)^2} = -\frac{\exp o_i}{\sum_k \exp o_k}\frac{\exp o_j}{\sum_k \exp o_k} = -y_i y_j$$

Use the Loss function $l = -\log y_c$ to find local derivative with respect to y_j.

- When j = c , where c is the true class index, we have

$$\frac{\partial l}{\partial y_j} = \frac{-\partial \log y_c}{\partial y_j} = -\frac{1}{y_c} = -\frac{1}{y_j}$$

- When $j \neq c$

$$\frac{\partial l}{\partial y_j} = \frac{-\partial \log y_c}{\partial y_j} = 0$$

- As piecewise function:

$$\frac{\partial l}{\partial y_j} = \begin{cases} -\frac{1}{y_j} & \text{if } j = c \\ 0 & \text{if } j \neq c \end{cases}$$

$$\frac{\partial y_i}{\partial o_j} = \begin{cases} y_j(1-y_j) & \text{if } i = j \\ -y_i y_j & \text{if } i \neq j \end{cases}$$

$Q2^{*1}$: Work out the derivative $\frac{\partial l}{\partial o_j}$. Why is this not strictly necessary for a neural network, if we already have the two derivatives we worked out above?

A:

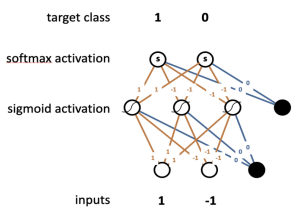
Use the chain rule and the results from the previous steps to get local derivative of l with respect to o_j

$$\frac{\partial l}{\partial o_j} = \sum_i \frac{\partial l}{\partial y_i}\frac{\partial y_i}{\partial o_j} = -t_j + y_j \sum_j t_j = \begin{cases} y_j - 1 & \text{if } j = c \\ y_j - 0 & \text{if } j \neq c \end{cases} = y_j - t_j$$

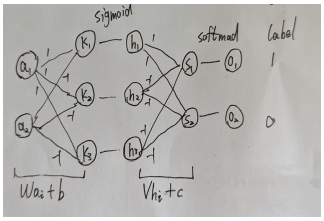
where t is the target class

- Because we the two derivatives we worked out above are the local derivatives of the loss function and the network output in scalar terms. They are not sufficient to calculate the gradient of the loss function with respect to all of the network's parameters. We use backpropagation to update the parameters.

Q3. Implement the network drawn in the image below.



(a) 2-layer Network



(b) Workflow

A:

1. Apply the parameters from the network:

```
inputs = [1., -1.]
W = [[1., 1., 1.], [-1., -1., -1.]]
b = [0., 0., 0.]
V = [[1., 1.], [-1., -1.], [-1., -1.]]
c = [0., 0.]
```

2. Forward pass:

```
k = np.dot(inputs, W) + b
h = sigmoid.forward(k)
s = np.dot(h, V) + c
o = softmax.forward(s)
```

3. Per diagram, the first element in output is classified 1. Cross-entropy loss derivative for correct class will be:

```
l, dl = [(-np.log(o[i]), -1/o[i]) for i in range(len(o)) if t[i] == 1][0]
>>> print(l,dl)
0.6931471805599453 -2.0
```

4. Backward pass:

```
ds = np.array(softmax.backward(dl)) #dl/ds, c
dh = np.dot(ds, np.array(V).T) #dl/dh, b
dk = sigmoid.backward(dh) #dl/dk
dw = np.dot(np.array([inputs]).T, np.array([dk]))
dv = np.dot(np.array([h]).T, np.array([ds])) #dl/dv, V
```

5. Output for the *Loss* with respect to the parameters:
   print("W:", dw)
   W: [[0. 0. 0.] [0. 0. 0.]]
   print("b", dh)
   b [0. 0. 0.]
   print("V", dv)
   V [[-0.44039854 0.44039854] [-0.44039854 0.44039854] [-0.44039854 0.44039854]]
   print("c", ds)
   c [-0.5 0.5]

Q4: Implement a training loop for your network and show that the training loss drops as training progresses.

A:

Normalize the synthetic data to the range $(0, 1)$. Set the network layers *input_dim*, *hidden_dim* and *output_dim* to 2, 3 and 2 respectively. Convert the true labels to One-hot format. Apply stochastic gradient descent algorithm. For each epoch, randomly select one train data to calculate the loss and update the weights $w1$&$w2$ and biases $b1$&$b2$ with *learning_rate* of 0.01. Repeat 100 epochs. Print training loss from the last 4 epochs and final results.

Epoch 97/100
Loss:13.430531652042282
Epoch 98/100
Loss: 13.31580743092673
Epoch 99/100
Loss: 9.824139011435912
Epoch 100/100
Loss: 9.739905146114124
w1:[[0.00460806,-0.00507444,-0.00223027],[0.01274007,0.011903,-0.03032788]]
b1:[0.00414125,0.00157709,0.00222446]
w2: [[-0.04568013, 0.05823134], [-0.03630828, 0.04085528], [-0.04702419, 0.0362655
]]

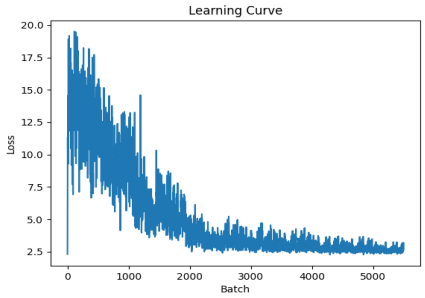
b2: [-0.096495, 0.096495]
final loss: 9.739905146114124

Q5&6: Implement a neural network for the MNIST data. Use two linear layers as before, with a hidden layer size of 300, a sigmoid activation, and a softmax activation over the output layer, which has size 10. Batch the workflow(Q6*).
A:

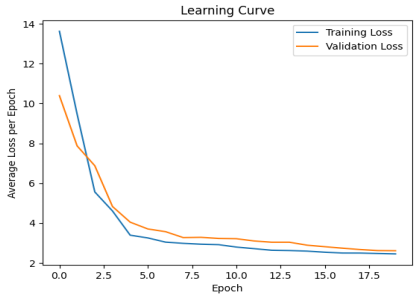
Normalize the synthetic data to the range (0,1). Set the network layers *input_dim*, *hidden_dim* and *output_dim* to 784, 300 and 10 respectively. Convert the true labels to One-hot format with 10 levels. Set the batch size of 100 for faster performance. Average the gradients for each batch and update the parameters per batch.
Print the batch average loss from last 4 batches of epoch 100.
Batch loss:
2.319679248787786
2.3145017813227646
2.3316020977610856
2.355157514421069

Q7: Train the network on MNIST and plot the loss of each batch or instance against the timestep. This is called a learning curve or a loss curve.

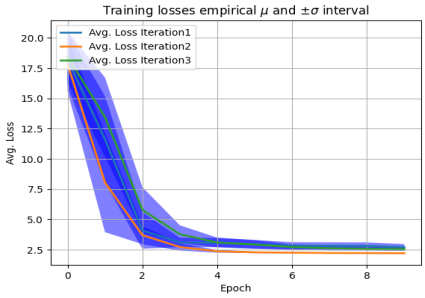
A:
1) Figure(c) shows loss from all epochs; Figure(d) shows average loss from each epoch. The validation set is split from the training data. The training loss line decreases steadily as the epoch increases, while the validation loss line decreases at the beginning, but not as fast as the training loss line after 5 epochs. This results in an over-fitted model that performs well on the training data but poorly on new data. This could happen since the parameters are not optimized but used for validation per epoch.
2) Figure(e) shows the average training loss by epochs with three iterations of random initial parameters. Average loss of training data converge after epochs despite of different initial parameters.
3) After running the SGD with different learning rates(i.e, 0.001, 0.003, 0.01, 0.03) with 1000 epochs, the average loss shows as following:
Learning rate: 0.001, Epoch: 1000, Avg. loss: 3.9511415751435145e-05
Learning rate: 0.003, Epoch: 1000, Avg. loss: 7.029711386482138e-06
Learning rate: 0.01, Epoch: 1000, Avg. loss: 4.7097647394737164e-05
Learning rate: 0.03, Epoch: 1000, Avg. loss: 7.64090772728009e-05
The learning rate of 0.003 appears to provide the best balance between convergence speed and final accuracy. With a learning rate of 0.001, the model learns slowly and may not fully converge within the whole epochs. Higher learning rate can result in oscillation near the optimal solution point. On the other hand, it may avoid the model falling into the local optimal solution point.
4)Select the parameters trained with learning rate of 0.03, batch size of 100 and 10 epochs, the accuracy on canonical validation data is 0.1428



(c) All average training loss



(d) Avg. loss per epoch



(e) Avg. loss of iterations