Problem Set + Programming Assignment 3

Teera Tesharojanasup

Northeastern University, Boston

August 6th, 2024

Overview

Problem set 3 and Programming Assignment 3 are combined for CS 4100 Summer II. Taught by assistant teaching professor, Rajagopal Venkat. [1]

1 Loss, Gradients, and PyTorch

Preliminaries: Recall our discussion of loss functions from class. For a multi-class classification problem, given a prediction model f, a common loss function is the Categorical Cross Entropy loss, which is calculated as follows:

$$L(x_i, y_i) = -y_i \cdot log(f(x_i))$$

Here, y_i is a one-hot vector (i.e. a vector of length m, where m is the number of classes) with an entry of 1 in the position corresponding to the true class and 0 elsewhere. Similarly, the model output $f(x_i)$ is a vector, containing the probabilities that the input belongs to each of the classes.

Q1 **Consider a logistic regression ($\hat{y}_i = \sigma(w \cdot x_i + b)$) with two classes (0 and 1). Given the input example, $x_i = [6, 4, 2, 3, 1]$, model weights $w = [0.5, 0.1, -0.8, 0.9, 0.0]$, bias term, $b = 0.05$ and the true label encoded as $y_i = [0, 1]$, compute the cross entropy loss and show every step of your calculation. (Hint: a logistic regression gives you the probability of the '1' class.).** (4)

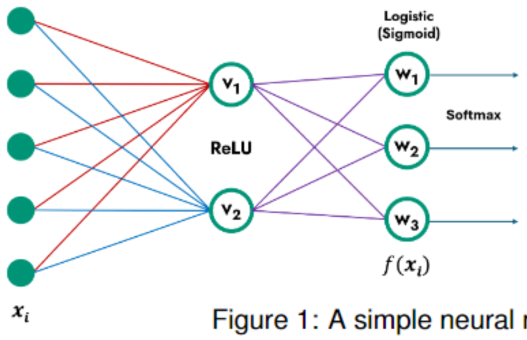
$$\begin{aligned}
\hat{y}_i &= \sigma\left(\begin{bmatrix}0.5 & 0.1 & -0.8 & 0.9 & 0.0\end{bmatrix} \cdot \begin{bmatrix}6\\4\\2\\3\\1\end{bmatrix} + 0.05\right)\\
&= \sigma\Big(4.5 + 0.05\Big)\\
&= \sigma(4.55)\\
&= \frac{1}{1+e^{-4.55}}\\
&= 0.9895
\end{aligned}$$

$$\begin{aligned}
L(x_i, y_i) &= -\begin{bmatrix}0 & 1\end{bmatrix} \cdot log(0.9895)\\
&= 0 - log(0.9895)\\
&= 0.0106
\end{aligned}$$

Q2 **The Rectified Linear Unit (ReLU) activation function is defined as:**

$$ReLU(x) = \frac{x + |x|}{2} = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Now consider the neural network in the figure below, with 3 possible classes instead. Assume that the hidden layer uses the ReLU activation function instead of the sigmoid function discussed in class. The final layer uses the sigmoid (logistic function) activation function, followed by a Softmax operation to ensure that the outputs add up to 1.



	Weights	Bias
$\mathbf{v_1}$	$[0.5, -0.1, -0.2, 0.1, -0.4]$	0.02
$\mathbf{v_2}$	$[-0.9, 0.7, 0.7, -0.5, 0.2]$	-0.01
$\mathbf{w_1}$	$[0.1, 0.6]$	0.0
$\mathbf{w_2}$	$[0.8, 0.4]$	0.05
$\mathbf{w_3}$	$[0.7, -0.2]$	0.04

Figure 1: A simple neural network and its parameters

For $x_i = [5, 7, 4, 3, 2]$, and the true class label 2 (where the possible classes are 0, 1, and 2), use the model from the diagram, and the given weights table (including nonzero biases this time) to compute the cross entropy loss. You may use NumPy/SciPy operations (but not the loss function methods from these packages) to perform the calculations for this question - if doing so, paste a screenshot of your function(s) and the final result into your written submission. You may find the SciPy expit function useful: [link]. (6)

ReLU Activation:

$$\begin{aligned} z_{v_1} &= w_{v_1} \cdot x_i + b \\ &= \begin{bmatrix} 0.5 & -0.1 & -0.2 & 0.1 & -0.4 \end{bmatrix} \begin{bmatrix} 5 \\ 7 \\ 4 \\ 3 \\ 2 \end{bmatrix} + 0.02 \\ &= 0.5 + 0.02 = 0.52 \end{aligned}$$

$$ReLU(0.52) = \frac{0.52 + |0.52|}{2} = 0.52$$

$$\begin{aligned} z_{v_2} &= w_{v_2} \cdot x_i + b \\ &= \begin{bmatrix} -0.9 & 0.7 & 0.7 & -0.5 & 0.2 \end{bmatrix} \begin{bmatrix} 5 \\ 7 \\ 4 \\ 3 \\ 2 \end{bmatrix} - 0.01 \\ &= 2.1 - 0.01 = 2.09 \end{aligned}$$

$$ReLU(2.09) = \frac{2.09 + |2.09|}{2} = 2.09$$

Sigmoid Activation:

$$\begin{aligned}
z_{w_1} &= w_{w_1} \cdot \begin{bmatrix} 0.52 \\ 2.09 \end{bmatrix} + b \\
&= \begin{bmatrix} 0.1 & 0.6 \end{bmatrix} \begin{bmatrix} 0.52 \\ 2.09 \end{bmatrix} + 0.0 \\
&= 1.306 + 0.0 = 1.306 \\
\sigma(1.306) &= \frac{1}{1 + e^{-1.306}} = 0.7868
\end{aligned}$$

$$\begin{aligned}
z_{w_2} &= w_{w_2} \cdot \begin{bmatrix} 0.52 \\ 2.09 \end{bmatrix} + b \\
&= \begin{bmatrix} 0.8 & 0.4 \end{bmatrix} \begin{bmatrix} 0.52 \\ 2.09 \end{bmatrix} + 0.05 \\
&= 1.252 + 0.0 = 1.302 \\
\sigma(1.302) &= \frac{1}{1 + e^{-1.302}} = 0.7862
\end{aligned}$$

$$\begin{aligned}
z_{w_3} &= w_{w_3} \cdot \begin{bmatrix} 0.52 \\ 2.09 \end{bmatrix} + b \\
&= \begin{bmatrix} 0.7 & -0.2 \end{bmatrix} \begin{bmatrix} 0.52 \\ 2.09 \end{bmatrix} + 0.04 \\
&= -0.054 + 0.04 = -0.014 \\
\sigma(-0.014) &= \frac{1}{1 + e^{0.014}} = 0.4965
\end{aligned}$$

Softmax Activation:

$$s_{w_1} = \frac{e^{0.7868}}{e^{0.7868} + e^{0.7862} + e^{0.4965}} = \frac{2.1964}{6.0344} = 0.3640$$

$$s_{w_2} = \frac{e^{0.7862}}{e^{0.7868} + e^{0.7862} + e^{0.4965}} = \frac{2.1950}{6.0344} = 0.3638$$

$$s_{w_3} = \frac{e^{0.4965}}{e^{0.7868} + e^{0.7862} + e^{0.4965}} = \frac{1.6430}{6.0344} = 0.2723$$

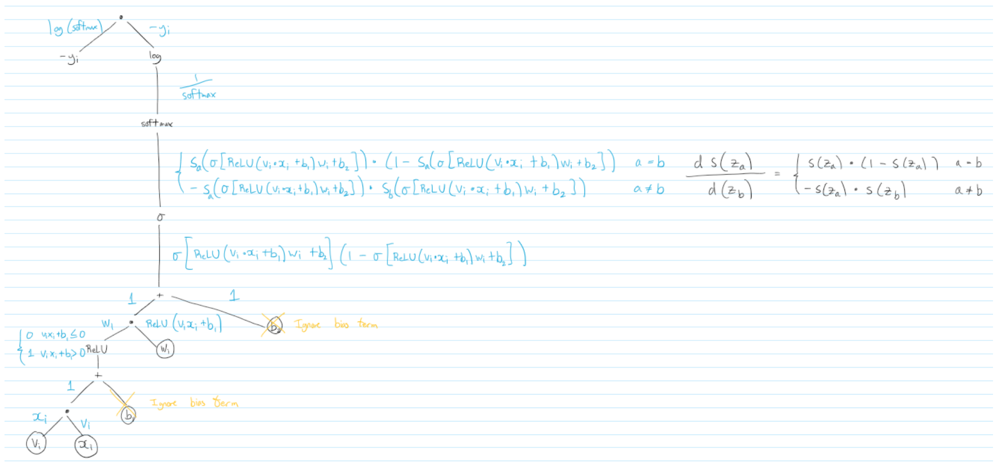
Cross Entropy Loss:

$$L(x_i, y_i) = -\begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \cdot \log \left(\begin{bmatrix} 0.3640 \\ 0.3638 \\ 0.2723 \end{bmatrix} \right)$$

$$= -1 \cdot log(0.2723) = 1.3009$$

Q3 **Using a computation graph (of the kind we discussed in class), compute the gradient of the cross entropy loss with respect to the model parameters in the above network. Repeatedly split the loss function by operators like we did in class, and ignore all bias terms. You do not need to split vector/matrix operators into elementwise/row-wise operations. For this question alone, you may scan a hand-drawn figure. Any writing must be perfectly legible. Finally, use NumPy to compute the value of the gradient of the loss function with respect to all parameters (w_1, w_2, w_3, v_1, v_2) for the given input**

and true label. Note that the *log* operator refers to the natural logarithm (base *exp*). Report your final values and a screenshot of your code. HINT: [Derivative of Softmax] (5 + 5)


log(softmax)
$-y_i$
$-y_i$
log
$\frac{1}{\text{softmax}}$
softmax
$\begin{cases} S_a(\sigma[\text{ReLU}(v_i \cdot x_i + b_1)w_i + b_2]) \cdot (1 - S_a(\sigma[\text{ReLU}(v_i \cdot x_i + b_1)w_i + b_2])) & a = b \\ -S_a(\sigma[\text{ReLU}(v_i \cdot x_i + b_1)w_i + b_2]) \cdot S_b(\sigma[\text{ReLU}(v_i \cdot x_i + b_1)w_i + b_2]) & a \neq b \end{cases}$
$\frac{d\,s(z_a)}{d(z_b)} = \begin{cases} s(z_a) \cdot (1 - s(z_a)) & a = b \\ -s(z_a) \cdot s(z_b) & a \neq b \end{cases}$
σ
$\sigma[\text{ReLU}(v_i \cdot x_i + b_1)w_i + b_2](1 - \sigma[\text{ReLU}(v_i \cdot x_i + b_1)w_i + b_2])$
1
$+$
1
w_i
$\text{ReLU}(v_i x_i + b_1)$
b_2
Ignore bias term
w_i
$\begin{cases} 0 & v_i x_i + b_1 \leq 0 \\ 1 & v_i x_i + b_1 > 0 \end{cases}$
ReLU
$+$
1
b_1
Ignore bias term
x_i
v_i
v_i
x_i

```python
import numpy as np


def ReLU_activation(w, x, b):
    ReLU_values = []
    for wi, bi in zip(w, b):
        z = wi.dot(x) + bi

        if z > 0:
            value = (z + np.abs(z)) / 2
        else:
            value = 0

        ReLU_values.append(value)

    return np.array(ReLU_values)


def sigmoid_activation(w, x, b):
    sigmoid_values = []
    for wi, bi in zip(w, b):
        z = wi.dot(x) + bi

        value = 1 / (1 + np.exp(-z))
        sigmoid_values.append(value)

    return np.array(sigmoid_values)


def softmax_activation(x):
    softmax_values = []

    for xi in x:
        value = np.exp(xi) / np.sum(np.exp(x))
        softmax_values.append(value)

    return np.array(softmax_values)


def cross_entropy(px, softmax_values):
    loss_value = -1 * (px.dot(np.log(softmax_values)))

    return loss_value


def ReLU_derivative(vi, xi):
    z = vi.dot(xi)

    return (z > 0).astype(int)


def sigmoid_derivative(sigmoid_values):
    derivative = sigmoid_values * (1 - sigmoid_values)

    return derivative


def softmax_derivative(S):
    derivative = np.zeros((len(S), len(S)))

    for i in range(len(S)):
        for j in range(len(S)):
            if i == j:
                derivative[i, j] = S[i] * (1 - S[i])
            else:
                derivative[i, j] = -S[i] * S[j]

    return derivative


def compute_w_gradient(yi, S, sig_val, ReLU_val):
    first_layer = yi * -1
    second_layer = 1 / S
    third_layer = softmax_derivative(S)
    fourth_layer = sigmoid_derivative(sig_val)
    fifth_layer = 1
    sixth_layer = ReLU_val

    result = first_layer * second_layer
    result = result.dot(third_layer)
    result = result * fourth_layer
    result = np.outer(result, sixth_layer)

    return result


def compute_v_gradient(yi, S, w, sig_val, ReLU_val, vi, xi):
    first_layer = yi * -1
    second_layer = 1 / S
    third_layer = softmax_derivative(S)
    fourth_layer = sigmoid_derivative(sig_val)
    fifth_layer = 1
    sixth_layer = w
    seventh_layer = ReLU_derivative(vi, xi)
    eigth_layer = 1
    ninth_layer = xi

    result = first_layer * second_layer
    result = result.dot(third_layer)
    result = result * fourth_layer
    result = result.dot(sixth_layer)
    result = result * seventh_layer
    result = np.outer(result, ninth_layer)

    return result
```

```python


def main():
    vi = np.array([[0.5, -0.1, -0.2, 0.1, -0.4],
                   [-0.9, 0.7, 0.7, -0.5, 0.2]])

    wi = np.array([[0.1, 0.6],
                   [0.8, 0.4],
                   [0.7, -0.2]])

    x = np.array([5, 7, 4, 3, 2])

    # we can set biases to 0, essentially ignoring them
    v_biases = np.array([0, 0])
    w_biases = np.array([0, 0, 0])

    px = np.array([0, 0, 1])

    ReLU_values = ReLU_activation(vi, x, v_biases)
    sigmoid_values = sigmoid_activation(wi, ReLU_values, w_biases)
    softmax_values = softmax_activation(sigmoid_values)
    loss_value = cross_entropy(px, softmax_values)

    w_gradient = compute_w_gradient(px, softmax_values, sigmoid_values,
       ReLU_values)
    v_gradient = compute_v_gradient(px, softmax_values, wi, sigmoid_values,
       ReLU_values, vi, x)

    print("W Gradient:")
    print(w_gradient)
    print()
    print("V Gradient:")
    print(v_gradient)

if __name__ == "__main__":
    main()
```

Listing 1: Python Code to Compute Derivative

```
W Gradient:
[[ 0.03070117  0.12894493]
 [ 0.03155628  0.13253638]
 [-0.09107858 -0.38253002]]

V Gradient:
[[-0.35439862 -0.49615807 -0.2835189  -0.21263917 -0.14175945]
 [ 0.49258931  0.68962504  0.39407145  0.29555359  0.19703572]]
```

Listing 2: Output of Listing 1

2 Fashion-MNIST Classification

In this section, you will be working with a dataset called Fashion-MNIST. Your task is to design and train a neural network that classifies each image of the dataset correctly. Data is downloaded directly from within the script (using PyTorch). The actual architecture of the neural network is up to you, and convolutional layers are optional.

Fashion-MNIST contains grayscale images of 28 x 28 pixels representing images of clothing. The dataset has 60000 training images, and 10000 testing images, and each image comes with an associated label (e.g. t-shirt, coat, bag, etc.). There are 10 classes, just like the MNIST handwritten digits dataset, so that it may serve as a direct drop-in replacement to test neural networks. Read the full details about this dataset at the repository.

The starter code for this part of the assignment is called "fashionmnist.py". The skeleton code serves as a general guide. There are some sections without any guidelines where you will be expected to research and experiment with various techniques to find a good approach. You must use PyTorch in this section, which is well-documented online. (30)

Your accuracy on the testing dataset must be greater or equal to <u>80%</u>.

Q4 **Submit two figures: A figure containing an image that is classified incorrectly by your model. Include a clear label in this figure that indicates the predicted class from your model and the true class the image belongs to (both human-readable labels, not just the class number). The second figure should be a single image classified correctly by your model and its corresponding class label. (3)**

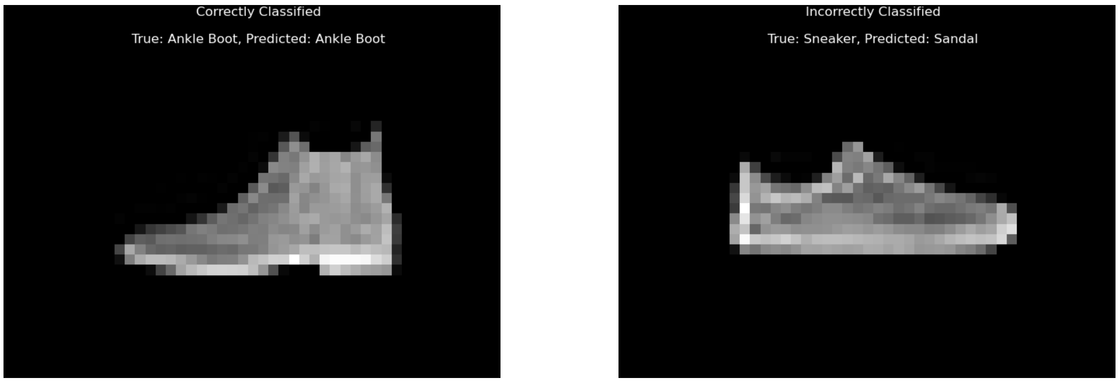

Correctly Classified
True: Ankle Boot, Predicted: Ankle Boot
Incorrectly Classified
True: Sneaker, Predicted: Sandal

Figure 1: Correctly and Incorrectly Classified Images

Q5 **Submit a plot showing your training loss over time. (2)**

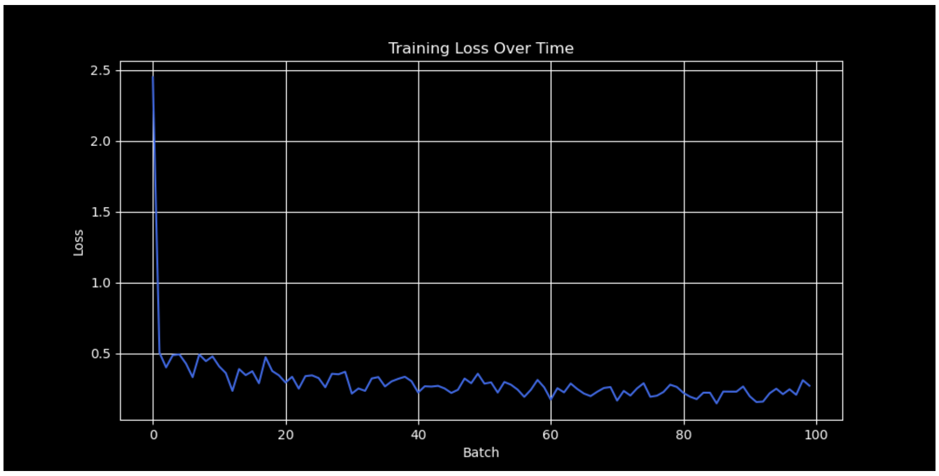


Figure 2: Training Loss

Q6 **Assume both the testing and training datasets have twice as many coat images as shirt images, and answer the following questions: (6)**

(a) **Would accuracy still be a good metric?**

Since there are now twice as many coat images as shirt images, if the model simply predicts coats more often, it will achieve a higher accuracy due to the data imbalance. This can be misleading since the model can perform poorly when classifying shirt images but still have high overall accuracy due to the abundance of coat images and its tendency to predict coats more often.

(b) **How would you modify your accuracy metric to account for data imbalance?**
We can implement weighted accuracy. This would mean that we assign weights to each class based on its prevalence in the dataset. Since there are twice as many coat images as shirt images, w_{coat} should be $\frac{1}{2}$ of w_{shirt}. This would mean that predicting shirts accurately would mean twice more than predicting coats accurately.

(c) **What other metrics can you use to evaluate model performance?**
We can use precision and/or recall instead of accuracy to evaluate model performance when facing an unbalanced dataset.

Q7 **Calculate the total number of tunable parameters your model has. Show your work, and don't forget the bias terms. Do not use code output to answer this question.** (4)

First Linear Layer:

$$\begin{aligned} &Weights : 28 \cdot 28 \cdot 512 = 401,408 \\ &Biases : 512 \\ &Total = 401,408 + 512 = 401,920 \end{aligned}$$

Second Linear Layer:

$$\begin{aligned} &Weights : 512 \cdot 512 = 262,144 \\ &Biases : 512 \\ &Total = 262,144 + 512 = 262,656 \end{aligned}$$

Third Linear Layer:

$$\begin{aligned} &Weights : 512 \cdot 10 = 5,120 \\ &Biases : 10 \\ &Total = 5,120 + 10 = 5,130 \end{aligned}$$

Total Number of Tunable Parameters (excluding BatchNorm): $401,920 + 262,656 + 5,130 = 669,706$

References

[1] R. Venkatesaramani, "Personal website." https://rajagopalvenkat.com/. Accessed: 2024-07-23.