Oregon State University

Assignment #2: CIFAR Image Classification using Fully-Connected Network

AI 535 Deep Learning

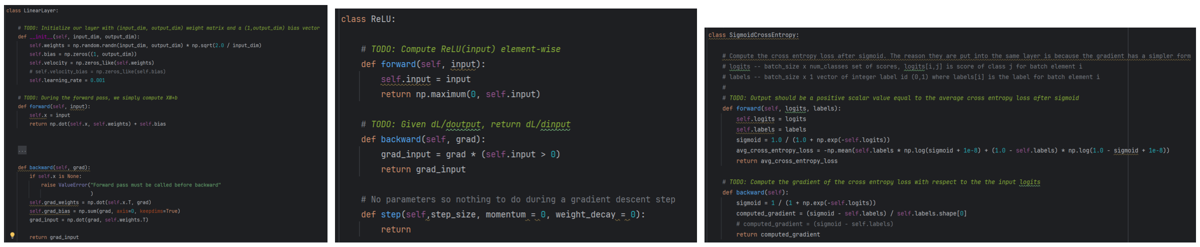
Hyuntaek Oh

ohhyun@oregonstate.edu

Due: Feb 9, 2025

1. Write a function that computes the linear, ReLU and SigmoidCrossEntropy layers (5 points each).
All three layer classes such as linear, ReLU and SigmoidCrossEntropy store the variables that have necessary information for computing backward while forwarding.

- *In the linear layer class, random, small-value weights and zero-value bias are initialized. Forward function uses input value to calculate for $XW + b$. On the other hand, backward function utilizes input "grad" to get the gradient of weights and bias, and then it returns "grad input".*
- *Forward function in the ReLU class compares the value of input with zero and returns the larger value. Like the linear layer, backward function returns "grad input", depending on the input value, which is zero if greater than 1, 0 if equal to or less than 0.*
- *SigmoidCrossEntropy class has forward function that takes logits and labels as inputs to compute sigmoid value. The result of this computation is used to produce the cross entropy loss. Backward function also gets the sigmoid value and returns the gradient of the loss function.*



```python
class LinearLayer:

    # TODO: Initialize our layer with (input_dim, output_dim) weight matrix and a (1,output_dim) bias vector
    def __init__(self, input_dim, output_dim):
        self.weights = np.random.randn(input_dim, output_dim) * np.sqrt(2.0 / input_dim)
        self.bias = np.zeros((1, output_dim))
        self.velocity = np.zeros_like(self.weights)
        # self.velocity_bias = np.zeros_like(self.bias)
        self.learning_rate = 0.001

    # TODO: During the forward pass, we simply compute XW+b
    def forward(self, input):
        self.x = input
        return np.dot(self.x, self.weights) + self.bias


    ...

    def backward(self, grad):
        if self.x is None:
            raise ValueError("Forward pass must be called before backward"
                             )
        self.grad_weights = np.dot(self.x.T, grad)
        self.grad_bias = np.sum(grad, axis=0, keepdims=True)
        grad_input = np.dot(grad, self.weights.T)

        return grad_input
```

```python
class ReLU:

    # TODO: Compute ReLU(input) element-wise
    def forward(self, input):
        self.input = input
        return np.maximum(0, self.input)

    # TODO: Given dL/doutput, return dL/dinput
    def backward(self, grad):
        grad_input = grad * (self.input > 0)
        return grad_input

    # No parameters so nothing to do during a gradient descent step
    def step(self,step_size, momentum = 0, weight_decay = 0):
        return
```

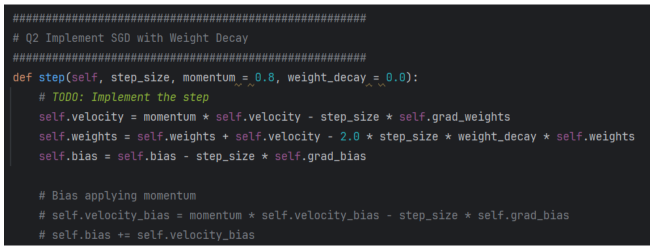
```python
class SigmoidCrossEntropy:

    # Compute the cross entropy loss after sigmoid. The reason they are put into the same layer is because the gradient has a simpler form
    # logits -- batch_size x num_classes set of scores, logits[i,j] is score of class j for batch element i
    # labels -- batch_size x 1 vector of integer label id (0,1) where labels[i] is the label for batch element i
    #
    # TODO: Output should be a positive scalar value equal to the average cross entropy loss after sigmoid
    def forward(self, logits, labels):
        self.logits = logits
        self.labels = labels
        sigmoid = 1.0 / (1.0 + np.exp(-self.logits))
        avg_cross_entropy_loss = -np.mean(self.labels * np.log(sigmoid + 1e-8) + (1.0 - self.labels) * np.log(1.0 - sigmoid + 1e-8))
        return avg_cross_entropy_loss

    # TODO: Compute the gradient of the cross entropy loss with respect to the the input logits
    def backward(self):
        sigmoid = 1 / (1 + np.exp(-self.logits))
        computed_gradient = (sigmoid - self.labels) / self.labels.shape[0]
        # computed_gradient = (sigmoid - self.labels)
        return computed_gradient
```

Figure 1: Linear(left), ReLU(middle), and SigmoidCrossEntropy(right)

2. Write a function that performs stochastic mini-batch gradient descent training with momentum (10 points).
According to the Lecture, since it is hard to find global minimum with only using traditional gradient descent method, the concepts of momentum and weight decay are used to handle its difficulty. The equations of momentum and weight decay are based on the Lecture slides, and they are all together in the gradient update at once.



```python
#######################################################
# Q2 Implement SGD with Weight Decay
#######################################################
def step(self, step_size, momentum = 0.8, weight_decay = 0.0):
    # TODO: Implement the step
    self.velocity = momentum * self.velocity - step_size * self.grad_weights
    self.weights = self.weights + self.velocity - 2.0 * step_size * weight_decay * self.weights
    self.bias = self.bias - step_size * self.grad_bias

    # Bias applying momentum
    # self.velocity_bias = momentum * self.velocity_bias - step_size * self.grad_bias
    # self.bias += self.velocity_bias
```

Figure 2: Implementation Stochastic Gradient Descent with Weight Decay

3. Train the network on the attached 2-class dataset extracted from CIFAR-10. The data has 10,000 training examples in 3072 dimensions and 2,000 testing examples. Train on all the training examples, tune your parameters (number of hidden units, learning rate, mini-batch size, momentum) until you reach a good performance on the testing set. What accuracy can you achieve? (15 points based on the report).

The following parameters are used to train dataset and to test model's accuracy:

$$\textit{batch size} : 16$$

$$\textit{max epochs} : 20$$

$$\textit{learning rate} : 0.001$$

$$\textit{number of layers} : 3$$

$$\textit{width of layers(hidden units)} : 128$$

$$\textit{weight decay} = 0.05$$

$$\textit{momentum} = 0.8$$

Based on these parameters, the plot of the training and validation accuracies for each epoch is:

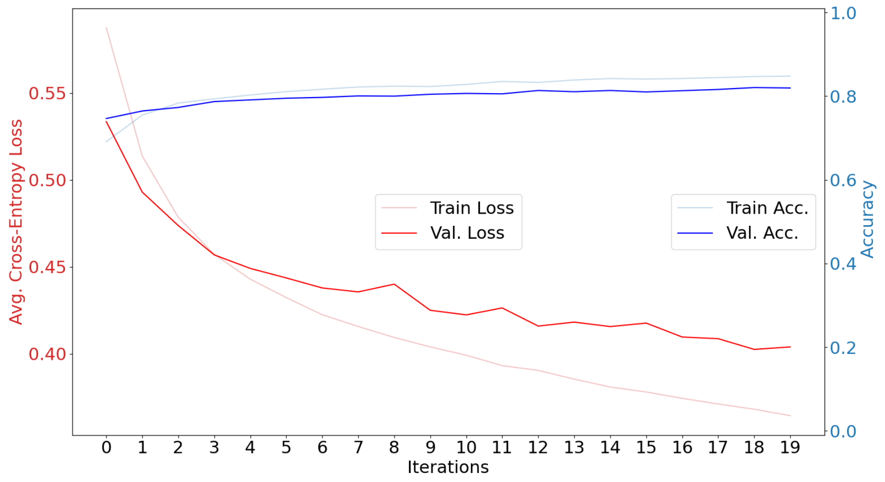
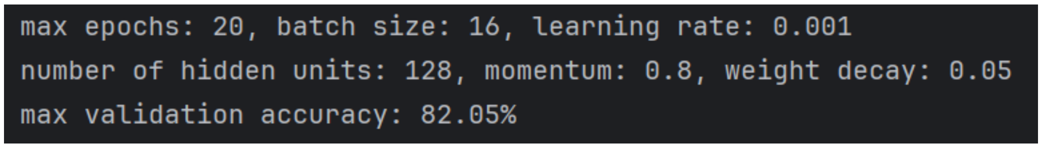


Figure 3: Training and validation accuracies

In this plot, the training loss gradually decreases and accuracy increases. Likely, validation loss and accuracy has similar tendency of training.



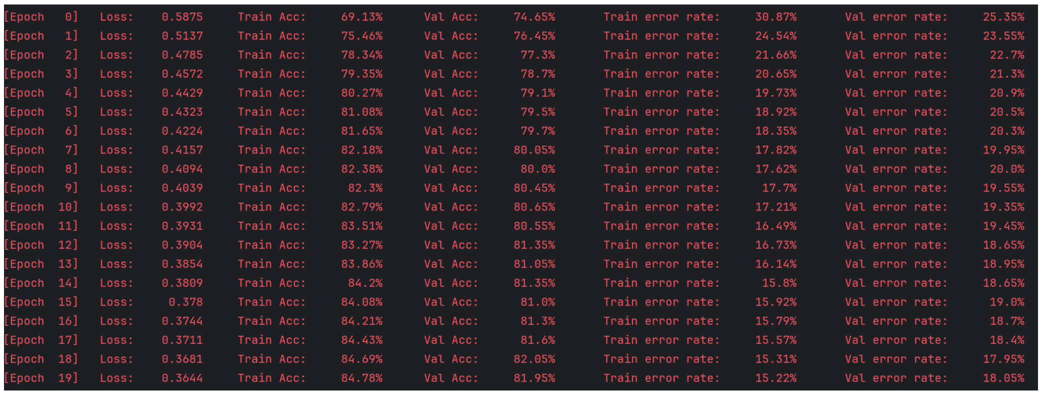
```
max epochs: 20, batch size: 16, learning rate: 0.001
number of hidden units: 128, momentum: 0.8, weight decay: 0.05
max validation accuracy: 82.05%
```

Figure 4: Max Validation Accuracy

As can be seen in Figure 4., the best validation accuracy is 82.05%.

4. Training Monitoring: For each epoch in training, your function should evaluate the training objective, training misclassification error rate (error is 1 for each example if misclassifies, 0 if correct), testing misclassification error rate (5 points).

The misclassification error rate is that error is 1 for each example if misclassifies, 0 if correct, meaning that it is opposite concept of accuracy, which is the number of correct predictions over the number of samples. Thus, the error rate can be computed by subtracting accuracy from one, error rate : $1 - accuracy$. *The results are as followed:*



```
[Epoch   0]   Loss:   0.5875   Train Acc:   69.13%   Val Acc:   74.65%   Train error rate:   30.87%   Val error rate:   25.35%
[Epoch   1]   Loss:   0.5137   Train Acc:   75.46%   Val Acc:   76.45%   Train error rate:   24.54%   Val error rate:   23.55%
[Epoch   2]   Loss:   0.4785   Train Acc:   78.34%   Val Acc:    77.3%   Train error rate:   21.66%   Val error rate:    22.7%
[Epoch   3]   Loss:   0.4572   Train Acc:   79.35%   Val Acc:    78.7%   Train error rate:   20.65%   Val error rate:    21.3%
[Epoch   4]   Loss:   0.4429   Train Acc:   80.27%   Val Acc:    79.1%   Train error rate:   19.73%   Val error rate:    20.9%
[Epoch   5]   Loss:   0.4323   Train Acc:   81.08%   Val Acc:    79.5%   Train error rate:   18.92%   Val error rate:    20.5%
[Epoch   6]   Loss:   0.4224   Train Acc:   81.65%   Val Acc:    79.7%   Train error rate:   18.35%   Val error rate:    20.3%
[Epoch   7]   Loss:   0.4157   Train Acc:   82.18%   Val Acc:   80.05%   Train error rate:   17.82%   Val error rate:   19.95%
[Epoch   8]   Loss:   0.4094   Train Acc:   82.38%   Val Acc:    80.0%   Train error rate:   17.62%   Val error rate:    20.0%
[Epoch   9]   Loss:   0.4039   Train Acc:    82.3%   Val Acc:   80.45%   Train error rate:    17.7%   Val error rate:   19.55%
[Epoch  10]   Loss:   0.3992   Train Acc:   82.79%   Val Acc:   80.65%   Train error rate:   17.21%   Val error rate:   19.35%
[Epoch  11]   Loss:   0.3931   Train Acc:   83.51%   Val Acc:   80.55%   Train error rate:   16.49%   Val error rate:   19.45%
[Epoch  12]   Loss:   0.3904   Train Acc:   83.27%   Val Acc:   81.35%   Train error rate:   16.73%   Val error rate:   18.65%
[Epoch  13]   Loss:   0.3854   Train Acc:   83.86%   Val Acc:   81.05%   Train error rate:   16.14%   Val error rate:   18.95%
[Epoch  14]   Loss:   0.3809   Train Acc:    84.2%   Val Acc:   81.35%   Train error rate:    15.8%   Val error rate:   18.65%
[Epoch  15]   Loss:    0.378   Train Acc:   84.08%   Val Acc:    81.0%   Train error rate:   15.92%   Val error rate:    19.0%
[Epoch  16]   Loss:   0.3744   Train Acc:   84.21%   Val Acc:    81.3%   Train error rate:   15.79%   Val error rate:    18.7%
[Epoch  17]   Loss:   0.3711   Train Acc:   84.43%   Val Acc:    81.6%   Train error rate:   15.57%   Val error rate:    18.4%
[Epoch  18]   Loss:   0.3681   Train Acc:   84.69%   Val Acc:   82.05%   Train error rate:   15.31%   Val error rate:   17.95%
[Epoch  19]   Loss:   0.3644   Train Acc:   84.78%   Val Acc:   81.95%   Train error rate:   15.22%   Val error rate:   18.05%
```

Figure 5: Accuracies and Error Rates of Training and Validation

5. Tuning Parameters: please create three figures with following requirements (5 points):
i) test accuracy with different number of batch size
ii) test accuracy with different learning rate
iii) test accuracy with different number of hidden units

(i) Batch Size
For test accuracy with different number of batch size, the batch sizes 16, 32, 64, 128, and 256 were used. Other parameters, learning rate and the number of hidden units, had fixed values, 0.001 and 256 respectively. Figure 6 showed that each batch size has different performance. As can be seen in Figure 6, the best accuracy is 83% *when the batch size is equal to 16. However, the increase in batch size not*

ensures better performance, even it can decrease the accuracy.

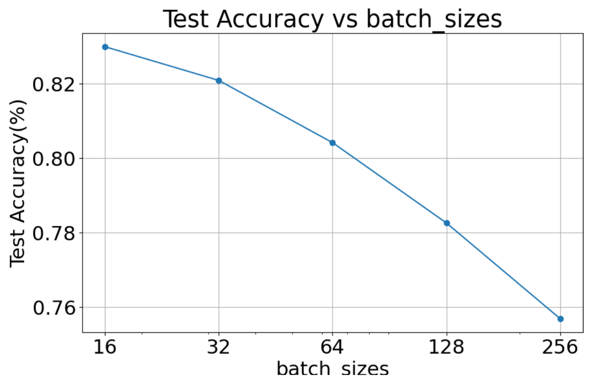


Figure 6: Test Accuracy vs. Batch Size

(ii) learning rate (step size)
For test accuracy with different learning rate (step size), the learning rate 0.001, 0.005, 0.01, 0.05, and 0.1 were used. Other parameters, batch size and the number of hidden units, had fixed values, 64 and 256 respectively. The Figure 7 indicates that each learning rate produces different performance. As shown in Figure 7, the best accuracy is 83.6% *when the learning rate is 0.005. Unlike batch sizes, the increase of the learning rate did not always decrease the accuracy. At certain point like 0.005, the accuracy is incredibly high, but right next value, 0.01, decreased model's performance.*

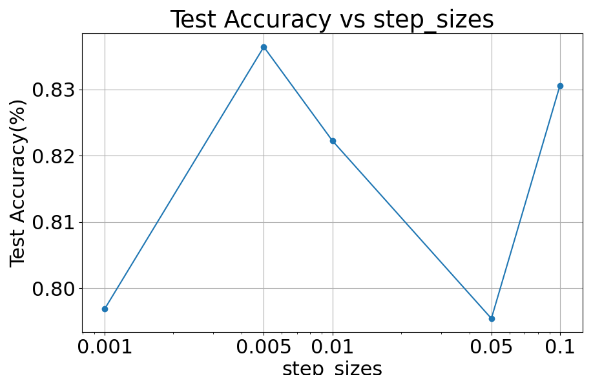


Figure 7: Test Accuracy vs. Learning Rate

(iii) Number of Hidden Units
For test accuracy with different number of hidden units, the values of hidden units 32, 64, 128, 256, and 512 were used. Other parameters, batch size and learning rate, have fixed values, 64 and 0.001 respectively. The Figure 8 implies that each hidden units size yielded different performance. In

particular, in Figure 8, the best accuracy 81.5% *when the number of hidden units is 128. Like learning rate, the large number of hidden units does not always guarantee the best performance when the number of hidden units are 256 and 512.*

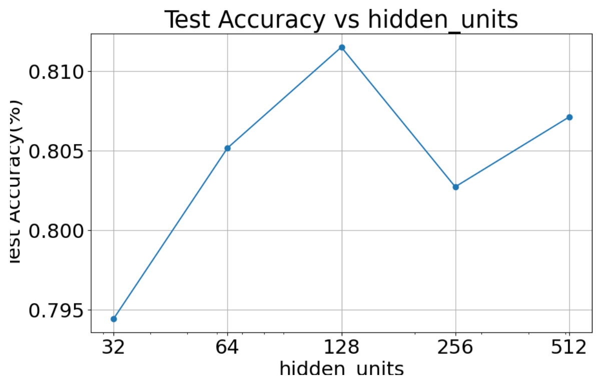


Figure 8: Test Accuracy vs. Number of Hidden Units

6. Discussion about the performance of your neural network.

As shown in the previous figures, the ideal hyperparameters for this problem setup are when $\textit{Batch Size} = 16,\ \textit{Step Size} = 0.001,\ \textit{Number of Hidden Units} = 128$ *with small value of momentum and weight decay, yielding the best accuracy* 82.05%.
In this project, we learned that the number of layers not always guarantees the better performance and that fine-tuning parameters can produce meaningful accuracy by manipulating the batch size, learning rate, and the number of hidden units, and so on.
The smaller batch size of 16 demonstrated the effectiveness of updating model since there are some batches that each training epoch tests on compared to larger batch sizes. It may be deputed when the maximum epochs increases since max epochs can affect the model performance (it may seem 20 epochs are not enough).
The tests across different learning rate indicated that there is a point to optimize neural network model effectively, meaning that learning rate is quite delicate parameter and should not too small or big.
The number of hidden units was impressive. Like learning rate parameter, it also implied that the number of hidden units has a optimal point for model update. It was interesting to see the drop off significantly when the hidden units 256. After that, however, the plot shows increase in accuracy, meaning that there exists an another optimal point.
As a result, it seems that there is an optimal combination of hyperparameters that producing best accuracy, and other conditions such as the maximum number of epoch can affect the model performance.