AI535: Deep Learning

Assignment 2: CIFAR Image Classification using Fully-Connected Network

Name: JuHyun Kim

1) Implementation of Layers

Linear Layer: Stores weights, biases, and their velocities for momentum updates. The forward pass computes XW + b, and the backward pass computes gradients for both inputs and parameters.

```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):
        # He initialization for weights
        self.weights = np.random.randn(input_dim, output_dim) * np.sqrt(2. / input_dim)
        self.bias = np.zeros((1, output_dim))
        self.velocity_weights = np.zeros_like(self.weights)
        self.velocity_bias = np.zeros_like(self.bias)
  # TODO: During the forward pass, we simply compute XW+b
  def forward(self, input):
    self.input = input
    output = np.dot(input, self.weights) + self.bias
    return output
  def backward(self, grad):
    self.grad_weights = np.dot(self.input.T, grad)
    self.grad_bias = np.sum(grad, axis=0, keepdims=True)
    grad_input = np.dot(grad, self.weights.T)
    return grad_input
```

ReLU Layer: On this function, stores the input to determine where to apply gradients during the backward pass, allowing gradients to flow only through the activations that were positive in the forward pass.

```python
class ReLU:
    def forward(self, input):
        self.input = input
        return np.maximum(0, input)

    def backward(self, grad):
        return grad * (self.input > 0).astype(grad.dtype)

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

Sigmoid Cross Entropy Layer: Combines the sigmoid activation with cross-entropy loss for binary classification. It calculates the loss during the forward pass and the gradient with respect to the input during the backward pass.

```python
class SigmoidCrossEntropy:

    def forward(self, logits, labels):
        self.sigmoid_output = 1 / (1 + np.exp(-logits))
        loss = -np.mean(labels * np.log(self.sigmoid_output + 1e-7) + (1 - labels) * np.log(1 - self.sigmoid_output + 1e-7))
        return loss

    def backward(self, logits, labels):
        grad = self.sigmoid_output - labels
        return grad
```

2)Mini-Batch Processing

Forward Pass: The model's forward method is called with the current mini-batch of inputs (X_batch), producing logits (the raw output scores from the model, before applying the sigmoid function).

Compute Loss: The SigmoidCrossEntropy loss between the logits and the actual labels (Y_batch) is computed. This loss quantifies how well the model's predictions match the true labels.

Backward Pass: The gradient of the loss with respect to the logits is computed using the backward method of the loss function. This gradient is then propagated backward through the network (via the net.backward method) to compute gradients for all model parameters.

Optimizer Step: The model parameters are updated in the direction that minimizes the loss, using stochastic gradient descent (SGD) with momentum and weight decay. This step adjusts the model parameters to reduce the loss on the current mini-batch.

Accuracy Calculation: The model's predictions are compared to the true labels to calculate the accuracy for the current mini-batch. The predictions are obtained by thresholding the logits at 0.5, and accuracy is computed as the mean of correct predictions.

Book-keeping: The loss and accuracy for each mini-batch are accumulated to compute the average loss and accuracy for the entire epoch.

```python
# Q2 TODO: For each epoch below max epochs
for epoch in range(max_epochs):
    epoch_avg_loss = 0.0
    epoch_total_acc = 0.0
    # Shuffle the training data
    permutation = np.random.permutation(num_examples)
    X_train_shuffled = X_train[permutation]
    Y_train_shuffled = Y_train[permutation]
    for i in range(0, num_examples, batch_size):
        # Get mini-batch
        X_batch = X_train_shuffled[i:i + batch_size]
        Y_batch = Y_train_shuffled[i:i + batch_size]
        # Forward pass
        logits = net.forward(X_batch)
        # Compute loss
        loss_func = SigmoidCrossEntropy()
        loss = loss_func.forward(logits, Y_batch)
        epoch_avg_loss += loss
        # Backward pass
        grad_loss = loss_func.backward(logits, Y_batch)
        net.backward(grad_loss)
        # Take optimizer step with momentum
        net.step(step_size, momentum, weight_decay)
        # Book-keeping for accuracy
        predictions = (logits >= 0.5).astype(int)
        accuracy = np.mean(predictions == Y_batch)
        epoch_total_acc += accuracy
    # Calculate average loss and accuracy for the epoch
    epoch_avg_loss /= (num_examples // batch_size)
    epoch_avg_acc = epoch_total_acc / (num_examples // batch_size)
    # Evaluate performance on test set after each epoch
    val_loss, val_acc = evaluate(net, X_test, Y_test, batch_size)
    val_accs.append(val_acc)
    # Add loss and accuracy to lists
    losses.append(epoch_avg_loss)
    accs.append(epoch_avg_acc)
    val_losses.append(val_loss)
```

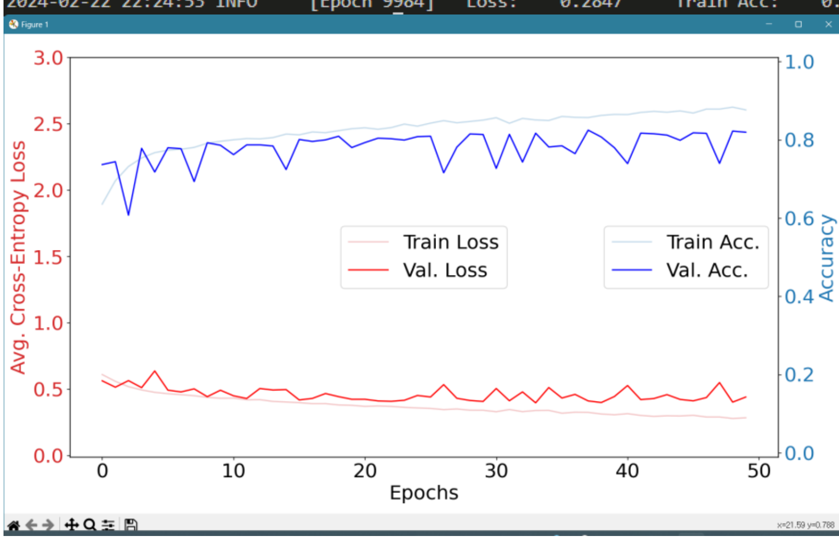
3) Accuracy

Tuned Batch size = 64, learning rate = 0.0001, and hidden layer = 100.

I got 0.248 of Loss, 87.64% of train accuracy and 81.93% of value accuracy at final epoch.

```python
def main():
    # Set optimization parameters
    batch_size = 64              #64,128,256
    max_epochs = 50

    step_size = 0.0001           #learning rate  0.01, 0.001, 0.0001
    number_of_layers = 2
    width_of_layers = 100        #hidden layter 50,100,200
    weight_decay = 0.0001
    momentum = 0.8

    # Load data
    data = pickle.load(open('cifar_2class_py3.p', 'rb'))
    X_train = data['train_data'] / 255.0
    Y_train = data['train_labels']
    X_test = data['test_data'] / 255.0
    Y_test = data['test_labels']
```

```
[Epoch  29]   Loss:   0.3422   Train Acc:   0.8469 Valid Acc:   0.8149
[Epoch  30]   Loss:   0.3415   Train Acc:   0.8504 Valid Acc:   0.8135
[Epoch  31]   Loss:   0.3306   Train Acc:   0.8565 Valid Acc:   0.7271
[Epoch  32]   Loss:   0.3472   Train Acc:   0.8423 Valid Acc:   0.8140
[Epoch  33]   Loss:   0.3311   Train Acc:   0.8547 Valid Acc:   0.7432
[Epoch  34]   Loss:   0.3400   Train Acc:   0.8508 Valid Acc:   0.8169
[Epoch  35]   Loss:   0.3406   Train Acc:   0.8494 Valid Acc:   0.7817
[Epoch  36]   Loss:   0.3194   Train Acc:   0.8597 Valid Acc:   0.7847
[Epoch  37]   Loss:   0.3267   Train Acc:   0.8575 Valid Acc:   0.7646
[Epoch  38]   Loss:   0.3255   Train Acc:   0.8569 Valid Acc:   0.8247
[Epoch  39]   Loss:   0.3134   Train Acc:   0.8624 Valid Acc:   0.8071
[Epoch  40]   Loss:   0.3075   Train Acc:   0.8649 Valid Acc:   0.7798
[Epoch  41]   Loss:   0.3158   Train Acc:   0.8645 Valid Acc:   0.7393
[Epoch  42]   Loss:   0.3027   Train Acc:   0.8700 Valid Acc:   0.8169
[Epoch  43]   Loss:   0.2956   Train Acc:   0.8727 Valid Acc:   0.8154
[Epoch  44]   Loss:   0.2999   Train Acc:   0.8705 Valid Acc:   0.8120
[Epoch  45]   Loss:   0.2985   Train Acc:   0.8738 Valid Acc:   0.7988
[Epoch  46]   Loss:   0.3036   Train Acc:   0.8684 Valid Acc:   0.8179
[Epoch  47]   Loss:   0.2908   Train Acc:   0.8784 Valid Acc:   0.8164
[Epoch  48]   Loss:   0.2909   Train Acc:   0.8784 Valid Acc:   0.7397
[Epoch  49]   Loss:   0.2796   Train Acc:   0.8834 Valid Acc:   0.8223
[Epoch  50]   Loss:   0.2847   Train Acc:   0.8764 Valid Acc:   0.8193
2024-02-22 22:24:53 INFO     [Epoch 9984]   Loss:    0.2847     Train Acc:    0.8764%       Val Acc:     81.93%
```



4) Training Monitoring – evaluate function

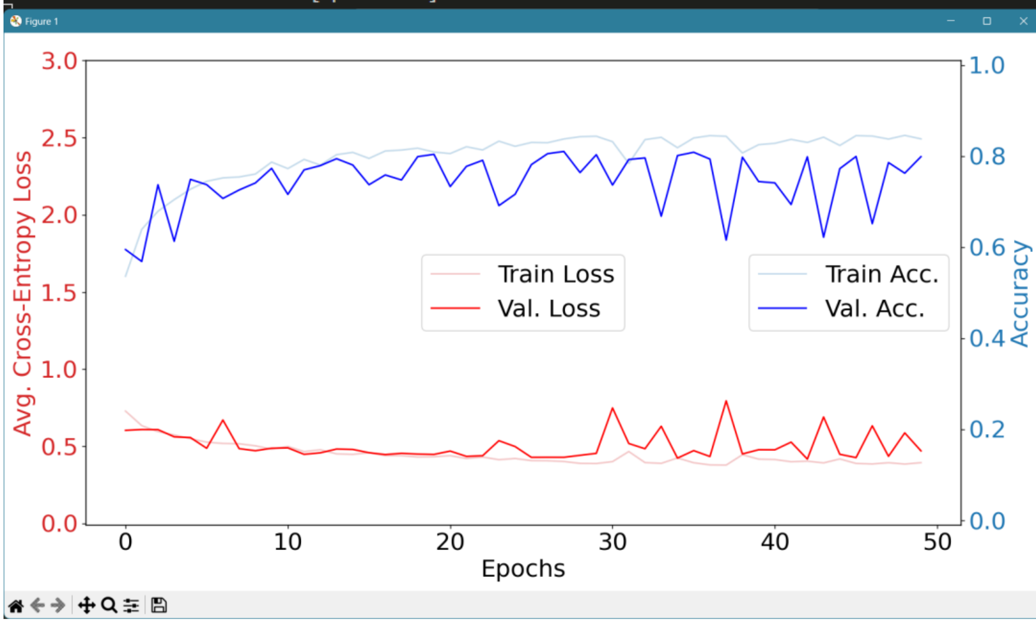
```python
######################################################
# Q4 Implement Evaluation for Monitoring Training
######################################################

# TODO: Given a model, X/Y dataset, and batch size, return the average cross-entropy loss and accuracy over the set
def evaluate(model, X_val, Y_val, batch_size):
    losses = []
    accuracies = []

    num_examples = X_val.shape[0]

    for i in range(0, num_examples, batch_size):
        X_batch = X_val[i:i+batch_size]
        Y_batch = Y_val[i:i+batch_size]

        logits = model.forward(X_batch)

        loss_func = SigmoidCrossEntropy()
        loss = loss_func.forward(logits, Y_batch)
        losses.append(loss)

        predictions = (logits >= 0.5).astype(int)
        accuracy = np.mean(predictions == Y_batch)
        accuracies.append(accuracy)

    avg_loss = np.mean(losses)
    avg_accuracy = np.mean(accuracies)

    return avg_loss, avg_accuracy
```

For each mini-batch, the X_batch and Y_batch are the subset of the input data and the corresponding labels are selected to form the mini-batch. And do loss Calculation which the SigmoidCrossEntropy object is instantiated, and its forward method is called with logits and Y_batch to compute the cross-entropy loss for the mini-batch. This loss is then added to the losses list. The accuracy for the mini-batch is computed as the mean of correct predictions (predictions == Y_batch) and is added to the accuracies list. After iterating through all mini-batches, the function calculates the average loss and accuracy across the entire dataset by computing the mean of the losses and accuracies lists, respectively. Finally the runction returns avg_loss and avg_accuracy.

5) Tuning Parameters
i) different number of batch size

Batch size = 256. Increased the batch size, but there are more loss and the accuracy values got lower.

```
[Epoch  40]   Loss:  0.4169   Train Acc:  0.8250 Valid Acc:  0.7440
[Epoch  41]   Loss:  0.4143   Train Acc:  0.8280 Valid Acc:  0.7411
[Epoch  42]   Loss:  0.4001   Train Acc:  0.8366 Valid Acc:  0.6936
[Epoch  43]   Loss:  0.4039   Train Acc:  0.8299 Valid Acc:  0.7984
[Epoch  44]   Loss:  0.3925   Train Acc:  0.8415 Valid Acc:  0.6218
[Epoch  45]   Loss:  0.4182   Train Acc:  0.8233 Valid Acc:  0.7724
[Epoch  46]   Loss:  0.3889   Train Acc:  0.8449 Valid Acc:  0.7992
[Epoch  47]   Loss:  0.3857   Train Acc:  0.8440 Valid Acc:  0.6516
[Epoch  48]   Loss:  0.3937   Train Acc:  0.8373 Valid Acc:  0.7858
[Epoch  49]   Loss:  0.3850   Train Acc:  0.8455 Valid Acc:  0.7624
[Epoch  50]   Loss:  0.3940   Train Acc:  0.8377 Valid Acc:  0.7987
2024-02-22 22:31:47 INFO     [Epoch 9984]   Loss:     0.394     Train Acc:    0.8377%     Val Acc:     79.87%
```



ii)different learning rate

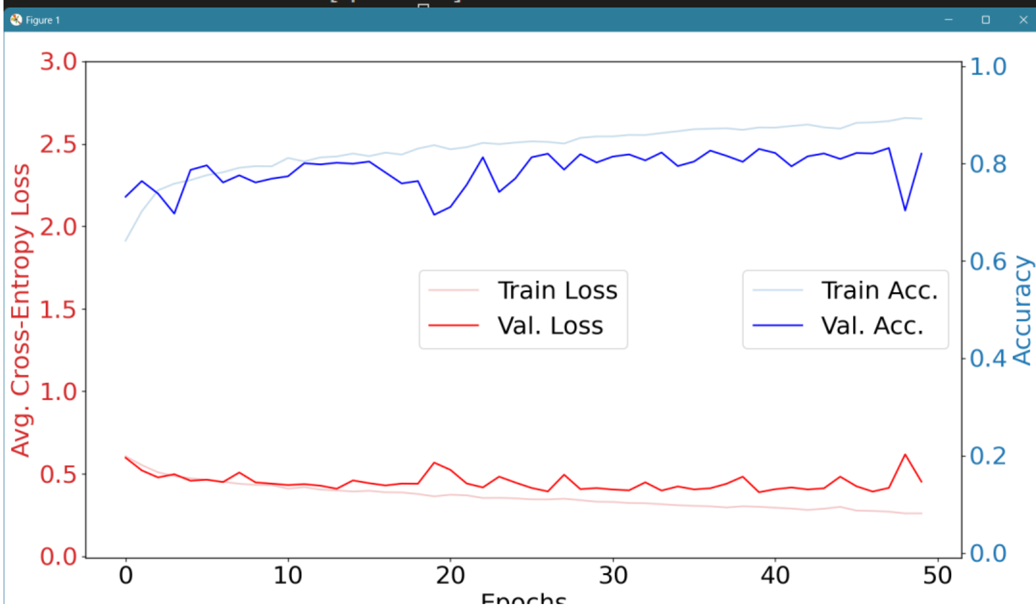
Learning rate = 0.01. Since the learning rate tuned higher, it can cause the model to overshoot the minimum of the loss function, leading to divergent behavior and instability in training.

```
[Epoch  39]   Loss:   0.7139   Train Acc:   0.5173 Valid Acc:   0.5002
[Epoch  40]   Loss:   0.7134   Train Acc:   0.5158 Valid Acc:   0.5002
[Epoch  41]   Loss:   0.7197   Train Acc:   0.5159 Valid Acc:   0.5002
[Epoch  42]   Loss:   0.7120   Train Acc:   0.5158 Valid Acc:   0.5002
[Epoch  43]   Loss:   0.7155   Train Acc:   0.5128 Valid Acc:   0.5002
[Epoch  44]   Loss:   0.7143   Train Acc:   0.5128 Valid Acc:   0.5002
[Epoch  45]   Loss:   0.7136   Train Acc:   0.5068 Valid Acc:   0.5002
[Epoch  46]   Loss:   0.7146   Train Acc:   0.5128 Valid Acc:   0.5002
[Epoch  47]   Loss:   0.7136   Train Acc:   0.5053 Valid Acc:   0.5002
[Epoch  48]   Loss:   0.7155   Train Acc:   0.5143 Valid Acc:   0.5002
[Epoch  49]   Loss:   0.7149   Train Acc:   0.5143 Valid Acc:   0.5002
[Epoch  50]   Loss:   0.7164   Train Acc:   0.5113 Valid Acc:   0.5002
2024-02-22 22:34:08 INFO     [Epoch 9984]   Loss:     0.7164     Train Acc:     0.5113%     Val Acc:     50.02%
```

iii) different number of hidden units

Hidden layers = 200. Adding 100 more hidden layers, therefore the output of training value got higher than when the number of hidden layers is 100. However, there is also has overfitting risk when we increase the data size. But still, this tuned model was the highest achievement I got.

```
[Epoch  39]   Loss:   0.3031   Train Acc:   0.8694 Valid Acc:   0.8037
[Epoch  40]   Loss:   0.3000   Train Acc:   0.8742 Valid Acc:   0.8301
[Epoch  41]   Loss:   0.2943   Train Acc:   0.8739 Valid Acc:   0.8218
[Epoch  42]   Loss:   0.2892   Train Acc:   0.8772 Valid Acc:   0.7944
[Epoch  43]   Loss:   0.2808   Train Acc:   0.8802 Valid Acc:   0.8149
[Epoch  44]   Loss:   0.2884   Train Acc:   0.8744 Valid Acc:   0.8208
[Epoch  45]   Loss:   0.2993   Train Acc:   0.8719 Valid Acc:   0.8096
[Epoch  46]   Loss:   0.2772   Train Acc:   0.8837 Valid Acc:   0.8218
[Epoch  47]   Loss:   0.2748   Train Acc:   0.8847 Valid Acc:   0.8208
[Epoch  48]   Loss:   0.2705   Train Acc:   0.8873 Valid Acc:   0.8320
[Epoch  49]   Loss:   0.2600   Train Acc:   0.8938 Valid Acc:   0.7036
[Epoch  50]   Loss:   0.2599   Train Acc:   0.8922 Valid Acc:   0.8203
2024-02-22 22:39:20 INFO     [Epoch 9984]   Loss:    0.2599    Train Acc:    0.8922%      Val Acc:     82.03%
```



6) Discussion

More things to improve, even if I got enough output, the overall performance could be significantly improved by transitioning to architectures more suited for image data, and by incorporating advanced training, regularization, and evaluation techniques. On the implementation of SGD with momentum, the performance of the network could be further enhanced by adopting more sophisticated optimization algorithms like Adam or RMSprop, which adapt the learning rates for each parameter, offering a more nuanced control over the training process. And the evaluation metrics implemented provide a basic understanding of the model's performance. However, relying solely on accuracy and cross-entropy loss may not always offer a complete picture, especially in cases where the dataset might be imbalanced.