CS 189:289A Introduction to Machine Learning
Fall 2021 Jennifer Listgarten and Jitendra Malik **HW2**

Due 10/04/21 at 11:59pm

- Homework 2 consists entirely of coding questions.
- We prefer that you typeset your answers using LaTeX or other word processing software. If you haven't yet learned LaTeX, one of the crown jewels of computer science, now is a good time! Neatly handwritten and scanned solutions will also be accepted.

Deliverables:

1. Submit a PDF of your homework to the Gradescope assignment entitled "HW2 Write-Up". **Please start each question on a new page.** If there are graphs, include those graphs in the correct sections. **Do not** put them in an appendix. We need each solution to be self-contained on pages of its own.
   - In your write-up, please state with whom you worked on the homework. This should be on its own page and should be the first page that you submit.
   - In your write-up, please copy the following statement and sign your signature next to it. (Mac Preview and FoxIt PDF Reader, among others, have tools to let you sign a PDF file.) We want to make it *extra* clear so that no one inadvertently cheats. *"I certify that all solutions are entirely in my own words and that I have not looked at another student's solutions. I have given credit to all external sources I consulted."*
   - **Replicate all your code in an appendix**.
   - **See the end of the assignment for the deliverables for each part!**

1 Implementing A Neural Network in NumPy

In this assignment, we will be implementing a neural network and the backpropagation algorithm in NumPy. Later on in this course, we will introduce you to PyTorch, which is extremely useful for quickly and correctly building neural networks. However, libraries such as PyTorch also hide many of the key algorithmic details behind implementing and training neural networks. To really understand neural networks and the backpropagation algorithm, it is very helpful to implement everything yourself. After this homework assignment, we hope that you gain this deeper understanding!

Our goal will be to train a neural network to classify a handwritten digit as either a 3 or a 9. We will be using a subset of the famous MNIST dataset. The neural network will be trained using mini-batch stochastic gradient descent (SGD). Recall that in standard SGD, at each timestep t, we compute the loss for a *single* data point and update the weights based on the gradient of the loss. In mini-batch SGD, we instead compute an average loss over a randomly sampled *batch* of n data points at each timestep. Note that your code should be general enough to handle SGD (by setting $n = 1$) as well as traditional gradient descent (by setting n equal to the total number of data points).

1.1 Getting started

Install the MNIST Python library by running `pip install mnist` in your terminal. The `get_mnist_threes_nines` function in `hw2.py` provides the training and test splits that we will use in this assignment. Familiarize yourself with the data and look at a few examples using the `display_image` function. Check for yourself that you can answer the following questions: What are the labels for an image of a 3? For a 9? What are the the dimensions of a single image? If we want to treat each image as a vector how could we do that and how many features would that image have?

Neural networks will take longer than other assignments to train. Therefore, it is necessary that you *vectorize* your computations in order to take advantage of the highly optimized operations provided to you by NumPy. Vectorize your code when possible to save you hours of time. A simple rule-of-thumb is to avoid explicit for-loops whenever possible.

A particularly useful function that you should learn is `numpy.einsum`. The syntax is tricky to learn at first and takes a while to get used to. However, many machine learning practitioners find this function incredibly useful, and for this assignment, you will too.

1.2 Questions

(a) Backpropagation is quite tricky to implement correctly. You will verify that the gradients computed by your backpropagation code are correct by comparing its output to gradients computed by finite differences. Let $f : \mathbb{R}^d \to \mathbb{R}$ be an arbitrary function (for example, the loss of a neural network with d weights and biases), and let a be a d-dimensional vector. You can approximately evaluate the partial derivatives of f as:

$$\frac{\partial f}{\partial x_k}(a) \approx \frac{f(\ldots, a_k + \epsilon, \ldots) - f(\ldots, a_k - \epsilon, \ldots)}{2\epsilon}$$

where ϵ is a small constant. As ϵ approaches 0 , this approximation theoretically becomes exact, but becomes prone to numerical precision issues. $\epsilon = 10^{-5}$ usually works well. Implement a finite differences checker, which you will use later to verify the correctness of your backpropagation implementation.

(b) Here, we will implement several helper functions that we will combine in the next two parts.

To perform binary classification, we will have the last layer of our neural network consist of a single neuron with the sigmoid activation function, $\sigma(s) = \frac{1}{1+e^{-s}}$. Implement a function `sigmoid_activation` which takes in a NumPy array of arbitrary shape and applies the sigmoid function elementwise. The function should return both the output and the gradient of the activation function with respect to the input (as this will be needed for the backward pass).

To vectorize the numerically stable version of this function (see the deliverables below), it may help to use the `numpy.where` function in combination with the optional `where` and `out` arguments inside of `numpy.exp`. Also note that floating point limitations may cause probabilities equal to 0 for very negative inputs, which will cause errors in the next part. To correct this, we can clip the output of the sigmoid function to be between $(\epsilon, 1 - \epsilon)$ where ϵ is a small number such as 10^{-15}. Apply this fix to your implementation of `sigmoid_activation` before proceeding.

Now, implement a function `logistic_loss(g, y)` where g is the output of the neural network and y is an array of the true labels. The function should return two arrays: `loss` and `dL_dg` where `loss[i]` is the negative of the log-likelihood defined in lecture for binary classification, i.e. $l(g[i], y[i]) = -\log\{g[i])^{y[i]}(1 - g[i])^{1-y[i]})\}$ and `dL_dg[i]` is the derivative of `loss[i]` with respect to `g[i]`.

Note: be careful about NumPy broadcasting rules. It may help to include assertions in your code about the dimensions of certain inputs and outputs to prevent unintended silent broadcasting from introducing bugs. See the NumPy documentation for more information.

For the intermediate layers, we will utilize the ReLU activation function, $\sigma(x) = \max\{0, x\}$. Write a function `relu_activation` which takes in an nxd array, s, and returns both the elementwise activation to s and the partial derivatives of the activation function with respect to the elements of s.

Finally, implement a function `layer_forward(x, W, b, activation_fn)` where `x` is the nx$d^{(l-1)}$ dimensional matrix consisting of the mini-batch neurons for layer $l - 1$, `W` is the $d^{(l-1)}$x$d^{(l)}$ weight matrix for layer l, `b` is 1x$d^{(l)}$ dimensional bias vector for layer l, and `activation_fn` is a function that applies the elementwise activation to $S^{(l)}$ (i.e. this should be either `relu_activation` or `sigmoid_activation`). The function should return two things, `out` and `cache`. `out` should be the nx$d^{(l)}$ dimensional vector corresponding the mini-batch neurons for layer l and `cache` is a tuple consisting of any information that we need to compute $\frac{\partial l(W,b)}{\partial W_{ij}}$ and $\frac{\partial l(W,b)}{\partial b_j}$ on the backward pass. By looking at the induction step that we derived in lecture, you should figure out what exactly `cache` needs to contain.

Solution: A naïve version of `sigmoid_activation` is:

```
def sigmoid_activation(s):
    out = 1 / (1 + np.exp(-s))
    grad = out * (1 - out)
    return out, grad

s = np.asarray([1., 0., -1])
out, grad = sigmoid_activation(s)
print(out)
print(grad)
[0.73105858 0.5        0.26894142]
[0.19661193 0.25       0.19661193]
```

The overflow is caused when the exponential function is given a very large positive input, i.e., a very negative s. We can fix this by applying the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$ for positive inputs and $\sigma(x) = \frac{e^x}{1+e^x}$ for negative inputs.

```
def sigmoid_activation(x):
    mask = (x >= 0)
    pos_sig = 1 / (1 + np.exp(-x, where=mask, out=np.ones_like(x)))
    neg_sig = np.exp(x, where=~mask, out=np.ones_like(x)) / \
        (1 + np.exp(x, where=~mask, out=np.ones_like(x)))
    out = np.where(mask,
                   pos_sig,
                   neg_sig)
    grad = out * (1 - out)
    return out, grad


s = np.asarray([-1000, 1000])
out, grad = sigmoid_activation(s)
print(out)
print(grad)
[0. 1.]
[0. 0.]
```

For logistic_loss, the function and derivative can be written as:

$$\begin{aligned} l(g, y) &= -\log\{g^y(1-g^{1-y})\} \\ &= -\Big[y \log g + (1-y)\log(1-g)\Big] \end{aligned}$$

$$\frac{\partial l(g, y)}{\partial g} = -\Big[\frac{y}{g} + \frac{(1-y)}{(1-g)}(-1)\Big]$$

$$= -\left[\frac{y}{g} + \frac{(y-1)}{(1-g)}\right]$$

The code is:

```
def sigmoid_activation(x):
    mask = (x >= 0)
    pos_sig = 1 / (1 + np.exp(-x, where=mask, out=np.ones_like(x)))
    neg_sig = np.exp(x, where=~mask, out=np.ones_like(x)) / \
        (1 + np.exp(x, where=~mask, out=np.ones_like(x)))
    out = np.where(mask,
                   pos_sig,
                   neg_sig)
    eps = 1e-15
    out = np.clip(out, eps, 1. - eps)
    grad = out * (1 - out)
    return out, grad


def logistic_loss(g, y):
    """
    Computes the loss and gradient for binary classification with logistic
    loss

    Inputs:
    - g: Output of final layer with sigmoid activation,
         of shape (n, 1)

    - y: Vector of labels, of shape (n,) where y[i] is the label for x[i]
         and y[i] in {0, 1}

    Returns a tuple of:
    - loss: array of losses
    - dL_dg: Gradient of the loss with respect to g
    """
    assert(g.ndim == 2 and g.shape[1] == 1)
    assert(y.ndim == 1)
    g = g.reshape(-1)
    loss = -(y * np.log(g) + (1 - y) * np.log(1 - g))
    dL_dg = -((y / g) + (y - 1)/(1 - g))
    return loss, dL_dg
```

The `relu_activation` function is given by:

```
def relu_activation(s):
    mask = s <= 0
    out = np.where(mask, 0, s)
    ds = np.where(mask, 0.0, 1.0)
    return out, ds
```

Finally, layer_forward is given by:

```
def layer_forward(x, W, b, activation_fn):
    assert(x.ndim == 2)
    assert(W.ndim == 2 and W.shape[0] == x.shape[1])
    assert(b.ndim == 2 and b.shape[0] == 1 and b.shape[1] == W.shape[1])
    h = x @ W + b
    out, ds = activation_fn(h)
    cache = (x, W, b, ds)
    return out, cache
```

(c) We are now ready to fully implement a forward pass of the neural network! We will specify the neural network by a list of the layer dimensions. For example, to implement a two layer neural network where the first layers maps the 784-dimensional image to 200 dimensional space and the second layer maps to a 1 dimensional output, we would use `layer_dims = [784, 200, 1]`. Similarly, we must specify the activation functions used at each layer, e.g., `activations = [relu_activation, sigmoid_activation]`. Write helper functions `create_weight_matrices(layer_dims)` and `create_bias_vectors(layer_dims)` that return a list of weight matrices and bias vectors, respectively, for each layer specified in `layer_dims`. You should initialize the weights by sampling each weight from a normal distribution with mean 0 and standard deviation 0.01. Finally, write a function `forward_pass(X_batch, weight_matrices, biases, activations)` which takes in an nx784 dimensional mini-batch of flattened images, a list of weight matrices, a list of bias vectors, and a list of activation functions, and returns a vector of outputs and a list of layer caches (defined previously).

You should now be able to compute the predictions, loss and the gradient with respect to the output for a mini-batch as follows:

```
# run forward pass
output, layer_caches = forward_pass(X_batch,
                                    weight_matrices,
                                    biases,
                                    activations)
# compute loss and derivative with respect to logits
loss, dL_dg = logistic_loss(output, y_batch)
print("Average Loss Across Batch", loss.mean())
```

Solution:

```
def create_weight_matrices(layer_dims):
    """
    Creates a list of weight matrices defining the weights of NN

    Inputs:
    - layer_dims: A list whose size is the number of layers. layer_dims[i] defin
      the number of neurons in the i+1 layer.

    Returns a list of weight matrices
    """
    weights = []
    for i in range(len(layer_dims) - 1):
        nrow = layer_dims[i]
        ncol = layer_dims[i+1]
        W = np.random.randn(nrow, ncol) * 0.01
        weights.append(W)
    return weights


def create_bias_vectors(layer_dims):
    """
    Creates a list of weight matrices defining the weights of NN

    Inputs:
    - layer_dims: A list whose size is the number of layers. layer_dims[i] defin
      the number of neurons in the i+1 layer.

    Returns a list of weight matrices
    """
    biases = [np.random.randn(1, h)* 0.01 for h in layer_dims[1:]]
    return biases

def forward_pass(X_batch, weight_matrices, biases, activations):
    assert(len(weight_matrices) == len(biases))
    assert(len(weight_matrices) == len(activations))
    layer_caches = []
    h = X_batch
    for layer in range(len(weight_matrices)):
        W = weight_matrices[layer]
        b = biases[layer]
        activation_fn = activations[layer]
        h, layer_cache = layer_forward(h, W, b, activation_fn)
        layer_caches.append(layer_cache)
    output = h
    return output, layer_caches
```

(d) We are now at the most difficult part of this problem: implementing the backward pass. Write a function called `backward_pass` that uses the derivative computed in `logistic_loss` and the layer caches to compute the gradients of each of the weight matrices and bias vectors. Implementing this correctly will require both a strong understanding of the backpropagation algorithm, as discussed in lecture, and careful coding/debugging to make sure each step is implemented appropriately. Your finite differences implementation will come in handy here to make sure that your backpropagation implementation is working correctly.

Solution:

```
def backward_pass(dL_dg, layer_caches):

    grad_Ws = []
    grad_bs = []

    ##########################################################
    ################### Base Case ############################
    ##########################################################
    layer_caches_reversed = layer_caches[::-1]
    x_l_min_1, W_l, b_l, g_prime = layer_caches_reversed[0]
    delta_l = dL_dg.reshape(-1, 1) * g_prime

    grad_W = np.einsum("ni,nj->nij", x_l_min_1, delta_l).mean(axis=0)
    grad_b = delta_l.mean(axis=0)

    grad_Ws.insert(0, grad_W)
    grad_bs.insert(0, grad_b)

    W_dot_delta = np.einsum("ij,nj->ni", W_l, delta_l)

    ##########################################################
    ################### Inductive Step ######################
    ##########################################################
    for layer in layer_caches_reversed[1:]:
        x_l_min_1, W_l, b_l, g_prime = layer
        delta_l = g_prime * W_dot_delta

        grad_W = np.einsum("ni,nj->nij", x_l_min_1, delta_l).mean(axis=0)
        grad_b = delta_l.mean(axis=0)

        grad_Ws.insert(0, grad_W)
        grad_bs.insert(0, grad_b)

        W_dot_delta = np.einsum("ij,nj->ni", W_l, delta_l)
    return grad_Ws, grad_bs
```

(e) Now it is time to bring it all together. Using a mini-batch size n of 100 and a step size of 0.1 for SGD, implement a training loop that iterates over the mini-batches of data, computes the forward pass, computes the loss, computes the backward pass, and updates the weights using SGD. Note that typically, instead of randomly sampling each mini-batch, the training data is shuffled and then divided up into mini-batches for efficiency. Each loop over the entire dataset is commonly referred to as an *epoch*, and the training data is reshuffled for each epoch. Train a two layer network with a hidden dimension of 200 (i.e. `layer_dims = [784, 200, 1]`) for 5 epochs. For each timestep, record the average loss across the training mini-batch, the average accuracy across the training mini-batch, the average loss across the entire test set, and the average accuracy across the entire test set. You should be able to achieve a test accuracy of around 98%.

Solution: When we increase the step size to 10.0, the loss no longer consistently goes down and the accuracy remains close to 50%. Simply put, the step size is too high to converge. Examining the images that we got wrong, we notice that many of the images seem to be tricky. Some are drawn oddly and appear different than a conventional 3 or a 9. Some of the 3's look a bit like 9's and vice versa.

```
layer_dims = [28*28, 200, 1]
activations = [relu_activation, sigmoid_activation]
weight_matrices = create_weight_matrices(layer_dims)
biases = create_bias_vectors(layer_dims)
bs = 100
step_size = 0.1
training_stats = []
for epoch in range(5):
    idxs = np.arange(X_train.shape[0])
    idxs = np.random.permutation(idxs)
    for batch in range(X_train.shape[0] // bs):
        batch_idxs = idxs[batch * bs : (batch + 1) * bs]
        X_batch = X_train[batch_idxs]
        X_batch = X_batch.reshape(X_batch.shape[0], -1)
        y_batch = y_train[batch_idxs]

        output, layer_caches = forward_pass(X_batch, weight_matrices, biases, ac
        loss, dL_dg = logistic_loss(output, y_batch)
        grad_Ws, grad_bs = backward_pass(dL_dg, layer_caches)
        #######################################################
        ################ Update Gradients ####################
        #######################################################
        for weight_matrix, bias, grad_W, grad_b in zip(weight_matrices, biases,
            weight_matrix -= (step_size * grad_W)
            bias -= (step_size * grad_b)

        y_hat = (output > 0.5).astype(np.int).reshape(-1)

        accuracy = (y_hat == y_batch).sum() / y_batch.shape[0]

        avg_train_loss = loss.mean()
        avg_train_acc = accuracy


        #######################################################
        ############### Test Loss ############################
        #######################################################
        output, _ = forward_pass(X_test.reshape(X_test.shape[0], -1), weight_mat
        loss, ds = logistic_loss(output, y_test)

        y_hat = (output > 0.5).astype(np.int).reshape(-1)
        accuracy = (y_hat == y_test).sum() / y_test.shape[0]

        avg_test_loss = loss.mean()
        avg_test_acc = accuracy

        training_stats.append([avg_train_loss, avg_train_acc, avg_test_loss, avg

        print(f"Test Loss: {loss.mean():.3f}")
        print(f"Test Accuracy: {accuracy:.3f}")

import pandas as pd
df = pd.DataFrame(training_stats, columns = ["Train_Loss", "Train_Acc", "Test_Lo
fig, ax = plt.subplots()
ax.plot(df.Train_Loss, label="Training Loss")
ax.plot(df.Test_Loss, label="Test Loss")
ax.set(ylim=(0, 5.))
ax.legend()
fig.tight_layout()
plt.savefig("train_test_losses_lr_01.png", dpi=300)

fig, ax = plt.subplots()
```

```
ax.plot(df.Train_Acc, label="Training Accuracy")
ax.plot(df.Test_Acc, label="Test Accuracy")
ax.legend()
fig.tight_layout()
plt.savefig("train_test_accuracy_lr_01.png", dpi=300)
```

1.3 Deliverables

We have provided inputs, weight matrices, and biases for a neural network with input layer of dimension 4, hidden layer of dimension 2 and output layer of dimension 1. The activation functions are again ReLU and sigmoid.

The test input and parameters can be loaded using the `pickle` module as follows:

```
with open("test_batch_weights_biases.pkl", "rb") as fn:
    (X_batch, y_batch, weight_matrices, biases) = pickle.load(fn)
```

(a) Compute the finite difference approximation to the gradients of the loss with respect to the weights and biases of the neural network. The code to produce the result may look like:

```
with open("test_batch_weights_biases.pkl", "rb") as fn:
    (X_batch, y_batch, weight_matrices, biases) = pickle.load(fn)

grad_Ws, grad_bs = my_nn_finite_difference_checker(X_batch,
                                                   y_batch,
                                                   weight_matrices,
                                                   biases,
                                                   activations)


with np.printoptions(precision=2):
    print(grad_Ws[0])
    print()
    print(grad_Ws[1])
    print()
    print(grad_bs[0])
    print()
    print(grad_bs[1])
```

Your answer should be the result of the printout of the gradients.

Solution:

```
[[ 4.12e-04  3.34e-05]
 [-6.56e-05  2.07e-05]
 [-1.24e-03  8.02e-05]
 [ 2.26e-04 -1.41e-06]]
```

```
[[-0.  ]
 [-0.01]]

[[-0.  0.]]

[[-0.5]]
```

(b) Answer the following questions:

(i) What does `sigmoid_activation` return for `s = np.asarray([1., 0., -1.])`?

Solution:

```
################################################
############ Naive Sigmoid #####################
################################################

def sigmoid_activation(s):
    out = 1 / (1 + np.exp(-s))
    grad = out * (1 - out)
    return out, grad

s = np.asarray([1., 0., -1])
out, grad = sigmoid_activation(s)
with np.printoptions(precision=2):
    print(out)
    print(grad)
[0.73 0.5  0.27]
[0.2  0.25 0.2 ]
```

(ii) What does `sigmoid_activation` return for `s = np.asarray([-1000, 1000])`? If you observe an overflow warning from NumPy: can you figure which input, -1000 or 1000, is causing it? Ensure that your implementation of `sigmoid_activation` is numerically stable and does not cause overflows, and explain how you achieved this. *Hint:* Consider the equivalent definitions of the sigmoid function: $\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x}$. Can we use both of these definitions to avoid overflow errors?

Solution: The issue arises because very large values of x will vause e^x to overflow and very negative values of x will cause e^{-x} to underflow. We can fix this problem by applying the implementation $\sigma(x) = \frac{1}{1+e^{-x}}$ when $x \geq 0$ and applying $\frac{e^x}{1+e^x}$ when $x < 0$.

```
def sigmoid_activation(s):
    mask = (s >= 0)
    out = np.where(mask,
                   1 / (1 + np.exp(-s, where=mask)),
                   np.exp(s, where=~mask) / (1 + np.exp(s, where=~mask)))
    grad = out * (1 - out)
    return out, grad
```

```
s = np.asarray([-1000, 1000])
out, grad = sigmoid_activation(s)
with np.printoptions(precision=2):
    print(out)
    print(grad)
[0. 1.]
[0. 0.]
```

(iii) What is the derivative of the negative log-likelihood loss with respect to g?

Solution: For `logistic_loss`, the function and derivative can be written as:

$$\begin{aligned} l(g, y) &= -\log\{g^y(1 - g^{1-y})\} \\ &= -\Big[y \log g + (1 - y)\log(1 - g)\Big] \end{aligned}$$

$$\begin{aligned} \frac{\partial l(g, y)}{\partial g} &= -\Big[\frac{y}{g} + \frac{(1 - y)}{(1 - g)}(-1)\Big] \\ &= -\Big[\frac{y}{g} + \frac{(y - 1)}{(1 - g)}\Big] \end{aligned}$$

(iv) Explain what is returned in `cache` in your `layer_forward` implementation.

Solution: The cache should return anything that is needed to compute the gradients during the backward pass. In this implementation we stored the inputs, x, weights, biases, and gradients of activation function with respect to the input. It is okay if your implementation did things differently, so long as it contains everything that is needed for the backward pass.

(c) Report the average loss for the test data, weight, and biases provided. The code needed to produce the response may look like:

```
with open("test_batch_weights_biases.pkl", "rb") as fn:
    (X_batch, y_batch, weight_matrices, biases) = pickle.load(fn)

activations = [relu_activation, sigmoid_activation]
output, _ = forward_pass(X_batch, weight_matrices, biases,
                         activations)
loss, dL_dg = logistic_loss(output, y_batch)
print(loss.mean())
```

Solution: The answer should be roughly 0.6985

```
with open("test_batch_weights_biases.pkl", "rb") as fn:
    (X_batch, y_batch, weight_matrices, biases) = pickle.load(fn)
activations = [relu_activation, sigmoid_activation]
output, _ = forward_pass(X_batch, weight_matrices, biases, activations)
```

```
loss, dL_dg = logistic_loss(output, y_batch)
print(f"{loss.mean():.4f}")
0.6985
```

(d) Report the gradients of the weights and biases computed from the test data using back-propagation. Compare these gradients to the finite differences approximation and make sure the two gradients are close. The code to produce the response might look like:

```
with open("test_batch_weights_biases.pkl", "rb") as fn:
    (X_batch, y_batch, weight_matrices, biases) = pickle.load(fn)

activations = [relu_activation, sigmoid_activation]
output, layer_caches = forward_pass(X_batch, weight_matrices, biases,
                                    activations)
loss, dL_dg = logistic_loss(output, y_batch)
grad_Ws, grad_bs = backward_pass(dL_dg, layer_caches)

with np.printoptions(precision=2):
    print(grad_Ws[0])
    print()
    print(grad_Ws[1])
    print()
    print(grad_bs[0])
    print()
    print(grad_bs[1])
```

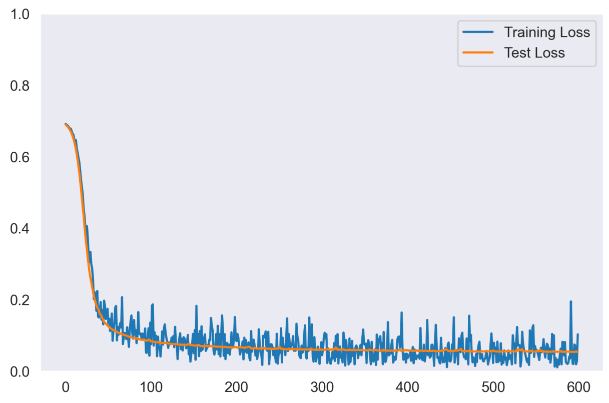
Solution:

```
with open("test_batch_weights_biases.pkl", "rb") as fn:
    (X_batch, y_batch, weight_matrices, biases) = pickle.load(fn)
activations = [relu_activation, sigmoid_activation]
output, layer_caches = forward_pass(X_batch, weight_matrices, biases, activat
loss, dL_dg = logistic_loss(output, y_batch)
grad_Ws, grad_bs = backward_pass(dL_dg, layer_caches)

with np.printoptions(precision=2):
    print(grad_Ws[0])
    print()
    print(grad_Ws[1])
    print()
    print(grad_bs[0])
    print()
    print(grad_bs[1])
[[ 4.12e-04  3.34e-05]
 [-6.56e-05  2.07e-05]
 [-1.24e-03  8.02e-05]
 [ 2.26e-04 -1.41e-06]]

[[-0.  ]
 [-0.01]]

[[-0.  0.]]

[[-0.5]]
```

(e) Answer the following questions:

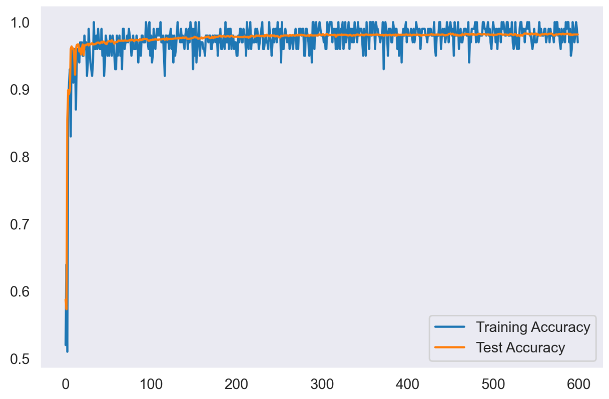
(i) Make a plot of the training and test losses per timestep.

Solution:



(ii) Make another plot of the training and test accuracies per timestep.

Solution:



(iii) Examine the images that your network guesses incorrectly, and explain at a high level what patterns you see in those images.

Solution: Examining the images that we got wrong, we notice that many of the images seem to be tricky. Some are drawn oddly and appear different than a conventional 3 or a 9. Some of the 3's look a bit like 9's and vice versa.

(iv) Rerun the neural network training but now increase the step size to 10.0. What happens? You do not need to include plots here.

Solution: When we increase the step size to 10.0, the loss no longer consistently goes down and the accuracy remains close to 50%. Simply put, the step size is too high to converge.