Final Project - Statistical Machine Learning

Francine Uwera

Ann Shehani Fernando Warnakulasuriya

Anthony Mark Wallace

April 2024

1 Section I : Handwritten Digit Recognition with Neural Network

1.1 Introduction

We trained a a single layer neural network and a multi layer neural network to classify handwritten digit images in MNIST dataset with 10 different labels from 0 to 9.

1.2 Data Description

The MNIST dataset consists images of handwritten digits from 0 to 9, where each image is represented as a 28x28 pixel grid. The training set contains 60000 images and the test set 10000 images. We scaled the data to a range of [0,1] by dividing it by 255.0.

1.3 Part I

A random sample of 10 images for each digit is displayed in Figure 1.

Figure 1: A random selection of 10 samples for each digit from the dataset, displayed using the *show_digit* function.

1.4 Single Layer Neural Network

1.4.1 Part II

We implement a function that computes a neural network as shown in Figure 2. The O_i is calculated as weighted sum of inputs x_j with weights $w_j i$ plus a bias term bi and the softmax function $S(x)$ is applied to the outputs to get the probability distribution over the 10 class labels. The neural network is computed as follows : $O_i = \sum_j w_{ji} x_j + b_i$. And this is the softmax function applied to the outputs: $S(x)_{ij} = \frac{e^{x_{ij}}}{\sum_k e^{x_{ik}}}$

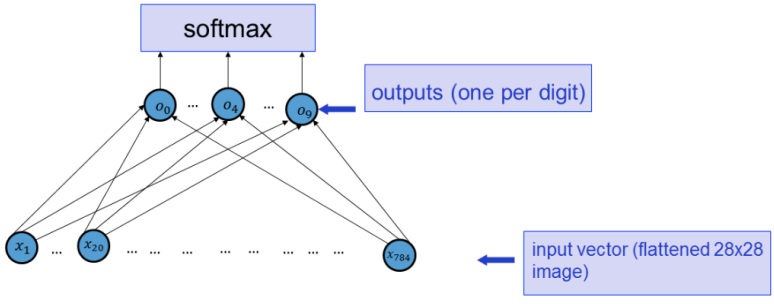

Figure 2: Single Layer Neural Network

The implementation of the neural network in R is as follow :

Softmax Function

```
softmax <- function(x) {
  S_exp <- rowSums(exp(x))
  S_exp <- replicate(ncol(x), S_exp)
  return((exp(x) / S_exp))
}
```

NN Function

```
Layer_1 <- function(x, W, b) {
  # x is the Train dataset with only predictors with dimensions n*784
  # W is the weight matrix of dimension 784 * 10
  # b is the beta0 weights with dimensions 10*1

  O_i <- (as.matrix(x) %*% W)
  biases <- replicate(nrow(O_i), b)
  O_i <- O_i + t(biases)
  Y <- softmax(O_i)

  return(Y)
}
```

1.4.2 Part III

The cross entropy loss is used as the cost function for the neural network. It is the negative log-probabilities of the correct label under the predicted probability distribution over training cases. The Cross-Entropy for case i : $H(y_i, y_i') = -\log(\mathbf{y}_i \cdot \mathbf{y'}_i)$ where y_i is the true label and y_i' is the predicted label. We then implemented a function for the gradient of the cost function above. The gradient of the cross entropy loss with respect to weights and biases is computed to find the parameters (W and b) that minimizes the loss function. The function starts by computing the predicted outputs for the given inputs using the neural network function defined above. Then it computes the cross entropy loss using the predictions and true labels. Finally, it calculates the gradients of the cost with respect to the parameters.

The implementation of the cost function and its gradient in R is as follow :

Cost Function

```
# cost function
  #y is the predicted matrix
  #y_ is the true values which is a one_hot_matrix
cost <- function(y, y_) {

  NLL <- matrix(0, nrow(y), 1)

  for (i in 1:nrow(y)) {
    NLL[i] <- (-log((y[i,]) %*% y_[i,]))
  }

  return(NLL)
}
```

Gradient Function

```
compute_cost_and_gradient <- function(x, y, W, b) {
  # x is the Train dataset with only predictors with dimensions n*784
  # y is the True value (number) in dimension n*1
  # W is the weight matrix of dimension 784 * 10
  # b is the beta0 weights with dimensions 10*1

  # One_hot_matrix
  # Forward pass
  Y_hat <- Layer_1(x, W, b)

  Y_H <- matrix(0, nrow = nrow(y), ncol = ncol(Y_hat))

  for (i in 1:nrow(y)) {
    Y_H[i, as.numeric(y[i]) + 1] <- 1
  }

  # Y_hat is matrix of dimension n*10

  # Negative log-likelihood
  nll <- cost(Y_hat, Y_H)
```

```
  # Backward pass for gradients
  d_logits <- (Y_hat - Y_H)
  grad_W <- t(x) %*% d_logits
  grad_b <- colSums(d_logits)

  return(list(nll = nll, grad_W = grad_W, grad_b = grad_b))
}
```

1.4.3 Part IV

We use a finite-difference approximation to compute the gradient for several coordinates of W and b. The finite-difference approximation is based of the principle that the gradient of a function can be computed numerically. The numerical approximation follows directly from the definition of derivative : $\frac{dE}{dw}\big|_w \approx \frac{E(w+h)-E(w)}{h}$ for a small stepsize h. This should yield an approximate estimate of the gradient. This method helps verify the accuracy of our previously implemented gradient computation algorithm.

Here is how the function was implemented in R :

```
# Finite difference approximation function
compute_finite_diff <- function(W, b, x, y_, h = 1e-5) {
  # Perturbation vector
  perturb <- matrix(0, nrow=nrow(W), ncol=ncol(W))
  grad_approx <- matrix(0, nrow=nrow(W), ncol=ncol(W))

  Y_H <- matrix(0, nrow = nrow(y_), ncol = ncol(W))

  for (i in 1:nrow(y_)) {
    Y_H[i, as.numeric(y_[i]) + 1] <- 1
  }

  for (i in 1:nrow(W)) {
    for (j in 1:ncol(W)) {
      # Perturb parameter
      perturb[i, j] <- h
      W_plus <- W + perturb
      W_minus <- W - perturb

      # Forward pass with perturbed weights
      result_plus <- Layer_1(x, W_plus, b)
      cost_plus <- cost(Y_H, result_plus)
```

```
      result_minus <- Layer_1(x, W_minus, b)
      cost_minus <- cost(Y_H, result_minus)

      # Compute gradient approximation
      grad_approx[i, j] <- ((cost_plus - cost_minus) / (2 * h)) %>% sum()

      # Reset perturbation vector
      perturb[i, j] <- 0
    }
  }

  perturb_b <- rep(0, length(b))
  grad_approx_b <- rep(0, length(b))

  for (k in 1:length(b)) {
    perturb_b[k] <- h
    b_plus <- b + perturb_b
    b_minus <- b - perturb_b

    # Forward pass with perturbed biases
    result_plus_b <- Layer_1(x, W, b_plus)
    cost_plus_b <- cost(Y_H, result_plus_b)

    result_minus_b <- Layer_1(x, W, b_minus)
    cost_minus_b <- cost(Y_H, result_minus_b)

    # Compute gradient approximation for bias
    grad_approx_b[k] <- ((cost_plus_b - cost_minus_b) / (2 * h)) %>% sum()

    # Reset perturbation vector
    perturb_b[k] <- 0

  }

  return(list(W_approx = grad_approx, b_approx = grad_approx_b))
}
```

```
# Example usage: x, y_ are your data and labels, W0, b0 are initialized
    weights and biases
# Initialize weights randomly (e.g., uniform distribution)
```

```
weights <- matrix(runif(num_inputs * num_outputs), nrow = num_inputs, ncol =
    num_outputs)

# Initialize biases (zeros) with dimensions 10*1 which is a vector
biases <- rep(0, num_outputs)

grad_approx_finite_diff <- compute_finite_diff(weights, biases, X_, Y_)

# Display the computed gradients
grad_approx_finite_diff

# Compare computed gradients with theoretical gradients
comparison <- cbind(gradient_Cost$grad_b, grad_approx_finite_diff$b_approx)
print(comparison)
```

Output:

Analytical Gradient	Finite Difference Approximation
−34.92651442	−34.92651444
0.34027081	0.34027081
0.09253228	0.09253228
19.56083310	19.56083310
3.06418524	3.06418524
0.88015153	0.88015153
1.90251056	1.90251056
1.62221489	1.62221489
4.51872676	4.51872676
2.94508927	2.94508927

We can confirm the correct implementation of our gradient descent algorithm, as the gradients obtained using the finite difference method match those derived from the algorithm.

1.4.4 Part V

To optimize our neural network model, we used the mini-batch gradient descent function, which involves updating the model's weights and biases incrementally using subsets of the training data. We configured our model to use a learning rate of 0.01 and processed the data in batches of 50 samples.We initialized the weights from a uniform distribution. The biases are initialized with zeros. The optimization was done on 10 epochs, during which we recorded the model's loss and accuracy for each epoch, both on the training set and the test set (Figure 3 and Figure 4).

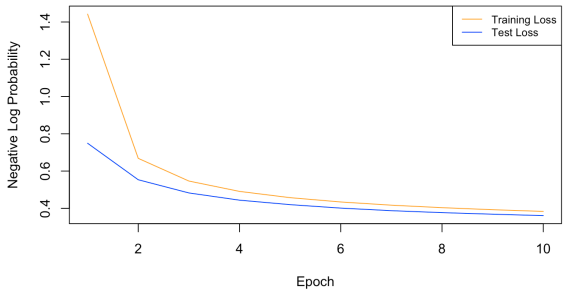

Figure 3: Loss per Epoch

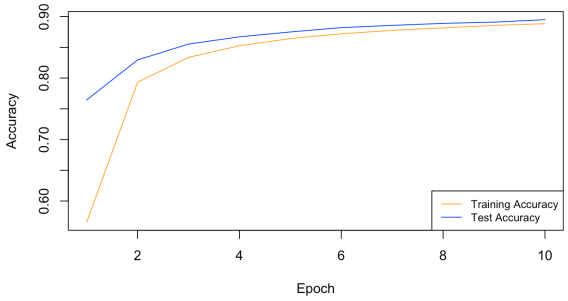

Figure 4: Accuracy per Epoch

We obtained a test accuracy of 90.15%. To illustrate the result, we sample 10 incorrectly classified digits from the test set in Figure 5 and 20 correctly classified digits in Figure 6.

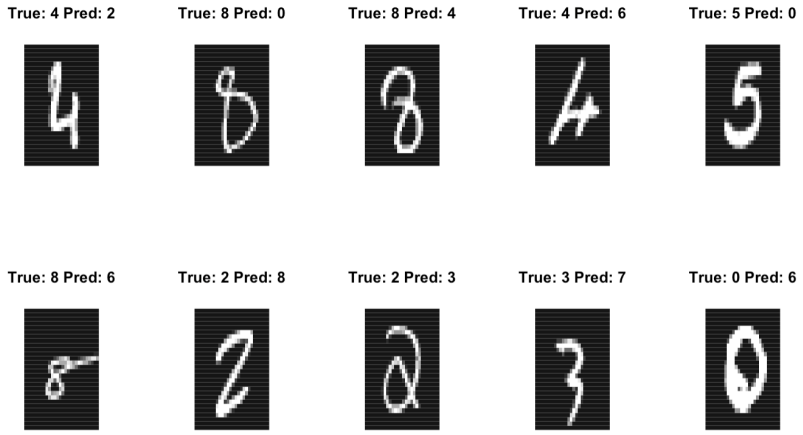

Figure 5: Sampled Images of Incorrectly Classified Digits

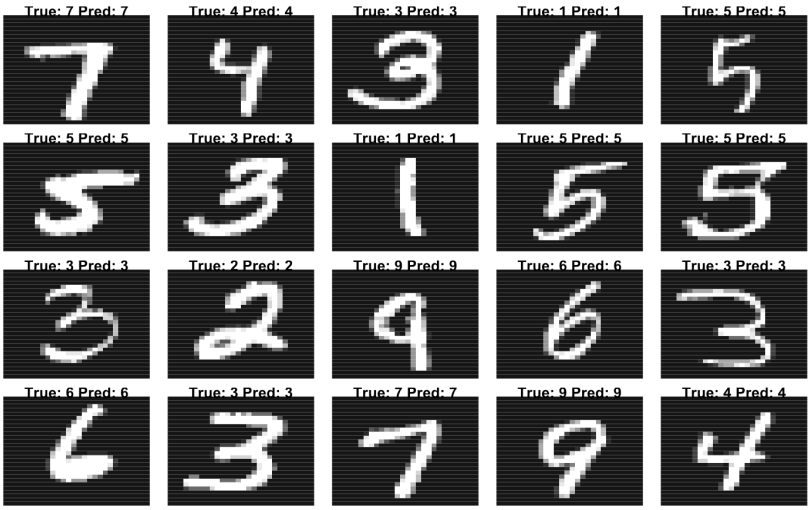

Figure 6: Sampled Images of Correctly Classified Digits

1.4.5 Part VI

The weight matrices connecting to the output units were visualized as digits in Figure 7. This visualization provides insight in how the neural network is working. We can observe patterns resembling the digits corresponding to the labels of our network.

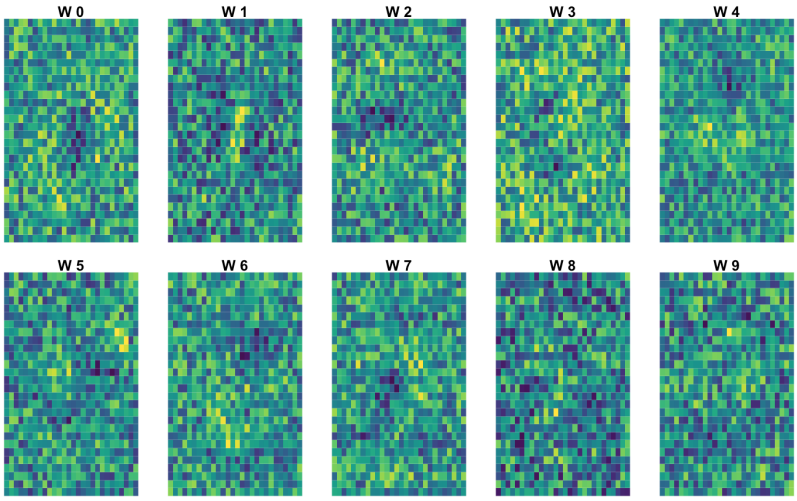

Figure 7: Heat Map of the Weights

1.5 Multi-Layer Neural Network

1.5.1 Part VII

We implement a function that computes a multi-layer neural network with 300 hidden layers using Keras Library as shown in figure 7. The neural network is computed as follow : Let $O_i^{(k)}$ be the output of the i-th neuron in the k-th layer, where k ranges from 1 to 300 for the 300 hidden layers. The output of each neuron can be described by the equation:

$$O_i^{(k)} = \tanh\left(\sum_j w_{ji}^{(k)} x_j^{(k-1)} + b_i^{(k)}\right)$$

where $w_{ji}^{(k)}$ are the weights connecting the j-th neuron of the $(k-1)$-th layer to the i-th neuron of the k-th layer, $x_j^{(k-1)}$ is the output of the j-th neuron from the $(k-1)$-th layer (or the input layer if $k = 1$), and $b_i^{(k)}$ is the bias term for the i-th neuron in the k-th layer. Each neuron uses the hyperbolic tangent activation function, defined as:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

The softmax function $S(x)$ is applied to the final outputs to get the probabilities.

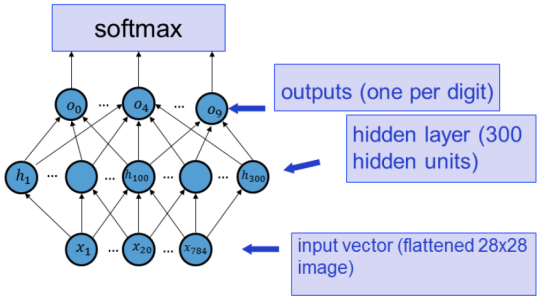

Figure 8: Multi-Layer Neural Network

We define our neural network model using the Keras library as follows:

```
model = Sequential([
    Dense(300, activation='tanh', input_shape=(784,)),
    Dense(10, activation='softmax')
])
```

1.5.2 Part VIII

To optimize our multi-layer neural network model, we used the mini-batch gradient descent algorithm. We configured our model to use a learning rate of 0.01 and processed the data in batches of 50 samples. The optimization was done on 10 epochs, during which we recorded the model's loss and accuracy for each epoch, both on the training set and the test set.

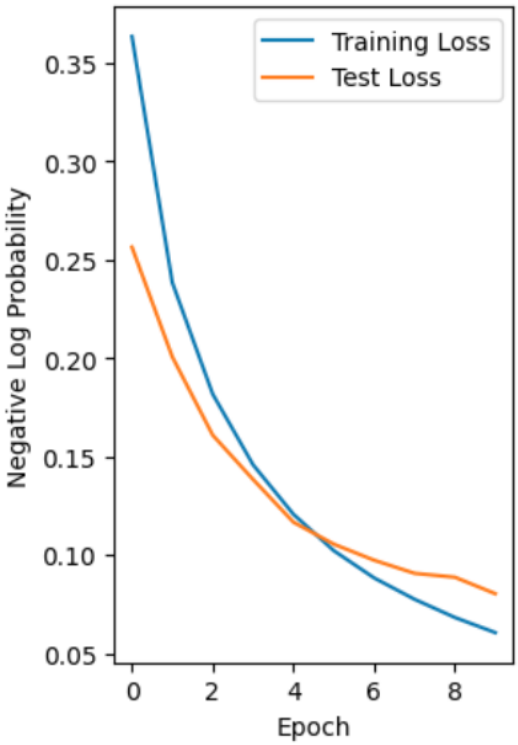

Figure 9: Loss per Epoch

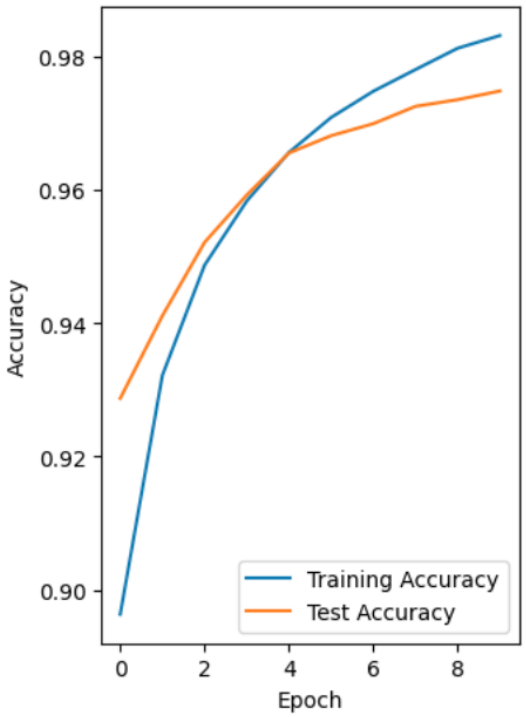

Figure 10: Accuracy per Epoch

The test classification performance was of 98%.To illustrate the result, we sample 10 incorrectly classified digits from the test set in Figure 11 and 20 correctly classified digits in Figure 12

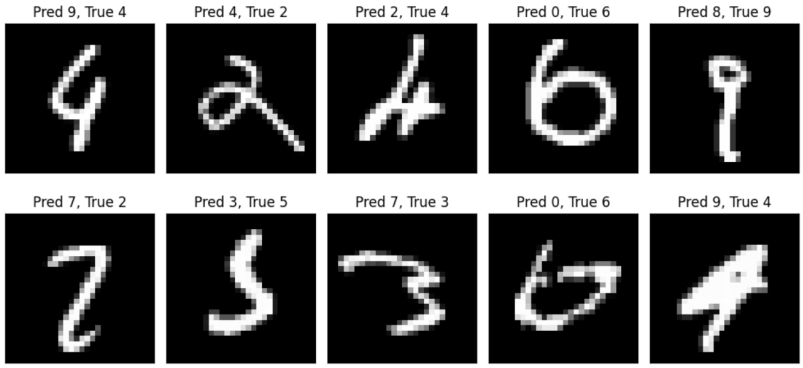

Figure 11: Sampled Images of Incorrectly Classified Digits

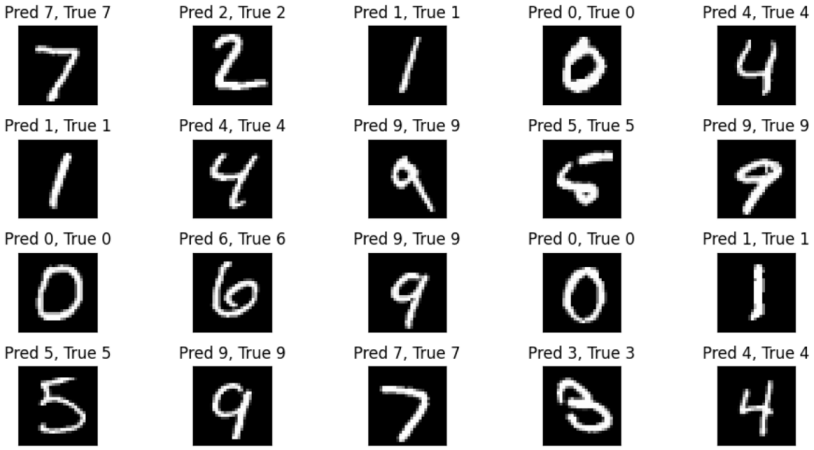

Figure 12: Sampled Images of Correctly Classified Digits

1.5.3 Part IX

We visualized two specific weight matrices,W_1 and W_300 corresponding to neurons 1 and 300 in the hidden layer. The weights connecting the hidden unit that corresponds to this W to the output units are :

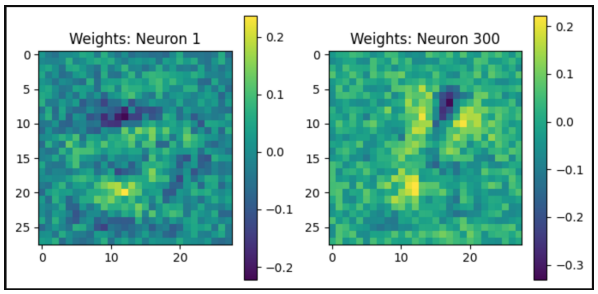

Figure 13: Heat Map of the Weights

Neuron 1 (w1): Since the weights shows strong positive weights on one side of the image and strong negative weights on the opposite side, this neuron could be detecting edges or orientation-specific features. For instance, it might be tuned to contrast differences that delineate the left side from the right side of the digits, which could be crucial for distinguishing between "6" and "9."

Neuron 300 (w2): Since the weights mostly highlight the center of the image area while diminishing the edges. This could suggest that Neuron 300 is primarily sensitive to features in the center of the image. This neuron might be specializing in detecting patterns that occur centrally in the digits, such as the loops in "0," "6," "8," or the top arch of "9."

1.6 Conclusion

In this report, we trained one layer neural network and a multi-layer neural network on the MNIST dataset. Both networks achieved a test accuracy exceeding 90%. Notably, the multi-layer network outperformed the single-layer one, attaining a higher accuracy, making it the optimal model for the task.

2 Section II : Reinforcement Learning and Markov Decision Processes

There are 3 basic paradigms in Machine learning: supervised learning, unsupervised learning, and reinforcement learning. Throughout our Machine learning course, we mainly focused on supervised learning: in this paradigm, data comes with labels and the goal is to predict labels for new data while minimizing error. In unsupervised learning, data is unlabeled and the goal is to organize it in some neat way.

Today, we take a look at the third paradigm: reinforcement learning, and how it is represented mathematically by a Markov Decision Process.

A Markov Decision Process is a decision-making process in which an agent interacts with, and learns from an environment in order to achieve or approximate an optimal outcome. A balance between exploration and exploitation must be achieved, with the goal of maximizing long-term reward, whose feedback may not be immediate. Markov Decision Processes (MDP) model situations where outcomes are partly random and partly under control of a decision maker.

We will attempt to elucidate these concepts by summarizing chapter 3 of Richard S. Sutton and Andrew G. Barto's book *Reinforcement Learning: An Introduction*.

First, we will introduce the important basic elements such as agent, environment, action, and reward. Then, we will define the concepts of a policy and a value function. Finally, we will explain how the Bellman equation is used in the optimization process.

The Markov decision process is described in terms of 4 basic elements: agent, environment, action, and reward.

The *agent* is the learner and decision-maker. The *environment* is the thing the agent interacts with. At each time step, the agent can evaluate the *state* of the environment and choose an *action* to take, which will bring the agent to a new state. Depending on this new state, the agent will receive a *reward*, which the agent will try to maximize.

It is important to note that the outcome of an action is not deterministic, and is instead determined by a probability distribution. That is to say, given a state s and an action a, the probability of attaining a different state s' and its corresponding reward r is given by a function $p(s', r|s, a)$. Crucially, we note that in a *Markov* decision process, the probability is determined by only the previous state. This means that in order to achieve desired outcomes, the state itself must contain important information about the past. We say a state satisfying this condition has the *Markov property*.

Despite its formal definition, MDP is quite flexible. The definition of an action is broad and can include external decisions like moving a robot arm or internal decisions like choosing to focus on some aspect of the environment rather than another. The definition of environment is also broad and should be thought of as even including parts of the agent like for example its motor if it's a robot.

There are many diverse examples of MDPs: Stirring a bioreactor, picking up garbage, and learning

to walk are all processes that can fit into the MDP framework. Each has its own agent, environment, actions and rewards. In a bioreactor, the reward might correspond to the concentration of a desired chemical element. For a garbage collecting robot, a reward might be $+1$ for every can picked up, and -3 every time it runs out of battery. For a robot learning to walk, the reward could correspond to the distance it managed to move itself forward.

The goal of MDP is always to maximize the cumulative reward. Here we must distinguish between *Episodic* and *Continuing* tasks. Episodic tasks are ones where the agent-environment interaction breaks up naturally into *episodes*, like for example individual games of chess. Continuing tasks are ones which potentially go on forever. In the former case, the cumulative reward for each episode is easily defined as the sum of the rewards at each step. However in the latter case, this sum could be infinite, so in this case we do something called *discounting* where each successful step gives a smaller and smaller (but still positive) reward, so that the infinite sum converges.

So far we have defined the basic concepts of MDPs, and explained how the agent-environment interaction yields rewards, which are to be summed up and maximized.

Next, we define the notions of a *policy* and a *value function*. A policy π is a rule giving us the probability $\pi(a|s)$ that an agent following that policy will pick an action a at a state s.

A value function under a specified policy π is a function $v_\pi(s)$ giving the expected return (cumulative reward) when starting from the state s and following the policy π. Informally, it tells us how good the state s is, given a policy.

We can also define the value of choosing action a at state s, denoted $q_\pi(s, a)$ and corresponding to the expected return when starting from s, choosing action a, and then following the policy π.

It is clear that the future cumulative reward G_t at step t equals R_{t+1} (the reward at step $t + 1$) plus the future cumulative reward G_{t+1} at step $t+1$. We have $G_t = R_{t+1} + G_{t+1}$. From this equation it can be deduced that

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')]$$

where γ is a discounting parameter. This equation is known as the *Bellman equation* for $v_\pi(s)$. Despite its daunting appearance, it simply states that the value of the state s equals the value of the expected next state, plus the expected reward along the way.

We have now described how the Bellman equation characterizes the value function of a given policy.

Finally, we will look at what makes a policy optimal, and how the Bellman optimality equation can be used to find the value function of an optimal policy.

We say that a policy π is *optimal* if, for every state s and every other policy π', we have $v_\pi(s) \geq v_{\pi'}(s)$. It is a fact that there is always at least one optimal policy, and there could be more than one, however all optimal policies will have the same value function, denoted v_*.

We can determine v_* by solving the *Bellman optimality equation* for v_*:

$$v_*(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')]$$

for each state. There is also a Bellman optimality equation for q_*:

$$q_*(s) = \sum_{s',r} p(s', r|s, a)[r + \gamma \max_a q_*(s', a')]$$

Once $v_*(s)$ is known, an optimal policy is one which only assigns nonzero probability to actions that maximize the Bellman optimality equation. In this sense, it is said to be *greedy*, that is to say, it only considers immediate consequences. But since it is maximizing the expected return, a policy obtained this way is actually optimal in the long run.

Unfortunately, in practice it is often computationally impractical to use the Bellman optimality equation to find an optimal policy. Since there is one equation to solve per state, solving them all becomes infeasible when there are a large number of states (for example there are about 1020 states in backgammon). There are many reinforcement learning methods focused on approximately solving the Bellman optimality equation, since in practice, this is often the best we can hope to do.

We have seen now how optimal policies can be determined, and what limitations there are to determining them.

In summary, Markov Decision Processes are a form of reinforcement learning in which an agent interacts with its environment, and attempts to maximize its cumulative reward by finding an optimal policy. Such a policy can be determined by solving the Bellman optimality equation, however in practice it is often more practical to find approximate solutions due to computational limitations.

3 References

- Brownlee , Jason . "How to Control the Stability of Training Neural Networks with the Batch Size." Machine Learning Mastery, 28 Aug. 2020, machinelearningmastery.com/how-to-control-the-speed-and-stability-of-training-neural-networks-with-gradient-descent-batch-size/.
- Brownlee, Jason. "A Gentle Introduction to Mini-Batch Gradient Descent and How to Configure Batch Size." Machine Learning Mastery, 19 Aug. 2019, machinelearningmastery.com/gentle-introduction-mini-batch-gradient-descent-configure-batch-size/.
- Hertzmann, Aaron , et al. CSC 411 / CSC D11 / CSC C11 Gradient Descent. 2015.
- "MNIST." Kaggle.com, www.kaggle.com/code/ddangman/mnist. Accessed 29 Apr. 2024.
- "Neural Networks - How to Set Mini-Batch Size in SGD in Keras." Cross Validated, stats.stackexchange.com/question to-set-mini-batch-size-in-sgd-in-keras.

- Olamendy, Juan C. "Mini-Batch Gradient Descent in Keras." Medium, 30 Nov. 2023, medium.com/@juanc.olamendy batch-gradient-descent-in-keras-95cfdd7dd7a5. Accessed 29 Apr. 2024.
- Sutton, Richard S., and Andrew G. Barto. Reinforcement Learning. Second Edition ed., Westchester Publishing Services, 2020, incompleteideas.net/book/RLbook2020.pdf.

4 Appendix : Code

4.1 Part I

```
    show_digit <- function(arr784, col=gray(12:1/12), ...) {
    image(matrix(arr784, nrow=28)[,28:1], col=col, ...)
}

par(mfrow=c(10, 10), mar=c(0.5, 0.5, 0.5, 0.5))

# Loop over all the digits
for (digit in 0:9) {
    # Loop over 10 images of each digit
    for (i in 1:10) {
        image(matrix(data[[paste0("train", digit)]][i,], nrow=28)[,28:1], col=
    gray(12:1/12), axes=FALSE)
    }
}

par(mfrow=c(1, 1), mar=c(5.1, 4.1, 4.1, 2.1))
```

4.2 Part II - IV

```
    library(R.matlab)
library(here)
library(tidyverse)

mat_data <- readMat(here("mnist_all.mat"))


##-------Creating Train and Test Datasets
    ----------------------------------------


df <- mat_data
```

```r
for (i in 1:length(df)) {
  # Get the name of the current matrix
  matrix_name <- names(df)[i]

  # Get the current matrix
  current_matrix <- df[[i]]

  # Create a new column with the matrix name
  new_column <- rep(matrix_name, nrow(current_matrix))

  # Bind the new column to the current matrix
  updated_matrix <- cbind(current_matrix, new_column)

  # Update the matrix in the list
  df[[i]] <- updated_matrix
}



DF <- df[[1]]

for (i in 2:20) {

  DDF <- rbind(DF, df[[i]])
  DF <- DDF

}

DF <- as.data.frame(DF)



summary(as.factor(DF$new_column))

DF$Y <- str_sub(DF$new_column, -1)
DF$Set <- str_sub(DF$new_column, end = -2)



summary(as.factor(DF$Y))
summary(as.factor(DF$Set))

DF_1 <- subset(DF, select = -c(new_column))
```

```
Train_df <- DF_1[DF_1$Set == 'train', ]
Test_df <- DF_1[DF_1$Set == 'test', ]


Train_df <- subset(Train_df, select = -c(Set))
Test_df <- subset(Test_df, select = -c(Set))

class(Train_df$Y)


summary(Train_df$Y)
summary(Test_df$Y)


#--------Adjusting the Values ----------------

for (i in 1:(ncol(Train_df)-1)) {

  Train_df[[i]] <- as.numeric(Train_df[[i]])

}



Train_df_X <- Train_df[,1:784]/255
Train_df <- cbind(Train_df_X, Train_df[,785])

colnames(Train_df)[colnames(Train_df) == "Train_df[, 785]"] <- "Number"


Maxi <- matrix(0, 784, 1)

for (i in 1:784) {
  Maxi[i,1] <- max(Train_df[,i])

}

summary(Maxi)
```

```
##---------Functions
    ------------------------------------------------------------



softmax <- function(x) {
  S_exp <- rowSums(exp(x))
  S_exp <- replicate(ncol(x), S_exp)
  return((exp(x)/S_exp))
}



Layer_1 <- function(x, W, b) {

  # x is the Train dataset with only predictors with dimensions n*784
  # W is the weight matrix of dimension 784 * 10
  # b is the beta0 weights with dimensions 10*1 which is a vector

  O_i <- (as.matrix(x) %*% W)
  biases <- replicate(nrow(O_i), b)
  O_i <- O_i + t(biases)
  Y <- softmax(O_i)

  return(Y)
}

# cost function
cost <- function(y, y_) {
  #y is the predicted matrix
  #y_ is the true values which is a one_hot_matrix

  NLL <- matrix(0, nrow(y), 1)

  for (i in 1:nrow(y)){

    NLL[i] <- (-log((y[i,]) %*% y_[i,]))

  }

  return(NLL)
}
```

```
compute_cost_and_gradient <- function(x, y, W, b ) {

  # x is the Train dataset with only predictors with dimensions n*784
  # y is the True value (number) in dimension n*1
  # W is the weight matrix of dimension 784 * 10
  # b is the beta0 weights with dimensions 10*1 which is a vector

  #one_hot_matrix

  Y_hat <- Layer_1(x, W, b)

  Y_H <- matrix(0, nrow = nrow(y), ncol = ncol(Y_hat))

  for (i in 1:nrow(y)) {
    Y_H[i, as.numeric(y[i]) + 1] <- 1
  }

  # Forward pass

  #Y_hat is matrix of dimension n*10

  # Negative log-likelihood

  nll <- cost(Y_hat, Y_H)

  # Backward pass for gradients
  d_logits <- (Y_hat - Y_H)
  grad_W <- t(x) %*% d_logits
  grad_b <- colSums(d_logits)

  return(list(nll = nll, grad_W = grad_W, grad_b = grad_b))

}




##----------------Weight Initialization
    ----------------------------------------
```

```r
set.seed(2024)

X_ <- Train_df_X[1:50, ] %>% as.matrix()

Y_ <- Train_df[1:50,785] %>% as.matrix()

# Number of input features (neurons)
num_inputs <- ncol(X_)

# Number of output neurons
num_outputs <- 10

# Initialize weights randomly (e.g., uniform distribution)
weights <- matrix(runif(num_inputs * num_outputs), nrow = num_inputs, ncol =
    num_outputs)

# Initialize biases (zeros) with dimensions 10*1 which is a vector
biases <- rep(0, num_outputs)

##---------Output
    ----------------------------------------------------------------



gradient_Cost <- compute_cost_and_gradient(X_, Y_, weights, biases)
gradient_Cost

##-------Part_4
    ------------------------------------------------------------------



# Finite difference approximation function
compute_finite_diff <- function(W, b, x, y_, h = 1e-5) {
  # Perturbation vector
  perturb <- matrix(0, nrow(W), ncol(W))
  grad_approx <- matrix(0, nrow(W), ncol(W))

  Y_H <- matrix(0, nrow = nrow(y_), ncol = ncol(W))

  for (i in 1:nrow(y_)) {
    Y_H[i, as.numeric(y_[i]) + 1] <- 1
```

```
}


for (i in 1:nrow(W)) {
  for (j in 1:ncol(W)) {
    # Perturb parameter
    perturb[i, j] <- h
    W_plus <- W + perturb
    W_minus <- W - perturb

    # Forward pass with perturbed weights
    result_plus <- Layer_1(x, W_plus, b)
    cost_plus <- cost(Y_H, result_plus)

    result_minus <- Layer_1(x, W_minus, b)
    cost_minus <- cost(Y_H, result_minus)

    # Compute gradient approximation
    grad_approx[i, j] <- ((cost_plus - cost_minus) / (2 * h)) %>% sum()

    # Reset perturbation vector
    perturb[i, j] <- 0
  }
}

perturb_b <- rep(0, length(b))
grad_approx_b <- rep(0, length(b))

for (k in 1:length(b)) {
  perturb_b[k] <- h
  b_plus <- b + perturb_b
  b_minus <- b - perturb_b

  # Forward pass with perturbed weights
  result_plus_b <- Layer_1(x, W, b_plus)
  cost_plus_b <- cost(Y_H, result_plus_b)

  result_minus_b <- Layer_1(x, W, b_minus)
  cost_minus_b <- cost(Y_H, result_minus_b)

  # Compute gradient approximation
```

```
    grad_approx_b[k] <- ((cost_plus_b - cost_minus_b) / (2 * h)) %>% sum()

    # Reset perturbation vector
    perturb_b[k] <- 0

  }
  return(list(W0_approx = grad_approx, b_approx = grad_approx_b))

}

##--------- Example usage: x, y_ are your data and labels, W0, b0 are
    initialized weights and biases------
grad_approx_finite_diff <- compute_finite_diff(weights, biases,X_, Y_)

grad_approx_finite_diff

cbind(gradient_Cost$grad_b, grad_approx_finite_diff$b_approx)

# Some other weights and biases

weights_1 <- matrix(rnorm(num_inputs * num_outputs), nrow = num_inputs, ncol =
     num_outputs)

biases_1 <- rep(1, num_outputs)

grad_approx_2 <- compute_cost_and_gradient(X_, Y_, weights_1, biases_1)

grad_finite_2 <- compute_finite_diff(weights_1, biases_1, X_, Y_)

cbind(grad_approx_2$grad_b, grad_finite_2$b_approx)
```

4.3 Part V- VI

```
    show_digit = function(arr784, col = gray(12:1 / 12), ...) {
  image(matrix(as.matrix(arr784[-785]), nrow = 28)[, 28:1], col = col, ...)
}

# load image files
load_image_file = function(filename) {
  ret = list()
  f = file(filename, 'rb')
```

```
  readBin(f, 'integer', n = 1, size = 4, endian = 'big')
  n    = readBin(f, 'integer', n = 1, size = 4, endian = 'big')
  nrow = readBin(f, 'integer', n = 1, size = 4, endian = 'big')
  ncol = readBin(f, 'integer', n = 1, size = 4, endian = 'big')
  x = readBin(f, 'integer', n = n * nrow * ncol, size = 1, signed = FALSE)
  close(f)
  data.frame(matrix(x, ncol = nrow * ncol, byrow = TRUE))
}

# load label files
load_label_file = function(filename) {
  f = file(filename, 'rb')
  readBin(f, 'integer', n = 1, size = 4, endian = 'big')
  n = readBin(f, 'integer', n = 1, size = 4, endian = 'big')
  y = readBin(f, 'integer', n = n, size = 1, signed = FALSE)
  close(f)
  y
}

# load images
train = load_image_file("train-images.idx3-ubyte")
test  = load_image_file("t10k-images.idx3-ubyte")

# load labels
train$y = load_label_file("train-labels.idx1-ubyte")
test$y  = load_label_file("t10k-labels.idx1-ubyte")
vectorize <- function(j) {
  k <- rep(0, 10)
  k[j + 1] <- 1
  k
}
y <- t( apply(matrix(train$y), 1, vectorize) )
train_x <- as.matrix(sapply(train[,-785], as.numeric))
test_x <- as.matrix(sapply(test[,-785], as.numeric))
test_x <- test_x / 255.0
test_y <- t( apply(matrix(test$y), 1, vectorize) )
train_x = train_x/255.0
softmax <- function(x) {
  S_exp <- rowSums(exp(x))
  S_exp <- replicate(ncol(x), S_exp)
  return((exp(x)/S_exp))
```

```
}
forward <- function(x, W, b) {

  # x is the Train dataset with only predictors with dimensions n*784
  # W is the weight matrix of dimension 784 * 10
  # b is the beta0 weights with dimensions 10*1

  O_i <- (as.matrix(x) %*% W)
  biases <- replicate(nrow(O_i), b)
  O_i <- O_i + t(biases)
  Y <- softmax(O_i)

  return(Y)
}
cost <- function(y, y_) {
  NLL <- numeric(nrow(y))
  for (i in 1:nrow(y)) {
    true_class_index <- which(y_[i,] == 1)  # Finds the index of '1' in the
    one-hot encoded vector
    NLL[i] <- -log(y[i, true_class_index])  # Negative log of the probability
    of the true class
  }
  return(NLL)
}

compute_cost_and_gradient <- function(x, y, W, b ) {

  # x is the Train dataset with only predictors with dimensions n*784
  # y is the True value (number) in dimension n*1
  # W is the weight matrix of dimension 784 * 10
  # b is the beta0 weights with dimensions 10*1

  #one_hot_matrix
  # Forward pass
  Y_hat <- forward(x, W, b)
  nll <- cost(Y_hat, y)

  # Backward pass for gradients
  d_logits <- (Y_hat - y)
  grad_W <- t(x) %*% d_logits
  grad_b <- colSums(d_logits)
```

```
  return(list(nll = nll, grad_W = grad_W, grad_b = grad_b))

}
calculate_accuracy <- function(predictions, labels) {
    if (is.matrix(labels) && ncol(labels) > 1) {
        labels <- max.col(labels) - 1
    }
    accuracy <- mean(predictions == labels)
    return(accuracy)
}
set.seed(123)
num_inputs <- 784
num_outputs <- 10
W <- matrix(runif(num_inputs * num_outputs), nrow = num_inputs, ncol =
    num_outputs)
b <- rep(0, num_outputs)
n <- dim(y) [1]
epoch_loss <- list()
epoch_accuracy <- list()
test_epoch_loss <- list()
test_epoch_accuracy <- list()
batch_size = 50
alpha = 0.01
epochs <- 10
  for (epoch in 1:epochs) {
    cat("Epoch:", epoch, "\n")
    indices <- sample(n)  # Shuffle indices
    epoch_losses <- numeric()
    epoch_accuracies <- numeric()

    for (i in seq(1, n, by = batch_size)) {
        end_idx <- min(i + batch_size - 1, n)
        rows <- indices[i:end_idx]
        X <- train_x[rows, ]  # Training input for this batch
        Y <- y[rows, ]  # True labels for this batch

        Y_hat <- forward(X, W, b)

        # Convert Y_hat (softmax probabilities) to class indices
        predicted_labels <- apply(Y_hat, 1, which.max) - 1
```

```
        grads <- compute_cost_and_gradient(X, Y, W, b)
        gradW <- grads$grad_W /nrow(X)
        gradB <- grads$grad_b/ nrow(X)
        W <- W - alpha * gradW  # Update weights
        b <- b - alpha * gradB # Update biases

        current_loss <- sum(grads$nll) / length(rows)
        current_accuracy <- calculate_accuracy(apply(Y_hat, 1, which.max) - 1,
    apply(Y, 1, which.max) - 1)
        epoch_losses <- c(epoch_losses, current_loss)
        epoch_accuracies <- c(epoch_accuracies, current_accuracy)

    }
    epoch_loss[[epoch]] <- mean(epoch_losses)
    epoch_accuracy[[epoch]] <- mean(epoch_accuracies)

    test_Y_hat <- forward(test_x, W, b)
    test_predicted_labels <- apply(test_Y_hat, 1, which.max) - 1
    test_loss <- sum(cost(test_Y_hat, test_y))/ nrow(test_y)
    test_accuracy <- calculate_accuracy(test_predicted_labels, apply(test_y,
    1, which.max) - 1)

    test_epoch_loss[[epoch]] <- test_loss
    test_epoch_accuracy[[epoch]] <- test_accuracy
  }
  # Plot for Loss
plot(1:length(epoch_loss), unlist(epoch_loss), type = "l", col = "orange",
    xlab = "Epoch", ylab = "Negative Log Probability", ylim = range(c(unlist(
    epoch_loss), unlist(test_epoch_loss))))
lines(1:length(test_epoch_loss), unlist(test_epoch_loss), type = "l", col = "
    blue")
legend("topright", legend = c("Training Loss", "Test Loss"), col = c("orange",
     "blue"), lty = 1, cex = 0.8)

# Plot for Accuracy
plot(1:length(epoch_accuracy), unlist(epoch_accuracy), type = "l", col = "
    orange", xlab = "Epoch", ylab = "Accuracy", ylim = range(c(unlist(
    epoch_accuracy), unlist(test_epoch_accuracy))))
lines(1:length(test_epoch_accuracy), unlist(test_epoch_accuracy), type = "l",
    col = "blue")
```

```
legend("bottomright", legend = c("Training Accuracy", "Test Accuracy"), col =
    c("orange", "blue"), lty = 1, cex = 0.8)
#predict y on the test set
guess_Y <- forward(test_x,W,b)
guess <- max.col(guess_Y)-1
round(head(guess_Y),4)
head(guess)
results <- data.frame(actual = test$y, guess = guess)
results_table <- table (results)
results_table
accuracy <- sum(diag(results_table)) / nrow(test_x) # sum of diagonal / total
    predictions
cat("Test Accuracy:", accuracy * 100, "%\n")
show_digit <- function(arr784, col=gray(1:12 / 12), ...) {
    image(matrix(arr784, nrow=28)[,28:1], col=col, axes = FALSE, ...)
}

# Plot images classified incorrectly
incorrect_indices <- which(results[,1] != results[,2])  # indices for
    predictions do not match true labels
set.seed(123)  # For reproducibility
sample_incorrect_indices <- sample(incorrect_indices, min(10, length(
    incorrect_indices)))
par(mfrow = c(2,5))

for (index in sample_incorrect_indices) {
  show_digit(test_x[index,])  # Display the digit image
  true_label <- results[index, 1]  # Get the true label
  predicted_label <- results[index, 2]  # Get the predicted label
  title(main = paste("True:", true_label, "Pred:", predicted_label))
}
#plot images classified correctly # indices for predictions match true labels
correct_indices <- which(results[,1] == results[,2])
set.seed(123)  # For reproducibility
sample_correct_indices <- sample(correct_indices, min(20, length(
    correct_indices)))
par(mar = c(0.6, 0.6, 0.6, 0.6))
par(mfrow = c(4, 5))
for (index in sample_correct_indices) {
  show_digit(test_x[index,])
  true_label <- results[index, 1]  # Get the true label
```

```
  predicted_label <- results[index, 2]  # Get the predicted label
  title(main = paste("True:", true_label, "Pred:", predicted_label))  # Add a
    title with true and predicted labels

}
visualize_weights <- function(W, num_cols = 5) {
    num_neurons <- ncol(W)  # Number of output neurons
    par(mfrow = c(ceiling(num_neurons / num_cols), num_cols), mar = c(1, 1, 1,
     1))
    gradual_palette <- colorRampPalette(c("blue", "green"))(256)
    for (i in 1:num_neurons) {
        w_image <- matrix(W[, i], nrow = 28, byrow = TRUE)
        rotated_image <- t(w_image)[, ncol(w_image):1]
        image(1:28, 1:28, rotated_image, col = viridis::viridis(256), main =
    paste("W", i - 1), axes = FALSE)
    }
}
visualize_weights(W)
```

4.4 Part VII - IX

```
    import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import SGD

import matplotlib.pyplot as plt
import numpy as np

(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Normalize the pixel values to [0, 1]
x_train, x_test = x_train / 255.0, x_test / 255.0

# Flatten the images to 1D array of 784 features (28*28)
x_train = x_train.reshape((x_train.shape[0], 784))
x_test = x_test.reshape((x_test.shape[0], 784))

model = Sequential([
    Dense(300, activation='tanh', input_shape=(784,)),
```

```
    Dense(10, activation='softmax')
])

model.compile(optimizer=SGD(learning_rate=0.1),  # Using SGD optimizer
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Mini-batch gradient descent with a batch size of 50
history = model.fit(x_train, y_train, epochs=10, batch_size=50,
    validation_data=(x_test, y_test))

test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print('\nTest accuracy:', test_acc)

# Plot learning curves
plt.figure(figsize=(5, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Negative Log Probability')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.tight_layout()
plt.show()

predictions = model.predict(x_test)
predicted_classes = np.argmax(predictions, axis=1)

correct_indices = np.nonzero(predicted_classes == y_test)[0]
incorrect_indices = np.nonzero(predicted_classes != y_test)[0]

plt.figure(figsize=(10, 5))
for i, correct in enumerate(correct_indices[:20]):
```

```
    plt.subplot(4, 5, i + 1)
    plt.imshow(x_test[correct].reshape(28, 28), cmap='gray', interpolation='
    none')
    plt.title("Predicted {}, Class {}".format(predicted_classes[correct],
    y_test[correct]))
    plt.xticks([])
    plt.yticks([])

plt.tight_layout()
plt.show()

plt.figure(figsize=(10, 5))
for i, incorrect in enumerate(incorrect_indices[:10]):
    plt.subplot(2, 5, i + 1)
    plt.imshow(x_test[incorrect].reshape(28, 28), cmap='gray', interpolation='
    none')
    plt.title("Pred {}, True {}".format(predicted_classes[incorrect], y_test[
    incorrect]))
    plt.xticks([])
    plt.yticks([])

plt.tight_layout()
plt.show()

weights = model.layers[0].get_weights()[0]  # This gets the weights from the
    input layer to the first hidden layer

# Assuming weights shape is (784, 300)
# Select two neurons, e.g., the first and the last neuron in the hidden layer
w1 = weights[:, 0].reshape(28, 28)  # Weights for neuron 1
w2 = weights[:, -1].reshape(28, 28) # Weights for neuron 300

plt.figure(figsize=(8, 4))
plt.subplot(1, 2, 1)
plt.imshow(w1, cmap='viridis', interpolation='none')
plt.title('Weights: Neuron 1')
plt.colorbar()
plt.subplot(1, 2, 2)
plt.imshow(w2, cmap='viridis', interpolation='none')
plt.title('Weights: Neuron 300')
plt.colorbar()
```

```
plt.show()
```