VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Natural Language Processing - Exercise (CO3086)

Lab 8

Math Exercises

Semester 2, Academic Year 2024 - 2025

Teacher: Bui Khanh Vinh
Students: Nguyen Quang Phu - 2252621

HO CHI MINH CITY, MARCH 2025

Contents

Chapter 1

Problem Description and Solution

1.1 Problem 1

1.1.1 Description

Consider a fully connected neural network with the following architecture:

- **Input Layer:** 2 neurons (i_1, i_2)
- **Hidden Layer:** 2 neurons (h_1, h_2) with sigmoid activation
- **Output Layer:** 2 neurons (o_1, o_2) with sigmoid activation
- **Bias terms:** b_1 for the hidden layer, b_2 for the output layer
- **Loss function:** Mean Squared Error (MSE), given by:

$$MSE = \frac{1}{2} \sum (o_{\text{expt}} - o_{\text{pred}})^2$$

Learning rate: $\eta = 0.5$

The neural network is structured as follows:

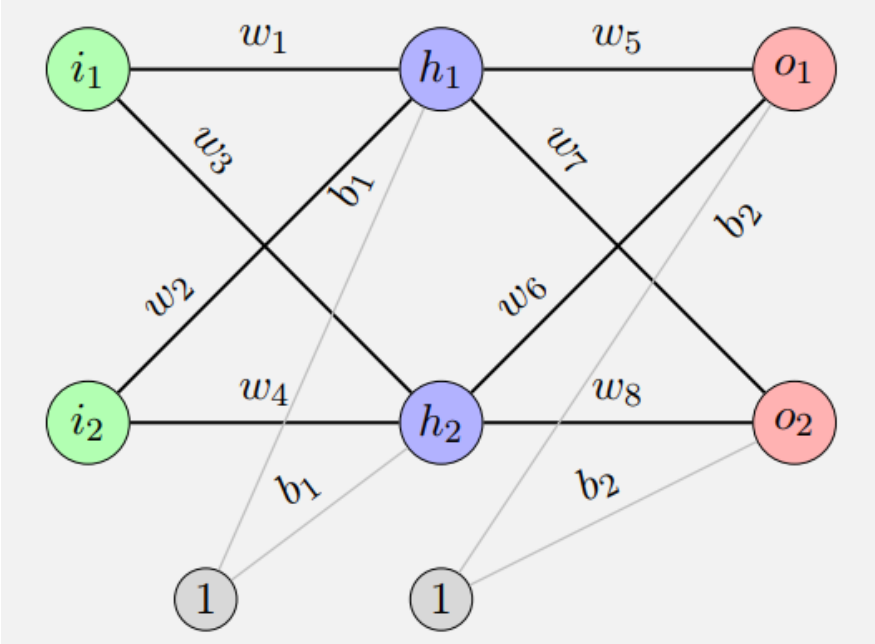


Figure 1.1: Neural Network Structure

The given parameter values are:

$$[w_1, w_2, w_3, w_4, b_1, w_5, w_6, w_7, w_8, b_2] = [0.15, 0.20, 0.25, 0.30, 0.35, 0.40, 0.45, 0.50, 0.55, 0.60]$$

The input and expected output values are:

$$[i_1, i_2] = [0.05, 0.10], \quad [o_1, o_2] = [0.01, 0.99]$$

Perform one forward pass and one backward pass through the network to compute the updated weights and biases with detailing each step of the calculations.

1.1.2 Solution

Forward Pass Calculation

Given neural network structure and parameters, we compute the forward pass step-by-step.

Step 1: Compute Activations for Hidden Layer

Each hidden neuron h_j is computed using the formula:

$$h_j = \sigma(z_j) = \frac{1}{1 + e^{-z_j}}$$

where

$$z_j = \sum_i w_{ij} i_i + b_1$$

For h_1:

$$\begin{aligned} z_1 = w_1 i_1 + w_3 i_2 + b_1 &= (0.15 \times 0.05) + (0.25 \times 0.10) + 0.35 \\ &= 0.0075 + 0.025 + 0.35 = 0.3825 \\ h_1 &= \frac{1}{1 + e^{-0.3825}} \approx 0.5948 \end{aligned}$$

For h_2:

$$\begin{aligned} z_2 = w_2 i_1 + w_4 i_2 + b_1 &= (0.20 \times 0.05) + (0.30 \times 0.10) + 0.35 \\ &= 0.0100 + 0.0300 + 0.35 = 0.3900 \\ h_2 &= \frac{1}{1 + e^{-0.3900}} \approx 0.5966 \end{aligned}$$

Step 2: Compute Activations for Output Layer

Each output neuron o_k is computed as:

$$o_k = \sigma(z_k) = \frac{1}{1 + e^{-z_k}}$$

where

$$z_k = \sum_j w_{jk} h_j + b_2$$

For o_1:

$$\begin{aligned} z_3 = w_5 h_1 + w_7 h_2 + b_2 &= (0.40 \times 0.5948) + (0.50 \times 0.5966) + 0.60 \\ &= 0.2379 + 0.2983 + 0.60 = 1.1362 \\ o_1 &= \frac{1}{1 + e^{-1.1362}} \approx 0.757 \end{aligned}$$

For o_2:

$$\begin{aligned} z_4 = w_6 h_1 + w_8 h_2 + b_2 &= (0.45 \times 0.5948) + (0.55 \times 0.5966) + 0.60 \\ &= 0.2676 + 0.3281 + 0.60 = 1.1957 \\ o_2 &= \frac{1}{1 + e^{-1.1957}} \approx 0.767 \end{aligned}$$

Backward Propagation

The loss function used is Mean Squared Error (MSE):

$$MSE = \frac{1}{2}\sum(o_{\text{expected}} - o_{\text{predicted}})^2$$

Step 1: Compute Error Gradients at Output Layer

The error term for each output neuron is:

$$\delta_k = (o_k - o_{\text{expected},k}) \cdot o_k(1 - o_k)$$

For o_1:

$$\begin{aligned}\delta_3 &= (0.757 - 0.01) \times 0.757 \times (1 - 0.757) \\ &= 0.747 \times 0.757 \times 0.243 \\ &\approx 0.136\end{aligned}$$

For o_2:

$$\begin{aligned}\delta_4 &= (0.767 - 0.99) \times 0.767 \times (1 - 0.767) \\ &= (-0.223) \times 0.767 \times 0.233 \\ &\approx -0.040\end{aligned}$$

Step 2: Compute Error Gradients at Hidden Layer

The error term for hidden neurons is:

$$\delta_j = h_j(1 - h_j)\sum_k \delta_k w_{jk}$$

For h_1:

$$\begin{aligned}\delta_1 &= 0.5948(1 - 0.5948)(0.136 \times 0.40 + (-0.040) \times 0.45) \\ &= 0.5948 \times 0.241 \times (0.0544) \\ &\approx 0.0078\end{aligned}$$

For h_2:

$$\begin{aligned}\delta_2 &= 0.5966(1 - 0.5966)(0.136 \times 0.50 + (-0.040) \times 0.55) \\ &= 0.5966 \times 0.241 \times (0.064) \\ &\approx 0.0092\end{aligned}$$

Step 3: Compute Weight and Bias Updates

The weight updates follow the gradient descent rule:

$$\Delta w = -\eta \cdot \frac{\partial MSE}{\partial w}$$

For output layer weights:

$$w_5' = w_5 - \eta\delta_3 h_1 = 0.40 - (0.5 \times 0.136 \times 0.5948)$$
$$= 0.40 - 0.040$$
$$= 0.36$$

$$w_6' = w_6 - \eta\delta_4 h_1 = 0.45 - (0.5 \times -0.040 \times 0.5948)$$
$$= 0.45 + 0.012$$
$$= 0.462$$

$$w_7' = w_7 - \eta\delta_3 h_2 = 0.50 - (0.5 \times 0.136 \times 0.5966)$$
$$= 0.50 - 0.041$$
$$= 0.459$$

$$w_8' = w_8 - \eta\delta_4 h_2 = 0.55 - (0.5 \times -0.040 \times 0.5966)$$
$$= 0.55 + 0.012$$
$$= 0.562$$

For hidden layer weights:

$$w_1' = w_1 - \eta\delta_1 i_1 = 0.15 - (0.5 \times 0.0078 \times 0.05)$$
$$= 0.15 - 0.0002$$
$$= 0.1498$$

$$w_2' = w_2 - \eta\delta_2 i_1 = 0.20 - (0.5 \times 0.0092 \times 0.05)$$
$$= 0.20 - 0.0002$$
$$= 0.1998$$

$$w_3' = w_3 - \eta\delta_1 i_2 = 0.25 - (0.5 \times 0.0078 \times 0.10)$$
$$= 0.25 - 0.0004$$
$$= 0.2496$$

$$w_4' = w_4 - \eta\delta_2 i_2 = 0.30 - (0.5 \times 0.0092 \times 0.10)$$

$$= 0.30 - 0.0005$$
$$= 0.2995$$

For bias updates:

$$b_1' = b_1 - \eta\delta_1 = 0.35 - (0.5 \times 0.0078) = 0.346$$
$$b_2' = b_2 - \eta\delta_3 = 0.60 - (0.5 \times 0.136) = 0.532$$

Final updated weights and biases:

$$w_1' = 0.1498, \quad w_2' = 0.1998, \quad w_3' = 0.2496, \quad w_4' = 0.2995$$
$$w_5' = 0.36, \quad w_6' = 0.462, \quad w_7' = 0.459, \quad w_8' = 0.562$$
$$b_1' = 0.346, \quad b_2' = 0.532$$

1.2 Problem 2

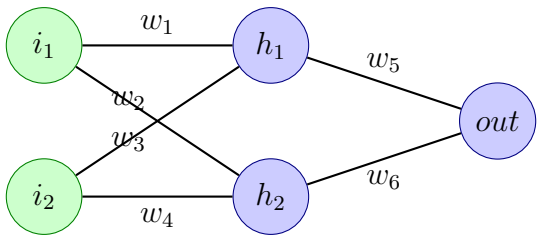
1.2.1 Description

Consider a fully connected neural network with the following architecture:

- **Input Layer:** 2 neurons (i_1, i_2)
- **Hidden Layer:** 2 neurons (h_1, h_2) without activation
- **Output Layer:** 1 neuron (out) without activation
- **Learning rate:** $\eta = 0.05$
- **Loss function:** Squared Error (SE), given by:

$$SE = \frac{1}{2}(y - out)^2$$

The neural network is structured as follows:



The given parameter values are:

$$[w_1, w_2, w_3, w_4, w_5, w_6] = [0.11, 0.21, 0.12, 0.08, 0.14, 0.15]$$

The input and expected output values are:

$$[i_1, i_2] = [2, 3], \quad [out] = [1]$$

Perform one forward pass and one backward pass through the network to compute the updated weights and biases with detailing each step of the calculations.

1.2.2 Solution

Forward Pass Calculation

Given the neural network structure and parameters, we compute the forward pass step-by-step.

Step 1: Compute Activations for Hidden Layer

Each hidden neuron h_j is computed as:

$$h_j = z_j$$

since there is no activation function.

$$z_j = \sum_i w_{ij} i_i$$

For the hidden layer neurons, we compute:

$$h_1 = z_1 = w_1 i_1 + w_3 i_2 = (0.11 \times 2) + (0.12 \times 3) = 0.22 + 0.36 = 0.58$$

$$h_2 = z_2 = w_2 i_1 + w_4 i_2 = (0.21 \times 2) + (0.08 \times 3) = 0.42 + 0.24 = 0.66$$

Step 2: Compute Activation for Output Layer

Since there is no activation function at the output layer, the output neuron is simply:

$$out = w_5 h_1 + w_6 h_2$$

$$= (0.14 \times 0.58) + (0.15 \times 0.66)$$

$$= 0.0812 + 0.099 = 0.1802$$

Backward Propagation

The loss function used is Squared Error (SE):

$$SE = \frac{1}{2}(y - out)^2$$

Step 1: Compute Error Gradient at Output Layer

The gradient of the loss function with respect to the output is:

$$\frac{\partial SE}{\partial out} = -(y - out)$$

$$= -(1 - 0.1802) = -0.8198$$

The error term for the output neuron is:

$$\delta_{\text{out}} = \frac{\partial SE}{\partial out} = -0.8198$$

Step 2: Compute Error Gradients at Hidden Layer

The error term for hidden neurons is given by:

$$\delta_j = \sum_k \delta_k w_{jk}$$

For the hidden layer neurons, we compute the error terms:

$$\delta_1 = \delta_{\text{out}} w_5 = -0.8198 \times 0.14 = -0.1148$$

$$\delta_2 = \delta_{\text{out}} w_6 = -0.8198 \times 0.15 = -0.1229$$

Step 3: Compute Weight Updates

The weight updates follow the gradient descent rule:

$$\Delta w = -\eta \cdot \frac{\partial SE}{\partial w}$$

For the output layer weights, we update:

$$w_5' = w_5 - \eta \delta_{\text{out}} h_1 = 0.14 - (0.05 \times -0.8198 \times 0.58) = 0.14 + 0.0238 = 0.1638$$

$$w_6' = w_6 - \eta \delta_{\text{out}} h_2 = 0.15 - (0.05 \times -0.8198 \times 0.66) = 0.15 + 0.0271 = 0.1771$$

For the hidden layer weights, we update:

$$w_1' = w_1 - \eta\delta_1 i_1 = 0.11 - (0.05 \times -0.1148 \times 2) = 0.11 + 0.0115 = 0.1215$$

$$w_2' = w_2 - \eta\delta_2 i_1 = 0.21 - (0.05 \times -0.1229 \times 2) = 0.21 + 0.0123 = 0.2223$$

$$w_3' = w_3 - \eta\delta_1 i_2 = 0.12 - (0.05 \times -0.1148 \times 3) = 0.12 + 0.0172 = 0.1372$$

$$w_4' = w_4 - \eta\delta_2 i_2 = 0.08 - (0.05 \times -0.1229 \times 3) = 0.08 + 0.0184 = 0.0984$$

Final updated weights:

$$w_1' = 0.1215, \quad w_2' = 0.2223, \quad w_3' = 0.1372, \quad w_4' = 0.0984$$

$$w_5' = 0.1638, \quad w_6' = 0.1771$$

1.3 Problem 3

1.3.1 Description

Softmax takes in an n-dimensional vector x and outputs another n-dimensional vector y:

$$y_i \equiv \frac{e^{x_i}}{\sum_k e^{x_k}}$$

In this question, we're going to compute the gradient of y with respect to x. Let

$$\delta_{ij} = \frac{\partial y_i}{\partial x_j}.$$

1. Derive an expression for δ_{ii}.
2. Now derive an expression for δ_{ij}, where $i \neq j$.
3. Write a general expression for δ_{ij}. Hint: Feel free to use curly brace notation to distinguish between the two cases where $i = j$ and $i \neq j$. For a concrete example, see how ELU was defined in the previous question.

1.3.2 Solution

Let's begin by computing the gradient of the softmax function.

The softmax function is given as:

$$y_i = \frac{e^{x_i}}{\sum_k e^{x_k}}$$

To find $\delta_{ij} = \frac{\partial y_i}{\partial x_j}$, we differentiate y_i with respect to x_j.

Case 1: $i = j$

$$\delta_{ii} = \frac{\partial y_i}{\partial x_i}$$

Using the quotient rule:

$$\frac{\partial y_i}{\partial x_i} = \frac{e^{x_i} \sum_k e^{x_k} - e^{x_i} e^{x_i}}{\left(\sum_k e^{x_k}\right)^2}$$

$$= \frac{e^{x_i} \left(\sum_k e^{x_k} - e^{x_i}\right)}{\left(\sum_k e^{x_k}\right)^2}$$

Since $y_i = \frac{e^{x_i}}{\sum_k e^{x_k}}$, we can rewrite this as:

$$\delta_{ii} = y_i(1 - y_i)$$

Case 2: $i \neq j$

For $i \neq j$:

$$\delta_{ij} = \frac{\partial y_i}{\partial x_j}$$

Using the quotient rule:

$$\frac{\partial y_i}{\partial x_j} = \frac{0 \cdot \sum_k e^{x_k} - e^{x_i} e^{x_j}}{\left(\sum_k e^{x_k}\right)^2}$$

$$= -\frac{e^{x_i} e^{x_j}}{\left(\sum_k e^{x_k}\right)^2}$$

Rewriting in terms of y_i and y_j:

$$\delta_{ij} = -y_i y_j, \quad \text{for } i \neq j$$

General Expression

We can express δ_{ij} in a single equation:

$$\delta_{ij} = \begin{cases} y_i(1 - y_i), & \text{if } i = j, \\ -y_i y_j, & \text{if } i \neq j. \end{cases}$$

This completes the derivation.

1.4 Problem 4

1.4.1 Description

You have a dataset where each example contains two features, x_1 and x_2, and a binary label. Here's a plot of the dataset and You want to develop a model to perform binary classification. Suppose you're using a small neural network with the architecture shown below.

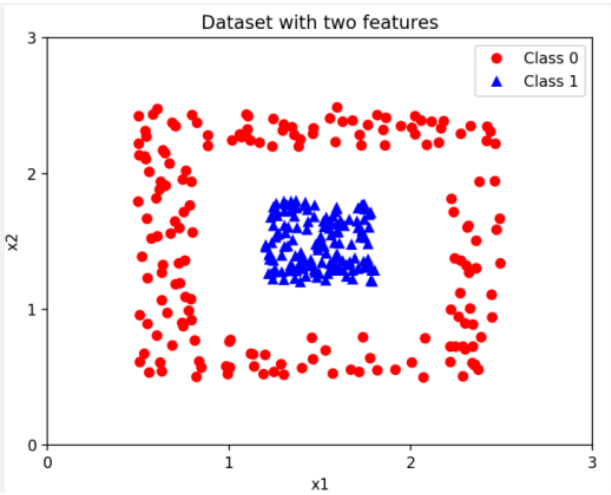


Figure 1.2: Dataset with two features

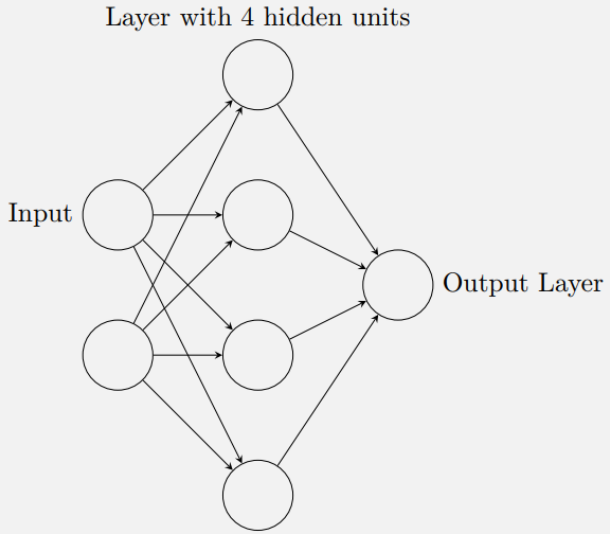


Figure 1.3: Neural network architecture

Below is a precise definition of the network. Note h is the activation function, $w_{i,j}$ denotes the jth weight in the ith hidden unit in the network, and $w_{\text{output},j}$ denotes the jth weight in the output layer. The biases follow similar rules.

$$a_i = h(w_{i,1} * x_1 + w_{i,2} * x_2 + b_i)$$

$$\text{output} = f\left(\sum_{j=1}^{4}(a_j * w_{\text{output},j}) + b_{\text{output}}\right)$$

Note that h is the activation function for the hidden units and f is the activation function for the output layer. For all of these questions, f is defined as:

$$f(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$$

1. If you used the function $h(x) = c * x$ for some $c \in \mathbb{R}$, is it possible for this model to achieve perfect accuracy on this dataset? If so, provide a set of weights that achieves perfect accuracy. If not, briefly explain why.

2. Now assume h is a modified version of the sign function:

$$h(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$$

Is it possible for this model to achieve perfect accuracy? If so, provide a set of weights that achieves perfect accuracy. If not, briefly explain why.

3. Recall the activation function $\text{ReLU}(x) = \max(0, x)$. Consider an alternative to ReLU called Exponential Linear Unit (ELU).

$$\text{ELU}(x) = \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

What is the gradient for ELU?

4. Name one advantage of using ELU over ReLU.

1.4.2 Solution to 1:

No; the decision boundary for this network would become linear due to the composition of matrix multiplication (something of the form prediction $1\{wx + b > 0\}$). Since the dataset is **not linearly separable** (it has a ring structure), a linear decision boundary **cannot** perfectly classify the data.

Using $h(x) = c \cdot x$ as an activation function does not introduce non-linearity into the model. Since the entire network would be a composition of linear transformations, the output remains a linear function of the inputs. This means that the network behaves like a single-layer perceptron with a linear decision boundary.

However, the dataset requires a **nonlinear** decision boundary to distinguish between the inner and outer points. A linear classifier cannot separate these two regions, meaning that **perfect accuracy is not achievable** with this activation function.

1.4.3 Solution to 2:

Yes, it is possible for this model to achieve perfect accuracy. The key is to have each hidden node evaluate one of the sides of the separating square, and the output layer checks that all conditions are true (or false, depending on how the hidden weights are set).

Each hidden neuron detects whether a point is inside or outside a boundary:

- $w_1 = (0, -1)$, $b_1 = 2$ *(Detects if $x_2 > 2$, upper boundary)*
- $w_2 = (-1, 0)$, $b_2 = 2$ *(Detects if $x_1 > 2$, right boundary)*

- $w_3 = (0, 1)$, $b_3 = -1$ *(Detects if $x_2 < 1$, lower boundary)*
- $w_4 = (1, 0)$, $b_4 = -1$ *(Detects if $x_1 < 1$, left boundary)*

For the output neuron, it activates only if all four conditions are met:

$$w_{\text{output}} = (1, 1, 1, 1), \quad b_{\text{output}} = -4$$

Since this structure correctly classifies points inside and outside the separating square, **perfect accuracy is achieved**.

1.4.4 Solution to 3:

To compute the gradient of the Exponential Linear Unit (ELU) function:

$$\text{ELU}(x) = \begin{cases} x, & x \geq 0 \\ \alpha(e^x - 1), & x < 0 \end{cases}$$

Differentiating each case:

- For $x \geq 0$, $\text{ELU}(x) = x$ so:
$$\frac{d}{dx}\text{ELU}(x) = 1.$$
- For $x < 0$, $\text{ELU}(x) = \alpha(e^x - 1)$, so:
$$\frac{d}{dx}\text{ELU}(x) = \alpha e^x.$$

Thus, the gradient of ELU is:

$$\frac{d}{dx}\text{ELU}(x) = \begin{cases} 1, & x \geq 0 \\ \alpha e^x, & x < 0 \end{cases}$$

1.4.5 Solution to 4:

One major advantage of ELU is that it has a **non-zero gradient everywhere**, unlike ReLU, which has a gradient of zero for all negative inputs. This helps avoid the **dying ReLU problem**, where neurons stop learning if they receive consistently negative inputs.

1.5 Problem 5

1.5.1 Description

You want to build a model to predict whether a startup will raise funding (label = 1) or not (label = 0) in its first year. You have access to a dataset of examples with 300 input features, including the number of founders, the number of employees, the variance in the age of the employees, etc. As a baseline model, you decide to train a logistic regression that outputs a probability $\hat{y}^{(i)} \in [0, 1]$ that a startup described by a vector of features $x^{(i)}$ raises funding in its first year, where the predicted label of an input is chosen to be 1 when $\hat{y}^{(i)} \geq 0.5$ and 0 otherwise.

1. If the shape of $x^{(i)}$ is $(n_x, 1)$, what should n_x be?
2. A logistic regression has trainable weights w and bias b. How many weights does the logistic regression model have? What is the dimension of b?
3. Consider the prediction $\hat{y}^{(i)} \in [0, 1]$. What is the dimension of $\hat{y}^{(i)}$? Write the output $\hat{y}^{(i)}$ in terms of the input $x^{(i)}$ and the parameters w and b.
4. After training your baseline logistic regression model, you decide to train a single hidden layer neural network with 80 hidden neurons on the same dataset. How many weights and biases does this neural network have?
5. You decide to use ReLU as your hidden layer activation, and also insert a ReLU before the sigmoid activation such that

   $$\hat{y} = \sigma(\text{ReLU}(z)),$$

   where z is the pre-activation value for the output layer. What problem are you going to encounter?

1.5.2 Solution to 1:

The input feature vector $x^{(i)}$ is given to have shape $(n_x, 1)$. Since the dataset consists of 300 input features, we conclude:

$$n_x = 300$$

1.5.3 Solution to 2:

A logistic regression model has trainable parameters w (weights) and b (bias). Since the dataset consists of 300 input features, we conclude:

- The number of weight w the model has is 300 (300 x 1).
- The bias b is a scalar, hence its dimension is 1 (1 x 1).

1.5.4 Solution to 3:

Since logistic regression outputs a single probability value per input $x^{(i)}$, the dimension of $\hat{y}^{(i)}$ is 1 (1 x 1)

The prediction is computed using the sigmoid function:

$$\hat{y}^{(i)} = \sigma(w^T x^{(i)} + b)$$

where $\sigma(z)$ is the sigmoid activation function:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Thus, the logistic regression model maps the input $x^{(i)}$ to a probability value, which is then thresholded at 0.5 to determine the final classification.

1.5.5 Solution to 4:

We consider a single hidden-layer neural network with 80 hidden neurons and an input size of 300.

Weight Parameters:

- Input to Hidden Layer: Each hidden neuron connects to all 300 input features, resulting in:
$$300 \times 80 = 24000$$
- Hidden to Output Layer: The single output neuron connects to all 80 hidden neurons:
$$80 \times 1 = 80$$
- **Total Weights:**
$$24000 + 80 = 24080$$

Bias Parameters:

- Each of the 80 hidden neurons has one bias term:
$$80$$

- The single output neuron has one bias term:

$$1$$

- **Total Biases:**

$$80 + 1 = 81$$

Total Parameters in the Neural Network:

$$\text{Total Parameters} = 24080 + 81 = 24161$$

1.5.6 Solution to 5:

Given that the final activation is computed as:

$$\hat{y} = \sigma(\text{ReLU}(z))$$

where z is the pre-activation value, we analyze its effect.

ReLU Function:

$$\text{ReLU}(z) = \begin{cases} z, & z \geq 0 \\ 0, & z < 0 \end{cases}$$

Effect on Sigmoid Output:

- Since **ReLU never outputs negative values**, the input to the sigmoid function is always ≥ 0.
- The sigmoid function outputs values in:

$$\sigma(x) = \begin{cases} \geq 0.5, & x \geq 0 \\ < 0.5, & x < 0 \end{cases}$$

- Because negative values are eliminated by ReLU, the sigmoid function will always output ≥ 0.5.

Final Issue: Since the sigmoid function will always produce values ≥ 0.5, the model will **always classify inputs as the positive class (1)**, making it impossible to predict the negative class correctly.

1.6 Problem 6

1.6.1 Description

You are building a classification model to distinguish between labels from a *synthetically* generated dataset. More specifically, you are given a dataset,

$$\{x^{(i)}, y^{(i)}\}_{i=1}^{m}, \quad \text{where } x^{(i)} \in \mathbb{R}^2, \quad y^{(i)} \in \{0, 1\}$$

The data is generated with the scheme:

$$X \mid Y = 0 \sim \mathcal{N}(2, 2)$$
$$X \mid Y = 1 \sim \mathcal{N}(0, 3)$$

You can assume the dataset is perfectly balanced between the two classes.

1. As a baseline, you decide to use a logistic regression model to fit the data. Since the data is synthesized easily, you can assume you have infinitely many samples (i.e., $m \to \infty$). Can your logistic regression model achieve 100% training accuracy? Explain your answer.

2. After training on a large training set of size M, your logistic regressor achieves a training accuracy of T. Can the following techniques, *applied individually*, improve over this *training accuracy*? Please justify your answer in a single sentence.

   (a) Adding a regularizing term to the binary cross-entropy loss function for the logistic regressor.

   (b) Standardizing all training samples to have *mean zero* and *unit variance*.

   (c) Using a 5-hidden layer feedforward network *without* non-linearities in place of logistic regression.

   (d) Using a 2-hidden layer feedforward network with ReLU in place of logistic regression.

1.6.2 Solution to 1:

No, logistic regression **cannot** achieve 100% training accuracy due to the inherent overlap in the data distributions.

Understanding the Problem

We are using a logistic regression model to classify data generated from two different Gaussian distributions. The dataset consists of feature-label pairs:

$$\{x^{(i)}, y^{(i)}\}_{i=1}^{m}, \quad x^{(i)} \in \mathbb{R}^2, \quad y^{(i)} \in \{0, 1\}$$

Given an infinitely large dataset ($m \to \infty$), we want to determine whether logistic regression can achieve **100% training accuracy**.

Why Logistic Regression Might Fail

1. Logistic Regression Assumption:

- Logistic regression finds a **linear decision boundary** of the form:

$$w^T x + b = 0$$

- This means logistic regression can only perfectly classify data that is **linearly separable**.

2. Understanding the Data Distributions:

- The feature vector X is generated according to the following distributions:

$$X|Y = 0 \sim \mathcal{N}(2, 2), \quad X|Y = 1 \sim \mathcal{N}(0, 3)$$

- This implies:
  - Class $Y = 0$ is centered around 2 with a variance of 2, meaning most values are near 2.
  - Class $Y = 1$ is centered around 0 with a variance of 3, meaning most values are near 0.

3. Overlap Between the Distributions:

- Since both distributions have relatively large standard deviations ($\sigma_0 = \sqrt{2}$, $\sigma_1 = \sqrt{3}$), a significant proportion of points from both classes will overlap.
- Some points from $Y = 0$ will have values close to 0, and some from $Y = 1$ will have values close to 2.

- Because of this overlap, **no linear decision boundary can perfectly separate all points**.

4. Effect on Training Accuracy:

- Even if we have **infinitely many samples** ($m \to \infty$), the nature of the Gaussian distributions means that points will always exist that are misclassified by a linear boundary.
- Logistic regression will find the **best possible linear boundary**, but it **cannot perfectly classify all points** due to the overlap in feature space.

Conclusion

- Since the two distributions have overlapping probability density functions, logistic regression cannot perfectly separate the classes.
- Even with infinite data, **some misclassification is inevitable**, as no linear boundary can correctly classify all points.
- Therefore, **logistic regression cannot achieve 100% training accuracy**.

1.6.3 Solution to 2:

The key observation is that logistic regression is a **linear classifier** and suffers from **high bias**. Since we assume the model has been trained to convergence, the only techniques that can improve training accuracy are those that **increase model capacity** or **reduce bias**. We do not need to worry about overfitting, as we are only concerned with training performance.

(a) Adding a regularizing term to the binary cross-entropy loss function

No, regularization does not improve training accuracy.

- Regularization (e.g., L2 or L1) helps prevent overfitting by penalizing large weights.
- However, since we are focused on **training accuracy**, regularization may actually reduce accuracy by making the model less flexible.
- Thus, adding a regularization term would not help the model fit the training data better.

(b) Standardizing all training samples to have mean zero and unit variance

No, standardization does not impact training accuracy.

- Standardizing data improves **numerical stability** and **convergence training speed** training.
- However, it does **not change the model's capacity** or **decision boundary**.
- It may speed up training but does not improve the final training accuracy.

(c) Using a 5-hidden layer feedforward network without non-linearities

No, the decision boundary remains linear.

- A neural network without non-linearities is mathematically equivalent to a **single-layer linear model**.
- Stacking multiple linear transformations:

  $$W_5W_4W_3W_2W_1x + b$$

  is equivalent to:

  $$W'x + b'$$

  where $W' = W_5W_4W_3W_2W_1$, meaning model still learns a **linear decision boundary**.
- Since logistic regression is already a linear classifier, this technique does not provide any advantage.

(d) Using a 2-hidden layer feedforward network with ReLU

Yes, this improves training accuracy by allowing non-linear decision boundaries.

- Adding **non-linear activation functions (ReLU)** allows the model to learn **non-linear decision boundaries**.
- This reduces the **bias** of the model and helps capture complex patterns in the data.
- Correct solutions commonly refer to:
  - **Reduced bias** due to higher model capacity.
  - **Non-linear decision boundary**, allowing better class separation.
  - **Increased model complexity**, leading to higher training accuracy.
- **Caveat:** While a deep network can approximate any function (by the Universal Approximation Theorem), it may require an **infeasibly large number of neurons** to separate certain types of distributions.

1.7 Problem 7

1.7.1 Description

The softmax function has the desirable property that it outputs a probability distribution, and is often used as an activation function in many classification neural networks. Consider a 2-layer neural network for K-class classification using softmax activation and cross-entropy loss, as defined below:

$$\mathbf{z}^{[1]} = W^{[1]}\mathbf{x} + \mathbf{b}^{[1]}$$
$$\mathbf{a}^{[1]} = \text{LeakyReLU}(\mathbf{z}^{[1]}, \alpha = 0.01)$$
$$\mathbf{z}^{[2]} = W^{[2]}\mathbf{a}^{[1]} + \mathbf{b}^{[2]}$$
$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{z}^{[2]})$$

$$L = -\sum_{i=1}^{K} y_i \log(\hat{y}_i)$$

where the model is given input $\mathbf{x}$ of shape $D_x \times 1$, and one-hot encoded label $\mathbf{y} \in \{0, 1\}^K$. Assume that the hidden layer has D_a nodes, i.e., $\mathbf{z}^{[1]}$ is a vector of size $D_a \times 1$. Recall the softmax function is computed as follows:

$$\hat{\mathbf{y}} = \left[\frac{\exp(z_1^{[2]})}{Z}, \ldots, \frac{\exp(z_K^{[2]})}{Z}\right]$$

where

$$Z = \sum_{j=1}^{K} \exp(z_j^{[2]})$$

1. What are the shapes of $W^{[2]}, b^{[2]}$? If we were vectorizing across m examples, i.e., using a batch of samples $X \in \mathbb{R}^{D_x \times m}$ as input, what would be the shape of the output of the hidden layer?
2. What is $\frac{\partial \hat{y}_k}{\partial z_k^{[2]}}$? Simplify your answer in terms of element(s) of $\hat{\mathbf{y}}$.
3. What is $\frac{\partial \hat{y}_k}{\partial z_i^{[2]}}$ for $i \neq k$? Simplify your answer in terms of element(s) of $\hat{\mathbf{y}}$.
4. Assume that the label $\mathbf{y}$ has 1 at its k^{th} entry, and 0 elsewhere. What is $\frac{\partial L}{\partial z_i^{[2]}}$? Simplify your answer in terms of $\hat{\mathbf{y}}$. *Hint: Consider both cases where $i = k$ and $i \neq k$.*
5. What is $\frac{\partial z^{[2]}}{\partial a^{[1]}}$? Refer to this result as δ_1.

6. What is $\frac{\partial a^{[1]}}{\partial z^{[1]}}$? Refer to this result as δ_2. Feel free to use curly brace notation.

7. Denote $\frac{\partial L}{\partial z^{[2]}}$ with δ_0. What is $\frac{\partial L}{\partial W^{[1]}}$ and $\frac{\partial L}{\partial b^{[1]}}$? You can reuse notations from previous parts. *Hint: Be careful with the shapes.*

8. To avoid running into issues with numerical stability, one trick may be used when implementing the softmax function. Let $m = \max_{i=1}^{K} z_i$ be the max of z_i, then

$$\hat{y}_i = \frac{\exp(z_i^{[2]} - m)}{\sum_{j=1}^{K} \exp(z_j^{[2]} - m)}$$

   What is the numerical problem with the initial softmax computation? Why does the modified formula help resolve that problem?

1.7.2 Solution

(1) Shapes of $W^{[2]}$ and $b^{[2]}$

The second layer applies the transformation:

$$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$$

where:

- $W^{[2]}$ maps D_a hidden activations to K output classes.
- $b^{[2]}$ provides a bias for each of the K outputs.

Thus, the parameter dimensions are:

$$W^{[2]} \in \mathbb{R}^{K \times D_a}, \quad b^{[2]} \in \mathbb{R}^{K \times 1}$$

For a batch of m inputs:

$$a^{[1]} \in \mathbb{R}^{D_a \times m}$$

resulting in:

$$z^{[2]} \in \mathbb{R}^{K \times m}$$

(2) Computing $\frac{\partial \hat{y}_k}{\partial z_k^{[2]}}$

We need to compute the derivative of the softmax output $\hat{y}_k$ with respect to its corresponding logit $z_k^{[2]}$.

Step 1: Define the Softmax Function

The softmax function is given by:

$$\hat{y}_k = \frac{\exp(z_k^{[2]})}{Z}, \quad \text{where } Z = \sum_{j=1}^{K} \exp(z_j^{[2]})$$

Step 2: Differentiate $\hat{y}_k$ with Respect to $z_k^{[2]}$

Applying the quotient rule:

$$\frac{\partial \hat{y}_k}{\partial z_k^{[2]}} = \frac{\exp(z_k^{[2]})Z - \exp(z_k^{[2]})^2}{Z^2}$$

Factoring out $\exp(z_k^{[2]})$ from the numerator:

$$= \frac{\exp(z_k^{[2]})(Z - \exp(z_k^{[2])})}{Z^2}$$

Since $\hat{y}_k = \frac{\exp(z_k^{[2]})}{Z}$, we rewrite:

$$= \hat{y}_k \cdot \frac{Z - \exp(z_k^{[2]})}{Z}$$

Since $\frac{Z - \exp(z_k^{[2]})}{Z} = 1 - \hat{y}_k$, we obtain this final result:

$$\frac{\partial \hat{y}_k}{\partial z_k^{[2]}} = \hat{y}_k(1 - \hat{y}_k)$$

(3) Computing $\frac{\partial \hat{y}_k}{\partial z_i^{[2]}}$ for $i \neq k$

We need to compute the derivative of the softmax output $\hat{y}_k$ with respect to a different logit $z_i^{[2]}$, where $i \neq k$.

Step 1: Recall the Softmax Function

The softmax function is given by:

$$\hat{y}_k = \frac{\exp(z_k^{[2]})}{Z}, \quad Z = \sum_{j=1}^{K} \exp(z_j^{[2]})$$

which ensures that all probabilities sum to 1.

Step 2: Differentiate $\hat{y}_k$ with Respect to $z_i^{[2]}$

Applying the quotient rule:

$$\frac{\partial \hat{y}_k}{\partial z_i^{[2]}} = \frac{\exp(z_k^{[2]}) \cdot \frac{\partial}{\partial z_i^{[2]}} Z - Z \cdot \frac{\partial}{\partial z_i^{[2]}} \exp(z_k^{[2]})}{Z^2}$$

Since $\frac{\partial}{\partial z_i^{[2]}} \exp(z_k^{[2]}) = 0$, we obtain:

$$\frac{\partial \hat{y}_k}{\partial z_i^{[2]}} = \frac{\exp(z_k^{[2]}) \cdot \exp(z_i^{[2]})}{Z^2}$$

Using the softmax property:

$$\hat{y}_k = \frac{\exp(z_k^{[2]})}{Z}, \quad \hat{y}_i = \frac{\exp(z_i^{[2]})}{Z}$$

Thus, we conclude:

$$\frac{\partial \hat{y}_k}{\partial z_i^{[2]}} = -\hat{y}_k \hat{y}_i$$

(4) Computing $\frac{\partial L}{\partial z_i^{[2]}}$

We need to compute the derivative of the cross-entropy loss L with respect to the logits $z_i^{[2]}$, given that the label y is one-hot encoded.

Step 1: Recall the Cross-Entropy Loss

The cross-entropy loss for a single example is:

$$L = -\sum_{j=1}^{K} y_j \log(\hat{y}_j)$$

Since the label y is one-hot encoded, meaning only $y_k = 1$, we simplify:

$$L = -\log(\hat{y}_k)$$

Step 2: Compute $\frac{\partial L}{\partial \hat{y}_k}$

Differentiating the loss function with respect to the predicted probability $\hat{y}_k$:

$$\frac{\partial L}{\partial \hat{y}_k} = -\frac{1}{\hat{y}_k}$$

Step 3: Apply Chain Rule to Compute $\frac{\partial L}{\partial z_i^{[2]}}$

Using the chain rule:

$$\frac{\partial L}{\partial z_i^{[2]}} = \frac{\partial L}{\partial \hat{y}_k} \cdot \frac{\partial \hat{y}_k}{\partial z_i^{[2]}}$$

Case 1: When $i = k$ (Correct Class Logit)

From softmax differentiation:

$$\frac{\partial \hat{y}_k}{\partial z_k^{[2]}} = \hat{y}_k(1 - \hat{y}_k)$$

Applying the chain rule:

$$\frac{\partial L}{\partial z_k^{[2]}} = -\frac{1}{\hat{y}_k} \cdot \hat{y}_k(1 - \hat{y}_k)$$

$$= -(1 - \hat{y}_k)$$

$$= \hat{y}_k - 1$$

Case 2: When $i \neq k$ (Incorrect Class Logit)

From softmax differentiation:

$$\frac{\partial \hat{y}_k}{\partial z_i^{[2]}} = -\hat{y}_k \hat{y}_i$$

Applying the chain rule:

$$\frac{\partial L}{\partial z_i^{[2]}} = -\frac{1}{\hat{y}_k} \cdot (-\hat{y}_k \hat{y}_i)$$

$$= \hat{y}_i$$

Final Conclusion

Thus, we derived:

$$\frac{\partial L}{\partial z_k^{[2]}} = \hat{y}_k - 1, \quad \frac{\partial L}{\partial z_i^{[2]}} = \hat{y}_i$$

(5): Compute $\boldsymbol{\delta_1}$

From the forward propagation equation:

$$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$$

Since the derivative of a constant term $b^{[2]}$ is 0, we compute:

$$\frac{\partial z^{[2]}}{\partial a^{[1]}} = W^{[2]}$$

Thus, we define:

$$\delta_1 = W^{[2]}$$

This result indicates that the change in $z^{[2]}$ with respect to $a^{[1]}$ is determined by the weight matrix $W^{[2]}$.

(6): Compute $\boldsymbol{\delta_2}$

The activation function used is Leaky ReLU:

$$a_i^{[1]} = \begin{cases} z_i^{[1]}, & \text{if } z_i^{[1]} \geq 0 \\ 0.01 z_i^{[1]}, & \text{if } z_i^{[1]} < 0 \end{cases}$$

Differentiating:

$$\frac{\partial a_i^{[1]}}{\partial z_i^{[1]}} = \begin{cases} 1, & \text{if } z_i^{[1]} \geq 0 \\ 0.01, & \text{otherwise} \end{cases}$$

Thus, we define:

$$\delta_2 = \begin{cases} 1, & \text{if } z_i^{[1]} \geq 0 \\ 0.01, & \text{otherwise} \end{cases}$$

(7): Compute $\frac{\partial L}{\partial W^{[1]}}$ and $\frac{\partial L}{\partial b^{[1]}}$

We need to compute the gradients of the loss function L with respect to the first layer's weight matrix $W^{[1]}$ and bias $b^{[1]}$.

Step 1: Recall Given Notations

From previous derivations:

- $\delta_0 = \frac{\partial L}{\partial z^{[2]}}$, the gradient of the loss function with respect to the second layer's pre-activation.
- $\delta_1 = \frac{\partial z^{[2]}}{\partial a^{[1]}} = W^{[2]}$.
- $\delta_2 = \frac{\partial a^{[1]}}{\partial z^{[1]}}$, which follows the derivative of the Leaky ReLU activation.

Step 2: Compute $\frac{\partial L}{\partial W^{[1]}}$

Applying the chain rule:

$$\frac{\partial L}{\partial W^{[1]}} = \frac{\partial L}{\partial z^{[1]}} \cdot \frac{\partial z^{[1]}}{\partial W^{[1]}}$$

Expanding $\frac{\partial L}{\partial z^{[1]}}$:

$$\frac{\partial L}{\partial z^{[1]}} = \delta_0 \circ \delta_2 \circ \delta_1$$

where $\circ$ represents the Hadamard (element-wise) product.

Since the first layer pre-activation is:

$$z^{[1]} = W^{[1]}x + b^{[1]}$$

we differentiate with respect to $W^{[1]}$:

$$\frac{\partial z^{[1]}}{\partial W^{[1]}} = x^T$$

Thus, we obtain:

$$\frac{\partial L}{\partial W^{[1]}} = \delta_1^T * (\delta_0 \circ \delta_2) * x^T$$

Step 3: Compute $\frac{\partial L}{\partial b^{[1]}}$

Although the bias term $b^{[1]}$ does not depend on the input x, we still need to properly apply the chain rule.

Step 1: Chain Rule for $\frac{\partial L}{\partial b^{[1]}}$

Expanding using the chain rule:

$$\frac{\partial L}{\partial b^{[1]}} = \frac{\partial L}{\partial z^{[1]}} \cdot \frac{\partial z^{[1]}}{\partial b^{[1]}}$$

From the pre-activation equation:

$$z^{[1]} = W^{[1]}x + b^{[1]}$$

taking the derivative with respect to $b^{[1]}$:

$$\frac{\partial z^{[1]}}{\partial b^{[1]}} = I$$

where I is the identity matrix.

Step 2: Compute $\frac{\partial L}{\partial b^{[1]}}$

Using the previous result:

$$\frac{\partial L}{\partial z^{[1]}} = \delta_1^T * (\delta_0 \circ \delta_2)$$

Thus:

$$\frac{\partial L}{\partial b^{[1]}} = I \cdot \left(\delta_1^T * (\delta_0 \circ \delta_2)\right)$$

Since multiplication by identity does not change the vector:

$$\frac{\partial L}{\partial b^{[1]}} = \delta_1^T * (\delta_0 \circ \delta_2)$$

(8): Solution

The softmax function is widely used in multi-class classification but suffers from numerical instability when computed directly. We analyze the problem and provide a stable solution.

Step 1: Understanding the Standard Softmax Function

The softmax function is defined as:

$$\hat{y}_i = \frac{\exp(z_i^{[2]})}{\sum_{j=1}^{K} \exp(z_j^{[2]})}$$

where:

- $z_i^{[2]}$ are the logits (pre-activation values).
- The denominator ensures that all probabilities sum to 1.

However, direct computation of the softmax function can result in numerical issues.

Step 2: Identifying the Numerical Problems

The main issues with the direct computation of softmax are:

1. Overflow Issue (Large $z_i^{[2]}$)

- Exponentiation grows exponentially for large values.
- If $z_i^{[2]}$ is very large (e.g., 1000), then exp(1000) exceeds floating-point limits, leading to overflow errors.
- This makes the denominator approach infinity, distorting the probability calculation.

2. Underflow Issue (Small $z_i^{[2]}$)

- If $z_i^{[2]}$ is very small, exponentiation may round down to zero due to floating-point precision limits.
- This causes division-by-zero errors or inaccurate probabilities.

Step 3: The Numerical Stabilization Trick

To prevent these issues, we modify softmax function by subtracting the maximum logit m:

$$m = \max_{i=1}^{K} z_i^{[2]}$$

The modified softmax function is:

$$\hat{y}_i = \frac{\exp(z_i^{[2]} - m)}{\sum_{j=1}^{K} \exp(z_j^{[2]} - m)}$$

Why Does This Help?

- **Prevents Overflow:**
  - Subtracting m ensures that the largest exponentiation result is $\exp(0) = 1$, preventing large numerical values.
- **Prevents Underflow:**
  - Since all values are relatively close to zero, smaller exponentials do not vanish to zero, avoiding division errors.
- **Maintains Probability Distribution:**
  - Since we subtract a constant from all logits, the relative differences remain unchanged, ensuring correct classification.

1.8 Problem 8

1.8.1 Description

You want to build a classifier that predicts the musical instrument given an audio clip. There are 50 unique types of instruments, and you've collected a dataset of sounds of each, first collecting audio clips of guitars, then clips of violins, and so on for all the different instruments.

1. You use batch gradient descent (BGD) to optimize your loss function, but you have been getting poor training loss. You search your code for potential bugs and realize that you're not shuffling the training data. Would shuffling the training fix this problem? Explain your reasoning. (1-2 sentences)

2. You are deciding whether you should optimize your network parameters using mini-batch gradient descent (MBGD) or stochastic gradient descent (SGD) (i.e. batch size of 1). Give one reason to choose MBGD over SGD. (1-2 sentences)

3. Give one reason to use MBGD over BGD. (1-2 sentences)

4. Label the training loss curves A, B, and C with whether they were likely generated with SGD, MBGD, or BGD *(select one for each)*.

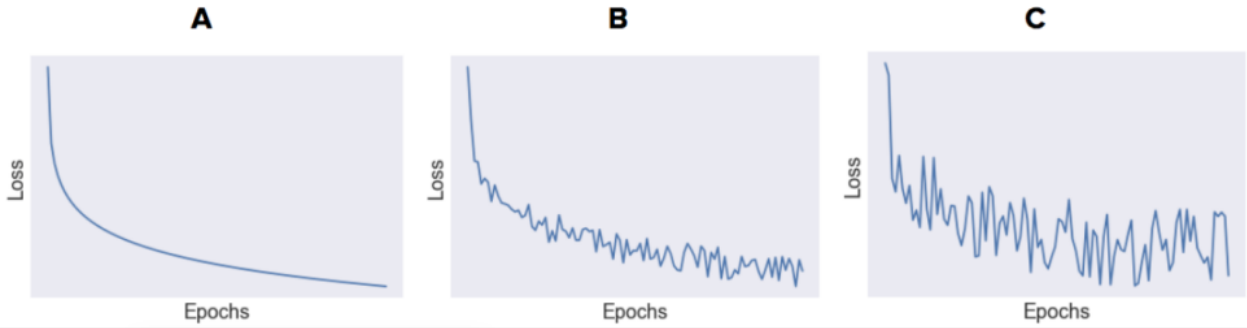


Figure 1.4: Loss curves A, B, and C

5. You decide to tune your model's learning rate. What is one typical sign of a learning rate being too large?

6. What is one typical sign of a learning rate being too small?

7. You now decide to use gradient descent with momentum. For gradient descent with momentum, write down the update rule for a particular trainable weight W. Use learning rate α and momentum hyperparameter β, and let J denote the cost function. (2 equations)

8. Explain how momentum speeds up learning compared to standard gradient descent. (1 sentence)

1.8.2 Solution to 1:

Batch Gradient Descent (BGD) is an optimization method that updates model parameters based on the entire dataset. We analyze whether shuffling affects BGD.

Step 1: Understanding BGD

BGD computes the gradient as:

$$\nabla L = \frac{1}{m} \sum_{i=1}^{m} \nabla L_i$$

where L is the loss function and m is the number of training examples.

- Since **all** data points contribute to each update, BGD does not depend on their order.
- The computed gradient remains the same regardless of how the dataset is arranged.

Step 2: Does Shuffling Affect BGD?

- **No**, because BGD aggregates the loss over the full dataset in each step.
- Changing the order of training data does not alter the total gradient calculation.

Step 3: When Does Shuffling Matter?

- **Stochastic Gradient Descent (SGD)** and **Mini-Batch Gradient Descent** use subsets of the data.
- Without shuffling, SGD and mini-batch may be **biased**, lead to poor generalization.
- In those cases, shuffling ensures batches are diverse and representative.

Final Conclusion

Shuffling is unnecessary for BGD but is other technique like **SGD and Mini-Batch Gradient Descent**.

1.8.3 Solution to 2:

Final Conclusion

MBGD is preferred over SGD because it benefits from **vectorized computation**, making training more **efficient**. Additionally, it **reduces noise in updates**, leading to **smoother and more stable convergence**.

1.8.4 Solution to 3:

MBGD is preferred over BGD because it is **more memory-efficient**, enables **faster convergence**, and helps **avoid poor local optima**. **BGD is impractical for large datasets**, as it requires **too much memory and time** for updates.

We analyze each loss curve and identify which optimization method likely generated it.

1.8.5 Solution to 4:

Step 1: Understanding Gradient Descent Behavior

- **Batch Gradient Descent (BGD)**: Uses the **entire dataset** for each update, resulting in a **smooth, steadily decreasing loss curve**.
- **Mini-Batch Gradient Descent (MBGD)**: Uses **small batches** of data, leading to **some fluctuations** in the loss curve, but it still follows a **general decreasing trend**.
- **Stochastic Gradient Descent (SGD)**: Updates parameters **after each sample**, leading to **highly noisy and fluctuating** loss curves.

Step 2: Identifying the Loss Curves

- **Curve A (Batch GD)**: **Smooth** and **steadily decreasing** loss, indicating the entire dataset was used per update.
- **Curve B (Mini-Batch GD)**: Shows **some fluctuations** but maintains an **overall downward trend**, characteristic of mini-batch updates.
- **Curve C (Stochastic GD)**: **Highly noisy** with large fluctuations, due to frequent updates from single samples.

Final Answer

- **Batch Gradient Descent (BGD) → Curve A**
- **Mini-Batch Gradient Descent (MBGD) → Curve B**
- **Stochastic Gradient Descent (SGD) → Curve C**

1.8.6 Solution to 5 and 6:

We analyze the effects of an improper learning rate and how to detect it.

Step 1: Understanding Learning Rate

- The **learning rate** (α) determines how much the model updates parameters per iteration.
- If α is **too large**, updates **overshoot** the optimal solution.
- If α is **too small**, updates are **tiny**, leading to slow learning.

Step 2: Identifying Signs of an Improper Learning Rate

(i) Learning Rate is Too Large

- **Sign**: The cost function does not converge and may even **diverge**.
- **Why?** Updates are too large, causing **wild oscillations** or increasing loss.
- **Detection**:
  - **Plot cost vs. iterations**.
  - If the cost fluctuates or increases, the learning rate is too large.

(ii) Learning Rate is Too Small

- **Sign**: The cost function **decreases very slowly**.
- **Why?** Updates are **too small**, causing **slow convergence**.
- **Detection**:
  - **Plot cost vs. iterations**.
  - If the cost **decreases too slowly**, try a higher learning rate.

Final Answer

- **Too large** $\rightarrow$ **Wild oscillations** in loss, may **diverge**.
- **Too small** $\rightarrow$ **Slow convergence**, training takes **too long**.

1.8.7 Solution to 7 and 8:

Step 1: Deriving the Momentum Update Rule

Standard gradient descent updates the weight W directly:

$$W = W - \alpha \frac{\partial J}{\partial W} \tag{1.1}$$

However, this approach suffers from **oscillations** in steep valleys and slow convergence when the gradient direction is consistent. To address these issues, we introduce a **momentum term** V:

$$V = \beta V + (1 - \beta) \frac{\partial J}{\partial W} \tag{1.2}$$

$$W = W - \alpha V \tag{1.3}$$

where:

- V is the **velocity term** that stores an exponentially weighted moving average of past gradients.
- β is the **momentum coefficient** (typically 0.9).
- $(1 - \beta)$ determines how much of the **current gradient** contributes to V.
- α is the **learning rate**.

Step 2: Why Momentum Speeds Up Learning

Final Answer: Momentum speeds up learning by canceling out oscillations in the gradients and ensuring updates are performed in the direction of maximum change using exponentially weighted averages of past gradients.