Question 1: Stochastic Gradient Descent Improvements (**10 pts**)

Part 1. (**5 pts**)

Read this blog on medium and describe in your own words how momentum leads to a faster convergence of the loss function.

ANSWER TO PART 1)

Momentum in gradient descent accelerates convergence by reducing oscillations in the optimization path, allowing the algorithm to move more directly toward the minimum of the loss function.

In standard gradient descent, the weight updates are purely based on the current gradient. When optimizing functions with steep, narrow valleys (such as an elongated cost function contour), this approach often results in oscillations in the vertical direction while slowly progressing toward the minimum in the horizontal direction. These oscillations slow down convergence and may limit the learning rate, as a higher learning rate would increase the vertical oscillations, making training unstable.

Momentum addresses this issue by incorporating past gradients into the update process. Instead of using just the current gradient to adjust parameters, it maintains an exponentially weighted moving average of past gradients. This means that updates are influenced by both recent and previous gradients, creating a smoothing effect.

Mathematically, momentum is applied by introducing velocity terms for weight updates:

1. Compute the exponentially weighted average of gradients
   VdW = beta times VdW plus (1 minus beta) times dW
   Vdb = beta times Vdb plus (1 minus beta) times db

   Here, beta (momentum factor) determines how much past gradients influence the current update.

2. Update weights using the velocity term
   W = W minus learning rate times VdW
   b = b minus learning rate times Vdb

By doing this:

- Oscillations in directions with high gradient variance (like the vertical direction in an elongated valley) are dampened, as the momentum averages out these fluctuations.
- The movement in the desired direction (horizontal movement toward the minimum) accumulates speed over iterations, leading to a faster convergence.

In essence, momentum helps the optimizer move more efficiently by smoothing the path toward the minimum, reducing unnecessary oscillations, and allowing for a higher effective learning rate without instability. This leads to a more direct and faster convergence of the loss function compared to standard gradient descent.

Part 2. (**5 pts**)

Read this article and explain in your own words how early stopping helps prevent overfitting.

ANSWER TO PART 2)

Early stopping is a regularization technique that prevents overfitting by halting the training process when the model's performance on a validation set starts to degrade. It balances learning by ensuring that the model generalizes well to unseen data rather than memorizing the training set.

How Early Stopping Prevents Overfitting

During training, the model updates its parameters iteratively to minimize the loss function. Initially, both the training and validation errors decrease, meaning the model is learning useful patterns. However, after a certain point, while the training error continues to decrease, the validation error begins to increase. This indicates that the model is no longer improving its ability to generalize but is instead memorizing specific details of the training data, leading to overfitting.

Early stopping helps by:

1. Identifying the optimal stopping point – It tracks validation performance and halts training at the moment validation error starts increasing.
2. Preventing unnecessary training – Continuing training beyond this point only benefits the training set but hurts generalization.
3. Reducing computational costs – Since the model is stopped early, it requires fewer training iterations, making the process more efficient.

Key Benefits

- **Improves generalization** – The model retains the best-performing parameters before overfitting begins.

- **Reduces training time** – Less computation is needed compared to other regularization methods.
- **Simple to implement** – It requires only monitoring the validation loss over epochs.

Potential Limitations

- **Risk of underfitting** – If training stops too early, the model might not have learned enough useful patterns.
- **Validation set dependency** – A poorly chosen validation set may lead to suboptimal stopping.

In summary, early stopping acts as an implicit regularization method by ensuring the model stops learning at the right moment, achieving a balance between learning useful patterns and preventing memorization of training data.

Question 2: SGD Derivation (**15 pts**)

Use stochastic gradient descent to derive the coefficent updates for the 4 coefficients w_0, w_1, w_2, w_3 in this model:

$$y = w_0 + w_1 log(x_1) + w_2 x_2^2 + w_3 x_1 e^{-x_2}$$

Stochastic Gradient Descent (SGD) Derivation

1 Given Model

We are given the following model:

$$y = w_0 + w_1 \log(x_1) + w_2 x_2^2 + w_3 x_1 e^{-x_2}$$

where we need to derive the coefficient updates for w_0, w_1, w_2, w_3 using **stochastic gradient descent (SGD)**.

2 Gradient Descent Update Rule

The general update rule for SGD is:

$$w_{new} = w_{old} - learningrate \times \frac{\partial L}{\partial w}$$

where we assume the loss function is the **squared error**:

$$L = \frac{1}{2}(y_{pred} - y_{actual})^2$$

Taking the derivative with respect to any weight w_i:

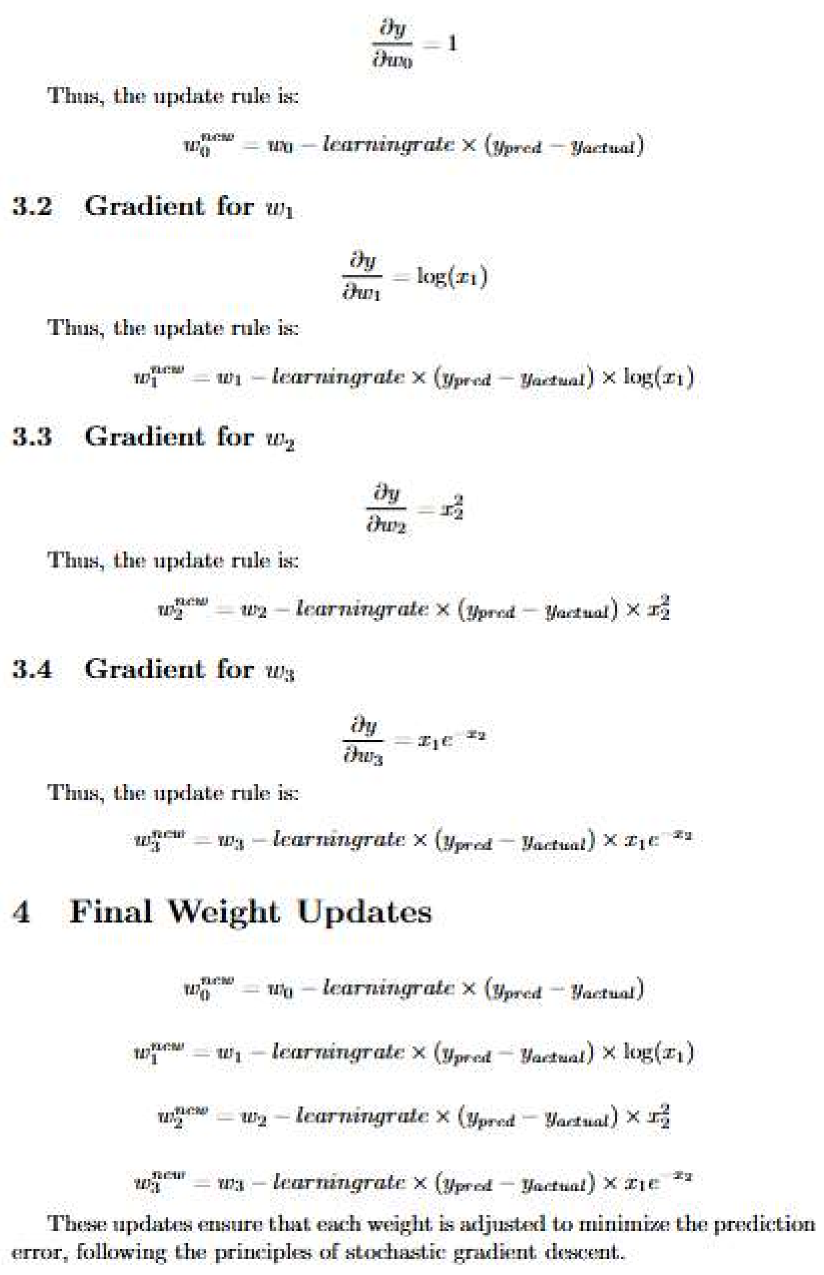
$$\frac{\partial L}{\partial w_i} = (y_{pred} - y_{actual})\frac{\partial y}{\partial w_i}$$

where y_{pred} is the predicted value from the model.

3 Deriving Partial Derivatives

3.1 Gradient for w_0

Since w_0 appears as a direct term:



$$\frac{\partial y}{\partial w_0} = 1$$

Thus, the update rule is:

$$w_0^{new} = w_0 - learningrate \times (y_{pred} - y_{actual})$$

3.2 Gradient for w_1

$$\frac{\partial y}{\partial w_1} = \log(x_1)$$

Thus, the update rule is:

$$w_1^{new} = w_1 - learningrate \times (y_{pred} - y_{actual}) \times \log(x_1)$$

3.3 Gradient for w_2

$$\frac{\partial y}{\partial w_2} = x_2^2$$

Thus, the update rule is:

$$w_2^{new} = w_2 - learningrate \times (y_{pred} - y_{actual}) \times x_2^2$$

3.4 Gradient for w_3

$$\frac{\partial y}{\partial w_3} = x_1 e^{-x_2}$$

Thus, the update rule is:

$$w_3^{new} = w_3 - learningrate \times (y_{pred} - y_{actual}) \times x_1 e^{-x_2}$$

4 Final Weight Updates

$$w_0^{new} = w_0 - learningrate \times (y_{pred} - y_{actual})$$

$$w_1^{new} = w_1 - learningrate \times (y_{pred} - y_{actual}) \times \log(x_1)$$

$$w_2^{new} = w_2 - learningrate \times (y_{pred} - y_{actual}) \times x_2^2$$

$$w_3^{new} = w_3 - learningrate \times (y_{pred} - y_{actual}) \times x_1 e^{-x_2}$$

These updates ensure that each weight is adjusted to minimize the prediction error, following the principles of stochastic gradient descent.

Question 3. (15 points) Tensorflow Playground

In this question, you will be playing with Tensorflow Playground.

Select **Classification** as the Problem Type. Among the four datasets shown in DATA, please select the top left dataset.

Use the following settings as the DEFAULT settings for all subquestions:

- Learning rate = 0.01
- Activation = ReLU
- Regularization = None

- Ratio of training to test data = 50%
- Noise = 0
- Batch Size = 30
- Input features as X_1 and X_2
- One hidden layer with 4 neurons

For all questions below, it is ok if you did not run to the exact number of epochs specified as long as it is within $\pm$ 20 epochs. For example, if you are asked to run 1000 epochs and you ran 1012 epochs, it is fine because it's within 980 and 1020.

Part 1. (5 pts) Effect of activation function (keep other settings the same as in DEFAULT)

- Using ReLU as the activation function
- Using the Linear activation function

(a) Report the train and test losses for both at the end of 1000 epochs.
(b) What qualitative difference do you observe in the decision boundaries obtained and what do you think is the reason for this?

(a)

Activation	ReLU	Linear
Train Loss	.021	.511
Test Loss	.019	0.495

(b) ReLU activation yields a diamond-shaped, nonlinear decision boundary that clearly separates the classes, while a linear activation produces a smoother, nearly straight boundary that results in poor classification. This contrast arises because ReLU introduces nonlinearity, enabling the network to capture more complex patterns. In contrast, a linear activation cannot model nonlinear relationships, limiting the model to simple linear separations that do not suit this dataset. Consequently, ReLU significantly outperforms the linear activation by adapting to the dataset’s structure, whereas the linear activation fails due to its inability to handle nonlinearity.

Part 2. (5 pts) Effect of number of hidden units (keep other settings the same as in DEFAULT)

- Use 2 neurons in the hidden layer
- Use 8 neurons in the hidden layer

(a) Report the train and test losses at the end of 1000 epochs.
(b) What do you observe in terms of the decision boundary obtained as the number of neurons increases and what do you think is the reason for this?

(a)

# Hidden Units	2	8
Train Loss	0.276	0.007
Test Loss	0.252	0.008

(b) As the neuron count grows, the decision boundary becomes more precise and better aligns with the dataset's structure. With only two neurons (first image), the boundary is relatively crude, displaying gaps and a rigid, angular shape. In contrast, with four neurons (second image), the boundary appears smoother and more rounded, closely matching the data distribution.

This enhancement arises because adding more neurons increases the network's ability to capture complex patterns. A larger number of neurons enables the model to approximate the true decision boundary more accurately, reducing classification errors. However, too many neurons can lead to overfitting, causing the model to memorize noise rather than generalize effectively to unseen data.

Part 3. (5 pts) Effect of Learning rate and number of epochs (keep other settings the same as in DEFAULT)

Set the learning rate to the following numbers

- 10
- 0.1
- 0.0001

(a) Report the train and test losses at the end of 100 epochs, 500 epochs and 1000 epochs respectively.
(b) What do you observe from the change of loss vs learning rate, and the change of loss vs epoch numbers? Also report your observations on the training and test loss curve (observe if you see noise for certain learning rates and reason why this is happening).

(a)

Learning rate: 10

Epoch	100	500	1000
Train Loss	0.642	0.605	0.545
Test Loss	0.598	0.562	0.506

(Copy and fill the tables for other learning rates below)

(b) When the learning rate becomes very large, the model struggles to converge, leading to higher loss values. In this case, the model oscillates around the optimal point rather than settling, which causes instability and poor generalization. Conversely, when the learning rate is too low, the loss decreases slowly, requiring more epochs to reach a satisfactory level of accuracy. Although the model can eventually achieve similar performance with enough training, it demands substantially more time. In all scenarios, increasing the number of epochs reduces the loss, but the rate of improvement varies depending on the learning rate.

With a learning rate of 10, the step sizes are so large that the model's loss fluctuates significantly, causing erratic jumps instead of smooth progress.

Question 4: MLP (10 points)

Hint: Use a unit step function $u(t)$ as the activation

Part 1: NOT Function (2 points)

Give a perceptron network that calculates the NOT of a binary input X, ie, $Y = \quad \sim X$. Atleast how many hidden layers do you need?

Part 2: NAND Function (3 points)

Give a perceptron network that calculates the NAND of two binary inputs X_1 and X_2 ie, $Y = NAND(X_1, X_2)$. Atleast how many hidden layers do you need?

Part 3: XOR Function (5 points)

Give a perceptron network that calculates the XOR of two binary inputs X_1 and X_2 ie, $Y = XOR(X_1, X_2)$. Atleast how many hidden layers do you need?

A perceptron network is a type of artificial neural network used to compute boolean logic functions. In this problem, we determine the required architecture for computing the NOT, NAND, and XOR functions using perceptrons with a unit step activation function $u(t)$.

6 Part 1: NOT Function

The NOT function is defined as:

$$Y = \neg X$$

A perceptron can implement the NOT function using a single neuron with appropriate weights and bias. The equation is:

$$Y = u(-X + b)$$

where the bias $b = 0.5$ ensures that when $X = 0$, the output is 1, and when $X = 1$, the output is 0.

Minimum number of hidden layers: 0 (Single-layer perceptron)

7 Part 2: NAND Function

The NAND function is defined as:

$$Y = \neg(X_1 \wedge X_2)$$

A perceptron can implement the NAND function using a single neuron with the equation:

$$Y = u(-w_1 X_1 - w_2 X_2 + b)$$

Choosing $w_1 = w_2 = 1$ and $b = 1.5$ satisfies the NAND truth table.

Minimum number of hidden layers: 0 (Single-layer perceptron)

8 Part 3: XOR Function

The XOR function is defined as:

$$Y = X_1 \oplus X_2$$

Since XOR is not linearly separable, a single-layer perceptron cannot compute it. However, a multi-layer perceptron with at least one hidden layer can represent XOR. The typical implementation involves two neurons in the hidden layer that compute the NAND and OR functions, followed by a neuron in the output layer that computes AND:

$$h_1 = NAND(X_1, X_2) = u(-X_1 - X_2 + 1.5)$$

$$h_2 = OR(X_1, X_2) = u(X_1 + X_2 - 0.5)$$

$$Y = AND(h_1, h_2) = u(h_1 + h_2 - 1.5)$$

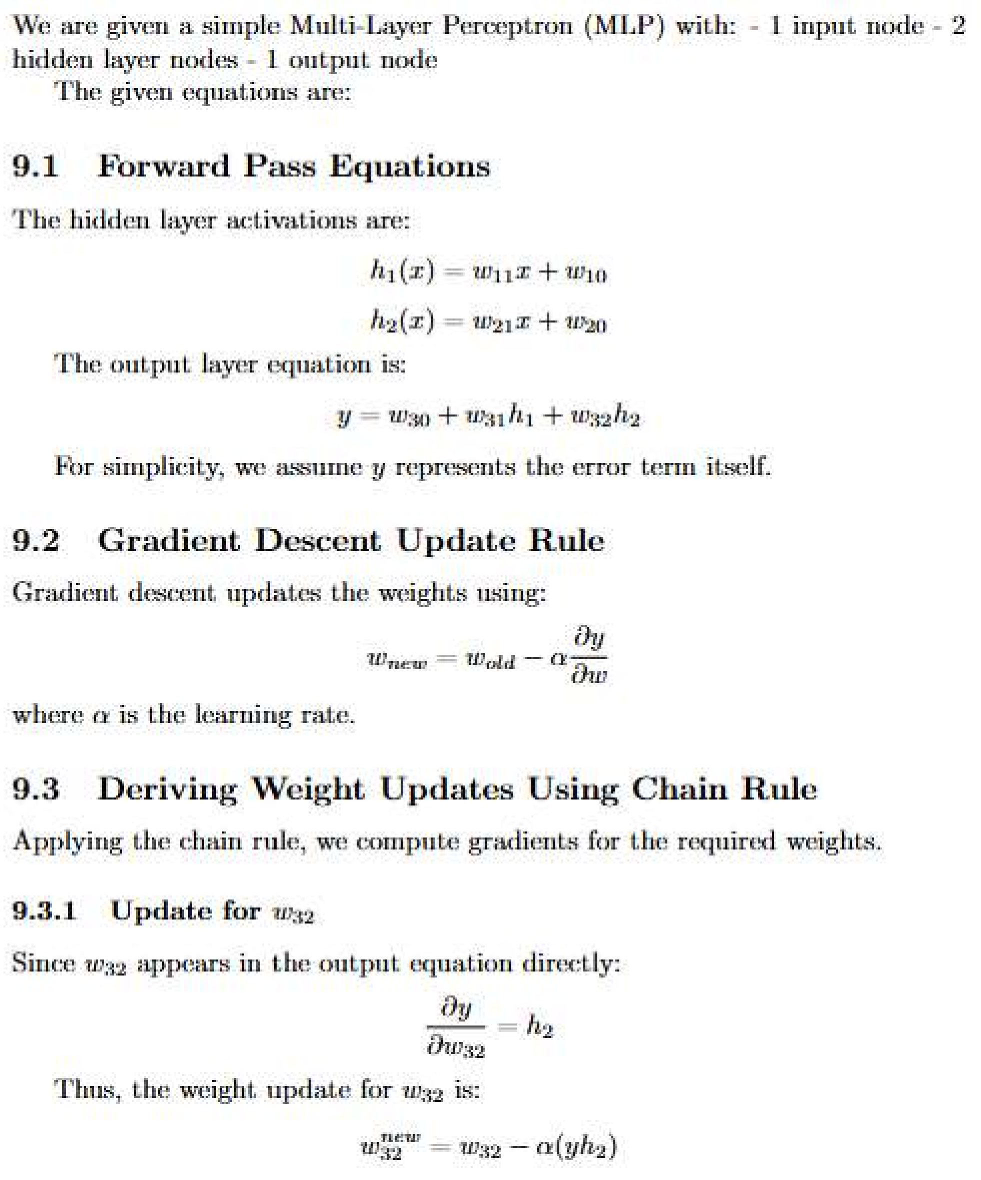
Minimum number of hidden layers: 1 (Multi-layer perceptron required)

Question 5: MLP Backpropagation (15 points)

You are given a simple MLP with 1 node in the input layer, 2 nodes in the hidden layer, and 1 node in the output layer.

$$h_1(x) = w_{11}x + w_{10}$$
$$h_2(x) = w_{21}x + w_{20}$$
$$y = w_{30} + w_{31}h_1 + w_{32}h_2$$

Derive the updates for w_{11}, w_{20} and w_{32} using algebraic expressions. Clearly show how Chain-Rule helps in efficient update calculation for these weights. For simplicity, consider y as the error term itself.



We are given a simple Multi-Layer Perceptron (MLP) with: - 1 input node - 2 hidden layer nodes - 1 output node

The given equations are:

9.1 Forward Pass Equations

The hidden layer activations are:

$$h_1(x) = w_{11}x + w_{10}$$
$$h_2(x) = w_{21}x + w_{20}$$

The output layer equation is:

$$y = w_{30} + w_{31}h_1 + w_{32}h_2$$

For simplicity, we assume y represents the error term itself.

9.2 Gradient Descent Update Rule

Gradient descent updates the weights using:

$$w_{new} = w_{old} - \alpha\frac{\partial y}{\partial w}$$

where α is the learning rate.

9.3 Deriving Weight Updates Using Chain Rule

Applying the chain rule, we compute gradients for the required weights.

9.3.1 Update for w_{32}

Since w_{32} appears in the output equation directly:

$$\frac{\partial y}{\partial w_{32}} = h_2$$

Thus, the weight update for w_{32} is:

$$w_{32}^{new} = w_{32} - \alpha(yh_2)$$

9.3.2 Update for w_{11}

Since w_{11} contributes to y through h_1, we apply the chain rule:

$$\frac{\partial y}{\partial w_{11}} = \frac{\partial y}{\partial h_1} \times \frac{\partial h_1}{\partial w_{11}}$$

From the output equation:

$$\frac{\partial y}{\partial h_1} = w_{31}$$

From the hidden layer equation:

$$\frac{\partial h_1}{\partial w_{11}} = x$$

Thus:

$$\frac{\partial y}{\partial w_{11}} = w_{31}x$$

The weight update for w_{11} is:

$$w_{11}^{new} = w_{11} - \alpha(yw_{31}x)$$

9.3.3 Update for w_{20}

Since w_{20} contributes to y through h_2, we apply the chain rule:

$$\frac{\partial y}{\partial w_{20}} = \frac{\partial y}{\partial h_2} \times \frac{\partial h_2}{\partial w_{20}}$$

From the output equation:

$$\frac{\partial y}{\partial h_2} = w_{32}$$

From the hidden layer equation:

$$\frac{\partial h_2}{\partial w_{20}} = 1$$

Thus:

$$\frac{\partial y}{\partial w_{20}} = w_{32}$$

The weight update for w_{20} is:

$$w_{20}^{new} = w_{20} - \alpha(yw_{32})$$

9.4 Final Weight Update Equations

$$w_{32}^{new} = w_{32} - \alpha(yh_2)$$

$$w_{11}^{new} = w_{11} - \alpha(yw_{31}x)$$

$$w_{20}^{new} = w_{20} - \alpha(yw_{32})$$

9.6 Role of the Chain Rule in Efficient Update Calculation

The chain rule is essential in backpropagation as it enables us to systematically compute gradients by decomposing them into simpler derivative computations. This allows efficient weight updates by propagating the error from the output layer back to the input layer.

For each weight w_i, the update follows the general gradient descent rule:

$$w_i^{new} = w_i - \alpha \frac{\partial y}{\partial w_i}$$

where α is the learning rate.

Since the output y depends on the hidden layer activations h_1 and h_2, we apply the chain rule to compute derivatives for weights that influence these activations.

Chain Rule Application for w_{11} The weight w_{11} affects y through h_1, so we compute its derivative using:

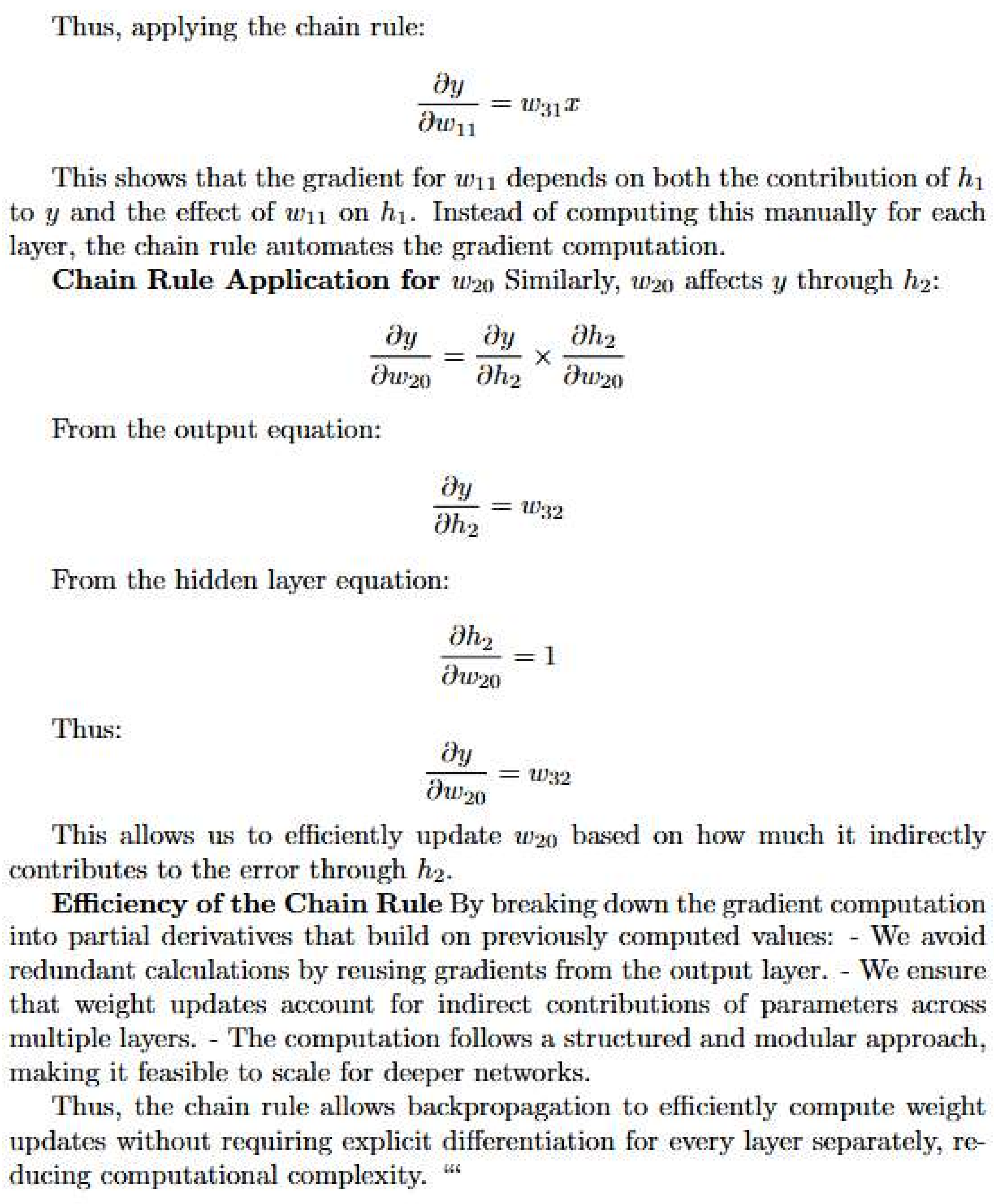
$$\frac{\partial y}{\partial w_{11}} = \frac{\partial y}{\partial h_1} \times \frac{\partial h_1}{\partial w_{11}}$$

From the output equation:

$$\frac{\partial y}{\partial h_1} = w_{31}$$

From the hidden layer equation:

$$\frac{\partial h_1}{\partial w_{11}} = x$$



Thus, applying the chain rule:

$$\frac{\partial y}{\partial w_{11}} = w_{31}x$$

This shows that the gradient for w_{11} depends on both the contribution of h_1 to y and the effect of w_{11} on h_1. Instead of computing this manually for each layer, the chain rule automates the gradient computation.

Chain Rule Application for w_{20} Similarly, w_{20} affects y through h_2:

$$\frac{\partial y}{\partial w_{20}} = \frac{\partial y}{\partial h_2} \times \frac{\partial h_2}{\partial w_{20}}$$

From the output equation:

$$\frac{\partial y}{\partial h_2} = w_{32}$$

From the hidden layer equation:

$$\frac{\partial h_2}{\partial w_{20}} = 1$$

Thus:

$$\frac{\partial y}{\partial w_{20}} = w_{32}$$

This allows us to efficiently update w_{20} based on how much it indirectly contributes to the error through h_2.

Efficiency of the Chain Rule By breaking down the gradient computation into partial derivatives that build on previously computed values: - We avoid redundant calculations by reusing gradients from the output layer. - We ensure that weight updates account for indirect contributions of parameters across multiple layers. - The computation follows a structured and modular approach, making it feasible to scale for deeper networks.

Thus, the chain rule allows backpropagation to efficiently compute weight updates without requiring explicit differentiation for every layer separately, reducing computational complexity. "‘

Question 6: SGD Variations (10 points)

Read this blog on SGD variants and provide short descriptions about your understanding of:

1. Adagrad
2. Adadelta
3. RMSProp
4. Adam

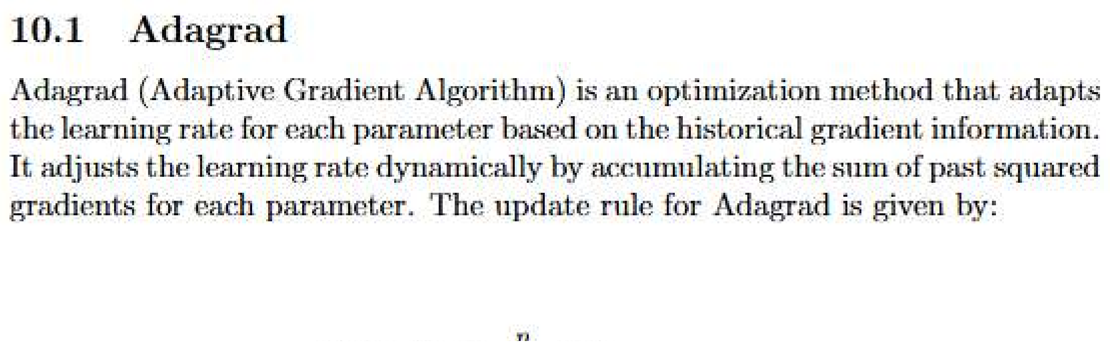
Gradient descent is a fundamental optimization technique used to minimize an objective function by iteratively updating parameters in the direction of the negative gradient. However, standard gradient descent methods can face challenges such as slow convergence, diminishing learning rates, and difficulty navigating complex loss surfaces. To address these issues, several adaptive optimization algorithms have been developed. Below are detailed descriptions of four widely used gradient descent optimization algorithms:

1. Adagrad (Adaptive Gradient Algorithm)

Adagrad adapts the learning rate for each parameter individually based on the historical gradients. It accumulates the sum of squared gradients over time and adjusts the learning rate accordingly. This approach allows Adagrad to perform larger updates for infrequent features and smaller updates for frequent ones, making it particularly useful for sparse data.

Update Rule:

1. Compute the sum of squared gradients up to time step **t**: G_t,i = G_t-1,i + (gradient_t,i)^2



10.1 Adagrad

Adagrad (Adaptive Gradient Algorithm) is an optimization method that adapts the learning rate for each parameter based on the historical gradient information. It adjusts the learning rate dynamically by accumulating the sum of past squared gradients for each parameter. The update rule for Adagrad is given by: