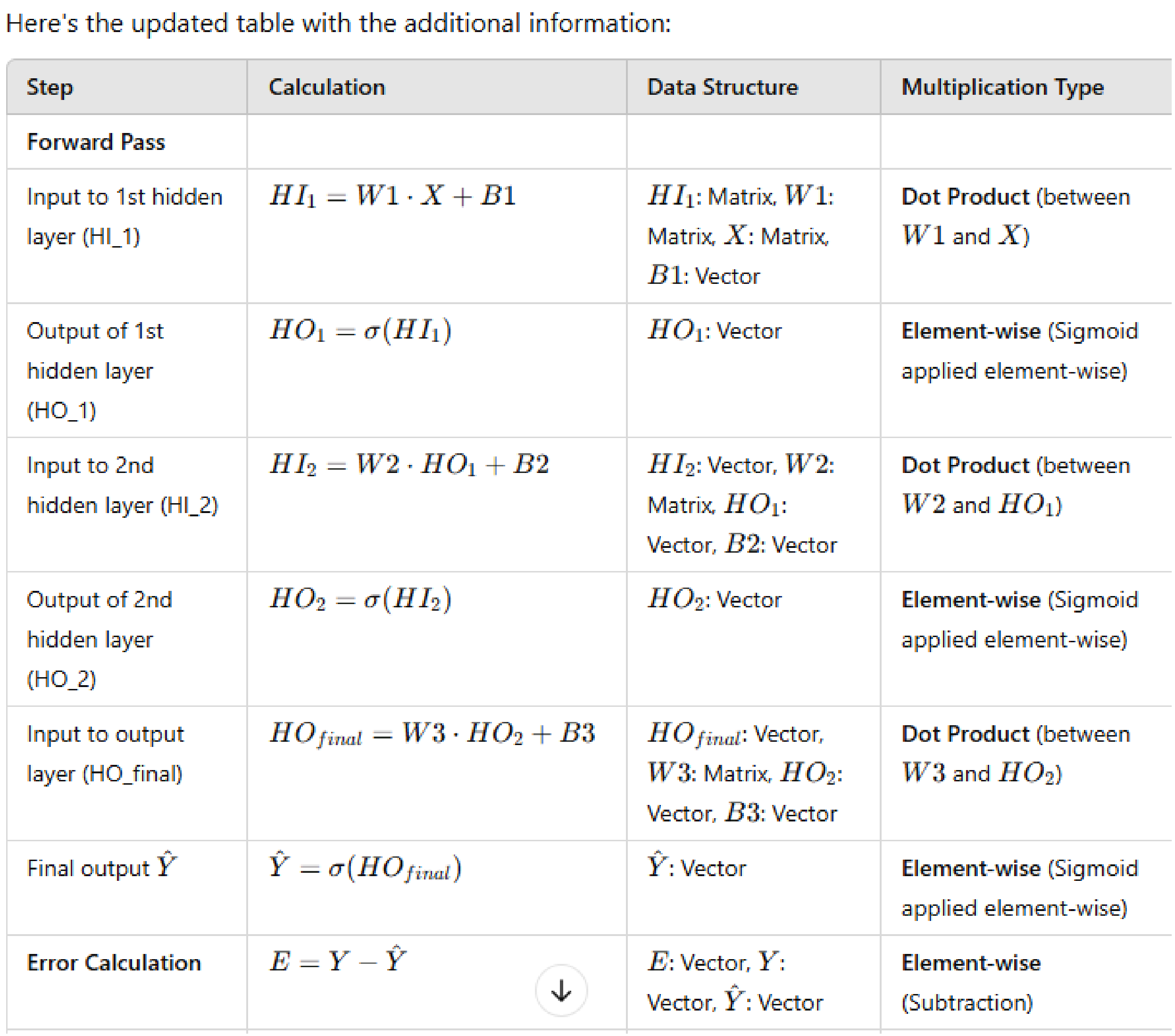
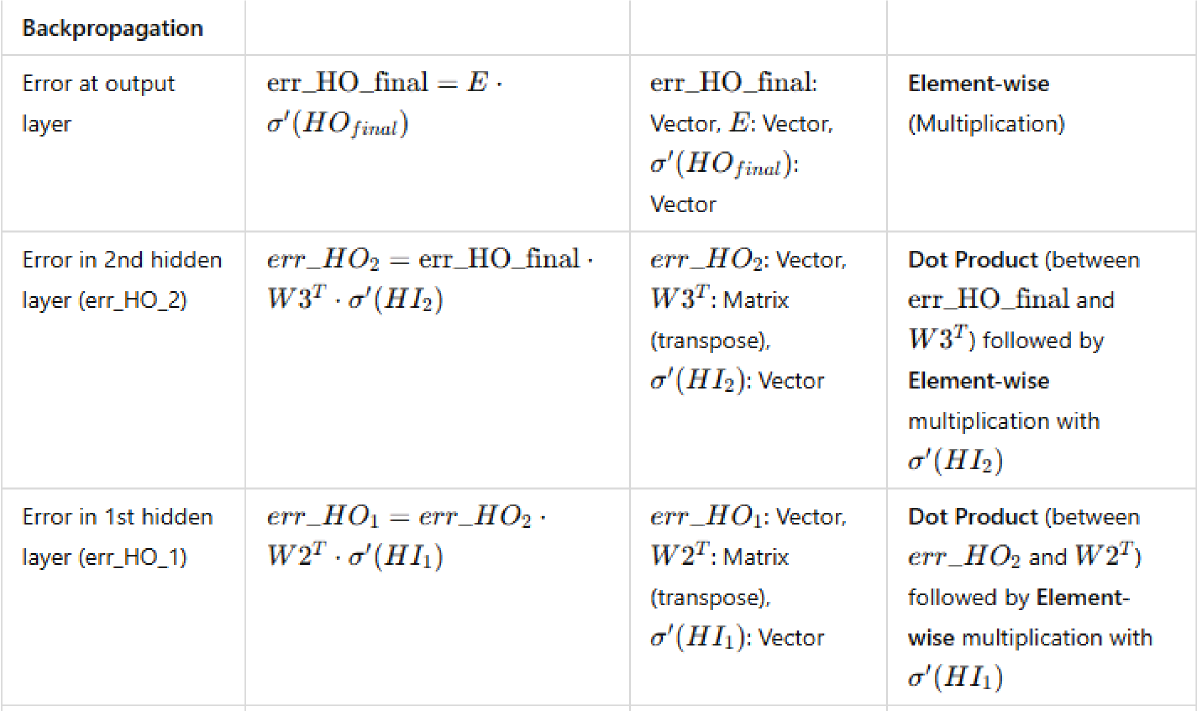
Forward pass



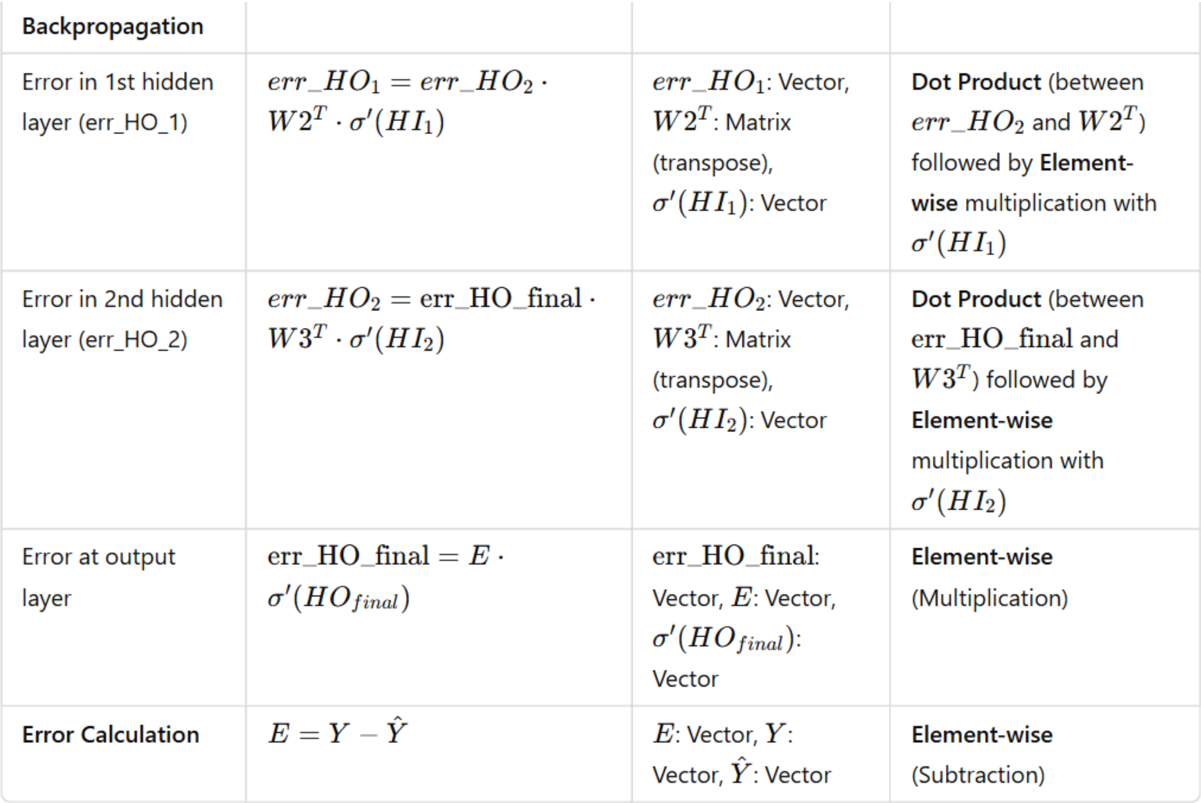
Here's the updated table with the additional information:

Step	Calculation	Data Structure	Multiplication Type
Forward Pass			
Input to 1st hidden layer (HI_1)	$HI_1 = W1 \cdot X + B1$	HI_1: Matrix, $W1$: Matrix, X: Matrix, $B1$: Vector	**Dot Product** (between $W1$ and X)
Output of 1st hidden layer (HO_1)	$HO_1 = \sigma(HI_1)$	HO_1: Vector	**Element-wise** (Sigmoid applied element-wise)
Input to 2nd hidden layer (HI_2)	$HI_2 = W2 \cdot HO_1 + B2$	HI_2: Vector, $W2$: Matrix, HO_1: Vector, $B2$: Vector	**Dot Product** (between $W2$ and HO_1)
Output of 2nd hidden layer (HO_2)	$HO_2 = \sigma(HI_2)$	HO_2: Vector	**Element-wise** (Sigmoid applied element-wise)
Input to output layer (HO_final)	$HO_{final} = W3 \cdot HO_2 + B3$	HO_{final}: Vector, $W3$: Matrix, HO_2: Vector, $B3$: Vector	**Dot Product** (between $W3$ and HO_2)
Final output $\hat{Y}$	$\hat{Y} = \sigma(HO_{final})$	$\hat{Y}$: Vector	**Element-wise** (Sigmoid applied element-wise)
Error Calculation	$E = Y - \hat{Y}$	E: Vector, Y: Vector, $\hat{Y}$: Vector	**Element-wise** (Subtraction)

Backpropagation



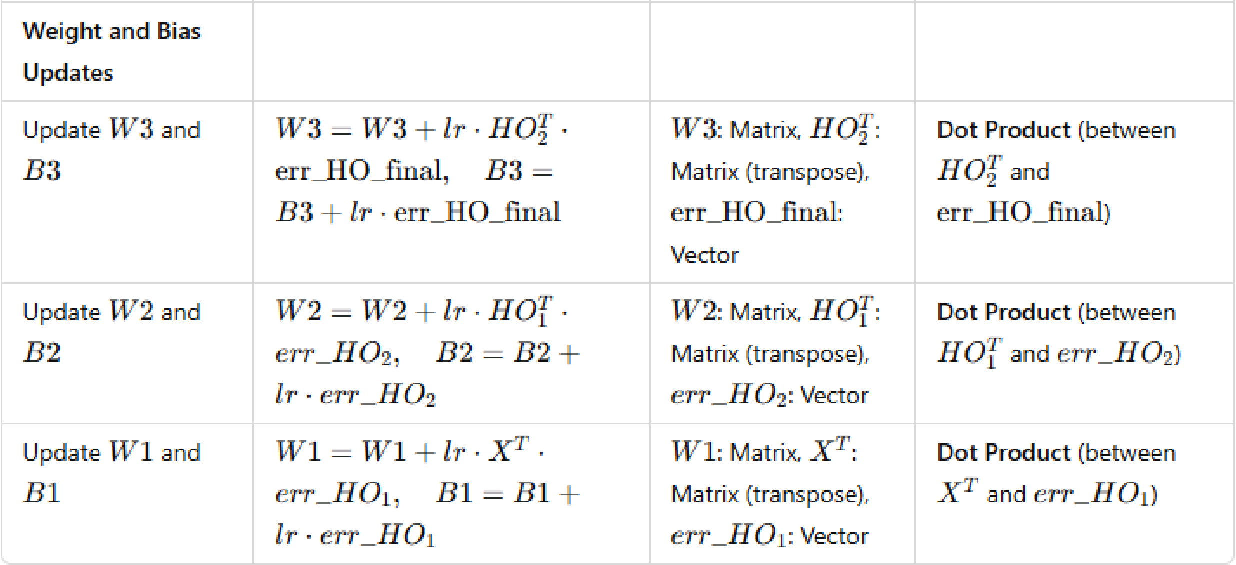
Backpropagation			
Error at output layer	$\text{err\_HO\_final} = E \cdot \sigma'(HO_{final})$	err_HO_final: Vector, E: Vector, $\sigma'(HO_{final})$: Vector	**Element-wise** (Multiplication)
Error in 2nd hidden layer (err_HO_2)	$err\_HO_2 = \text{err\_HO\_final} \cdot W3^T \cdot \sigma'(HI_2)$	err_HO_2: Vector, $W3^T$: Matrix (transpose), $\sigma'(HI_2)$: Vector	**Dot Product** (between err_HO_final and $W3^T$) followed by **Element-wise** multiplication with $\sigma'(HI_2)$
Error in 1st hidden layer (err_HO_1)	$err\_HO_1 = err\_HO_2 \cdot W2^T \cdot \sigma'(HI_1)$	err_HO_1: Vector, $W2^T$: Matrix (transpose), $\sigma'(HI_1)$: Vector	**Dot Product** (between err_HO_2 and $W2^T$) followed by **Element-wise** multiplication with $\sigma'(HI_1)$



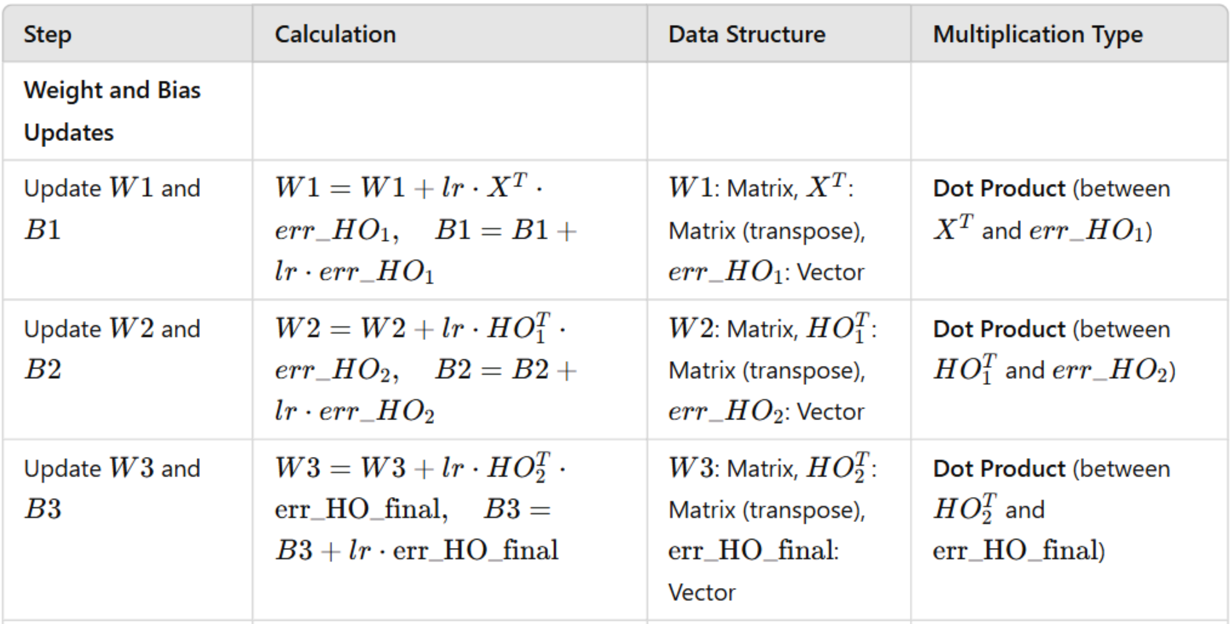
Backpropagation			
Error in 1st hidden layer (err_HO_1)	$err\_HO_1 = err\_HO_2 \cdot W2^T \cdot \sigma'(HI_1)$	err_HO_1: Vector, $W2^T$: Matrix (transpose), $\sigma'(HI_1)$: Vector	**Dot Product** (between err_HO_2 and $W2^T$) followed by **Element-wise** multiplication with $\sigma'(HI_1)$
Error in 2nd hidden layer (err_HO_2)	$err\_HO_2 = \text{err\_HO\_final} \cdot W3^T \cdot \sigma'(HI_2)$	err_HO_2: Vector, $W3^T$: Matrix (transpose), $\sigma'(HI_2)$: Vector	**Dot Product** (between err_HO_final and $W3^T$) followed by **Element-wise** multiplication with $\sigma'(HI_2)$
Error at output layer	$\text{err\_HO\_final} = E \cdot \sigma'(HO_{final})$	err_HO_final: Vector, E: Vector, $\sigma'(HO_{final})$: Vector	**Element-wise** (Multiplication)
Error Calculation	$E = Y - \hat{Y}$	E: Vector, Y: Vector, $\hat{Y}$: Vector	**Element-wise** (Subtraction)

Error_ Layer_N = sigma_d(Input_N) (Error_ Layer_N+1) (W_N+1)^{T}	**Input_N = WX +B**

Weight and Bias update



Weight and Bias Updates			
Update $W3$ and $B3$	$W3 = W3 + lr \cdot HO_2^T \cdot \mathrm{err\_HO\_final}, \quad B3 = B3 + lr \cdot \mathrm{err\_HO\_final}$	$W3$: Matrix, HO_2^T: Matrix (transpose), $\mathrm{err\_HO\_final}$: Vector	**Dot Product** (between HO_2^T and $\mathrm{err\_HO\_final}$)
Update $W2$ and $B2$	$W2 = W2 + lr \cdot HO_1^T \cdot err\_HO_2, \quad B2 = B2 + lr \cdot err\_HO_2$	$W2$: Matrix, HO_1^T: Matrix (transpose), err_HO_2: Vector	**Dot Product** (between HO_1^T and err_HO_2)
Update $W1$ and $B1$	$W1 = W1 + lr \cdot X^T \cdot err\_HO_1, \quad B1 = B1 + lr \cdot err\_HO_1$	$W1$: Matrix, X^T: Matrix (transpose), err_HO_1: Vector	**Dot Product** (between X^T and err_HO_1)



Step	Calculation	Data Structure	Multiplication Type
Weight and Bias Updates			
Update $W1$ and $B1$	$W1 = W1 + lr \cdot X^T \cdot err\_HO_1, \quad B1 = B1 + lr \cdot err\_HO_1$	$W1$: Matrix, X^T: Matrix (transpose), err_HO_1: Vector	**Dot Product** (between X^T and err_HO_1)
Update $W2$ and $B2$	$W2 = W2 + lr \cdot HO_1^T \cdot err\_HO_2, \quad B2 = B2 + lr \cdot err\_HO_2$	$W2$: Matrix, HO_1^T: Matrix (transpose), err_HO_2: Vector	**Dot Product** (between HO_1^T and err_HO_2)
Update $W3$ and $B3$	$W3 = W3 + lr \cdot HO_2^T \cdot \mathrm{err\_HO\_final}, \quad B3 = B3 + lr \cdot \mathrm{err\_HO\_final}$	$W3$: Matrix, HO_2^T: Matrix (transpose), $\mathrm{err\_HO\_final}$: Vector	**Dot Product** (between HO_2^T and $\mathrm{err\_HO\_final}$)

W_new = W_old + lr* Own_Error * Own_Input ^{T}	Own_Input = X
	Own_Input_N = HO_N-1
	= Sigma (HI_N-1)

Yes, the general structure of **forward pass** and **backpropagation** is the same for all basic neural networks, regardless of how many hidden layers or neurons you have. The following principles hold true for most **feedforward neural networks** (also known as multilayer perceptrons or MLPs):

1. Forward Pass:

- **Basic Structure**:
  - The network consists of an input layer, one or more hidden layers, and an output layer.
  - Each layer computes a weighted sum of the inputs (or the output of the previous layer), adds a bias, and applies an activation function (like sigmoid, ReLU, or tanh) to produce an output.
- **Computation Flow**:
  - Data flows forward through the network from the input layer to the output layer.
  - The output of one layer becomes the input to the next layer.

2. Backpropagation:

- **Error Propagation**:
  - After computing the final output during the forward pass, the network compares the predicted output with the expected output (target).
  - The **error** is calculated (usually with a loss function like mean squared error or cross-entropy).
  - The error is propagated backward from the output layer to the hidden layers, adjusting the weights of the neurons to minimize the error.
- **Weight Update**:
  - The gradients (rate of change of error with respect to weights) are computed using the **chain rule** of calculus.
  - The weights and biases are adjusted to reduce the error using **gradient descent** (or its variants like stochastic gradient descent).
- **General Steps**:
  - Compute error at the output layer.
  - Backpropagate the error to each preceding layer, updating weights and biases.

Yes, the Process is the Same:

- The process of **forward pass** and **backpropagation** described earlier holds for all basic neural networks, regardless of:
  - The number of **hidden layers**.
  - The number of **neurons** per layer.
  - The type of **activation function** used.

Variations:

While the core steps remain the same, there are some variations in the types of networks and specific techniques:

- **Activation Functions**: Instead of sigmoid, modern neural networks often use **ReLU** (Rectified Linear Unit) or **tanh** as the activation function in hidden layers.
- **Loss Functions**: Different tasks use different loss functions. For example, classification problems typically use **cross-entropy loss**, while regression problems might use **mean squared error (MSE)**.
- **Learning Algorithms**: The way weights are updated can vary. Standard **gradient descent** may be replaced by more sophisticated algorithms like **Adam**, **RMSprop**, or **Adagrad**, which adaptively adjust the learning rate.

Summary:

- The **concept of forward pass and backpropagation** is **universal** to most feedforward neural networks.
- The main difference lies in the architecture (number of layers, neurons) and the type of activation, loss function, and optimization algorithm used.
- As your network becomes more complex (more layers, neurons), the same principles apply, but with more layers to propagate through. For example:
    - **Deep Neural Networks** (DNNs) have more hidden layers, but the core process of forward propagation and backpropagation is identical.
    - **Convolutional Neural Networks (CNNs)** and **Recurrent Neural Networks (RNNs)** also use forward and backward propagation, though with some specialized modifications.

In essence, for any **basic feedforward neural network**, this process of forward pass and backpropagation will remain the same.

Sure! Let's break down the weight update equations for standard **gradient descent** and compare them with more sophisticated optimization algorithms like **Adam**, **RMSprop**, and **Adagrad**. Each algorithm improves upon standard gradient descent by introducing mechanisms that adapt the learning rate or smooth out gradients to improve convergence.

1. Standard Gradient Descent

The basic idea of gradient descent is to update the weights and biases by moving them in the direction that minimizes the loss function. The update rule is:

W=W−η·∇WW = W - \eta \cdot \nabla WW=W−η·∇W

Where:

- WWW is the weight matrix.
- η\etaη is the **learning rate** (a small positive scalar that controls the size of the update step).
- ∇W\nabla W∇W is the **gradient** of the loss function with respect to the weights WWW.

The gradient tells us how much the weights should change to reduce the error. The learning rate controls how large the steps are as the weights are adjusted.

---

2. Adagrad (Adaptive Gradient Algorithm)

Adagrad modifies the learning rate based on the history of gradients. It adapts the learning rate for each parameter individually, so frequently updated parameters have smaller learning rates, and infrequent ones have larger learning rates.

The update rule is:

W=W−ηGt+ϵ·∇WW = W - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot \nabla WW=W−Gt+ϵη·∇W

Where:

- Gt=∑i=1t(∇Wi)2G_t = \sum_{i=1}^{t} (\nabla W_i)^2Gt=∑i=1t(∇Wi)2 is the sum of squares of past gradients for weight WWW.
- ϵ\epsilonϵ is a small constant (usually 10−810^{-8}10−8) added to prevent division by zero.
- ∇W\nabla W∇W is the current gradient.
- The learning rate η\etaη is scaled inversely proportional to the sum of the squares of the gradients.

Key Insight: Adagrad reduces the learning rate for parameters that have large gradients and allows for larger updates to parameters with small gradients, which is helpful for sparse data.

3. RMSprop (Root Mean Square Propagation)

RMSprop is a refinement of Adagrad that controls the accumulation of past gradients by using an exponentially decaying average, preventing the learning rate from shrinking too much over time.

The update rule is:

St=ρSt−1+(1−ρ)(∇Wt)2S_t = \rho S_{t-1} + (1 - \rho) (\nabla W_t)^2St=ρSt−1+(1−ρ)(∇Wt)2
W=W−ηSt+ϵ⋅∇WW = W - \frac{\eta}{\sqrt{S_t + \epsilon}} \cdot \nabla WW=W−St+ϵη⋅∇W

Where:

- StS_tSt is the exponentially weighted average of the squared gradients (like a moving average).
- ρ\rhoρ is the decay rate (usually set around 0.9).
- ϵ\epsilonϵ is a small constant added to avoid division by zero.
- ∇Wt\nabla W_t∇Wt is the current gradient at time ttt.

Key Insight: RMSprop maintains a balance between reducing the learning rate for parameters with large gradients and keeping it from shrinking too much over time. It works well for non-stationary problems like deep learning tasks.

4. Adam (Adaptive Moment Estimation)

Adam combines the ideas of **momentum** and **RMSprop** to create an adaptive learning rate optimization algorithm. It maintains both an exponentially decaying average of past gradients (first moment) and squared gradients (second moment).

The update rule involves two steps:

1. **First Moment (mean of gradients)**:

   mt=β1mt−1+(1−β1)∇Wtm_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla W_tmt=β1mt−1+(1−β1)∇Wt

   - mtm_tmt is the first moment (moving average of the gradient).
   - β1\beta_1β1 is the exponential decay rate for the first moment, typically β1=0.9\beta_1 = 0.9β1=0.9.
2. **Second Moment (mean of squared gradients)**:

   vt=β2vt−1+(1−β2)(∇Wt)2v_t = \beta_2 v_{t-1} + (1 - \beta_2) (\nabla W_t)^2vt=β2vt−1+(1−β2)(∇Wt)2

- vtv_tvt is the second moment (moving average of the squared gradient).
- β2\beta_2β2 is the exponential decay rate for the second moment, typically β2=0.999\beta_2 = 0.999β2=0.999.

3. **Bias Correction**: Since mtm_tmt and vtv_tvt are initialized to zero, they are biased towards zero at the beginning. To correct this, Adam introduces bias-corrected estimates:

   mt^=mt1−β1t,vt^=vt1−β2t\hat{m_t} = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v_t} = \frac{v_t}{1 - \beta_2^t}mt^=1−β1tmt,vt^=1−β2tvt

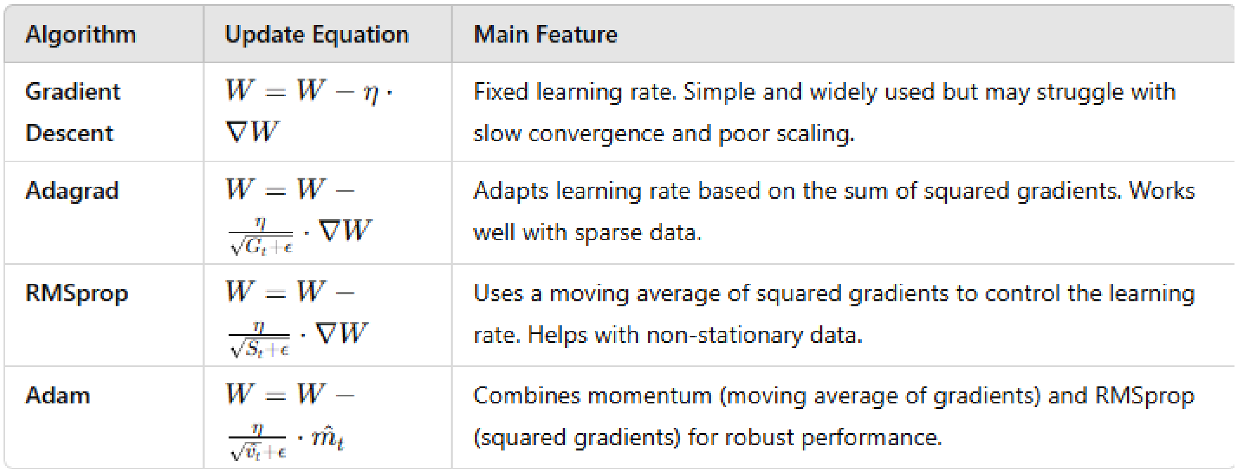
4. **Final Update**:

   W=W−ηvt^+ϵ⋅mt^W = W - \frac{\eta}{\sqrt{\hat{v_t}} + \epsilon} \cdot \hat{m_t}W=W−vt^+ϵη⋅mt^

   - mt^\hat{m_t}mt^ is the bias-corrected first moment.
   - vt^\hat{v_t}vt^ is the bias-corrected second moment.
   - ϵ\epsilonϵ is a small constant (usually 10−810^{-8}10−8) to prevent division by zero.

Key Insight: Adam adapts the learning rate based on both the **first moment** (mean of gradients) and the **second moment** (variance of gradients), making it highly effective for a wide variety of problems, especially in deep learning.

Optimizer Comparison Summary:



Algorithm	Update Equation	Main Feature
Gradient Descent	$W = W - \eta \cdot \nabla W$	Fixed learning rate. Simple and widely used but may struggle with slow convergence and poor scaling.
Adagrad	$W = W - \frac{\eta}{\sqrt{G_t+\epsilon}} \cdot \nabla W$	Adapts learning rate based on the sum of squared gradients. Works well with sparse data.
RMSprop	$W = W - \frac{\eta}{\sqrt{S_t+\epsilon}} \cdot \nabla W$	Uses a moving average of squared gradients to control the learning rate. Helps with non-stationary data.
Adam	$W = W - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon} \cdot \hat{m}_t$	Combines momentum (moving average of gradients) and RMSprop (squared gradients) for robust performance.

Conclusion:

- **Gradient Descent**: Simple and effective, but requires manual tuning of the learning rate.
- **Adagrad**: Adapts learning rate for each parameter, but can shrink the learning rate too much over time.
- **RMSprop**: Prevents the learning rate from decaying too much by using a moving average of squared gradients.

- **Adam**: Combines the advantages of both momentum and adaptive learning rates, making it a popular choice for many deep learning applications.

Adam is often the default optimizer in practice because it converges faster and more reliably on a wide range of deep learning problems. However, each algorithm has its strengths depending on the problem being solved.

In machine learning, **L1** and **L2 norms** refer to different ways of regularizing a model to prevent overfitting. These norms are applied as part of the **regularization** process, which adds a penalty to the model's loss function based on the magnitude of the weights. This encourages simpler models with smaller weights, leading to better generalization.

1. L1 Regularization (Lasso)

The **L1 norm** regularization (also called **Lasso** in linear models) adds the **absolute value** of the weights to the loss function. It helps promote sparsity in the model, meaning that some of the weights may become exactly zero, effectively removing certain features from the model.

L1 Regularization Formula:

For a weight vector WWW, the L1 norm is the sum of the absolute values of the weights:

L1_penalty=λ∑i=1n|Wi|L1_penalty = \lambda \sum_{i=1}^{n} |W_i|L1_penalty=λi=1∑n|Wi|

Where:

- WiW_iWi are the individual weights.
- λ\lambdaλ (lambda) is the **regularization strength** (hyperparameter that controls how much regularization is applied).

Regularized Loss Function:

The regularized loss function with L1 is:

L(y^,y)=Loss(y^,y)+λ∑i=1n|Wi|L(\hat{y}, y) = Loss(\hat{y}, y) + \lambda \sum_{i=1}^{n} |W_i|L(y^,y)=Loss(y^,y)+λi=1∑n|Wi|

Where:

- Loss(y^,y)Loss(\hat{y}, y)Loss(y^,y) is the original loss function (like mean squared error or cross-entropy).
- The second term is the L1 regularization penalty.

Effect of L1 Regularization:

- **Sparsity**: L1 regularization tends to drive certain weights to **exactly zero**, leading to sparse models. This is particularly useful for **feature selection**, as it can effectively remove less important features from the model.
- **Feature Elimination**: By setting some weights to zero, L1 regularization effectively "eliminates" certain features, helping models become simpler and more interpretable.

---

2. L2 Regularization (Ridge)

The **L2 norm** regularization (also called **Ridge** in linear models) adds the **squared magnitude** of the weights to the loss function. Unlike L1, L2 does not promote sparsity (i.e., weights do not become exactly zero), but it penalizes large weights more severely.

L2 Regularization Formula:

For a weight vector WWW, the L2 norm is the sum of the squared weights:

L2_penalty=λ∑i=1nWi2L2_penalty = \lambda \sum_{i=1}^{n} W_i^2L2_penalty=λi=1∑nWi2

Where:

- WiW_iWi are the individual weights.
- λ\lambdaλ (lambda) is the regularization strength (hyperparameter).

Regularized Loss Function:

The regularized loss function with L2 is:

L(y^,y)=Loss(y^,y)+λ∑i=1nWi2L(\hat{y}, y) = Loss(\hat{y}, y) + \lambda \sum_{i=1}^{n} W_i^2L(y^,y)=Loss(y^,y)+λi=1∑nWi2

Where:

- Loss(y^,y)Loss(\hat{y}, y)Loss(y^,y) is the original loss function.
- The second term is the L2 regularization penalty.

Effect of L2 Regularization:

- **No Sparsity**: L2 regularization does not set weights to zero but rather reduces the magnitude of all weights, making the model simpler and reducing overfitting.
- **Smoothness**: L2 regularization encourages smaller weights but does not eliminate any, leading to more **smoothness** and **stability** in the model.

---

3. Elastic Net

Elastic Net is a combination of **L1 and L2 regularization**. It applies both penalties to the loss function, allowing for both feature selection (through L1) and weight shrinkage (through L2).

Elastic Net Formula:

The regularized loss function for Elastic Net is:

L(y^,y)=Loss(y^,y)+λ1∑i=1n|Wi|+λ2∑i=1nWi2L(\hat{y}, y) = Loss(\hat{y}, y) + \lambda_1 \sum_{i=1}^{n} |W_i| + \lambda_2 \sum_{i=1}^{n} W_i^2L(y^,y)=Loss(y^,y)+λ1i=1∑n|Wi |+λ2i=1∑nWi2

Where:

- λ1\lambda_1λ1 controls the strength of L1 regularization.
- λ2\lambda_2λ2 controls the strength of L2 regularization.

Effect of Elastic Net:

- **Combines the benefits of both L1 and L2**: It can both eliminate features (L1) and shrink non-zero weights (L2).
- **Useful for correlated features**: Elastic Net is often preferred when the data has **highly correlated features**, where L1 regularization alone might perform poorly.

Use Cases in Machine Learning:

- **L1 Regularization (Lasso)**:
  - Useful for problems where **feature selection** is important (e.g., when you have many irrelevant features).
  - Commonly used in models like **Lasso regression**.
- **L2 Regularization (Ridge)**:
  - Used when you want to reduce overfitting without eliminating any features.
  - Helps in creating smoother models by shrinking weights uniformly.
  - Commonly used in models like **Ridge regression** and neural networks.
- **Elastic Net**:
  - Preferred when you need both **feature selection** and **weight shrinkage**, especially when features are **correlated**.
  - Balances between L1 and L2 regularization.
  - Often used in high-dimensional datasets where L1 alone might fail.

In Neural Networks:

- L1 and L2 regularization (or Elastic Net) can be added to neural networks to reduce overfitting and encourage sparsity or smoothness in the model.
- In deep learning frameworks like TensorFlow or PyTorch, you can easily apply L1/L2 regularization by adding it to the loss function during training.

Summary:

- **L1 Regularization (Lasso)** encourages sparsity by shrinking weights to zero, making it useful for feature selection.
- **L2 Regularization (Ridge)** shrinks weights uniformly, encouraging smoothness and reducing overfitting without eliminating features.
- **Elastic Net** combines the strengths of both L1 and L2, making it useful when there are correlated features and both feature selection and regularization are needed.

These regularization techniques help control the complexity of the model, especially when working with high-dimensional data or preventing overfitting.

Comparison of L1, L2, and Elastic Net:

Type	Penalty Term	Effect on Weights	Use Case
L1 (Lasso)	(\lambda \sum_{i=1}^{n}	W_i	)
L2 (Ridge)	$\lambda \sum_{i=1}^{n} W_i^2$	Reduces weight magnitudes	Regularizing, avoids overfitting
Elastic Net	(\lambda_1 \sum_{i=1}^{n}	W_i	+ \lambda_2 \sum_{i=1}^{n} W_i^2)

Yes, **regularizers** (like L1, L2, or Elastic Net) modify the **loss calculation** by adding a penalty term to the original loss function. The goal of this is to discourage the model from learning overly complex patterns by shrinking the model's weights, thus helping it to generalize better and avoid overfitting.

1. Standard Loss Function (Without Regularization)

Normally, a machine learning model uses a loss function to calculate the error between the predicted output Y^\hat{Y}Y^ and the actual output YYY. Common loss functions include:

- **Mean Squared Error (MSE)** for regression problems: Loss(Y,Y^)=1n∑i=1n(Yi−Y^i)2Loss(Y, \hat{Y}) = \frac{1}{n} \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2Loss(Y,Y^)=n1i=1∑n(Yi−Y^i)2
- **Cross-Entropy Loss** for classification problems: Loss(Y,Y^)=−1n∑i=1n[Yilog(Y^i)+(1−Yi)log(1−Y^i)]Loss(Y, \hat{Y}) = -

\frac{1}{n} \sum_{i=1}^{n} \left[Y_i \log(\hat{Y}_i) + (1 - Y_i) \log(1 - \hat{Y}_i) \right]Loss(Y,Y^)=−n1i=1∑n[Yilog(Y^i)+(1−Yi)log(1−Y^i)]

In both cases, the loss function measures the discrepancy between the predicted values Y^\hat{Y}Y^ and the actual target values YYY. Without regularization, the model optimizes this **loss function alone**.

2. Loss Function With Regularization

When you apply **regularization**, the penalty term is added to the standard loss function. The total loss that the model tries to minimize becomes a combination of:

- The original **loss** (error on the training data).
- The **regularization term** (a penalty based on the weights).

The general form of the **regularized loss function** is:

Regularized Loss=Original Loss+λ⋅Penalty Term\text{Regularized Loss} = \text{Original Loss} + \lambda \cdot \text{Penalty Term}Regularized Loss=Original Loss+λ⋅Penalty Term

Where:

- λ\lambdaλ is the **regularization strength** (a hyperparameter that controls how much the regularization influences the total loss).
- The **Penalty Term** depends on the type of regularization applied (L1, L2, or Elastic Net).

a. L1 Regularization (Lasso):

Regularized Loss=Loss(Y,Y^)+λ∑i=1n∣Wi∣\text{Regularized Loss} = Loss(Y, \hat{Y}) + \lambda \sum_{i=1}^{n} |W_i|Regularized Loss=Loss(Y,Y^)+λi=1∑n∣Wi∣

- This adds the **sum of the absolute values** of the weights to the loss function. It encourages sparsity by penalizing large weights and tends to make some weights exactly zero, effectively performing **feature selection**.

b. L2 Regularization (Ridge):

Regularized Loss=Loss(Y,Y^)+λ∑i=1nWi2\text{Regularized Loss} = Loss(Y, \hat{Y}) + \lambda \sum_{i=1}^{n} W_i^2Regularized Loss=Loss(Y,Y^)+λi=1∑nWi2

- This adds the **sum of the squared values** of the weights to the loss function. It penalizes large weights, but without setting any of them to zero, and helps prevent overfitting by ensuring the weights are smaller.

c. Elastic Net:

Regularized Loss=Loss(Y,Y^)+λ1∑i=1n∣Wi∣+λ2∑i=1nWi2\text{Regularized Loss} = Loss(Y, \hat{Y}) + \lambda_1 \sum_{i=1}^{n} |W_i| + \lambda_2 \sum_{i=1}^{n} W_i^2Regularized Loss=Loss(Y,Y^)+λ1i=1∑n∣Wi∣+λ2i=1∑nWi2

- This combines both **L1** and **L2** regularization, so it penalizes both the absolute values and the squared values of the weights.

3. How Regularization Affects the Weight Updates

When you include regularization, the model will still minimize the loss function, but now it's balancing between:

1. **Minimizing the training error** (the original loss function).
2. **Minimizing the size of the weights** (via the regularization term).

This means that during the **gradient descent** process, the weights are updated not only to reduce the error but also to reduce the size of the weights. This trade-off helps the model avoid fitting the training data too closely (which can lead to overfitting).

4. Effect on Gradient Calculation

When regularization is added to the loss function, it also affects the **gradients** of the weights during backpropagation. The gradient of the regularized loss function is the sum of:

- The gradient of the **original loss** function (e.g., MSE or cross-entropy).
- The gradient of the **regularization term**.

For L2 regularization, for example, the update rule for the weights during gradient descent becomes:

W=W−η⋅(∇WOriginal Loss+λ⋅W)W = W - \eta \cdot \left(\nabla_W \text{Original Loss} + \lambda \cdot W \right)W=W−η⋅(∇WOriginal Loss+λ⋅W)

Where:

- ∇WOriginal Loss\nabla_W \text{Original Loss}∇WOriginal Loss is the gradient of the original loss function with respect to the weights.
- λ⋅W\lambda \cdot Wλ⋅W is the gradient of the regularization term.
- η\etaη is the learning rate.

For **L1 regularization**, the gradient for the regularization term is λ⋅sign(W)\lambda \cdot \text{sign}(W)λ⋅sign(W), where sign(W)\text{sign}(W)sign(W) returns -1 for negative weights and 1 for positive weights.

5. Summary: Regularization Updates the Loss Function

- **Yes, regularizers modify the loss calculation** by adding a penalty term that depends on the weights of the model.
- This new **regularized loss function** influences the model's weight updates, encouraging smaller weights and simpler models that generalize better.
- Regularization is a **balance** between fitting the training data and preventing the model from becoming too complex (overfitting).

By modifying the loss function with regularization, you are effectively training the model to minimize both the prediction error and the complexity of the model.