Deep-Learning-Module-2-Important-Topics-PYQs

1. A supervised learning problem is given to model a deep feed forward neural network. Suggest solutions for a small sized dataset for training.

1. Data Augmentation

- Create new data from existing data by modifying it slightly.
- This makes the dataset look larger and more diverse.

2. Use Pretrained Models (Transfer Learning)

- Start with a model that was trained on a large dataset (like ImageNet), and **fine-tune it** on your small dataset.
- The pretrained model already knows general patterns like edges or shapes. You just teach it to adjust a little for your specific task.

3. Apply Regularization

- Regularization prevents overfitting.

4. Early Stopping

- Stop training when the model **starts doing worse** on the validation set.
- This avoids training for too long and overfitting to the small training set.

5. Cross-Validation

- Split your small data into **multiple folds**, train on some, and validate on others.
- This helps make **better use of all your data** and gives a more reliable estimate of performance.

6. Simplify the Model

- Don’t use a very deep or large network — it can easily overfit a small dataset.
- Start with fewer layers and neurons. Simpler models often work better on small datasets.

7. Use Synthetic Data (if possible)

- If applicable, generate **fake but realistic** data that mimics your real dataset.
- Tools like simulations, GANs (generative adversarial networks), or synthetic text/image generators can help.

---

2. Explain how L2 regularization improves the performance of deep feed forward neural networks.

What Is a Deep Feedforward Neural Network?

Imagine a big team of decision-makers working together to solve a problem.
Each person (called a **neuron**) takes input, processes it, and passes the result forward.

These neurons are organized into **layers**, and information flows from **input** ➡️ **hidden layers** ➡️ **output** — that's why it's called **feedforward** (it only goes forward).

The Problem: Overfitting in Deep Networks

Deep networks have **many layers** and **millions of parameters (weights)**.
They are **super powerful**, which is good, but also dangerous:

- They can **memorize the training data** instead of **learning the patterns**.
- This means they perform well on training data, but **fail on new, unseen data**.
  This is called **overfitting**.

How L2 Regularization Helps

L2 regularization **improves performance** by making the model **simpler, more general, and less extreme**. Let’s see how.

1. Prevents Weights from Becoming Too Large

Each connection in the network has a weight (importance).
Sometimes the network might make some weights **very large**, which makes the model **unstable** and **too sensitive** to small changes in input.

L2 regularization **adds a penalty** for large weights:

> "Hey model, if you want to increase a weight too much, you'll pay a price for it."

This keeps all the weights **small and smooth**, leading to **more stable predictions**.

2. Encourages Simpler Models

L2 pushes the model to:

- Spread the learning across **many small weights**, rather than a few large ones.
- This means the model is learning **general patterns** and not memorizing noise.

Simple models usually **generalize better** — they do well even on new, unseen data.

3. Reduces Overfitting

By penalizing complexity (large weights), L2 regularization:

- Makes the network **less likely to memorize** the training data.
- Helps it **focus on meaningful patterns** instead of noise.
  This improves **test performance** — the model performs better on real-world data.

4. Works Well with Gradient Descent

During training, the model updates weights to reduce error using a method called **gradient descent**.

L2 regularization fits smoothly into this process — it:

- Just adds a small extra term to the weight update formula.
- Helps the model **converge more smoothly** to a better solution.

---

3. Explain any two parameter initialization methods.

What Is Parameter Initialization?

Before a neural network starts learning, it needs to **start with some values** — called **parameters** (these are mainly weights and biases).
Imagine you're building a house (your model), and the **first step is laying the foundation**. If you lay it wrong, the whole structure becomes unstable.
So… how do we start with the **right kind of values**? That's what **initialization** is all about!

1. Zero Initialization

You set **all weights and biases to zero** before training.
It's like saying:

> "Let's just start everything from 0 — clean and equal."

Problems

It seems fair, but **it breaks learning** in neural networks.
Here's why:

- If all neurons in a layer start with **exactly the same weight**, they'll also produce **the same output**.
- That means they'll learn **the same thing** forever — no uniqueness or specialization.
- The network becomes **stuck** and doesn't learn well.

Key takeaway:

- **Simple** idea, but almost **never used in practice** because it causes **symmetry** problems.
- Your neurons become **clones** of each other.

2. Random Initialization (Better, but not perfect)

You give each weight a **small random value** (positive or negative) to start with.
It's like saying:

> "Let's give each neuron its own personality from the start — even if it's a tiny one."

Why it works better:

- Random weights break the symmetry, so each neuron starts learning something **slightly different**.

- This helps the network **learn properly** and **faster**.
  But:
- If the random numbers are **too small**, the learning is **very slow**.
- If they're **too big**, it can cause problems like **exploding or vanishing gradients** (where the math goes crazy during training).

3. Xavier Initialization (for tanh or sigmoid)

- This method chooses random values **carefully**, based on the number of neurons.
- Best when you use **tanh** or **sigmoid** (types of "yes/no" activations).

4. He Initialization (for ReLU)

- Like Xavier, but made specially for a different kind of brain cell called **ReLU** (used a lot today).

5. Orthogonal Initialization (for RNNs)

- Weights are set in a way that keeps signals clean and not mixed up.
- Best for special types of networks (like ones that work with time or sequences — like predicting words in a sentence).

---

4. Compare L1 and L2 regularization.

Imagine You're a Student Writing an Essay
You're told:

- "Write a great essay, but don't make it **too long or too complicated**."
  This is just like building a **machine learning model** — we want it to **fit the data well**, but **not be overly complex** (that leads to overfitting).

What is Regularization

Regularization is like a **rule** your teacher gives you:

> "Hey, keep your essay **simple and short**. If you add unnecessary fancy words, you'll lose marks!"

In machine learning, regularization does the same thing:

- It **adds a penalty** for being too complex.
  There are two common types of this rule: **L1 and L2**.

L1 Regularization (Also called Lasso)

> **L1 is like a strict teacher** who says:
> “Don’t just *reduce* the number of fancy words — **cut them out entirely** if you can!”

In ML terms:

- L1 tries to **make some weights exactly 0** (it completely removes unimportant features).
- It makes the model **sparse** — it focuses only on the most important things.
- Great when you want to **select only a few key features** from your data.

L2 Regularization (Also called Ridge)

> **L2 is like a gentle teacher** who says:
> “Okay, you can keep all your words, but **don’t overuse** any of them.”

In ML terms:

- L2 **shrinks** all the weights, but usually **doesn’t make them exactly zero**.
- It smoothens things out — keeps all features, but makes them less extreme.
- Good when **all features are useful**, but you still want to avoid overfitting.

Feature	L1 Regularization (Lasso)	L2 Regularization (Ridge)
Penalty Style	Sum of **absolute values** of weights	Sum of **squares** of weights
Effect on Weights	Can make some weights **exactly zero**	Makes all weights **smaller**, but not zero
Feature Selection?	✅ Yes (removes unimportant features)	❌ No (keeps all features)
Model Simplicity	Very simple (sparse)	Smooth but still includes everything
Use Case	When only **a few features are important**	When **all features contribute a bit**
Analogy	Tosses out junk from your desk	Organizes everything in your drawers

Feature	L1 Regularization (Lasso)	L2 Regularization (Ridge)
Math Sensitivity	More resistant to noise and outliers	More sensitive to outliers

5. What is meant by Data augmentation? List down the benefits of it.

- **Data augmentation** is a technique used in machine learning where we artificially increase the size of a dataset by making small changes to the existing data.4
- The idea is to create new data points by modifying the original data, which can help the model learn better and avoid overfitting (where the model memorizes the training data rather than generalizing to new, unseen data).

Key Concept:

- **Augmented data**: Modified versions of the original data (e.g., flipping, rotating images).
- **Synthetic data**: New data that is generated without using the original data
- Data augmentation is not limited to images. It can also be used for **text**, **audio**, and **video** data.

Why Use Data Augmentation?

Here are some **benefits** of using data augmentation

1. **Prevents Overfitting**: By artificially increasing the dataset size, the model sees more variations of the data, which helps it generalize better and not memorize the training data.
2. **Improves Model Accuracy**: With more diverse data, the model can learn to handle more situations and can become more accurate when making predictions.
3. **Helps with Small Datasets**: When you don't have enough data to train a good model, data augmentation helps by providing more examples.
4. **Reduces Cost of Labeling Data**: Instead of manually creating new labeled data, you can modify existing data and still use it for training.

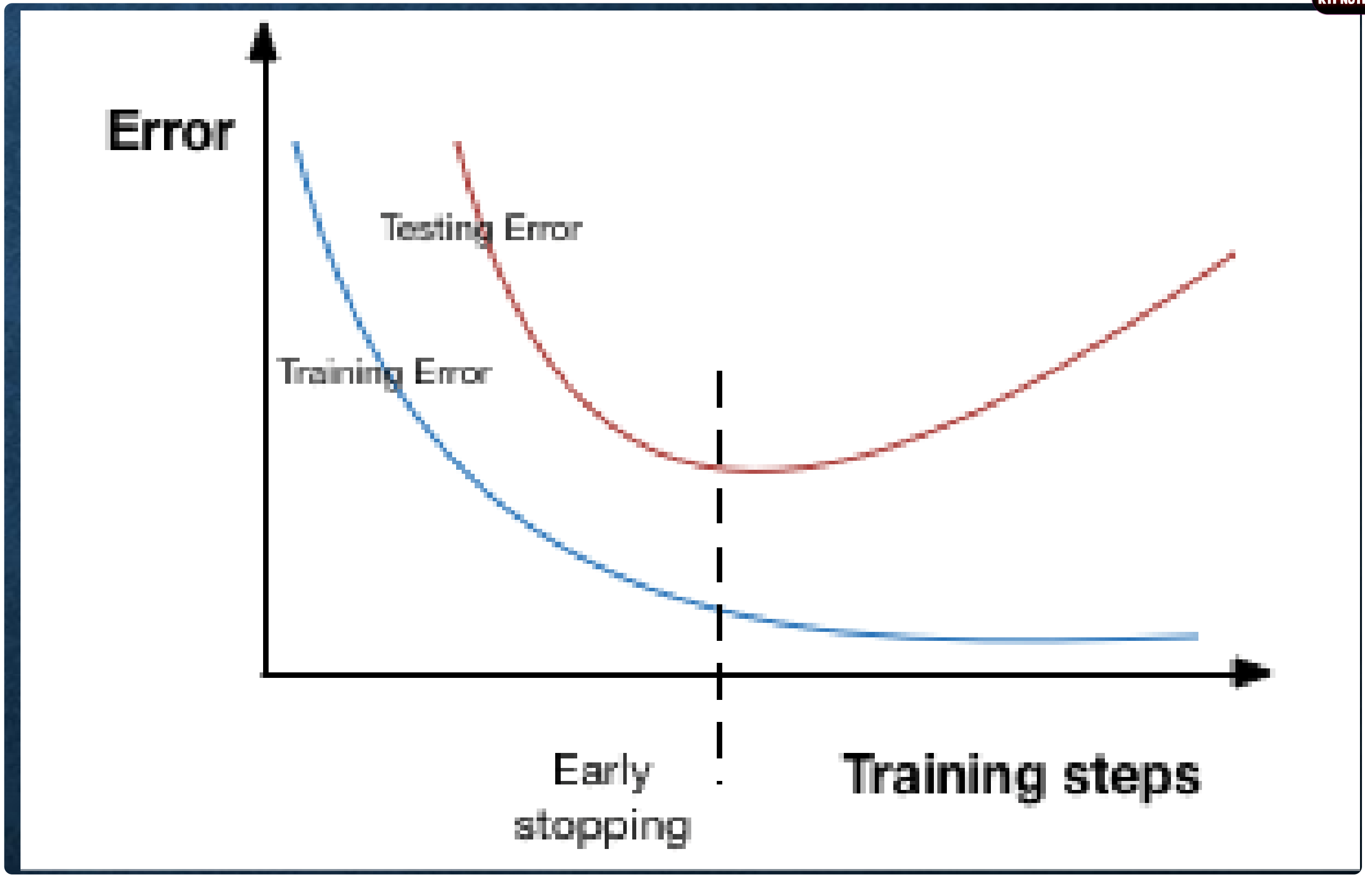
6. Describe Early stopping in neural network training.

Early stopping is a technique used in training neural networks to prevent overfitting (where the model becomes too specialized in the training data and fails to generalize to new, unseen data). It helps by stopping the training process before the model starts to overfit.

How Does Early Stopping Work?

1. **Training and Validation Split**: When training a model, we split the data into two parts:
   - **Training data**: The data used to train the model.
   - **Validation data**: The data that the model has not seen during training. We use this to check how well the model is performing on new data.
2. **Training the Model**: We begin training the neural network on the training data. During the training process, the model's performance is evaluated on both the training data and the validation data.
3. **Monitoring Performance**:
   - **Training error**: This tells us how well the model is fitting the training data.
   - **Validation error**: This tells us how well the model is performing on unseen data (the validation data).
4. **When to Stop**: Initially, as the model trains, both training and validation errors decrease, meaning the model is improving. But at some point, the model might start to "memorize" the training data, and the validation error begins to increase even though the training error keeps decreasing. This is a sign of **overfitting**.
   - **Early stopping** kicks in at the point where the validation error is the lowest. This is the **optimal point** where the model performs best on unseen data.
5. **Stopping the Training**: When the validation error starts increasing, it indicates that the model is overfitting. At this point, we stop the training, even if the training error is still decreasing.

Visualizing Early Stopping:



Think of the training process as a graph with two lines:

- **Training error (blue line)**: This line keeps going down as the model improves on the training data.
- **Validation error (red line)**: This line goes down initially as the model improves on unseen data, but at some point, it starts to go up because the model starts overfitting the training data.
  The **early stopping point** is the point where the validation error is at its **lowest**, just before it starts to increase again.

7. Differentiate stochastic gradient descent with and without momentum.

Imagine a **neural network** like a **robot brain** made of simple decision units (called **neurons**) that work together to make predictions — like:

- "Is this a cat or a dog?"
- "What's the best price for this product?"

- “Is this email spam?”
  Each neuron:
- Takes some numbers as input
- Does some simple math
- Sends its output to the next layer

What are Weights?

Each connection between neurons has a number called a **weight**.
Think of it like the importance of a path:

- A **high weight** = very important path
- A **low weight** = not so important

When we train a neural network, we’re trying to find the **best weights** that give us the **most correct answers**.

What is Loss or Error?

- After the network gives its answer (like "this is a dog"), we check if it was right.
- We calculate how **wrong** the network is — this is called the **loss** or **error**.
- Example:
  - If the answer should be **“dog”** and the network says **“cat”**, that’s a big error.
  The goal is to make this error as **small as possible**.

What is Gradient Descent?

- Imagine the error is like a **hill**, and the top is **bad** (high error), and the bottom is **good** (low error).
- We want to **go downhill** to reduce error.
- A **gradient** tells us which direction is **downhill**.
- “Gradient Descent” means:
  - Take steps in the direction that reduces the error
  Each step updates the **weights** to get closer to the best ones.

What is Stochastic Gradient Descent (SGD)?

“Stochastic” means **randomly picked**.

Instead of using the **whole training data at once**, we update the weights using **one example at a time**. This is faster and works well in practice.

When putting it altogether

Goal:

Find the best weights to make accurate predictions by **reducing the error**.

How?

We use **Stochastic Gradient Descent**:

- Take one example
- See how wrong the network is (loss)
- Use the **gradient** to take a step downhill (reduce the error)
- Update the weights

What is SGD without momentum?

This is like taking one step at a time — based **only on where you are right now**.
If the path is bumpy, you might:

- Go the wrong way
- Get stuck
- Move slowly

What is SGD with momentum?

This is like **rolling a ball downhill** instead of walking.

- You **remember your past steps**
- You **build up speed** in the right direction
- You go **faster and smoother**
  Momentum helps the network learn **better and quicker**.

Feature	SGD Without Momentum	SGD With Momentum
Memory of past steps	❌ No	✅ Yes
Speed of learning	⏳ Slower	⚡ Faster

Feature	SGD Without Momentum	SGD With Momentum
Stability	May oscillate	Smoother path
Equation	$w=w-\alpha\cdot\nabla L$	$v=\beta v-\alpha\nabla L v$, $w=w+v$

8. State how to apply early stopping in the context of learning using Gradient Descent. Why is it necessary to use a validation set (instead of simply using the test set) when using early stopping?

What is Early Stopping?

- When training a neural network using **Gradient Descent**, the model learns by gradually improving its performance. However, if we let it train for too long, it may start to **overfit**.
- Overfitting means the model **memorizes** the training data too well but doesn't perform well on **new, unseen data**.
- **Early stopping** is a technique used to prevent overfitting by **stopping the training process early** before the model starts overfitting. It helps you **save time** and **avoid unnecessary computation**.

How to Apply Early Stopping?

This is the approach to apply **early stopping** during training with **Gradient Descent**:

1. **Split your data** into:
    - **Training Set**: Used to train the model.
    - **Validation Set**: Used to evaluate the model's performance during training.
    - **Test Set**: Used to test the final model after training is complete.
2. **Monitor the performance** on the **validation set** at each training iteration (or after every few epochs).
    - If the **loss (error)** on the validation set starts to **increase** (after decreasing initially), this is a sign of overfitting.
3. **Stop the training**:
    - Stop training when the validation loss stops improving or starts increasing for a predefined number of consecutive iterations (or epochs). This number is called the

patience.

4. **Save the best model**:
    - The model with the lowest **validation loss** is saved. This is the model you use for final predictions.

Why Use a Validation Set Instead of the Test Set for Early Stopping?

Here's why the **validation set** is used for early stopping, not the **test set**:

- **Test Set**: The test set is meant to **evaluate** your model's performance **only once** after the model has finished training. Using the test set during training, including for early stopping, would give you a **biased estimate** of the model's performance, because the model would have been influenced by the test set during training.
- **Validation Set**: The validation set is used to **tune the model during training** (e.g., hyperparameters, early stopping). It allows you to **monitor the model's learning process** without affecting the final evaluation on the test set.

By using the **validation set** for early stopping, you ensure that the model generalizes well to new data while avoiding **leakage of information** from the test set into the training process.

9. Describe the effect in bias and variance when a neural network is modified with more number of hidden units followed with dropout regularization.

What are Bias and Variance?

Before diving into the effect of adding more hidden units and dropout, let's quickly define **bias** and **variance**:

- **Bias**: Bias refers to the error introduced by simplifying the model. If a model has high bias, it means it makes **incorrect assumptions** about the data and is **underfitting** — it doesn't capture the patterns in the data well.
- **Variance**: Variance refers to the error caused by the model being **too complex** and overly sensitive to small fluctuations in the training data. If a model has high variance, it means it's **overfitting** — it performs well on the training data but struggles to generalize to new, unseen data.

1. Adding More Hidden Units

- When you **increase the number of hidden units** (neurons in a layer), the neural network becomes **more complex**.
- It has more capacity to **learn and represent** different patterns in the data. Let’s see how this affects bias and variance:

Effect on Bias:

- **Lower Bias**: Adding more hidden units allows the model to **learn more complex patterns**. This can help **reduce bias** because the model can fit the training data better. The model becomes more flexible and can capture more intricate relationships in the data.

Effect on Variance:

- **Higher Variance**: With more hidden units, the model becomes more flexible and has **more parameters** to adjust.
- This increases the risk of the model becoming **too complex**, meaning it might **overfit** to the training data. As a result, the variance increases because the model may fit to the noise in the data rather than the true underlying pattern.

2. Applying Dropout Regularization

- Now, let's talk about **dropout regularization**. Dropout is a technique used to prevent overfitting by **randomly turning off (dropping out)** a fraction of neurons during training.
- This forces the network to **not rely too heavily on any single neuron** and encourages the network to learn **more robust features**.

Effect on Bias:

- **Small Increase in Bias**: Dropout can slightly **increase bias** because it reduces the capacity of the network (since some neurons are "dropped" randomly during training).
- This could mean the network doesn't capture the data as well as it could without dropout. However, this increase in bias is usually **small**.

Effect on Variance:

- **Lower Variance**: Dropout helps **reduce overfitting** by preventing the model from relying too much on specific neurons.

- This means that **variance is reduced** because the model becomes less sensitive to fluctuations in the training data and more generalizable to unseen data.

Modification	Effect on Bias	Effect on Variance
More Hidden Units	Decreases bias (lower)	Increases variance (higher)
Dropout Regularization	Slight increase in bias	Decreases variance (lower)

10. Describe the advantage of using Adam optimizer instead of basic gradient descent.

What is Gradient Descent?

Before we dive into **Adam**, let's first understand what **gradient descent** is:

- **Gradient Descent (GD)** is an optimization algorithm used to minimize the error (loss) of a model, like a neural network, by adjusting its weights.
- The basic idea is to update the model's weights in the opposite direction of the gradient (slope) of the loss function, so that the model learns to make better predictions over time.

What is Adam Optimizer?

The **Adam optimizer** is an **improvement** over basic gradient descent. **Adam** stands for **Adaptive Moment Estimation**, and it combines ideas from two important techniques:

1. **Momentum**: It helps the optimizer to remember the previous gradients, allowing it to move faster and smoother in the right direction.
2. **RMSProp**: It scales the learning rate for each parameter based on the past gradients, so it adapts to different magnitudes of gradients across parameters.

Advantages of Adam Over Basic Gradient Descent:

1. **Adaptive Learning Rate**:
    - **Gradient descent** uses a fixed learning rate for all weights, meaning it might either take very small steps (slowing down learning) or too large steps (which can make it jump over the minimum).

- **Adam**, however, **adjusts the learning rate** for each weight based on past gradients. This means it can make larger steps when the gradients are small and smaller steps when the gradients are large, which **improves convergence speed** and **stability**.

2. **Momentum**:
   - In basic gradient descent, each update is based only on the current gradient, which can lead to slow and erratic progress, especially if the gradient is noisy.
   - **Adam** uses **momentum** to accumulate past gradients, so it **remembers previous steps**. This helps it to move faster in the right direction and **smooths out noisy gradients**.
3. **Works Well with Sparse Gradients**:
   - If your data or model has sparse gradients (some parameters have very small gradients), **Adam** is more efficient because it can handle such situations by **adapting the learning rate** for each parameter, making sure the model still learns effectively.
4. **Reduces Tuning Effort**:
   - **Adam** usually performs well without much hyperparameter tuning. While basic gradient descent may require careful tuning of the learning rate (too large or too small can make training unstable), **Adam** automatically adjusts and is often more robust to these issues.

---

11. Explain different gradient descent optimization strategies used in deep learning.

1. Batch Gradient Descent (BGD)

How it works:

- **Batch Gradient Descent** computes the gradient of the entire dataset to update the weights. This means that before making any update to the weights, the algorithm goes through the entire training dataset to compute the gradient.

Advantages:

- **Stable updates** because it uses the whole dataset to calculate the gradient.
- **Converges to the minimum** smoothly and steadily.

Disadvantages:

- **Computationally expensive**: For large datasets, computing the gradient on the entire dataset can be very slow.
- It may take too long to train the model since every weight update requires going through the entire dataset.

When to use:

- When the dataset is small and can be processed all at once.

2. Stochastic Gradient Descent (SGD)

How it works:

- In **Stochastic Gradient Descent (SGD)**, the model updates weights **after each data point** (or training example).
- Instead of using the entire dataset, it picks a **single random training example** to compute the gradient and updates the weights based on that single example.

Advantages:

- **Faster updates**: Since it uses one data point at a time, updates are much faster and can start improving the model quickly.
- Can **escape local minima** because of the "noisy" updates, which help in exploring the solution space better.

Disadvantages:

- **Noisy updates**: The gradient estimation can be very noisy, causing fluctuations in the weight updates. This might make it harder to converge smoothly.
- The path to convergence is more **erratic** compared to Batch Gradient Descent.

When to use:

- When you have a large dataset or need fast updates.
- When the training time is a concern and computational resources are limited.

3. Mini-batch Gradient Descent

How it works:

- **Mini-batch Gradient Descent** is a hybrid approach that combines **Batch Gradient Descent** and **Stochastic Gradient Descent**.
- In this method, the dataset is divided into small batches (mini-batches) and the gradient is computed for each mini-batch. Typically, the mini-batches contain between 32 and 256 training examples.

Advantages:

- **Faster than batch gradient descent** because it computes the gradient over smaller batches, reducing the computational cost.
- **More stable than SGD** because the noise from a single training example is averaged over the mini-batch, making it easier to converge.
- It can benefit from **parallel processing** since batches can be processed in parallel.

Disadvantages:

- It still requires careful tuning of the **mini-batch size**. Too small a batch size can behave like SGD, and too large can behave like Batch Gradient Descent.

When to use

- This is the most commonly used optimization method because it provides a good trade-off between speed and stability, especially for large datasets.

4. Momentum (SGD with Momentum)

How it works:

- **Momentum** helps accelerate SGD by using the **previous updates** to adjust the current gradient update. It introduces a **velocity term** that helps the optimizer “remember” past gradients and smooth out the updates.

The update equation for momentum is:

$$v_t = \gamma v_{t-1} + \eta \nabla L(\theta)$$

$$\theta_{t+1} = \theta_t - v_t$$

Where:

v_t is the velocity (momentum)

γ is the momentum factor (typically 0.9)

η is the learning rate

$\nabla L(\theta)$ is the gradient of the loss function.

Advantages:

- **Speeds up convergence**: Momentum helps the optimizer move faster in the relevant directions and **avoids oscillations**.
- **Smoother updates**: The momentum term smooths out the updates, especially when the gradients are noisy.

Disadvantages:

- **Requires tuning of the momentum parameter**: Choosing the right momentum factor (γ) is important. Too much momentum can overshoot the optimal solution.

When to use:

- When you need faster convergence and want to smooth out the noisy updates from SGD.

5. AdaGrad (Adaptive Gradient Algorithm)

How it works:

- **AdaGrad** adapts the learning rate for each parameter based on the **history of gradients**.
- It assigns a **larger learning rate** to parameters with smaller gradients and a **smaller learning rate** to parameters with larger gradients.

The update equation for AdaGrad is:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla L(\theta)$$

Where:

- Gt is the sum of squared gradients up to time t
- ϵ is a small constant to prevent division by zero

Advantages:

- **Adaptive learning rate**: Automatically adjusts the learning rate for each parameter based on its past gradients
- **Good for sparse data**: Works well when some features are more important than others (e.g., in NLP or image data with sparse features).

Disadvantages:

- **Learning rate keeps decreasing**: Once the learning rate starts decreasing, it can become too small and halt learning prematurely.

When to use:

- When dealing with sparse data and features that vary in importance.

6. RMSprop (Root Mean Square Propagation)

How it works:

- **RMSprop** is an improvement over AdaGrad. It addresses the problem of continuously shrinking learning rates by using an **exponentially weighted moving average** of past gradients.

The update equation for RMSprop is:

$$v_t = \gamma v_{t-1} + (1-\gamma)\nabla L(\theta)^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t + \epsilon}}\nabla L(\theta)$$

Where:

- vt is the moving average of the squared gradients
- γ is a decay factor (typically around 0.9)

Advantages:

- **Solves the problem of vanishing learning rates** in AdaGrad by using a moving average.
- **Good for non-stationary problems** (problems where the data distribution changes over time).

Disadvantages:

- **Still requires tuning of learning rate** and decay factor.

When to use:

- In scenarios where learning rates need to be adapted, and for problems with noisy data or non-stationary distributions.

7. Adam (Adaptive Moment Estimation)

How it works:

- **Adam** combines the ideas of **Momentum** and **RMSprop**. It computes the **moving averages of the gradient** and the **squared gradient** to adjust the learning rates for each parameter.

Advantages:

- **Combines momentum and adaptive learning rates** for fast and efficient learning.
- **Works well in practice**: It usually requires **less tuning** of hyperparameters and performs well in many scenarios.

Disadvantages:

- **Can be more computationally expensive** compared to simpler optimizers like SGD.

When to use:

- **Most general-purpose** optimization algorithm and is widely used in practice for many deep learning tasks.

Summary of Gradient Descent Optimization Strategies:

Optimization Algorithm	Key Feature	Pros	Cons
Batch Gradient Descent (BGD)	Uses entire dataset for each update	Stable updates	Slow for large datasets
Stochastic Gradient Descent (SGD)	Uses single data point for each update	Faster updates	Noisy updates
Mini-batch Gradient Descent	Uses small batch of data	Balanced speed and stability	Need to choose optimal batch size
Momentum	Adds momentum term to smooth updates	Speeds up convergence	Requires tuning of momentum factor
AdaGrad	Adapts learning rate based on past gradients	Good for sparse data	Learning rate shrinks too quickly
RMSprop	Uses moving average of squared gradients	Solves AdaGrad issues	Requires tuning of parameters
Adam	Combines Momentum and RMSprop	Works well in practice with minimal tuning	Computationally more expensive

12. Explain the following. i)Early stopping ii) Drop out iii) Injecting noise at input iv)Parameter sharing and tying.

i) Early Stopping

- When training a neural network, the model can **start to memorize** the training data instead of learning general patterns. This is called **overfitting**.
- **Early stopping** is a technique where you **stop the training process early** when the model stops improving on a **validation set** (a small part of the data not used during training).

Why it's useful:

- Prevents **overfitting**
- Saves **training time**

Example (Simple analogy):

- Imagine you're studying for an exam. You practice with questions and get better. But if you keep practicing the same set over and over, you might just memorize the answers instead of understanding the concepts.
- Stopping your practice once you consistently do well on mock tests (validation) is like **early stopping**.

ii) Dropout

- Dropout is a trick to prevent a neural network from becoming too **dependent on specific neurons**.
- During training, **random neurons are “turned off” (dropped out)** in each round. This forces the network to learn in a more **robust and general** way.

Why it's useful:

- Reduces **overfitting**
- Encourages the network to **not rely too much on any one part**

Example (Simple analogy):

- Imagine you're playing a team sport. If every time you only pass the ball to your best player, the whole team never improves.
- If sometimes that player sits out, others have to learn and play better. That’s like **dropout**!

iii) Injecting Noise at Input

- This technique involves **adding random small “noise” (errors)** to the input data during training.
- This helps the model **learn more general and flexible patterns**, instead of just memorizing the exact input values.

Why it's useful:

- Makes the model **more resistant to errors or changes** in real-world data
- Improves generalization

Example (Simple analogy):

If you're learning to recognize handwritten numbers and sometimes people write “5” differently, you should still recognize it. Training with noisy examples helps your brain (or model) get **used to variety**.

iv) Parameter Sharing and Tying

- This is a method where the **same set of weights (parameters)** are used in **multiple places** in the model.
- It’s used mainly in **Convolutional Neural Networks (CNNs)** and **Recurrent Neural Networks (RNNs)**.

Why it's useful:

- Reduces the number of parameters
- Speeds up training
- Helps the model **generalize better**

Example (Simple analogy):

Imagine a sponge stamp with a pattern. Instead of drawing the same shape everywhere with a pencil, you can **reuse the same stamp** in multiple spots. That’s like **parameter sharing**—you reuse the same tool instead of creating a new one each time.

---

13. Discuss how the Gradient descent algorithm finds the values of the weight that minimizes the error function?

Goal of Training a Neural Network

When we train a neural network, we want it to **predict accurately**. To do that, we need to **adjust the weights** (the numbers that control how the network learns).

We measure how wrong the network is using something called an **error function** (also known as **loss function** or **cost function**). This function gives us a number—**lower is better**.

Think of the Error Function Like a Hill

Imagine a hilly landscape. Each point on the hill represents a different **set of weights** for the neural network. The **height** of the hill at each point represents the **error**.

We want to find the **lowest point** in this landscape — the place where the **error is smallest**. That's the best set of weights!

Enter Gradient Descent

Gradient Descent is like a hiker trying to walk downhill to find the lowest point. Here's how it works:

1. **Start somewhere random** (random weights).
2. **Look around and see which direction goes downhill fastest** (compute the gradient).
3. **Take a small step in that direction** (update the weights).
4. Repeat steps 2 and 3 until you're close to the bottom.

The Math Behind It

Let's say:

- `θ` (theta) is your weight
- `f(θ)` is the error function (what you're trying to minimize)
- `η` (eta) is the learning rate — a small number that controls how big your steps are

$\theta = \theta - \eta * \nabla f(\theta)$

This means:

- Take the current weight `θ`
- Move it in the opposite direction of the gradient ($\nabla f(\theta)$) — that's downhill!
- Multiply the gradient by `η` to take small steps, not big jumps

The **gradient** tells us the direction in which the error increases the most.
So moving in the **opposite direction** reduces the error — just like walking downhill!

Over time, the algorithm keeps adjusting the weights to reduce the error — until it gets close to a **minimum**.

14. Which optimization technique can be used to solve the problems caused by constant learning rate in Gradient

descent algorithms? Explain the method

There is a common issue in **Gradient Descent**: using a **constant learning rate** can cause problems like:

- **Too large** learning rate → the model jumps over the minimum and never converges
- **Too small** learning rate → the model takes forever to reach the minimum
- In some terrains (like narrow valleys), the model might oscillate and get stuck

Solution: Use Adaptive Learning Rate Techniques

- Instead of using **one fixed learning rate**, we can use optimization techniques that **automatically adjust the learning rate** during training.

Adam Optimizer (Adaptive Moment Estimation)

Adam is one of the best solutions for handling the problems caused by a constant learning rate.

How Adam Works

Adam adjusts the learning rate **for each weight individually**, based on how the error (loss) is changing.
It uses two key ideas:

1. **Momentum** – to keep track of the direction of the gradient (helps smooth things out)
2. **Adaptive Learning Rates** – to scale the learning rate based on how much change each weight has seen