MAL Exam answers

Neural network:

an artificial neuron takes in information, processes it, and decides what to do with it, similar to how neurons in our brain process signals.

Inputs: Think of the inputs as data that the neuron receives.

Weights: Each input has a weight. The weight determines how important that input is for the neuron. If a weight is high, that input is more important.

Summing: The neuron combines all the weighted inputs.

Bias: After summing the inputs, a bias is added. The bias helps adjust the output and allows the neuron to make better predictions.

Activation Function: The summed result is then passed through a special function (called an activation function) that decides if the neuron "fires" or not, which means it will pass on the information to the next layer. If the result is above a certain threshold, the neuron will fire.

Output: The final result from the neuron can then be used for making predictions or decisions.

Dense neuron

A **dense neuron** is just a type of artificial neuron that connects to every neuron in the layer before and after it. It's also called a **fully connected neuron** because each input is connected to each output.

Activation function

Activation: The **activation function** is applied to this total value (net input). This function decides if the neuron should output something useful or not.

Why Activation?: Without an activation function, a neuron would only pass along a linear value, which would make the network pretty boring and not able to learn complex patterns. The activation function adds "non-linearity," meaning it helps the network make decisions and learn complicated patterns (like predicting whether a picture has a cat or a dog).

Sigmoid: The sigmoid function squashes the input value to a number between 0 and 1. It's like a decision-maker that turns any value into a probability.

Why it's used: It's great when you need a simple "yes or no" decision (like predicting if something will happen or not).

ReLU: ReLU changes all negative values to 0 and keeps positive values the same. So, if the input is positive, ReLU just lets it through; if it's negative, it blocks it (turns it to 0).

Softmax: Softmax is used when there are multiple categories to choose from (like in classification problems). It turns the raw scores (logits) into probabilities that add up to 1.

Why it's used: It helps when you need to predict multiple classes (like choosing between cat, dog, or rabbit). It's especially useful for the final layer of a neural network for classification tasks.

Minimum Building blocks

These are the basic parts needed for a neural network to function:

1. **Neurons**:
    - Small processing units that take inputs, apply a weight, add bias, and use an activation function to decide what to pass forward.
2. **Layers**:
    - **Input Layer** → Receives raw data (e.g., pixels in an image).
    - **Hidden Layers** → Process data and learn patterns.
    - **Output Layer** → Produces the final result (e.g., cat or dog).
3. **Weights and Biases**:
    - **Weights** → Decide how much influence an input has.
    - **Biases** → Allow neurons to shift the decision boundary for better learning.
4. **Activation Functions**:
    - Applied in hidden layers to introduce non-linearity and help the network learn complex patterns. Examples: ReLU, Sigmoid, Softmax.
5. **Loss Function**:
    - Measures how wrong the network's prediction is. The goal is to minimize this error during training.

Loss functions vary depending on the type of problem:

- **For Regression (Predicting Numbers)**:
    1. **Mean Squared Error (MSE)** → Takes the average of squared differences between actual and predicted values.
    2. **Mean Absolute Error (MAE)** → Takes the average of absolute differences.
- **For Classification (Predicting Categories)**:
    1. **Cross-Entropy Loss** → Used for multi-class problems (e.g., predicting if an image is a cat, dog, or rabbit).
    2. **Binary Cross-Entropy** → Used for two-class problems (e.g., spam vs. not spam).

6. **Optimizer**:
    - Adjusts the weights and biases to reduce the error (e.g., **Gradient Descent** is commonly used).

• **Gradient Descent** → The simplest optimizer. It moves the weights in the direction that **reduces** the loss.

How Gradient Descent Works in a Neural Network

1. The network makes a prediction.
2. The **loss function** calculates how wrong the prediction is.
3. Gradient Descent looks at the **slope (gradient)** of the loss function.
4. The optimizer **adjusts the weights** slightly in the direction that reduces the loss.
5. This process repeats until the loss is as small as possible!

• **Stochastic Gradient Descent (SGD)** → Updates weights using small random batches of data instead of the full dataset, making it faster.
• **Adam (Adaptive Moment Estimation)** → A smarter optimizer that adjusts learning speed automatically, making training efficient and faster.

Adam (Adaptive Moment Estimation) is an advanced version of Gradient Descent that adjusts the learning rate **automatically** for each weight. This makes training **faster** and **more efficient** than normal Gradient Descent.

How Adam Works in Simple Steps

Adam improves on **Gradient Descent** using two key ideas:

1. **Momentum**: Remembers past updates and moves smoothly instead of making random jumps.
2. **Adaptive Learning Rate**: Adjusts the learning rate for each weight separately so that important weights learn faster and less important ones slow down.

1 Core Training Mechanism and Gradient Issues (Lecture 2)

1.1 Gradient descent

It is an optimization algorithm used to minimize the loss function by adjusting the model parameter(weight).

Find the best weights **θ** (theta) that minimize the loss function **L(θ)**

The loss function is denoted by **L(θ)**, and **θ** refers to all the parameters of the model.

We calculate the **gradient** (∇L(θ)), which tells us:

- The direction in which the loss increases.
- So, we **move in the opposite direction** to minimize the loss.

1.2 *Explain the fundamental process of training a deep neural network by minimizing a loss function using gradient descent.*

Overview: At its core, training a deep neural network is to find the best parameter (weight and biases) by minimizing the loss function using gradient descent.

The main task during training is to reduce this loss as much as possible. We want to find the best weights and biases that minimize this loss. However, because there are so many parameters in a deep neural network, we cannot simply try every possible combination. Instead, we use an approach called gradient descent, which helps us gradually improve the network's parameters.

Steps for gradient descent:

1. The networks take an input.
2. processes it through all the layers to produce a prediction — this is called the forward pass.
3. the loss is computed by comparing the prediction with the actual correct answer.
4. the network calculates the gradients of the loss with respect to every weight and bias — this is done during the backward pass (often called backpropagation).
5. the weights are updated using the gradient descent rule. we update the weight by moving it in the opposite direction of the gradient. Because the gradient points towards increasing the loss, so going the other way reduces it. we don't want to jump too far in one step because that might cause us to miss the best solution or make the loss worse. To control this, we multiply the gradient by a small number called the **learning rate**. This learning rate controls how big each step is when adjusting the weights.
6. By repeating these steps many times, the network gradually improves and the loss decreases, which means the predictions get better.

1.3. *Describe the backpropagation algorithm and how it efficiently calculates the gradients needed for parameter updates.*

Backpropagation is like a smart way for a neural network to learn from its mistakes. After the network makes a prediction, we want to know how to change all the tiny numbers inside it (called weights) so it can do better next time.

The problem is, the network has many layers, and each weight affects the final answer in a complicated way. Backpropagation helps by working backwards through the network to figure out how much each weight contributed to the mistake.

Steps:

1. **Start at the end:** We look at how wrong the network's final prediction was by calculating the loss.
2. **Calculate the error for the output layer:** We find out how much the loss would change if the output values changed a little bit.
3. **Move backward:** Using simple math (called the chain rule), we figure out how this error at the output depends on the weights in the layer before it.
4. **Repeat step-by-step:** We keep moving backward layer by layer, calculating how much each weight in each layer affected the final error.
5. **Collect all gradients:** These calculations give us the **gradients**, which tell us the direction and amount we should change each weight to reduce the error.

Because backpropagation reuses calculations from previous steps, it avoids doing the same work many times, making it efficient.

In short, **backpropagation helps the network "blame" each weight for the error and then learn how to fix it**, layer by layer, from the output back to the input.

1.3 *Discuss the phenomenon of vanishing and exploding gradients, which can hinder training stability and convergence.*

sometimes during backpropagation, the gradients can become **very, very small** or **very, very large** as they are passed backward through many layers. These two problems are called **vanishing gradients** and **exploding gradients**.

Vanishing Gradients

Vanishing gradients happen when the gradients get smaller and smaller as we move backward through the network layers. By the time the gradients reach the earliest layers (closest to the input), they are almost zero. Vanishing gradients make it **hard to learn** important features in early layers, slowing down or stopping training progress.

Exploding Gradients

On the other hand, **exploding gradients** happen when gradients become very large as they are backpropagated. This causes the weights to update with huge steps, which can make the network unstable. The loss might jump around wildly or even become "Not a Number" (NaN), causing training to fail. Exploding gradients make training **unstable**, causing weights to blow up and the model to fail to converge.

Why do these happen?

These problems often occur because of how gradients are computed using the chain rule, multiplying many derivatives together. If these derivatives are less than 1, multiplying many of them makes gradients shrink (vanish). If they are greater than 1, gradients grow exponentially (explode).

1.4 *explain how factors like the choice of activation functions can influence these gradient issues.*

Sigmoid: squashes input values into a small range between 0 and 1. Because the gradients multiply many small derivatives during backpropagation, they shrink quickly as they go backward, leading to the vanishing gradient problem.

Tanh: tanh squashes inputs into a small range (-1,1) but is zero-centred, which can help training. its derivatives are also **less than 1** and approach zero for large magnitude inputs, causing **vanishing gradients** similarly to sigmoid.

ReLU: Its derivative is 1 for positive inputs and 0 for negative inputs. ReLU can suffer from the "dying ReLU" problem where neurons output zero forever if they receive negative inputs, but it generally avoids vanishing gradients.

Leaky ReLU: Like ReLU, but for negative inputs, it outputs a small slope (e.g., 0.01 * x) instead of zero. Helps prevent the "dying ReLU" problem and still **avoids vanishing gradients** by maintaining a small gradient for negative inputs.

ELU: Like Leaky ReLU but uses an exponential function for negative inputs. This helps stabilize gradients and speeds up learning**, reducing vanishing gradient issues.**

Swish: it can output small negative values for negative inputs but also preserves positive linear behaviour for positive inputs. Swish tends to maintain stronger gradients than sigmoid or tanh and avoids saturating too quickly.

2 Improving Training Stability and Preventing Overfitting (Lecture 2)

2.1 Discuss how advanced optimizers (beyond basic SGD) such as Momentum or Adam improve the efficiency and stability of the parameter updates during training.

SGD- In basic **Stochastic Gradient Descent (SGD)**, we update the network's parameters (weights) using the gradient of the loss function with respect to each parameter.

Formula: New weight = Old weight − learning rate × gradient

This works, but it can be slow and unstable in some situations: if LR is small then training is slow, if LR is big, updates can jump over the minimum and cause the model to diverge.

Momentum Optimizer: Momentum helps accelerate gradient descent by remembering the past gradients and combining them with the current one. It keeps a "velocity" — a moving average of past gradients. Each new update is a combination of the current gradient and this velocity.

Adam - Adam combines Momentum and adaptive learning rates. Instead of using one fixed learning rate for all parameters, adaptive optimizers give each parameter its own learning rate, and this rate changes over time based on how the parameter is behaving.

How it works?

Keeps two things for parameter: Momentum, adaptive learning rate. Works well with noisy data, converges faster than SGD or momentum alone.

Scale Decay: As training goes on, we often want to reduce the learning rate slowly. This helps the model settle gently near the minimum of the loss function. Can be added to SGD, Momentum, Adam, etc. Improves final accuracy and convergence.

Weight Decay: Weight decay adds a small penalty to large weights during training. It encourages the network to keep weights small and simple. Can be added to any optimizer. Prevents overfitting and improves generalization.

2.2 *Explain the importance of dynamically adjusting the learning rate through learning rate scheduling.*

The **learning rate** controls how big the steps are when updating the model's weights during training.
A **learning rate schedule** means we **change the learning rate over time** — instead of keeping it constant.

Power Decay: Power decay reduces the learning rate. It gradually decreases the learning rate as training progresses. The "power" controls how quickly the learning rate decays. It's a smooth and controlled decay — slower than exponential. It is good for long training sessions where we want the learning rate to decline slowly.

Exponential Decay: Exponential decay decreases the learning rate **multiplicatively** after every few steps. This causes the learning rate to **drop very quickly** at first and then slow down.

It's a **more aggressive** form of decay than power decay. It is good for Quickly moving from large steps to fine-tuning.

Piecewise constant decay: the learning rate **remains constant for a while**, then **drops suddenly** at predefined steps or epochs. Define learning rate for some range of epochs. It is good for models that benefit from sharp learning rate drops at specific stages.

Performance: the learning rate is reduced **only when performance stops improving** (like validation loss or accuracy). If the model stops improving for a few epochs, the scheduler reduces the learning rate. This method **adapts automatically** to the training progress. It is goof for avoiding premature learning rate reduction and focusing on real performance.

2.3 ***Furthermore, explain how regularization techniques such as Dropout and Early Stopping are used to prevent overfitting and improve the generalization performance of deep neural networks.***

Overfitting happens when a model learns the **training data too well**, including the noise and small details that don't generalize to new data. To fix this, we use **regularization techniques.**

Dropout: Dropout is a regularization method where, during training, we randomly turn off (i.e., set to zero) a percentage of the neurons in the network. For example, with a dropout rate of 0.5, half the neurons in a layer are randomly ignored during one training step. This prevents the network from relying too heavily on specific neurons and encourages it to learn redundant and distributed representations.

Early Stopping: Early Stopping is a technique where we stop training the model before it starts overfitting. Here's how it works: Early Stopping monitors the validation loss during training. When validation loss stops improving and starts getting worse, the model is likely overfitting. Early stopping **halts training** at that point, keeping the model at its best generalization performance.

3 Convolutional Neural Networks and Feature Extraction (Lecture 3)

3.1 Explain how convolutional layers operate, including the role of filters, kernels, and receptive fields.

CNN- A **convolutional layer** is the core building block of a **Convolutional Neural Network (CNN)**, which is especially used in image processing. Instead of connecting every input neuron to every output neuron (like in fully connected layers), convolutional layers **focus on small local areas of the input** — this makes them **efficient** and good at detecting **patterns** like edges, textures, or shapes.

Filters and Kernels- A **filter** (also called a **kernel**) is a **small matrix of numbers** (usually 3×3 or 5×5) that **slides over the input image**. Each filter detects a **specific feature**, like a vertical edge, horizontal edge, blur, etc. A CNN usually uses **multiple filters**, so each one learns to detect a different pattern.

Receptive fields- The **receptive field** is the part of the input that a neuron "sees" or is connected to. In convolutional layers, each output value (neuron) is influenced only by a small area of the input (like a 3×3 patch). That small area is called its **receptive field**. As layers go deeper, the receptive fields get larger, helping the network understand more complex patterns.

How operation works?

1. The filter is placed over a small part of the image (called a patch).
2. Each value in the filter is multiplied with the corresponding pixel value in the image patch.
3. All the multiplied results are added up into a single number.
4. This number becomes one pixel in the output feature map.
5. The filter then slides (moves) across the image, repeating the process.

3.2 *How do CNN's learn hierarchical feature representations?*

Convolutional Neural Networks (CNN) learn to recognize patterns in a **hierarchical manner**, meaning that each layer builds on the patterns learned by the previous layers. This structure allows CNN to start with simple features and work their way up to more complex ones.

🧠 **How CNN Learn Hierarchies:**

🟢 **1. Early Layers – Low-Level Features:**

- These layers detect basic patterns like edges, lines, and colours.
- The convolutional filters at this stage act like edge detectors or texture extractors.

🟡 2. **Middle Layers – Mid-Level Features**

- These layers take combinations of the simple patterns and learn to detect **shapes**, **corners**, or **parts of objects** (e.g., an eye or an ear).
- Now the network starts understanding **spatial arrangements** of basic patterns

🔴 3. **Deep Layers – High-Level Features**

- These deeper layers combine shapes and parts to recognize **complex structures** or even **entire objects** like a face, a car, or a cat.

- These layers are very task-specific and help the model make decisions like classification.

3.3 Additionally, describe the purpose and effect of pooling layers such as max pooling.

A **pooling layer** is used in **Convolutional Neural Networks (CNNs)** to reduce the **spatial dimensions** (height and width) of feature maps. This makes the model **faster**, **less complex**, and more **robust to small changes** in the input, like slight shifts or distortions in images.

Max pooling- Max pooling looks at a small patch of the feature map and **selects the largest (maximum) value** in that patch. For example, if a 2×2 patch contains the values [1, 3; 2, 4], max pooling would return 4. This way, max pooling keeps the most important features and ignores minor variations.

Max pooling **shrinks the size** of the data while keeping strong signals like edges or corners. It also adds a degree of **invariance to translation**, meaning the model can still recognize features even if they shift slightly in the image. Max pooling keeps the **most prominent** features (like edges or textures).

4 Autoencoders and Representation Learning (Lecture 4)

4.1 What is an autoencoder, and how does it learn useful representations of the input data?

An **autoencoder** is a type of **neural network** used to **learn compressed and meaningful representations** of data. Think of it like a smart data shrinker: it learns how to compress data into a smaller form (encoding), and then how to rebuild the original data from that smaller version (decoding).

It has two main parts:

1. **Encoder**: Compresses the input into a smaller, dense representation (called a **latent vector** or **bottleneck**).
2. **Decoder**: Tries to reconstruct the original input from that compressed version.

The main purpose of an autoencoder is to learn **useful, compact features** from data in an **unsupervised** way — meaning it doesn't require labeled data. During training, the autoencoder is given an input and tries to make its output as similar as possible to the input. This is done by minimizing the **reconstruction loss**, which measures how close the output is to the original input.

Because the encoder has to squeeze the data into a smaller space (bottleneck), the model learns to focus on the **most important features** of the data while ignoring noise or irrelevant information. These compressed features can be useful for tasks like **data compression**, **denoising**, **dimensionality reduction**, or even as a **preprocessing step** for other machine learning models.

4.2 Beyond dimensionality reduction, describe at least one practical application of autoencoders, such as denoising, anomaly detection, or unsupervised pretraining.

Dimensionality reduction: **Dimensionality reduction** is the process of **reducing the number of features** (dimensions) in the data while **keeping the important information**.

Autoencoders are deep learning models that can perform dimensionality reduction automatically. They work by **compressing the input data into a lower-dimensional latent space** using an encoder and then reconstructing the original input using a decoder.

Practical Application of Autoencoders: Denoising

Denoising means **removing unwanted noise** (like grain, blur, or distortion) from data — especially from images. In many cases, images might be corrupted due to low-quality cameras, bad lighting, or transmission issues. We want to **clean these images up**.

How do autoencoder helps?

A special type of autoencoder called a **denoising autoencoder** is trained to **ignore noise** and **reconstruct the clean version** of a noisy input. During training, the model is given **noisy inputs** and asked to produce the **original, clean outputs**. For example, we might take a clean image, add random noise to it (like grain or blur), and then train the model to reconstruct the clean version. Over time, the autoencoder learns to **filter out the noise** and preserve the important features.

Denoising autoencoders are widely used in **image processing, medical imaging, and data cleaning**, helping improve data quality and clarity.

5 Modern Generative Models: GANs and Diffusion (Lectures 5 and 10)

5.1 Describe the architecture and training process of Generative Adversarial Networks (GANs), highlighting the competition between generator and discriminator.

A **GAN** is a type of deep learning model where **two neural networks compete** with each other to produce realistic data (like images, audio, etc.).

🧱 Architecture of a GAN

A GAN has **two main components**:

1. Generator (G):

- Takes **random noise** as input (usually a vector of random numbers).
- Tries to **generate fake data** (e.g., a realistic image).
- Its goal is to **fool** the discriminator into thinking the fake data is real.

2. Discriminator (D):

- Takes input data (either real from the training dataset or fake from the generator).
- Tries to **classify** whether the input is real or fake.
- Outputs a probability (real = 1, fake = 0)

🔁 Training Process of a GAN (Adversarial Training)

Step 1: Generator tries to fool the Discriminator

- Generator creates **fake data** from random noise.
- Discriminator tries to **detect** if the data is real or fake.

Step 2: Discriminator gets trained

1. It's trained on:
    a. **Real data** (labeled as real)
    b. **Fake data** from the generator (labeled as fake)
2. It learns to **increase accuracy** in distinguishing real vs fake

Step 3: Generator gets trained

- It is trained to **improve the quality** of its fake data.
- The goal: make the discriminator **classify fake data as real**.
- This means the Generator is rewarded when it **fools the Discriminator**.

🔥 The Competition Between Generator and Discriminator in GANs

GAN is a competitive "game" between two neural networks — the Generator and the Discriminator — each with opposing goals:

- **Generator's Goal:** To create fake data (like images) that look so real, the Discriminator **cannot tell they're fake**. It wants to **fool the Discriminator** into believing its outputs are genuine.
- **Discriminator's Goal:** To correctly **distinguish between real data** from the training set and **fake data generated** by the Generator. It tries to be a perfect "detective" who **never gets fooled**.

How the Competition Works:

1. **Generator makes fake data** from random noise.
2. **Discriminator evaluates this data** along with real data.
3. Discriminator tries to **assign a high probability to real data** and a **low probability to fake data**.
4. The Generator gets **feedback based on how well it fooled the Discriminator**.
5. Both networks **update their parameters**:
    - The Discriminator learns to better detect fakes.
    - The Generator learns to create more realistic fakes.

5.2 If time permits, explain the concept behind Diffusion Models, focusing on the forward noising and reverse denoising process, and compare the key differences between how GANs and Diffusion Models generate samples.

Diffusion models are a type of **generative model** that create data (like images) by **gradually transforming random noise into realistic samples** through a step-by-step process.

🔄 Forward Noising Process

- The model starts with **real data** (like an image).
- It **gradually adds Gaussian noise step-by-step** over many iterations.
- This process follows a **Markov chain**, meaning each step only depends on the previous one.
- By the end, the original data is almost completely turned into **pure noise**.
- This process is called the **forward diffusion or noising** process.
- It's like slowly "destroying" the data by adding small amounts of noise each step.

🔙 Reverse Denoising Process

- The model then **learns how to reverse this noising process**.
- The model (often based on **CNN** or **U-Net** architecture) learns to **predict and remove the noise** added in each time step.
- Starting from pure noise, it **removes the noise step-by-step**, trying to recover the original data.
- The training objective is typically to **minimize the Mean Squared Error (MSE)** between the actual noise added and the predicted noise.

- This reverse process is called **denoising**.
- By following these learned denoising steps, the model can generate new, realistic samples **from random noise**.
- Essentially, the model learns a pathway from noise back to data.
- More advanced models try to predict not just the noise, but the **mean and variance** of the data at each step — essentially **learning a distribution** to sample the clean data from.

Differences:

GANs create data in a single step by transforming a random noise vector into a realistic output using a generator network, which competes against a discriminator network in an adversarial training setup. This competition can make GANs challenging to train, sometimes causing instability and issues like mode collapse, where the generator produces limited varieties of samples. On the other hand, Diffusion Models generate data through a slow, multi-step process. They start with pure noise and gradually remove noise step-by-step, effectively reversing a learned noising process. This iterative denoising approach makes training more stable and less prone to collapse, although it typically requires more computation time. Unlike GANs, which use two networks working adversarial, Diffusion Models rely on a single network trained to predict and remove noise at each step. While GANs can quickly produce sharp and realistic outputs, Diffusion Models are often better at generating diverse and high-quality samples but with slower generation speed.

6 Recurrent Neural Networks (RNNs) and Memory (Lecture 6)

6.1 How do recurrent connections in RNNs allow the network to process sequential data?

Recurrent Neural Networks (RNNs) are designed to handle data that comes in a sequence, like text, music, or time-series data. What makes RNNs special is that they have **recurrent connections**—which means they can **remember past information**.

At each step, an RNN not only looks at the current input (like a word or number) but also **uses information from the previous step**. This way, it builds a kind of "memory" of what it has seen so far.

For example, if you're reading a sentence, knowing the previous words helps understand the current word better. RNNs do this using **recurrent layers**, which pass information forward in time.

To train RNNs, a method called **Backpropagation Through Time (BPTT)** is used. It helps the model learn which past information is important by updating the weights through multiple time steps.

Because of this design, RNNs are very **versatile** and useful in things like text generation, language translation, and predicting future values in time-series data.

Recurrent Neural Networks (RNNs) are specially designed to handle **sequential data** by using **recurrent connections** inside their architecture. At the core are **recurrent neurons**, which don't just take the current input, but also receive feedback from their own output at the previous time step. This means that at each time step, a recurrent neuron processes the new input along with information from the past, allowing it to maintain a form of **memory** over the sequence.

These neurons are organized into **recurrent layers**, where each layer passes not only the output forward but also loops the hidden state back into itself. This looping enables the network to carry information from earlier steps in the sequence forward to influence later steps, capturing temporal dependencies such as word order in language or trends in time series data.

Training RNNs involves a technique called **Backpropagation Through Time (BPTT)**. Since the same recurrent neurons are used at every time step, BPTT unfolds the network across the sequence length, computing gradients at each step and propagating errors backward through time. This allows the network to learn how earlier inputs affect later outputs by adjusting weights based on the entire sequence context.

One of the strengths of RNNs is their **versatility**. Because they can process sequences of varying length and remember past information, they are widely used in many applications like natural language processing, speech recognition, and even video analysis. Their ability to model temporal patterns and dependencies makes them a fundamental tool for any task involving sequential data.

6.2 Discuss the limitations of simple RNNs for long-term dependencies and explain how architectures like LSTMs and GRUs use gating mechanisms to address these issues.

Simple Recurrent Neural Networks (RNNs) are powerful for processing sequential data, but they struggle with learning **long-term dependencies**—that is, remembering information from many steps back in a sequence. This problem arises mainly due to the **vanishing and exploding gradient issues** during training. When backpropagating errors through many time steps, the gradients either become extremely small (vanish) or extremely large (explode). As a result, simple RNNs have difficulty updating weights effectively for distant past information, causing them to forget important context over long sequences.

To solve this, advanced architectures called **Long Short-Term Memory (LSTM)** networks and **Gated Recurrent Units (GRUs)** were developed. Both LSTMs and GRUs introduce **gating mechanisms** that control the flow of information inside the network, helping it to retain or forget information selectively.

An **LSTM** uses three main gates: the **input gate**, **forget gate**, and **output gate**. The forget gate decides which parts of the previous cell state to keep or discard. The input gate controls how much new information to add to the cell state. The output gate decides what information to output at the current step. By maintaining a separate **cell state** that runs through the sequence with minor linear interactions, LSTMs avoid the vanishing gradient problem and effectively preserve long-term dependencies.

Similarly, **GRUs** simplify this process with only two gates: the **reset gate** and the **update gate**. The reset gate determines how much past information to forget, while the update gate controls how much new information to incorporate. GRUs are computationally simpler than LSTMs but still effectively address the problem of long-term dependencies by controlling information flow.

In summary, by using these gating mechanisms, LSTMs and GRUs can selectively remember or forget information, allowing them to model long-term relationships in sequential data much better than simple RNNs. This makes them highly effective in applications like language modeling, speech recognition, and time series prediction where long-range context matters.

7 Text Generation with RNNs and Embeddings (Lecture 7)

7.1 Explain how a character-level RNN (Char-RNN) can be trained to generate text.

A character-level RNN generates text by learning to predict the next character in a sequence, one step at a time. The training process involves three main layers:

1. **Embedding Layer:**
   First, each input character is converted from a simple index or one-hot vector into a dense, meaningful vector representation by the embedding layer. This transformation helps the model understand relationships and similarities between different characters.

2. **RNN Layer:**
   Next, the embedded character vector is fed into the recurrent neural network (RNN) layer. This is the heart of the model. The **RNN layer** reads one character at a time and remembers what came before using its **hidden state**. So if the input is "hel", it remembers "h" and "e" while predicting "l". At every time step, the RNN combines the current embedded input with its previous hidden state to create a new hidden state that reflects the sequence so far.

3. **Fully Connected (Output) Layer:**
   This layer takes the RNN's output and predicts the next character by assigning **probabilities to all possible characters** (using a softmax). The character with the highest score is the model's guess.

During training, the model compares the predicted next character with the actual next character from the training text, calculates the error using a loss function, and then updates its weights through backpropagation to minimize this error. By repeating this process across many sequences, the Char-RNN learns to generate text that resembles the training data.

7.2 Describe the role of character embeddings and the SoftMax output layer. How does temperature sampling influence the creativity and coherence of the generated text?

In a character-level RNN, the **character embeddings** play a crucial role at the input stage. Instead of using one-hot encoded vectors (which are sparse and don't capture relationships), each character is mapped to a **dense, continuous vector** using an embedding layer. These embeddings help the model learn patterns between characters more efficiently.

After processing the input sequence through the RNN, the final hidden state is passed to the **Softmax output layer**. This layer generates a **probability distribution** over all possible characters, predicting which character is most likely to come next. The softmax ensures all probabilities add up to 1 and allows us to sample a character from this distribution.

To control the **style of the generated text**, we use a technique called **temperature sampling**. The **temperature** is a value used when applying the softmax function: Temperature controls how **creative or confident** the model is when picking the next character:

- A **low temperature** (e.g., 0.2) Makes the model more **predictable and safe** —it strongly prefers the highest probability character, resulting in **predictable but less creative** text.
- A **high temperature** (e.g., 1.0 or higher) flattens the probability distribution, making the model more likely to pick less probable characters, which can result in **more diverse or creative** text—but also **more mistakes or less coherence**.

By adjusting the temperature, we can **balance creativity and accuracy** when generating new sequences from a trained model.

8 Large Language Models and Transformers (Lecture 8)

Explain the encoder-decoder architecture of transformers and the purpose of attention mechanisms. How does self-attention work, and why was it a breakthrough that enabled the development of large language models (LLMs)? Describe one advantage of transformer-based models over earlier RNN-based language models.

8.1 Explain the encoder-decoder architecture of transformers and the purpose of attention mechanisms.

The Transformer is a model mainly used for **sequence-to-sequence tasks**, like **language translation**.It uses an **encoder-decoder** architecture:

🧠 **Encoder:**

- The **encoder** reads the **input sentence** (e.g., in English) and turns it into **embeddings** — numerical representations of the words.
- It has **multiple layers** that use something called **self-attention**, which lets each word "look at" other words in the sentence to understand context (like knowing "bank" means a riverbank, not a money bank).
- This helps the model understand the **meaning and structure** of the input.

🗣️ **Decoder:**

- The **decoder** takes the encoder's output and **generates the output sentence** (e.g., in French), one word at a time.
- It uses **two types of attention**:
  - **Self-attention** to look at the words it has already generated.
  - **Cross-attention** to focus on the encoder's output (the input sentence).
- This helps it decide **what to say next**, based on both the original sentence and what it has said so far.

Both the encoder and decoder are built using **attention mechanisms**, particularly **self-attention** and **cross-attention**.

🔹 **Purpose of Attention Mechanisms:**

The **attention mechanism** is what makes transformers so powerful. It allows the model to **focus on different parts of the input** when processing each word. Instead of treating all words equally, the model decides **which words are most relevant** for a given word. This is different from older models (like RNNs), which processed sequences word by word. Transformers can handle whole sentences at once, which is faster and captures better relationships.

- **Self-attention** helps the model understand relationships **within the same sentence**. For example, in the sentence "The cat that chased the mouse was fast," the word "cat" and "was" are far apart, but self-attention helps the model understand they are related.

- In the decoder, **cross-attention** lets the decoder look back at the encoder's output, focusing on the most relevant parts of the input sentence to generate the next word in the output sentence.

This attention-based approach allows transformers to **handle long sequences**, **learn context**, and **process all words in parallel**, making them much faster and more accurate than older models like RNNs.

8.2 How does self-attention work, and why was it a breakthrough that enabled the development of large language models (LLMs)? Describe one advantage of transformer-based models over earlier RNN-based language models.

🔹 How does Self-Attention Work?

Self-attention is a mechanism that allows a model to **look at all the words in a sentence at once** and decide **which words are important for understanding each word**.

Imagine the sentence:
"The cat sat on the mat."
When the model is trying to understand the word **"sat"**, self-attention helps it **focus on "cat"** because the cat is the one doing the action.

each word is turned into three vectors:

- **Query (Q)** – What am I looking for?
- **Key (K)** – What do I have?
- **Value (V)** – What information do I give?

The model compares the **Query** of a word (like "sat") to the **Keys** of all the words (like "cat", "the", "mat") to calculate **attention scores** — this tells the model **how much to focus on each word**.

Then it uses those scores to mix the **Values**, and this gives a new, better version of the word's meaning — with **context** added.

This process happens **in parallel for all words**, and it helps the model understand **context**, especially for words that depend on others (like pronouns or long-range dependencies).

🔹 Why Was Self-Attention a Breakthrough for LLMs?

Self-attention was a major breakthrough because:

- It allows models to **process entire sentences at once**, rather than one word at a time like RNNs.
- It **captures relationships between words**, no matter how far apart they are.
- It's **highly parallelizable**, meaning it can use GPUs efficiently and scale to very large datasets and models — which is critical for building large language models (LLMs) like GPT.

This made training much faster and enabled models to learn **deeper and more meaningful representations of language**.

- **Advantage of Transformers over RNNs:**

Transformers process entire sequences in parallel, while RNNs process them word-by-word.
This makes transformers **much faster to train**, especially on long sequences, and they **don't suffer from problems like vanishing gradients** as severely as RNNs do.

9 Reinforcement Learning Paradigms (Lecture 9)

Describe the components of a reinforcement learning system: agent, environment, states, actions, rewards, and returns. Compare value-based methods (such as Q-learning/DQN) with policy-based methods (such as policy gradients) and explain how each approach learns an optimal policy.

9.1 Describe the components of a reinforcement learning system: agent, environment, states, actions, rewards, and returns.

Reinforcement Learning is a type of machine learning where an **agent** learns to make decisions by interacting with an **environment**. The agent receives **rewards** for good actions and tries to learn a **strategy (policy)** that gives it the highest total reward over time.

- **Components of Reinforcement Learning**

- **Agent**
  The **agent** is the learner or decision-maker. It's the one that **interacts with the environment**, chooses actions, and learns from the outcomes. For example, in a chess game, the agent is the player.
- **Environment**
  The **environment** is everything the agent interacts with. It responds to the agent's actions and provides feedback. In the chess example, the board and rules of the game are part of the environment.
- **States** **(s)**
  A **state** is a snapshot of the environment at a specific moment. It gives the agent information about where it currently is. For instance, in a maze, a state might be the agent's current position.
- **Actions** **(a)**
  Actions are the choices the agent can make in a state. After observing a state, the agent picks an action. In a video game, moving left, right, or jumping would be actions.
- **Reward** **(r)**
  A **reward** is the feedback signal from the environment after taking an action. It tells the agent **how good or bad** its action was. The goal of the agent is to maximize the total reward it gets over time.

- **Return** **(G)**
  The **return** is the **total accumulated reward** an agent expects to receive in the future, often **discounted** over time. It helps the agent not just focus on immediate rewards, but also on rewards it can earn later

9.2 Compare value-based methods (such as Q-learning/DQN) with policy-based methods (such as policy gradients) and explain how each approach learns an optimal policy.

🔹 **What is a Policy?**

a **policy** is the agent's strategy — it tells the agent what action to take in each state.

🔵 **Value-Based Methods (like Q-learning / DQN)**

- **What it learns**: These methods **learn a value function**, such as the **Q-function** (Q(s, a)), which estimates how good an action is in a given state.
- **How it works**:
  The agent learns values for different actions in each state, then follows the **best action** (highest value).
- **Example**: In **Q-learning**, the agent updates Q-values based on the reward received and the estimated value of the next state.
- **DQN (Deep Q-Network)**: Uses a neural network to estimate Q-values for complex problems like playing Atari games.

Main idea:
👉 First learn the values → then extract the best policy (by picking the highest-valued action).

🔴 **Policy-Based Methods (like Policy Gradients)**

- **What it learns**: These methods **directly learn the policy** — that is, the probabilities of taking actions in each state.
- **How it works**:
  They **don't estimate action values**. Instead, they adjust the policy to **maximize expected reward** using techniques like gradient ascent.
- **Example**: In **policy gradient**, the agent improves its policy by calculating how small changes would affect the expected return and adjusting accordingly.

Main idea:
👉 Directly learn the best policy without estimating action values.

In reinforcement learning, both value-based and policy-based methods aim to learn the optimal policy, but they do it in different ways. Value-based methods like Q-learning learn a value function, which estimates how good each action is in a state. The optimal policy is then derived by picking the action with the highest estimated value. In contrast, policy-based methods like

policy gradients learn the policy directly — they adjust the probabilities of actions in each state to maximize expected rewards. Value-based methods are typically more sample efficient and work well in discrete action spaces, but they can become unstable when used with deep networks. Policy-based methods are more stable and are better suited for continuous or stochastic environments, although they may have higher variance in updates.

10 New Frontiers in AI, Agentic Systems, and Ethical Challenges (Lecture 10)

Discuss emerging directions in deep learning, particularly Large Language Models (LLMs) and generative models. What are the known limitations of frontier AI systems, such as hallucinations, generalization issues, and uncertainty? Identify key ethical challenges in AI systems in the real world.

10.1 Discuss emerging directions in deep learning, particularly Large Language Models (LLMs) and generative models.

In recent years, **deep learning** has moved far beyond basic image and speech recognition. Two of the most exciting and powerful new directions are:

1. 🔠 Large Language Models (LLMs)

LLMs like **GPT (Generative Pre-trained Transformer)**, **BERT**, and **Claude** are deep learning models trained on massive amounts of text. They are built on the **transformer architecture**, which uses a mechanism called **self-attention** to understand the meaning of words in context.

- **Training:** They are first trained using a huge amount of unlabelled text, where they learn to predict the next word or fill in missing words.
- **Capabilities:** LLMs can write essays, summarize articles, translate languages, answer questions, write code, and even carry on a conversation.
- **Few-shot and zero-shot learning:** These models can do new tasks with little or no training data, just by being given an example in the prompt.

Why it matters:
LLMs are revolutionizing natural language understanding and generation. They are being used in virtual assistants, customer support, content generation, education, and research.

2. 🎨 Generative Models

These models **generate new data** — such as images, text, music, or video — that looks similar to the data they were trained on.

There are a few main types:

a. GANs (Generative Adversarial Networks):

- Two networks — a **generator** and a **discriminator** — compete with each other.
- The generator tries to create fake data, and the discriminator tries to detect if the data is real or fake.

- Over time, the generator gets so good that it can make very realistic images.

b. Diffusion Models:

- These models add random noise to data and then learn to reverse that noise step by step to create new images.
- Examples: **DALL·E 2**, **Stable Diffusion**, **MidJourney**.

c. VAEs (Variational Autoencoders):

- These learn a compact version of the data (latent space) and generate new examples from that space.

Applications of generative models:

- Creating realistic art and images from text prompts
- Making synthetic data for training
- Denoising images or reconstructing missing data
- Generating music and videos

10.2 *What are the known limitations of frontier AI systems, such as hallucinations, generalization issues, and uncertainty?*

Even though frontier AI systems like Large Language Models (LLMs) are very powerful, they still have **significant limitations** that researchers are actively working to improve. The main ones include:

1. 🤯 Hallucinations (Making Up Information)

One of the most serious limitations is **hallucination**, where the AI **confidently generates false or made-up information**. For example, it might:

- Invent a fake quote or source,
- Give a wrong math answer with confidence,
- Generate code that looks real but doesn't work.

This happens because LLMs are **not truly reasoning** or checking facts — they are generating what **looks likely based on patterns in their training data**.

2. 📉 Generalization Issues

Frontier models can generalize well in some areas but still:

- Struggle with **out-of-distribution data** (data very different from what they saw during training),
- Fail at **logical reasoning** or **multi-step problem solving**,
- Underperform when **domain-specific expertise** is required (like advanced medical advice or legal interpretations).

These issues arise because models **learn patterns**, not deep understanding.

3. ❓ Uncertainty and Lack of Confidence Estimation

AI models don't **know when they don't know**.

- They often give **confident-sounding answers** even when they're unsure.
- They do not natively provide **reliable uncertainty estimates**, unless extra methods are used (like confidence scoring or calibration).

This can be dangerous in high-stakes applications like healthcare, finance, or autonomous driving.

10.3 Identify key ethical challenges in AI systems in the real world.

There are several major ethical challenges in real-world AI systems:

1. **Do No Harm & Proportionality** – AI must not be used in ways that cause unnecessary harm or go beyond legitimate use. Risk assessments should be in place to minimize potential damage.
2. **Safety and Security** – AI systems are vulnerable to attacks or malfunction, which can lead to safety risks. Ensuring robustness and protection against cyber threats is crucial.
3. **Privacy and Data Protection** – AI often uses personal data, raising concerns about privacy. It's important to have strong data protection laws and transparent data handling practices.
4. **Lack of Transparency and Explainability** – Many AI systems act as "black boxes," making it hard to understand how they make decisions. This reduces trust and accountability.
5. **Responsibility and Accountability** – It can be unclear who is responsible when AI causes harm — the developer, the user, or the company. There must be clear accountability structures.
6. **Bias and Discrimination** – AI can reflect or amplify biases present in data, leading to unfair or discriminatory outcomes, especially in hiring, policing, or lending.
7. **Human Oversight** – In some critical domains like healthcare or justice, AI must not replace human judgment. Humans must retain control and responsibility.
8. **Sustainability** – Training large AI models consumes a lot of energy and resources. AI should be aligned with environmental and sustainability goals.
9. **Governance and Global Collaboration** – There is a lack of global standards for AI. Ethical AI development requires cooperation across countries and sectors.
10. **Awareness and Literacy** – Many people don't understand how AI works or affects them. Increasing public awareness and digital literacy is key to ethical use.

11 Final Assignment

Test- Train split

Splited into complete pair

Normalization:
This means we scale pixel values from the usual range of 0–255 to a range of **-1 to 1**. This step is very important because deep learning models tend to perform better and converge faster when the input data is normalized.

Data Preparation:

BATCH_SIZE = 8

- During training, instead of processing one image at a time, we **group images into mini-batches**.
- This improves training speed and stability (e.g., via gradient averaging).
- **Smaller batch size (like 8)** uses less GPU memory, but might result in noisier updates.

SHUFFLE_BUFFER = 1000

- Defines the size of a **buffer** used when **shuffling** the data.
- A buffer of 1000 means it holds 1000 samples in memory at once and shuffles them randomly before yielding them.
- Large buffers = better randomness = better training (but more memory used).

Prefetching:

- Tells TensorFlow to **prepare the next batch while the current one is being processed**.
- This reduces training time by overlapping **data loading and model execution**.

12 Defining Model

The Config class organizes all hyperparameters and training settings in one place, which is critical for managing experiments and controlling training dynamics. Stable Diffusion is a powerful pretrained generative model used here because it efficiently produces high-quality images and can be adapted to generate detailed images from sketches, leveraging learned representations from large-scale image datasets.

I set gradient_accumulation_steps to 2 because directly using a batch size of 16 would not fit into my GPU memory. Accumulating gradients over two smaller batches of 8 allows me to simulate an effective batch size of 16, improving training stability and convergence, while working within the memory constraints of my hardware.

Stable diffusion model:

How Stable Diffusion is Trained — Simply Explained

1. **Start with lots of images and their descriptions**
   You need a big collection of images and matching text descriptions (like captions).
2. **Convert images into a smaller, simpler form (latent space)**
   Instead of working directly with huge images, the model compresses them into a smaller "latent" space — kind of like a summary that's easier to handle.
3. **Add noise to these latent images gradually**
   The model learns by starting with a clear image (in latent space) and then adding random noise step-by-step until the image looks like pure noise.
4. **Train the model to reverse the noise process**
   The core task for the model is to learn how to take a noisy image and gradually remove the noise to get back the original image — basically learning how to "denoise."
5. **Use text descriptions as guidance**
   While denoising, the model also uses the text description (prompt) to guide the image it's creating — so it learns to generate images that match the text.
6. **Repeat many times over the dataset**
   The model sees millions of images and their noise-added versions and keeps learning to improve its denoising skill.
7. **After training, it can create images from scratch**
   Once trained, you can start with random noise and feed in a text prompt — the model will slowly remove noise step-by-step, generating an image that fits the prompt.

ControlNet

ControlNet is an extension of Stable Diffusion designed to give you more precise control over the image generation process using extra input conditions, like sketches, depth maps, edge maps, or any guiding structure.

Normally, **Stable Diffusion** can create amazing images from just a text prompt. Stable Diffusion by itself **doesn't always follow the structure well** — it may create something different from your sketch.

ControlNet solves this problem by "controlling" the generation process with your input image (like a sketch).

What does ControlNet do that Stable Diffusion cannot do alone?

- Without ControlNet, if you give just a sketch and a text prompt, Stable Diffusion **might ignore or loosely follow the sketch**.
- It can create an image, but **it won't respect the details or shape of your sketch properly**.
- ControlNet **adds a special "control branch"** that processes your sketch and tells the diffusion model: *"Stick to this shape and details!"*
- This helps generate images that **look exactly like your input structure**, but with realistic textures/colors from the text prompt.

Guidance: During the diffusion process (when noise is gradually removed to form an image), ControlNet influences the UNet layers, **making sure the output image respects the input structure**.

UNet

How does UNet work inside Stable Diffusion?

1. **Input:** UNet gets a noisy image (the current state in the diffusion process) plus some conditioning info (like text embeddings).
2. **Downsampling:** It compresses the noisy image into smaller representations, extracting important features.
3. **Bottleneck:** It processes these compressed features in the middle, capturing global context.
4. **Upsampling:** It then reconstructs the image step-by-step, combining features from the downsampling path (skip connections) with the upsampled features.
5. **Output:** A cleaner image prediction with less noise.

Why is VAE important in Stable Diffusion?

- Stable Diffusion doesn't work directly on full images (which are large and complex).
- Instead, it works on **compressed representations** of images in a smaller latent space.
- **VAE is the tool that converts between real images and these smaller latent codes**.
- This compression saves memory and speeds up training and generation.