Deep Learning Notes

Himanshu Kesarvani

October 6, 2024

A **Perceptron** is the simplest form of a neural network model, often considered the building block of more complex neural networks like multilayer perceptrons and deep learning models. It is a type of **binary classifier that maps an input vector to an output using a simple mathematical model.

1 Key Concepts of the Perceptron:

1. **Inputs**: The perceptron receives inputs (features) from a dataset.
2. **Weights**: Each input is assigned a weight that determines its importance.
3. **Weighted Sum**: The inputs are multiplied by their respective weights and summed.
4. **Activation Function**: The sum is passed through an activation function to produce an output.
5. **Bias**: An additional term is added to shift the output, allowing for more flexibility.

2 Structure of a Perceptron:

- **Inputs**: $(x_1, x_2, ..., x_n)$ (features of the data)
- **Weights**: $(w_1, w_2, ..., w_n)$ (weights corresponding to the inputs)
- **Bias**: (b) (a constant term added to the weighted sum)
- **Output**: (y) (the final classification result)

The perceptron computes a **weighted sum** of the inputs, adds the bias, and applies an **activation function** to produce an output. The formula for the weighted sum is:

$$z = \sum_{i=1}^{n} w_i x_i + b$$

Where:

- (w_i) are the weights,
- (x_i) are the inputs,
- (b) is the bias.

3 Weighted Sum (Net Input)

The weighted sum (or linear combination) is calculated as:

$$z = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$$

This **z** value is a linear combination of the inputs and their corresponding weights. It essentially tells how strongly the inputs contribute to the final output.

4 Activation Function

The **activation function** determines the output of the perceptron based on the value of the weighted sum** (z)**. For the perceptron, we typically use the **step function**, also known as the **Heaviside function**, for binary classification. It outputs either 0 or 1 depending on whether (z) is greater than or less than a certain threshold (usually 0).

The step function is:

$$y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

This means:

- If the weighted sum (z) is positive or zero, the output will be **1** (positive class).
- If the weighted sum (z) is negative, the output will be **0** (negative class).

5 Full Perceptron Formula:

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} w_ix_i + b \geq 0 \\ 0 & \text{if } \sum_{i=1}^{n} w_ix_i + b < 0 \end{cases}$$

In this equation:

- **Inputs** $(x_1, x_2, ..., x_n)$ are features (e.g., pixel values of an image, measurements in a dataset).
- **Weights** $(w_1, w_2, ..., w_n)$ are the coefficients that adjust the influence of each feature.
- **Bias** (b) is a constant term that shifts the decision boundary.

6 Bias Term

The **bias** (b) helps control the position of the decision boundary. Without the bias, the decision boundary is forced to pass through the origin, which might not always be the best fit for the data. The bias gives more flexibility by shifting the boundary to better separate the data.

7 Perceptron Example:

Let's consider a simple binary classification example with two inputs.

7.1 Example:

Suppose you have two inputs, (x_1) and (x_2), with weights ($w_1 = 0.5$) and ($w_2 = -0.6$), and a bias of ($b = 0.2$).

The formula becomes:

$$z = 0.5 \cdot x_1 + (-0.6) \cdot x_2 + 0.2$$

Now, let's input the values ($x_1 = 1$) and ($x_2 = 2$):

$$z = 0.5 \cdot 1 + (-0.6) \cdot 2 + 0.2 = 0.5 - 1.2 + 0.2 = -0.5$$

Now, apply the step function:

$$y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

Since ($z = -0.5$), which is less than 0, the output is ($y = 0$). Therefore, the perceptron classifies this input as belonging to the negative class (0).

8 Training a Perceptron

During training, a perceptron uses a method called **gradient descent** (or in simpler cases, **perceptron learning algorithm**) to adjust its weights based on the error between the predicted and actual output. The update rule for weights is:

$$w_i = w_i + \Delta w_i$$

Where the weight update (Δw_i) is computed as:

$$\Delta w_i = \eta \cdot (y_{\text{true}} - y_{\text{predicted}}) \cdot x_i$$

- (η) is the **learning rate**, which controls how much the weights change in each step.
- (y_{true}) is the actual label of the data point.
- ($y_{\text{predicted}}$) is the output of the perceptron.

9 Perceptron as a Linear Classifier

The perceptron can only solve linearly separable problems because its decision boundary is a **hyperplane**. It struggles with problems like XOR, where the classes are not linearly separable, which led to the development of more complex neural networks.

10 Example of Linearly Separable Problem:

Consider the problem of classifying points in 2D space into two categories (positive or negative) based on their location. If the points can be separated by a straight line, a perceptron can learn the weights to classify them correctly.

11 Limitations of Perceptron:

- **Linearly Separable Data**: A perceptron can only classify data that is linearly separable. If the classes cannot be separated by a straight line, the perceptron will fail.
- **No Non-linearity**: It uses a step function as the activation function, which is not differentiable. This limits its ability to learn more complex patterns.

12 Summary:

- The **perceptron** is the simplest form of a neural network model and serves as a linear binary classifier.
- It works by calculating a **weighted sum** of its inputs, adding a **bias**, and passing the result through an **activation function** (step function).
- **Weights** and **bias** are adjusted during training based on errors in classification using a **learning rule**.

This simple yet powerful idea laid the foundation for modern neural networks and deep learning models. —

A **Convolutional Neural Network (CNN)** is a type of deep learning model specifically designed for processing grid-like data, such as images. It has proven highly effective in image recognition, classification, and various other computer vision tasks. CNNs automatically learn spatial hierarchies of features from input images by applying filters in multiple layers.

13 Key Concepts in CNNs:

1. **Convolution Operation**
2. **Filters/Kernels**
3. **Activation Function**
4. **Pooling/Downsampling**
5. **Fully Connected Layers**
6. **Training and Backpropagation**

14 1. Convolution Operation

The core idea behind CNNs is the **convolution** operation. In convolution, a small matrix called a **filter** or **kernel** slides across the input image to detect patterns, such as edges, textures, or colors.

14.1 Example:

Suppose you have a grayscale image of size 5x5 pixels:

```
Input Image (5x5):
[[10, 20, 30, 40, 50],
 [60, 70, 80, 90, 100],
 [110, 120, 130, 140, 150],
 [160, 170, 180, 190, 200],
 [210, 220, 230, 240, 250]]
```

Now, let's say you have a 3x3 filter (kernel):

```
Filter (3x3):
[[1, 0, -1],
 [1, 0, -1],
 [1, 0, -1]]
```

The filter slides over the image and performs an element-wise multiplication with the section of the image it's currently covering. The result of the element-wise multiplication is summed up to produce a single value. This value is stored in a new output matrix called a **feature map or convolved feature**.

15 Filters/Kernels

A **filter** is a small matrix of weights that captures features like edges, textures, or other patterns in an image. Multiple filters are applied in a CNN to capture different features. In the early layers, filters detect basic patterns (e.g., edges), and in deeper layers, they capture more complex patterns (e.g., shapes, objects).

Filters are **learned** during training. The network adjusts the weights of these filters using backpropagation to optimize feature detection for a given task.

16 Activation Function

After convolution, an activation function (commonly **ReLU**, or Rectified Linear Unit) is applied to introduce non-linearity. Without non-linearity, the CNN would behave like a linear model, which cannot learn complex patterns.

- **ReLU** is defined as:

$$\text{ReLU}(x) = \max(0, x)$$

This helps the network to focus on positive values and discard negative ones, which introduces the necessary non-linear properties to learn complex patterns.

16.1 Example of ReLU Activation:

```
Convolved feature (pre-activation):
[[5, -10],
 [-3, 7]]

After ReLU:
[[5, 0],
 [0, 7]]
```

17 Pooling/Downsampling

After convolution and activation, CNNs typically use a **pooling** layer to downsample the feature map. This reduces the spatial size, reducing the computational load and helping to extract dominant features.

- **Max Pooling** is the most common type, where the maximum value is taken from each region of the feature map.
- **Average Pooling** takes the average value instead.

17.1 Example of Max Pooling (2x2 filter with stride 2):

```
Input:
[[1, 3, 2, 4],
 [5, 6, 7, 8],
 [9, 2, 3, 1],
 [0, 1, 5, 2]]

Max Pooling result:
[[6, 8],
 [9, 5]]
```

Max pooling extracts the most important feature (highest value) in each 2x2 block.

18 Fully Connected Layers

Once convolution and pooling operations have been applied, the resulting feature map is flattened into a 1D vector. This vector is then passed to one or more **fully connected layers** (dense layers), where every node is connected to every other node. These layers act as a standard neural network that makes predictions.

18.1 Example:

If you have a 4x4 feature map after pooling:

```
Feature map:
[[1, 0, 2, 3],
 [4, 6, 5, 2],
 [0, 1, 3, 1],
 [5, 2, 2, 0]]

Flattened vector:

[1, 0, 2, 3, 4, 6, 5, 2, 0, 1, 3, 1, 5, 2, 2, 0]
```

The fully connected layers use this flattened vector to classify the image.

19 6. Training and Backpropagation

CNNs are trained through **backpropagation**, where errors from the final prediction are sent backward through the network to update the weights of the filters and neurons. This process is done over several iterations (epochs) to minimize the error.

The loss function typically used for image classification is **categorical cross-entropy**.

20 Example CNN for Image Classification (Simplified)

Let's assume we want to classify images of cats and dogs (binary classification).

1. **Input**: 64x64 color image (3 channels for RGB).
2. **Convolution Layer 1**: Apply a set of filters (e.g., 32 filters of size 3x3). The result is a set of feature maps, each highlighting specific features in the image.
3. **ReLU Activation**: Apply ReLU to introduce non-linearity.
4. **Max Pooling**: Downsample the feature maps by pooling (e.g., 2x2 filter).
5. **Convolution Layer 2**: Apply another set of filters (e.g., 64 filters of size 3x3).
6. **ReLU Activation**: Again, apply non-linearity.
7. **Max Pooling**: Pool the result to reduce dimensionality.
8. **Fully Connected Layer**: Flatten the pooled feature maps into a vector and pass them to a fully connected layer.
9. **Output**: Use a sigmoid function to output a probability for the two classes (cat or dog).

21 Why CNNs Are Effective for Images:

- **Local Connectivity**: Filters are small and learn local patterns, such as edges or textures.
- **Parameter Sharing**: The same filter is applied across different regions of the image, reducing the number of parameters and ensuring that the network learns useful global features.
- **Translation Invariance**: Pooling and convolution operations make the model robust to small shifts and distortions in the image.

22 Example Use Cases:

1. **Image Classification**: Recognizing whether an image contains a cat, dog, or other objects (e.g., using the popular ImageNet dataset).
2. **Object Detection**: Identifying and locating objects within an image (e.g., using YOLO or Faster R-CNN).

3. **Face Recognition**: Detecting and verifying faces in an image (e.g., using VGGFace).

4. **Medical Image Analysis**: Detecting tumors or abnormalities in medical scans (e.g., using CNNs in MRI or X-ray image analysis).

CNNs have revolutionized computer vision tasks due to their ability to learn and recognize complex patterns in images.

23 Perceptron

The **Perceptron** is one of the simplest types of artificial neural networks and a fundamental building block for more complex models. It is a binary classifier that attempts to separate data into two classes by learning a linear decision boundary.

24 Concept of a Perceptron:

- The perceptron takes a set of inputs, applies weights to them, and computes a weighted sum.

- It then passes this sum through an activation function (often a step function) to produce an output (usually 0 or 1, representing binary classification).

A perceptron works well with linearly separable data, meaning data that can be separated by a straight line (in 2D) or a hyperplane (in higher dimensions).

25 Structure of a Perceptron:

1. **Inputs**: $(x_1, x_2, ..., x_n)$ are the features of the input vector.

2. **Weights**: Each input (x_i) is associated with a weight (w_i). Weights determine the importance of each feature.

3. **Bias**: A bias term (b) is added to the weighted sum to help shift the decision boundary.

4. **Weighted Sum**: The perceptron computes a weighted sum of the inputs and bias.

$$z = \sum_{i=1}^{n} w_i x_i + b = \mathbf{w}^T \mathbf{x} + b$$

Where $(\mathbf{w} = [w_1, w_2, ..., w_n])$ is the weight vector and $(\mathbf{x} = [x_1, x_2, ..., x_n])$ is the input vector.

5. **Activation Function**: The perceptron uses an activation function (often the step function) to produce the final output.

$$y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

26 Perceptron Learning Rule (Mathematically):

The perceptron learning algorithm adjusts the weights and bias to minimize classification errors. It iterates through the dataset, updates the weights, and tries to correctly classify each input.

The **update rule** for the weights is based on the difference between the predicted output ($\hat{y}$) and the true label (y):

1. **Prediction**: Calculate the predicted output:

$$\hat{y} = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

2. **Weight Update Rule**: If the perceptron makes an incorrect prediction, the weights are updated using the following rule:

$$w_i \leftarrow w_i + \Delta w_i$$

Where (Δw_i) is the weight change, and is given by:

$$\Delta w_i = \eta \cdot (y - \hat{y}) \cdot x_i$$

Similarly, the bias (b) is updated as:

$$b \leftarrow b + \eta \cdot (y - \hat{y})$$

Here: - (η) is the **learning rate**, a small positive value that controls how large the updates are. - ($(y - \hat{y})$) is the **error term**, which tells whether the model's prediction was too high or too low.

The weights are only updated if the perceptron makes an incorrect prediction, meaning ($y \neq \hat{y}$).

27 Steps in the Perceptron Learning Algorithm:

1. **Initialize**: Start with random weights and bias (often small values).
2. **For each training example**:
   - Compute the weighted sum ($z = \mathbf{w}^T \mathbf{x} + b$).
   - Apply the activation function to get the prediction ($\hat{y}$).

- If the prediction is incorrect (i.e., $(\hat{y} \neq y)$), update the weights and bias:

$$w_i \leftarrow w_i + \eta \cdot (y - \hat{y}) \cdot x_i$$
$$b \leftarrow b + \eta \cdot (y - \hat{y})$$

3. **Repeat**: Iterate over the training data multiple times (epochs) until the model converges or a maximum number of iterations is reached.

28 Perceptron Example:

Suppose we have two inputs (x_1) and (x_2), and their corresponding weights (w_1) and (w_2). Let the bias be (b).

- Input: $(x = [x_1, x_2])$
- Weights: $(w = [w_1, w_2])$
- Bias: (b)

The decision function would be:

$$z = w_1 x_1 + w_2 x_2 + b$$

If the weighted sum (z) is greater than or equal to 0, the output is 1. Otherwise, it is 0.

For example, suppose we have:

- Inputs: $(x_1 = 2)$, $(x_2 = 3)$
- Weights: $(w_1 = 0.5)$, $(w_2 = -1)$
- Bias: $(b = 0.5)$

The weighted sum:

$$z = (0.5 \times 2) + (-1 \times 3) + 0.5 = 1 - 3 + 0.5 = -1.5$$

Since $(z < 0)$, the perceptron will output 0.

29 Perceptron Limitations:

- **Linearly Separable Data**: The perceptron can only classify data that is linearly separable. For more complex, non-linearly separable problems (e.g., XOR problem), the perceptron cannot find a decision boundary.
- **Single Layer**: The basic perceptron is a single-layer model (no hidden layers), which limits its ability to solve complex problems.

To overcome these limitations, we use **multilayer perceptrons (MLPs)** and more advanced models like deep neural networks, where multiple layers can learn more complex representations.

30 Geometric Interpretation:

- The perceptron attempts to find a **hyperplane** (in 2D, a line) that separates two classes of data points. The weights and bias define this hyperplane.
- The decision boundary is represented by the equation ($\mathbf{w}^T\mathbf{x} + b = 0$). Points for which this equation is true lie on the boundary, and the model classifies points on either side of this boundary.

In summary, the perceptron is a simple yet powerful concept, laying the foundation for more complex neural networks. The mathematical framework allows it to iteratively adjust its weights to classify linearly separable data.

31 CNN

A **Convolutional Neural Network (CNN)** is a specialized type of deep neural network primarily used for analyzing structured grid data, such as images. It is highly effective in tasks like image classification, object detection, and even natural language processing tasks where spatial hierarchies exist in the data.

32 Key Components of CNN:

1. **Convolutional Layers**: The core building block of a CNN where feature extraction occurs through convolution operations.
2. **Activation Function**: Non-linear activation (usually ReLU) applied after convolution to introduce non-linearity.
3. **Pooling (Subsampling) Layers**: Used to reduce the spatial dimensions of the input and thus reduce computational complexity.
4. **Fully Connected (Dense) Layers**: The last layers that perform high-level reasoning, combining all the extracted features into predictions.
5. **Output Layer**: Provides the final classification (or regression) output.

33 Mathematical Concepts Behind CNN:

33.1 1. Convolution Operation:

A convolution operation involves applying a filter (also known as a kernel) to an input (like an image) to extract features.

For a 2D input (e.g., an image), the convolution operation is mathematically defined as:

$$\text{Output}(i,j) = \sum_{m=1}^{M} \sum_{n=1}^{N} X(i+m-1, j+n-1) \cdot K(m,n)$$

Where:

- (X) is the input image or feature map (a 2D matrix).
- (K) is the convolution kernel (filter), typically a smaller matrix (e.g., 3x3 or 5x5).
- (M) and (N) are the dimensions of the kernel.
- (i,j) are the spatial indices of the output feature map.

This operation slides the kernel across the image, computing the dot product of the filter and a local region of the input.

33.2 Example:

For a 3x3 kernel (K) and an input image (X) of size 5x5, the convolution operation results in a smaller output (feature map). Stride and padding can control the output size.

33.3 2. Padding:

To preserve the spatial dimensions of the input after convolution, we often use padding. There are two types of padding:

- **Valid Padding (No Padding)**: Shrinks the output size as no extra pixels are added around the border.
- **Same Padding (Zero Padding)**: Adds zeros around the edges to keep the output size the same as the input.

The relationship between the input size (I), filter size (F), stride (S), and padding (P) is:

$$\text{Output Size} = \frac{I - F + 2P}{S} + 1$$

Where:

- (I) is the input size (height or width of the image).
- (F) is the filter size (height or width of the kernel).
- (S) is the stride (how far the filter moves each step).
- (P) is the padding (amount of zero-padding around the border).

33.4 3. Stride:

Stride defines how much the filter shifts over the input matrix at each step. A stride of 1 means the filter moves by 1 pixel, and larger strides lead to down-sampling by skipping pixels. This reduces the output size.

33.5 4. Activation Function:

After the convolution, a **non-linear activation function** is applied to the output of the convolution operation. The most commonly used activation function is **ReLU (Rectified Linear Unit)**, which is defined as:

$$f(x) = \max(0, x)$$

This introduces non-linearity to the model, allowing it to learn more complex patterns.

33.6 Pooling (Subsampling):

Pooling is used to reduce the dimensions of the feature maps while retaining important features. The most common types of pooling are:

- **Max Pooling**: Takes the maximum value from each region of the input. For a pooling window size (2×2):

$$\text{MaxPool}(x) = \max(x_1, x_2, x_3, x_4)$$

Where (x_1, x_2, x_3, x_4) are values in a 2x2 region.

- **Average Pooling**: Takes the average of the values in a pooling window.

Max pooling is often preferred in practice because it helps retain the most prominent features.

33.7 Fully Connected Layers:

After several convolutional and pooling layers, the output is typically flattened into a 1D vector and passed through fully connected (dense) layers, which perform high-level reasoning for classification or regression tasks. The output of the fully connected layer is computed as:

$$y = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

Where:

- $(\mathbf{W})$ is the weight matrix.

- $(\mathbf{x})$ is the input vector (flattened feature map).

- ($\mathbf{b}$) is the bias term.

- (σ) is the activation function (usually softmax for classification).

33.8 7. Softmax (For Classification):

In the case of multiclass classification, the final layer uses a **softmax** activation function, which converts the output into a probability distribution over (C) classes:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}$$

Where:

- (z_i) is the output score (logit) for class (i).

- (C) is the number of classes.

34 Backpropagation in CNN:

Training a CNN involves adjusting the weights of the filters and fully connected layers to minimize the loss function (e.g., cross-entropy loss for classification). This is done using **backpropagation** with **gradient descent**.

34.1 Steps in Backpropagation:

1. **Forward Pass**: Compute the output of the CNN for a given input using the convolution, activation, pooling, and fully connected layers.

2. **Compute Loss**: Calculate the error (loss) between the predicted output and the true labels.

3. **Backward Pass**:

   - **Gradient of Loss w.r.t Output**: Calculate how the loss changes with respect to the output (e.g., softmax probabilities).
   - **Chain Rule for Derivatives**: Use the chain rule to propagate the gradients back through the fully connected layers, pooling layers, and convolutional layers, updating weights accordingly.

4. **Update Weights**: Adjust the weights using gradient descent:

   $$w \leftarrow w - \eta \cdot \frac{\partial L}{\partial w}$$

   Where (η) is the learning rate, and ($\frac{\partial L}{\partial w}$) is the gradient of the loss function with respect to the weight.

35 Example Workflow of CNN:

Consider an input image of size $(32 \times 32 \times 3)$ (height, width, and 3 channels for RGB), and the goal is to classify this image into one of 10 classes.

1. **Convolutional Layer**: Apply a set of filters (e.g., 16 filters of size (3×3)), generating feature maps.
2. **ReLU Activation**: Apply ReLU to each feature map.
3. **Pooling Layer**: Max pooling with a (2×2) window reduces the spatial size, but retains important features.
4. **Repeat**: Additional convolutional and pooling layers extract more complex features.
5. **Flattening**: Convert the final feature maps into a 1D vector.
6. **Fully Connected Layer**: Pass the flattened vector through fully connected layers for final classification.
7. **Softmax**: Output probabilities for each class.

36 Summary:

- **Convolution Operation**: Extracts local features by sliding filters over the input.
- **Activation Functions**: Introduce non-linearity (ReLU is the most common).
- **Pooling**: Reduces the size of feature maps while retaining important information.
- **Fully Connected Layers**: Perform classification after feature extraction.
- **Backpropagation**: Used to update the weights during training by minimizing a loss function.

CNNs are effective for spatial data (like images) because of their ability to capture local and hierarchical patterns, making them fundamental to deep learning applications such as computer vision.

37 RNN

Recurrent Neural Networks (RNNs) are a type of neural network designed to handle sequential data by maintaining a memory of past inputs. This is achieved through the recurrence in its architecture, allowing it to process inputs sequentially while retaining information across time steps.

38 Mathematical Explanation of RNN

Let's denote:

- (x_t) as the input at time step (t),
- (h_t) as the hidden state (or memory) at time step (t),
- (y_t) as the output at time step (t),
- (W_x), (W_h), (W_y), (b_h), and (b_y) as learnable parameters (weights and biases).

38.1 Hidden State Update (Recurrence):

The hidden state at time (t), (h_t), depends on both the current input (x_t) and the previous hidden state (h_{t-1}):

$$h_t = f(W_x x_t + W_h h_{t-1} + b_h)$$

Here:

- (W_x) is the weight matrix that connects the input (x_t) to the hidden state.
- (W_h) is the weight matrix that connects the previous hidden state (h_{t-1}) to the current hidden state.
- (b_h) is a bias term.
- (f) is the activation function (often a (tanh) or ReLU function).

38.2 Output:

The output at each time step (t), (y_t), is typically calculated as:

$$y_t = g(W_y h_t + b_y)$$

Where:

- (W_y) is the weight matrix that connects the hidden state (h_t) to the output.
- (b_y) is a bias term for the output.
- (g) is typically a softmax function for classification tasks or a linear function for regression tasks.

39 Backpropagation Through Time (BPTT)

To train an RNN, we use **Backpropagation Through Time (BPTT)**, which is an extension of the standard backpropagation algorithm. Since the hidden states are dependent on previous states, errors need to be propagated back through all previous time steps. The gradients are computed with respect to each parameter (e.g., (W_x), (W_h)) by summing the contributions from every time step.

The gradients are calculated using the chain rule and accumulate across time steps. This can lead to two problems:

1. **Vanishing gradients**: Gradients shrink as they propagate backward in time, which makes learning long-term dependencies difficult.
2. **Exploding gradients**: Gradients can grow exponentially during backpropagation, leading to unstable updates.

These issues are often mitigated by using variants like **LSTM** (Long Short-Term Memory) or **GRU** (Gated Recurrent Units), which introduce gates to better control the flow of information.

40 Summary of Operations:

For each time step (t):

1. Compute the new hidden state:
$$h_t = f(W_x x_t + W_h h_{t-1} + b_h)$$
2. Compute the output:
$$y_t = g(W_y h_t + b_y)$$
3. Compute the loss and backpropagate the gradients through time using BPTT.

This structure allows the RNN to capture sequential dependencies and patterns in time series data, natural language, and other sequential tasks.

41 Exploding and Vanishing Gradient Problem

The **vanishing** and **exploding gradient problems** are common challenges when training deep neural networks, especially recurrent neural networks (RNNs) and deep feedforward networks. These problems occur during backpropagation when gradients either shrink exponentially (vanishing) or grow exponentially (exploding) as they propagate backward through layers.

Let's break down how these problems occur and the mathematical solutions to mitigate them.

42 Vanishing Gradient Problem

42.1 Problem Overview:

The **vanishing gradient problem** arises when gradients become very small during backpropagation. The weight updates get smaller and smaller, leading to very slow learning or no learning at all in deep networks.

42.2 Mathematical Explanation:

During backpropagation, the gradient at a layer is computed using the chain rule:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial h_n} \cdot \frac{\partial h_n}{\partial h_{n-1}} \cdot \cdots \cdot \frac{\partial h_1}{\partial W}$$

where L is the loss function, W is the weight matrix, and h_n are the activations.

The gradients for each layer are multiplied together. If the derivatives $\frac{\partial h_i}{\partial h_{i-1}}$ (which are influenced by activation functions) are small, multiplying many small numbers together causes the gradient to vanish. For example, if the derivative of the activation function $f'(z)$ is less than 1 (as with the sigmoid or tanh function), the gradient shrinks exponentially as it moves backward through the layers.

42.3 Solution 1: Activation Function Choice (ReLU)

Using the **ReLU (Rectified Linear Unit)** activation function helps reduce the vanishing gradient problem:

$$f(z) = \max(0, z)$$

The derivative of ReLU is:

$$f'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$$

For positive values, the derivative is 1, which prevents the gradients from vanishing during backpropagation. This helps maintain the magnitude of the gradient across layers.

42.4 Solution 2: Weight Initialization (Xavier and He Initialization)

Careful initialization of weights can also mitigate the vanishing gradient problem. If the initial weights are too small, gradients shrink even more. Two popular weight initialization strategies are:

- **Xavier initialization (Glorot initialization)**: Used when the activation function is tanh or sigmoid. It ensures that the variance of the activations and gradients remains roughly constant across layers.

$$W \sim \mathcal{N}(0, \frac{1}{n_{in}})$$

where n_{in} is the number of input units to the neuron.

- **He initialization**: Used with ReLU activation to preserve the variance of activations through layers.

$$W \sim \mathcal{N}(0, \frac{2}{n_{in}})$$

42.5 Solution 3: LSTMs and GRUs in RNNs

For RNNs, **LSTM** (Long Short-Term Memory) and **GRU** (Gated Recurrent Units) architectures help overcome the vanishing gradient problem by incorporating gating mechanisms (forget gate, input gate, and output gate). These gates control how much information is retained or forgotten over time, ensuring gradients can flow better over long sequences.

43 Exploding Gradient Problem

43.1 Problem Overview:

The **exploding gradient problem** occurs when gradients grow exponentially large as they are propagated backward through layers. This leads to unstable updates, causing the model's weights to diverge and make training unstable.

43.2 Mathematical Explanation:

Similar to the vanishing gradient problem, the chain rule applies:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial h_n} \cdot \frac{\partial h_n}{\partial h_{n-1}} \cdot \cdots \cdot \frac{\partial h_1}{\partial W}$$

If the derivatives $\frac{\partial h_i}{\partial h_{i-1}}$ are large, then the gradient grows exponentially. For example, with activation functions like ReLU, large gradients can accumulate as they propagate backward, leading to an explosion of values.

43.3 Solution 1: Gradient Clipping

Gradient clipping is a simple and effective method to prevent exploding gradients. It works by setting a threshold τ and scaling the gradient if it exceeds this threshold:

$$\text{if } \|\nabla W\| > \tau, \quad \nabla W = \tau \cdot \frac{\nabla W}{\|\nabla W\|}$$

This prevents the gradient from becoming too large and causing instability.

43.4 Solution 2: Weight Regularization (L2 Regularization)

L2 regularization (also known as weight decay) helps control the growth of weights by adding a penalty term to the loss function. This keeps the weights from becoming too large, indirectly mitigating the exploding gradient problem:

$$L_{\text{reg}} = L + \lambda \sum W^2$$

where λ is the regularization strength.

43.5 Solution 3: Batch Normalization

Batch normalization normalizes the activations within a layer to have zero mean and unit variance. This helps stabilize the training process, preventing large updates that can lead to exploding gradients:

$$\hat{h}_i = \frac{h_i - \mu_B}{\sigma_B}$$

where μ_B and σ_B are the batch mean and standard deviation, respectively.

44 Summary:

1. **Vanishing Gradients**:
   - Use ReLU instead of sigmoid or tanh.
   - Use better weight initialization (Xavier or He).
   - For RNNs, use LSTMs or GRUs.
2. **Exploding Gradients**:
   - Use gradient clipping.
   - Apply weight regularization (L2 regularization).
   - Use batch normalization.

By combining these techniques, deep neural networks and RNNs can effectively mitigate both vanishing and exploding gradient problems, leading to more stable and efficient training.

45 LSTM

Long Short-Term Memory (LSTM) is a type of Recurrent Neural Network (RNN) architecture designed to overcome the limitations of traditional RNNs, particularly the **vanishing gradient problem**. LSTMs are effective at learning long-term dependencies by using a memory cell and a set of gates that control the flow of information.

46 LSTM Architecture

An LSTM consists of four main components at each time step:

1. **Cell state** C_t: This is the "memory" of the LSTM, which runs through the entire sequence with only minor linear interactions. The cell state is the key component that carries information forward over long time periods.
2. **Hidden state** h_t: The hidden state is the output of the LSTM at each time step, passed to the next time step and also used for predictions.
3. **Gates**: The gates regulate the flow of information into and out of the cell state. There are three gates:
   - **Forget gate** f_t
   - **Input gate** i_t
   - **Output gate** o_t

46.1 Mathematical Notation

Let:

- x_t be the input at time step t,
- h_{t-1} be the hidden state from the previous time step,
- C_{t-1} be the previous cell state.

The LSTM equations use the following weight matrices and biases:

- W_f, W_i, W_o, W_c: Weight matrices for the forget, input, output gates, and the candidate memory respectively.
- b_f, b_i, b_o, b_c: Biases for the respective gates and memory.

Now, let's look at the mathematics behind each step of the LSTM.

47 Step-by-Step Explanation

47.1 1. Forget Gate f_t

The **forget gate** determines what portion of the previous cell state C_{t-1} should be "forgotten." It uses a sigmoid function to output a value between 0 (completely forget) and 1 (completely keep).

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

Where: - σ is the sigmoid activation function. - $[h_{t-1}, x_t]$ is the concatenation of the previous hidden state and the current input.

The forget gate output f_t is a vector of values between 0 and 1, which will be used to scale the previous cell state.

47.2 2. Input Gate i_t and Candidate Cell State $\tilde{C}_t$

The **input gate** controls how much of the new candidate memory $\tilde{C}_t$ should be added to the current cell state.

First, the input gate is calculated using a sigmoid function:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

Then, the candidate cell state $\tilde{C}_t$, which represents the new information that could be added to the cell state, is computed using a tanh activation:

$$\tilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$

The input gate i_t determines how much of this candidate state $\tilde{C}_t$ is added to the current memory.

47.3 3. Update Cell State C_t

Now, we update the **cell state** C_t using both the forget gate and the input gate. The forget gate determines how much of the previous memory C_{t-1} to retain, while the input gate determines how much of the new candidate memory $\tilde{C}_t$ to add.

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

Where: - $\odot$ represents element-wise multiplication.

The new cell state C_t carries the updated memory of the sequence.

47.4 4. Output Gate o_t and Hidden State h_t

Finally, the **output gate** controls how much of the cell state is used to compute the next hidden state, which is also the output of the LSTM for this time step. The output gate is computed using a sigmoid function:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

The hidden state h_t is then computed by applying the output gate o_t to the tanh-transformed cell state C_t:

$$h_t = o_t \odot \tanh(C_t)$$

This hidden state h_t is passed to the next time step and can also be used for making predictions at this time step.

48 Summary of LSTM Equations

At each time step t, the LSTM performs the following operations:

1. Forget gate:
$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$
2. Input gate:
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
3. Candidate memory:
$$\tilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$
4. Cell state update:
$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$
5. Output gate:
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$
6. Hidden state (output):
$$h_t = o_t \odot \tanh(C_t)$$

49 Advantages of LSTM

1. **Long-term dependency retention**: LSTMs can remember information over long time periods thanks to their cell state and gating mechanisms.
2. **Effective gradient flow**: LSTMs help alleviate the vanishing gradient problem by allowing gradients to flow through the cell state more easily during backpropagation.

50 Visual Representation of LSTM Gates

To summarize visually:

- **Forget gate**: Controls what to "forget" from the previous cell state.
- **Input gate**: Controls what new information to add to the cell state.
- **Output gate**: Controls what to output from the updated cell state.

The LSTM structure enables efficient and effective learning of long-term dependencies in sequence data, making it useful in various tasks like time series forecasting, speech recognition, and natural language processing.

51 Encoder and Decoder

An **encoder-decoder** architecture is commonly used in tasks that involve converting input sequences to output sequences, such as **machine translation**, **summarization**, and **image captioning**. This architecture is typically implemented using RNNs, LSTMs, GRUs, or, more recently, **transformers**.

The architecture consists of two parts:

- **Encoder**: Processes the input sequence and compresses it into a fixed-size vector (context or latent representation).
- **Decoder**: Takes this context vector and generates the output sequence step by step.

52 Encoder-Decoder Architecture

52.1 Encoder

The **encoder** is responsible for encoding the input sequence into a context vector that captures the information needed to generate the output. Mathematically, for an input sequence $X = (x_1, x_2, \ldots, x_T)$, where x_t is the input at time step t, the encoder computes a hidden state representation for each time step t.

1. For each time step t, the hidden state h_t is computed using an RNN, LSTM, or GRU:
$$h_t = f_{\text{enc}}(x_t, h_{t-1})$$
Where: - f_{enc} is the recurrent function (RNN, LSTM, or GRU). - h_t is the hidden state at time t. - h_{t-1} is the hidden state from the previous time step.

2. At the final time step T, the last hidden state h_T (or a combination of all hidden states) is often taken as the **context vector** c, which summarizes the entire input sequence:
$$c = h_T$$
The context vector c is passed to the decoder, containing the encoded information from the input sequence.

52.2 Decoder

The **decoder** generates the output sequence $Y = (y_1, y_2, \ldots, y_{T'})$ using the context vector c produced by the encoder. The decoder generates one output at a time in an auto-regressive manner, meaning it produces one output and feeds it back as an input to generate the next output.

For each time step t in the decoder, the hidden state s_t is computed as:

$$s_t = f_{\text{dec}}(y_{t-1}, s_{t-1}, c)$$

Where:

- f_{dec} is the recurrent function in the decoder (RNN, LSTM, or GRU).
- s_t is the hidden state at time step t in the decoder.
- y_{t-1} is the output from the previous time step (or an initial start token).
- s_{t-1} is the hidden state from the previous time step in the decoder.
- c is the context vector from the encoder.

The final output y_t at each time step is generated by applying a softmax layer to the decoder's hidden state s_t, producing a probability distribution over the possible outputs (in the case of translation, a vocabulary of words):

$$y_t = \text{softmax}(W \cdot s_t + b)$$

Where: - W and b are learnable parameters (weight matrix and bias). - y_t is the predicted output at time t.

53 Attention Mechanism (Extension to Basic Encoder-Decoder)

While the basic encoder-decoder model uses a single context vector c to summarize the input sequence, it can suffer from limitations when handling long sequences. The **attention mechanism** was introduced to address this issue by allowing the decoder to focus on different parts of the input sequence during each step of the output generation.

53.1 Attention Computation

In the attention-based encoder-decoder model, the decoder does not use a single fixed context vector c. Instead, it computes a dynamic context vector at each time step based on the hidden states of the encoder.

1. **Score calculation**: For each decoder time step t, we calculate a score $e_{t,j}$ between the decoder hidden state s_t and each encoder hidden state h_j:

   $$e_{t,j} = \text{score}(s_t, h_j)$$

   The score function can be dot product, multiplicative, or additive.

2. **Alignment (Attention) weights**: The scores are converted to probabilities using a softmax function:

   $$\alpha_{t,j} = \frac{\exp(e_{t,j})}{\sum_{k=1}^{T} \exp(e_{t,k})}$$

   The weights $\alpha_{t,j}$ determine how much attention the decoder should pay to each encoder hidden state h_j when producing the next output.

3. **Context vector**: The context vector c_t for the decoder at time step t is computed as a weighted sum of the encoder hidden states:

$$c_t = \sum_{j=1}^{T} \alpha_{t,j} h_j$$

This context vector c_t dynamically summarizes the most relevant parts of the input sequence for the current decoding step.

4. **Final decoder output**: The decoder hidden state is updated using both the previous hidden state and the dynamic context vector. The output at time t is computed as before:

$$y_t = \text{softmax}(W \cdot [s_t, c_t] + b)$$

54 Transformer Architecture (Without Recurrent Units)

The **Transformer** model is another form of encoder-decoder architecture but does away with the recurrence seen in RNNs and LSTMs. Instead, it relies entirely on **self-attention mechanisms** to process both the input and output sequences.

54.1 Self-Attention (Key, Query, and Value)

In transformers, self-attention is computed at each layer of both the encoder and decoder.

1. **Key, Query, Value**: For each word in the sequence, three vectors are computed: the **query** Q, **key** K, and **value** V vectors. These are derived by multiplying the input vector x_i by learnable weight matrices:

$$Q = W_Q \cdot x_i, \quad K = W_K \cdot x_i, \quad V = W_V \cdot x_i$$

2. **Attention score**: The attention score between a query and a key is computed using the dot product. The result is normalized using the softmax function:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{Q \cdot K^T}{\sqrt{d_k}}\right) \cdot V$$

Where d_k is the dimensionality of the key vectors.

3. **Multi-head attention**: The transformer employs **multi-head attention**, where the attention mechanism is run multiple times in parallel (with different sets of weight matrices) and the results are concatenated.

The transformer architecture significantly improves the speed and scalability of sequence-to-sequence tasks by removing the sequential dependencies inherent in RNNs, and it excels in long-range dependency tasks like translation and text generation.

55 Summary of Encoder-Decoder Architectures:

- **Basic encoder-decoder**: Uses RNNs or LSTMs to encode an input sequence into a fixed-size context vector, which is used to generate the output sequence.
- **Attention mechanism**: Allows the decoder to focus on different parts of the input sequence at each time step by dynamically computing a context vector.
- **Transformers**: A self-attention-based architecture that replaces recurrence with attention mechanisms, making it more efficient for parallel processing and handling long-range dependencies.