W6. Neural Network

Guang Cheng

University of California, Los Angeles

guangcheng@ucla.edu

Week 6

Limitation of Linear Models

- In earlier settings such as regression, we used a weighted linear combination of feature values x_j and weights β_j to model relationships.

Limitation of Linear Models

- In earlier settings such as regression, we used a weighted linear combination of feature values x_j and weights β_j to model relationships.
- A standard regression model example:

$$\hat{y} = \beta_0 + \sum_{j=1}^{d} \beta_j x_j$$

Limitation of Linear Models

- In earlier settings such as regression, we used a weighted linear combination of feature values x_j and weights β_j to model relationships.
- A standard regression model example:

$$\hat{y} = \beta_0 + \sum_{j=1}^{d} \beta_j x_j$$

- However, linear models struggle with more complex value relationships, for example:

Limitation of Linear Models

- In earlier settings such as regression, we used a weighted linear combination of feature values x_j and weights β_j to model relationships.
- A standard regression model example:

$$\hat{y} = \beta_0 + \sum_{j=1}^{d} \beta_j x_j$$

- However, linear models struggle with more complex value relationships, for example:
  - Any value in the range $[0; 5]$ is equally good.

Limitation of Linear Models

- In earlier settings such as regression, we used a weighted linear combination of feature values x_j and weights β_j to model relationships.
- A standard regression model example:

$$\hat{y} = \beta_0 + \sum_{j=1}^{d} \beta_j x_j$$

- However, linear models struggle with more complex value relationships, for example:
  - Any value in the range $[0; 5]$ is equally good.
  - Values over 8 are considered bad.

Limitation of Linear Models

- In earlier settings such as regression, we used a weighted linear combination of feature values x_j and weights β_j to model relationships.
- A standard regression model example:

$$\hat{y} = \beta_0 + \sum_{j=1}^{d} \beta_j x_j$$

- However, linear models struggle with more complex value relationships, for example:
  - Any value in the range $[0; 5]$ is equally good.
  - Values over 8 are considered bad.
  - Being higher than 10 is not worse than values just over 8.

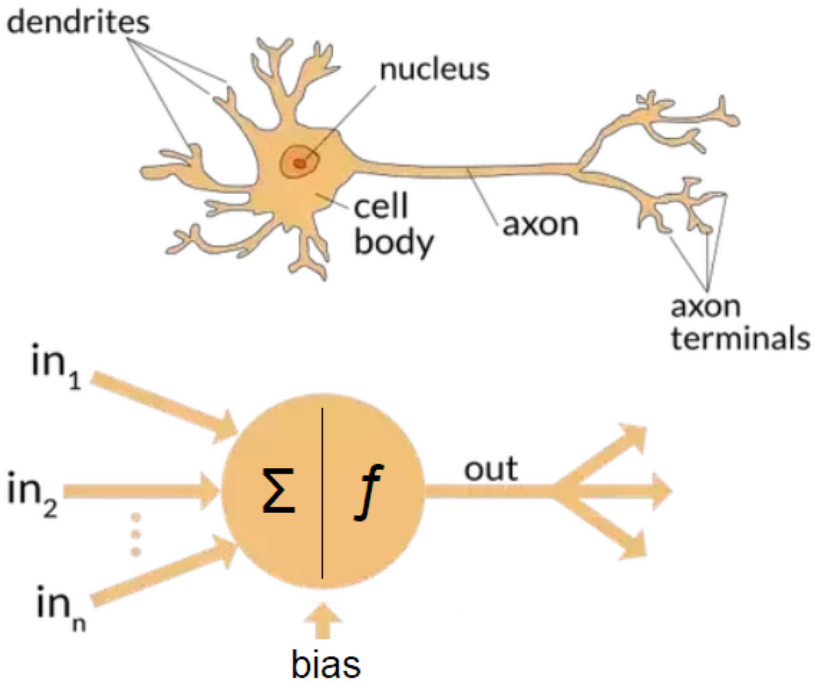
Limitation of Linear Models

- In earlier settings such as regression, we used a weighted linear combination of feature values x_j and weights β_j to model relationships.
- A standard regression model example:

$$\hat{y} = \beta_0 + \sum_{j=1}^{d} \beta_j x_j$$

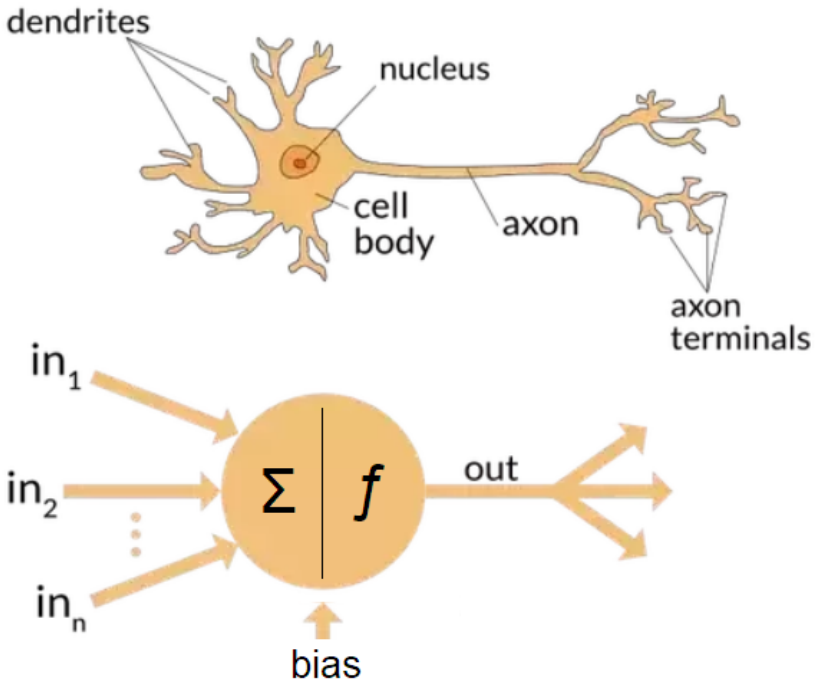
- However, linear models struggle with more complex value relationships, for example:
  - Any value in the range $[0; 5]$ is equally good.
  - Values over 8 are considered bad.
  - Being higher than 10 is not worse than values just over 8.
- These nuances require a modeling approach that goes beyond linear combinations – neural networks.

Neurons



- **Activation:** An artificial neuron (or McCulloch–Pitts "unit") fires when its input signals reach a certain threshold, just like a biological one.

Neurons



- **Activation:** An artificial neuron (or McCulloch–Pitts "unit") fires when its input signals reach a certain threshold, just like a biological one.
- **Processing Non-linear Messages:** Through activation, neurons can process complex non-linear pattern and decision boundries.

Activation Functions

- Activation function controls the activation pattern of neurons.

Activation Functions

- Activation function controls the activation pattern of neurons.
- On top of the linear combinations, we add a non-linear function:

$$\hat{y} = \sigma\left(\beta_0 + \sum_{j=1}^{d} \beta_j x_j\right)$$

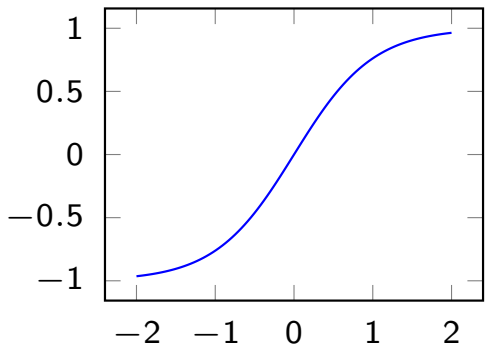
Activation Functions

- Activation function controls the activation pattern of neurons.
- On top of the linear combinations, we add a non-linear function:

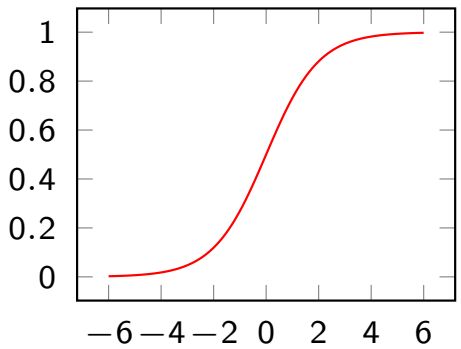
$$\hat{y} = \sigma\left(\beta_0 + \sum_{j=1}^{d} \beta_j x_j\right)$$

- Popular activation functions $\sigma(\cdot)$:

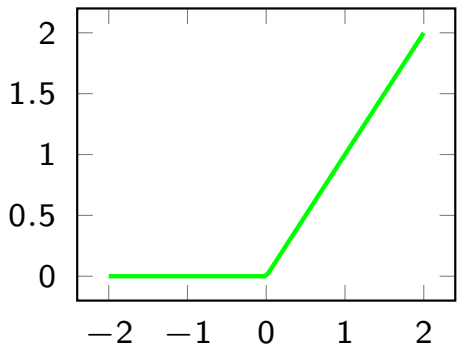
$\tanh(x)$



$\sigma(x) = \frac{1}{1+e^{-x}}$



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



Why Activation Functions?

- **Introduction of Non-linearity:** Activation functions introduce non-linearity, enabling neural networks to learn complex patterns beyond what is possible with linear models.

Why Activation Functions?

- **Introduction of Non-linearity:** Activation functions introduce non-linearity, enabling neural networks to learn complex patterns beyond what is possible with linear models.
- **Enabling Deep Learning:** Each layer in a neural network acts as a distinct processing step, and having multiple such steps allows for the modeling of complex functions.

Why Activation Functions?

- **Introduction of Non-linearity:** Activation functions introduce non-linearity, enabling neural networks to learn complex patterns beyond what is possible with linear models.
- **Enabling Deep Learning:** Each layer in a neural network acts as a distinct processing step, and having multiple such steps allows for the modeling of complex functions.
- **Complex Function Implementation:** Through the use of deep architectures, neural networks can implement intricate functions, capturing a wide variety of relationships in data.

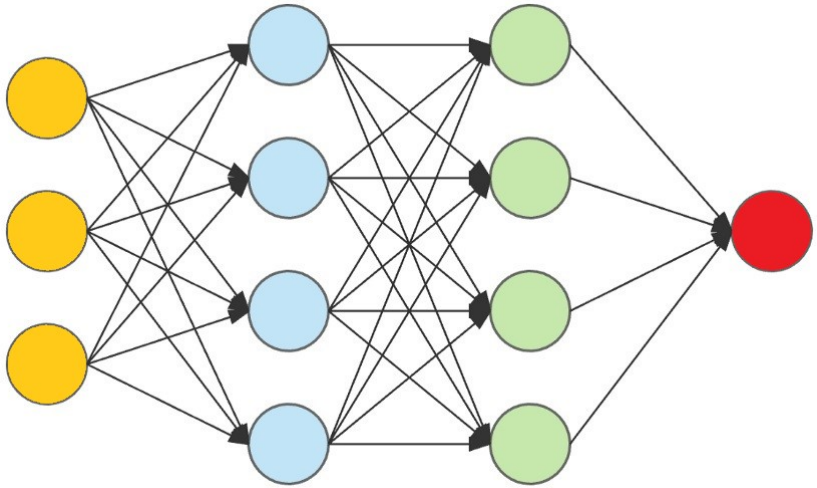
Why Activation Functions?

- **Introduction of Non-linearity:** Activation functions introduce non-linearity, enabling neural networks to learn complex patterns beyond what is possible with linear models.
- **Enabling Deep Learning:** Each layer in a neural network acts as a distinct processing step, and having multiple such steps allows for the modeling of complex functions.
- **Complex Function Implementation:** Through the use of deep architectures, neural networks can implement intricate functions, capturing a wide variety of relationships in data.
- **Constraint Range of Value:** Through activation, output of layers can be constrained to some desired ranges. For example, sigmoid activation limits output to $[0, 1]$, which can represents a probability.

Why Activation Functions?

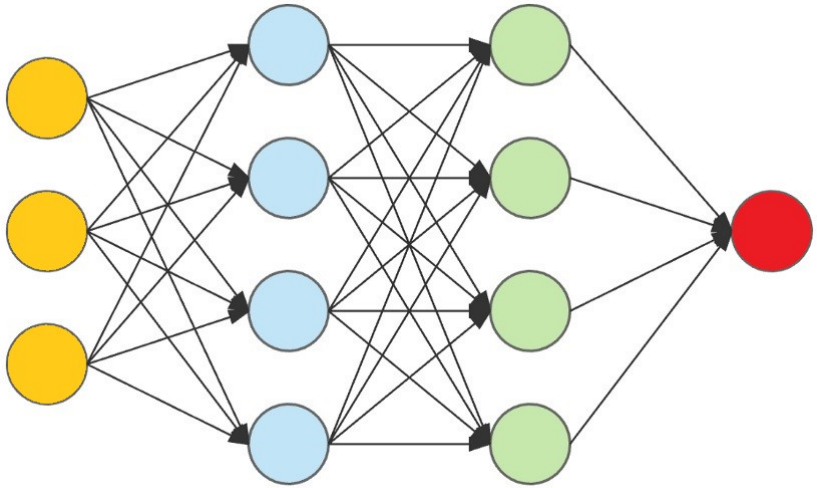
- **Introduction of Non-linearity:** Activation functions introduce non-linearity, enabling neural networks to learn complex patterns beyond what is possible with linear models.
- **Enabling Deep Learning:** Each layer in a neural network acts as a distinct processing step, and having multiple such steps allows for the modeling of complex functions.
- **Complex Function Implementation:** Through the use of deep architectures, neural networks can implement intricate functions, capturing a wide variety of relationships in data.
- **Constraint Range of Value:** Through activation, output of layers can be constrained to some desired ranges. For example, sigmoid activation limits output to $[0, 1]$, which can represents a probability.
- **No Curse of Dimensionality.**

From Single Neurons to a Network



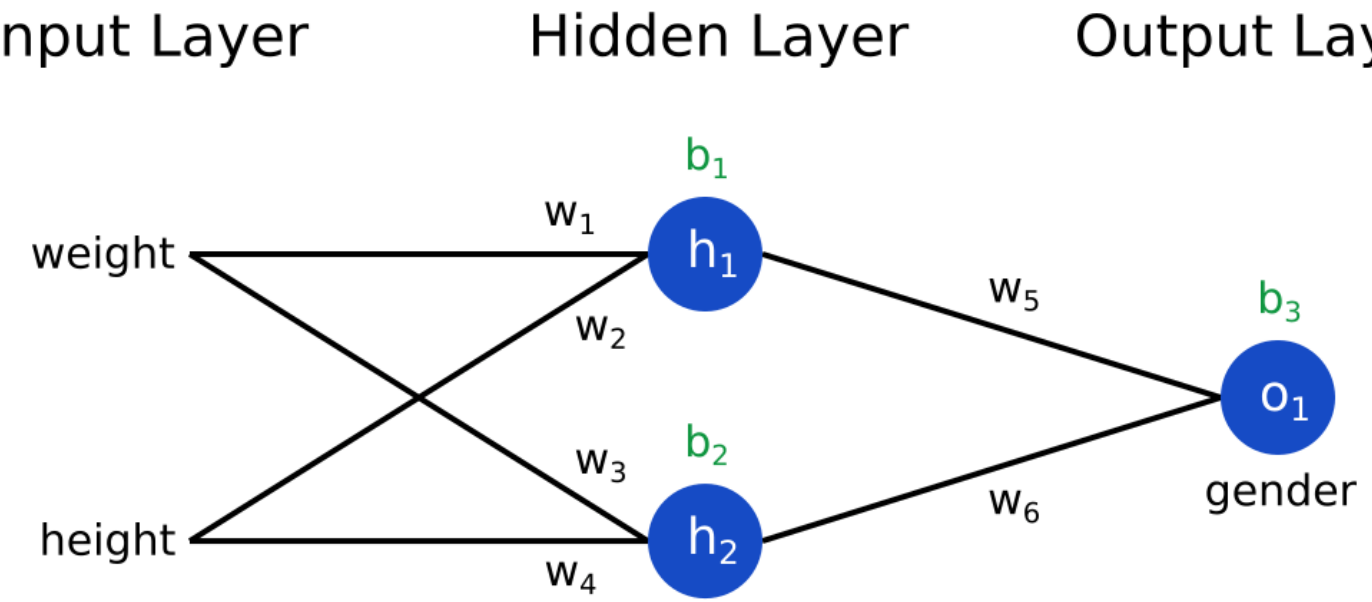
- **Building Blocks:** Each neuron in a neural network acts as a fundamental unit that receives inputs, applies weights to them, and generates an output using an activation function.

From Single Neurons to a Network



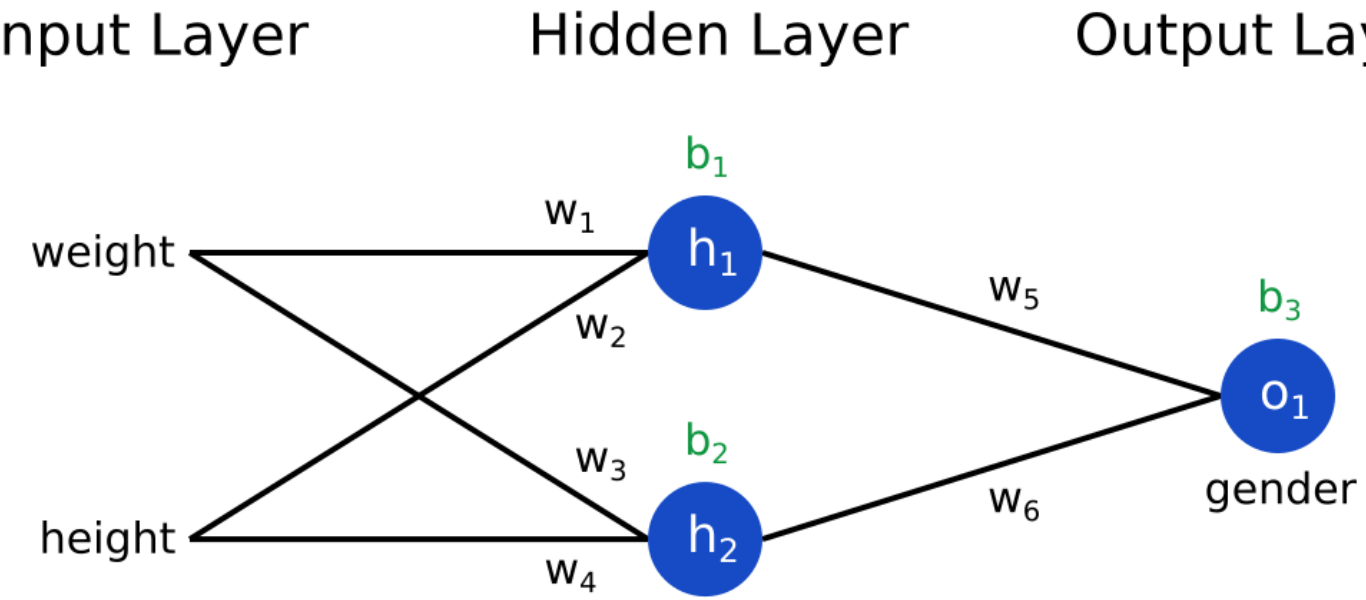
- **Building Blocks:** Each neuron in a neural network acts as a fundamental unit that receives inputs, applies weights to them, and generates an output using an activation function.
- **Layered Structure:** Neurons are organized into layers, where the output of one layer are weighted, and serves as the input to the next. This enables the network to model complex relationships through intricate patterns of weights and activations.

Connecting Neurons Into Network



Mathematical Definition: In the example network above, output, y, is calculated as $y = \sigma(w_5 \cdot h_1 + w_6 \cdot h_2 + b_3)$ where $h_1 = \sigma(w_1 \cdot x_1 + w_2 \cdot x_2 + b_1)$ and $h_2 = \sigma(w_3 \cdot x_1 + w_4 \cdot x_2 + b_2)$. Here, x_1 and x_2 are the input features (weight and height), σ denotes an activation function, and b_3 is the bias for the output neuron.

From Single Neurons to a Network



Example Calculation: Given inputs $x_1 = 70$ (weight) and $x_2 = 1.75$ (height), with weights $w_1 = 0.2, w_2 = 0.4, w_3 = -0.5$, $w_4 = 0.3, w_5 = 1, w_6 = -1.5$, and biases $b_1 = 0.1, b_2 = -0.1, b_3 = 0.5$. The hidden layer activations: $h_1 = \sigma(0.2 \cdot 70 + 0.4 \cdot 1.75 + 0.1)$ and $h_2 = \sigma(-0.5 \cdot 70 + 0.3 \cdot 1.75 - 0.1)$. Then $y = \sigma(1 \cdot h_1 - 1.5 \cdot h_2 + 0.5)$.

From Single Neurons to a Network

In-class exercise: Given inputs $x_1 = 20$ (weight) and $x_2 = 3$ (height), with weights $w_1 = -0.2, w_2 = 1, w_3 = -0.5$, $w_4 = 0.3, w_5 = -1, w_6 = 0$, and biases $b_1 = 1, b_2 = -0.2, b_3 = 0.5$. Let σ be the ReLu function. Please calculate hidden nodes h_1, h_2 and final outcome y.

From Single Neurons to a Network

In-class exercise: Given inputs $x_1 = 20$ (weight) and $x_2 = 3$ (height), with weights $w_1 = -0.2, w_2 = 1, w_3 = -0.5,\ w_4 = 0.3, w_5 = -1, w_6 = 0$, and biases $b_1 = 1, b_2 = -0.2, b_3 = 0.5$. Let σ be the ReLu function. Please calculate hidden nodes h_1, h_2 and final outcome y.

$h_1 = ReLU(20 * (-0.2) + 3 * 1 + 1) = 0$
$h_2 = ReLU(-0.5 * 20 + 0.3 * 3 - 0.2) = 0$
$y = ReLU(0 + 0 + 0.5) = 0.5.$

Network Definition in Matrix Form

The network computation can be expressed in vector/matrix form as:

$$y = \sigma(\mathbf{W}_2\mathbf{h} + b_3)$$

where

$$\mathbf{h} = \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b})$$

and

$$\mathbf{W}_1 = \begin{bmatrix} w_1 & w_2 \\ w_3 & w_4 \end{bmatrix}, \quad \mathbf{W}_2 = \begin{bmatrix} w_5 \\ w_6 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

Network Definition in Matrix Form

The network computation can be expressed in vector/matrix form as:

$$y = \sigma(\mathbf{W}_2\mathbf{h} + b_3)$$

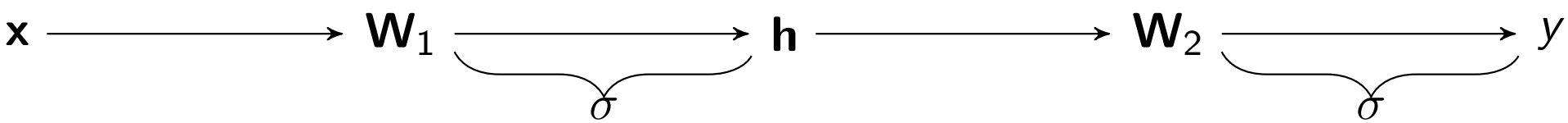
where

$$\mathbf{h} = \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b})$$

and

$$\mathbf{W}_1 = \begin{bmatrix} w_1 & w_2 \\ w_3 & w_4 \end{bmatrix}, \quad \mathbf{W}_2 = \begin{bmatrix} w_5 \\ w_6 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

Here, $\mathbf{x}$ represents the input features (weight and height), σ is the activation function, $\mathbf{h}$ is called the hidden features and $\mathbf{b}$ and b_3 are the biases.



Loss Function

As long as we have data $\{x_i, y_i\}_{i=1}^n$, we can learn the network weights w_j's by minimizing a loss function. Loss functions measures the difference between the network's predictions $\hat{y}$ and the actual target values y.

Loss Function

As long as we have data $\{x_i, y_i\}_{i=1}^n$, we can learn the network weights w_j's by minimizing a loss function. Loss functions measures the difference between the network's predictions $\hat{y}$ and the actual target values y.

Popular loss functions:

- **Mean Squared Error (MSE):**

Loss Function

As long as we have data $\{x_i, y_i\}_{i=1}^n$, we can learn the network weights w_j's by minimizing a loss function. Loss functions measures the difference between the network's predictions $\hat{y}$ and the actual target values y.

Popular loss functions:

- **Mean Squared Error (MSE):**
  - $\text{MSE} = \frac{1}{n}\sum_{i=1}^n (y_i - \hat{y}_i)^2$

Loss Function

As long as we have data $\{x_i, y_i\}_{i=1}^n$, we can learn the network weights w_j's by minimizing a loss function. Loss functions measures the difference between the network's predictions $\hat{y}$ and the actual target values y.

Popular loss functions:

- **Mean Squared Error (MSE):**
  - $\text{MSE} = \frac{1}{n}\sum_{i=1}^n (y_i - \hat{y}_i)^2$
  - Typically used for regression tasks, where the goal is to predict continuous values closest to the target value.

Loss Function

As long as we have data $\{x_i, y_i\}_{i=1}^n$, we can learn the network weights w_j's by minimizing a loss function. Loss functions measures the difference between the network's predictions $\hat{y}$ and the actual target values y.

Popular loss functions:

- **Mean Squared Error (MSE):**
  - $\text{MSE} = \frac{1}{n}\sum_{i=1}^n (y_i - \hat{y}_i)^2$
  - Typically used for regression tasks, where the goal is to predict continuous values closest to the target value.
- **Cross-Entropy:**

Loss Function

As long as we have data $\{x_i, y_i\}_{i=1}^n$, we can learn the network weights w_j's by minimizing a loss function. Loss functions measures the difference between the network's predictions $\hat{y}$ and the actual target values y.

Popular loss functions:

- **Mean Squared Error (MSE):**
  - MSE $= \frac{1}{n}\sum_{i=1}^n (y_i - \hat{y}_i)^2$
  - Typically used for regression tasks, where the goal is to predict continuous values closest to the target value.
- **Cross-Entropy:**
  - Binary Cross-Entropy $= -\frac{1}{n}\sum_{i=1}^n [y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$

Loss Function

As long as we have data $\{x_i, y_i\}_{i=1}^n$, we can learn the network weights w_j's by minimizing a loss function. Loss functions measures the difference between the network's predictions $\hat{y}$ and the actual target values y.

Popular loss functions:

- **Mean Squared Error (MSE):**
  - MSE $= \frac{1}{n}\sum_{i=1}^n (y_i - \hat{y}_i)^2$
  - Typically used for regression tasks, where the goal is to predict continuous values closest to the target value.
- **Cross-Entropy:**
  - Binary Cross-Entropy $= -\frac{1}{n}\sum_{i=1}^n [y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$
  - Used for classification tasks, especially binary and multi-class classification, to measure the difference between two probability distributions: output and target.

Loss Function

As long as we have data $\{x_i, y_i\}_{i=1}^n$, we can learn the network weights w_j's by minimizing a loss function. Loss functions measures the difference between the network's predictions $\hat{y}$ and the actual target values y.

Popular loss functions:

- **Mean Squared Error (MSE):**
  - MSE $= \frac{1}{n}\sum_{i=1}^n (y_i - \hat{y}_i)^2$
  - Typically used for regression tasks, where the goal is to predict continuous values closest to the target value.
- **Cross-Entropy:**
  - Binary Cross-Entropy $= -\frac{1}{n}\sum_{i=1}^n [y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$
  - Used for classification tasks, especially binary and multi-class classification, to measure the difference between two probability distributions: output and target.

Loss Function

As long as we have data $\{x_i, y_i\}_{i=1}^n$, we can learn the network weights w_j's by minimizing a loss function. Loss functions measures the difference between the network's predictions $\hat{y}$ and the actual target values y.

Popular loss functions:

- **Mean Squared Error (MSE):**
  - MSE $= \frac{1}{n}\sum_{i=1}^n (y_i - \hat{y}_i)^2$
  - Typically used for regression tasks, where the goal is to predict continuous values closest to the target value.
- **Cross-Entropy:**
  - Binary Cross-Entropy $= -\frac{1}{n}\sum_{i=1}^n [y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$
  - Used for classification tasks, especially binary and multi-class classification, to measure the difference between two probability distributions: output and target.

Brief intro to "double descent phenomenon"

How to optimize the loss function ?

- How to optimize the loss function in simple function?

How to optimize the loss function ?

- How to optimize the loss function in simple function?
- Use gradient descent!

How to optimize the loss function ?

- How to optimize the loss function in simple function?
- Use gradient descent!
- How to optimize the loss function in neural network (composite function) ?

How to optimize the loss function ?

- How to optimize the loss function in simple function?
- Use gradient descent!
- How to optimize the loss function in neural network (composite function) ?
- Use gradient descent with back propagation !

Why back propagation?

- **Efficient Learning**: Back-propagation efficiently computes the gradient of the loss function with respect to each weight in the network by using the chain rule, making it possible to update all weights in a direction that minimizes the loss.

Why back propagation?

- **Efficient Learning**: Back-propagation efficiently computes the gradient of the loss function with respect to each weight in the network by using the chain rule, making it possible to update all weights in a direction that minimizes the loss.
- Review what is chain rule in calculus

Why back propagation?

- **Efficient Learning**: Back-propagation efficiently computes the gradient of the loss function with respect to each weight in the network by using the chain rule, making it possible to update all weights in a direction that minimizes the loss.
- Review what is chain rule in calculus
- **Deep Learning Capability**: It enables neural networks to learn from complex data and perform tasks like image recognition, language translation (large language model), and playing games (reinforcement learning) at a high level.

Why back propagation?

- **Efficient Learning**: Back-propagation efficiently computes the gradient of the loss function with respect to each weight in the network by using the chain rule, making it possible to update all weights in a direction that minimizes the loss.
- Review what is chain rule in calculus
- **Deep Learning Capability**: It enables neural networks to learn from complex data and perform tasks like image recognition, language translation (large language model), and playing games (reinforcement learning) at a high level.
- **Scalability**: This method scales well with large neural networks and datasets, which is crucial for modern deep learning applications that involve vast amounts of data and complex models.

Simple Example of Computing Gradient

- Input: $x = 2$

Simple Example of Computing Gradient

- Input: $x = 2$
- Actual output: $y = 1$

Simple Example of Computing Gradient

- Input: $x = 2$
- Actual output: $y = 1$
- Weight: w, Bias $b = 0$ (for simplicity, let's consider an initial value $w = 0.5$)

Simple Example of Computing Gradient

- Input: $x = 2$
- Actual output: $y = 1$
- Weight: w, Bias $b = 0$ (for simplicity, let's consider an initial value $w = 0.5$)
- Sigmoid activation function: $\sigma(z) = \frac{1}{1+(\exp -z)}$

Simple Example of Computing Gradient

- Input: $x = 2$
- Actual output: $y = 1$
- Weight: w, Bias $b = 0$ (for simplicity, let's consider an initial value $w = 0.5$)
- Sigmoid activation function: $\sigma(z) = \frac{1}{1+(\exp -z)}$
- Prediction: $\widehat{y} = \sigma(w \cdot x + b)$

Simple Example of Computing Gradient

- Input: $x = 2$
- Actual output: $y = 1$
- Weight: w, Bias $b = 0$ (for simplicity, let's consider an initial value $w = 0.5$)
- Sigmoid activation function: $\sigma(z) = \frac{1}{1+(\exp -z)}$
- Prediction: $\widehat{y} = \sigma(w \cdot x + b)$
- Loss Function (Mean Squared Error): $L = \frac{1}{2}(y - \widehat{y})^2$

Simple Example of Computing Gradient

- Forward Pass:

Simple Example of Computing Gradient

- Forward Pass:
  - Compute the weighted input: $z = w \cdot x + b = 0.5 \cdot 2 + 0 = 1$

Simple Example of Computing Gradient

- Forward Pass:
  - Compute the weighted input: $z = w \cdot x + b = 0.5 \cdot 2 + 0 = 1$
  - Compute the output (prediction) using the sigmoid activation : $\widehat{y} = \sigma(1) = \frac{1}{1+e^{-1}}$

Simple Example of Computing Gradient

- Forward Pass:
  - Compute the weighted input: $z = w \cdot x + b = 0.5 \cdot 2 + 0 = 1$
  - Compute the output (prediction) using the sigmoid activation : $\widehat{y} = \sigma(1) = \frac{1}{1+e^{-1}}$
  - Calculate the loss $L = \frac{1}{2}(1 - \widehat{y})^2$

Simple Example of Computing Gradient

- Backward Pass (Computing the Gradient):

Simple Example of Computing Gradient

- Backward Pass (Computing the Gradient):
  - To update w, we need the gradient of L with respect to w, which involves applying the chain rule: $\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \widehat{y}} \cdot \frac{\partial \widehat{y}}{\partial z} \cdot \frac{\partial z}{\partial w}$

Simple Example of Computing Gradient

- Backward Pass (Computing the Gradient):
  - To update w, we need the gradient of L with respect to w, which involves applying the chain rule: $\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \widehat{y}} \cdot \frac{\partial \widehat{y}}{\partial z} \cdot \frac{\partial z}{\partial w}$
  - $\frac{\partial L}{\partial \widehat{y}} = -(y - \widehat{y})$

Simple Example of Computing Gradient

- Backward Pass (Computing the Gradient):
  - To update w, we need the gradient of L with respect to w, which involves applying the chain rule: $\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \widehat{y}} \cdot \frac{\partial \widehat{y}}{\partial z} \cdot \frac{\partial z}{\partial w}$
  - $\frac{\partial L}{\partial \widehat{y}} = -(y - \widehat{y})$
  - $\frac{\partial \widehat{y}}{\partial z} = \widehat{y} \cdot (1 - \widehat{y})$ (derivative of sigmoid function)

Simple Example of Computing Gradient

- Backward Pass (Computing the Gradient):
  - To update w, we need the gradient of L with respect to w, which involves applying the chain rule: $\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \widehat{y}} \cdot \frac{\partial \widehat{y}}{\partial z} \cdot \frac{\partial z}{\partial w}$
  - $\frac{\partial L}{\partial \widehat{y}} = -(y - \widehat{y})$
  - $\frac{\partial \widehat{y}}{\partial z} = \widehat{y} \cdot (1 - \widehat{y})$ (derivative of sigmoid function)
  - $\frac{\partial z}{\partial w} = x$

Simple Example of Computing Gradient

- Let's compute the actual numbers for this example, including the gradient.

Simple Example of Computing Gradient

- Let's compute the actual numbers for this example, including the gradient.
- For our simplified example, the computations yield the following results:

Simple Example of Computing Gradient

- Let's compute the actual numbers for this example, including the gradient.
- For our simplified example, the computations yield the following results:
  - Prediction ($\widehat{y}$): $\frac{1}{1+e^{-1}} = 0.731$

Simple Example of Computing Gradient

- Let's compute the actual numbers for this example, including the gradient.
- For our simplified example, the computations yield the following results:
  - Prediction ($\widehat{y}$): $\frac{1}{1+e^{-1}} = 0.731$
  - Loss (L): $(1 - 0.731)^2/2 = 0.036$

Simple Example of Computing Gradient

- Let's compute the actual numbers for this example, including the gradient.
- For our simplified example, the computations yield the following results:
  - Prediction ($\widehat{y}$): $\frac{1}{1+e^{-1}} = 0.731$
  - Loss (L): $(1 - 0.731)^2/2 = 0.036$
  - Gradient of Loss with respect to w: $(\frac{\partial L}{\partial w} = -0.106)$

Simple Example of Computing Gradient

- Let's compute the actual numbers for this example, including the gradient.
- For our simplified example, the computations yield the following results:
  - Prediction ($\widehat{y}$): $\frac{1}{1+e^{-1}} = 0.731$
  - Loss (L): $(1 - 0.731)^2/2 = 0.036$
  - Gradient of Loss with respect to w: $(\frac{\partial L}{\partial w} = -0.106)$
- Putting it all together gives us the gradient of the loss with respect to w, which is used to update w in the direction that reduces the loss.

Simple Example of Computing Gradient

- Let's compute the actual numbers for this example, including the gradient.
- For our simplified example, the computations yield the following results:
  - Prediction ($\widehat{y}$): $\frac{1}{1+e^{-1}} = 0.731$
  - Loss (L): $(1 - 0.731)^2/2 = 0.036$
  - Gradient of Loss with respect to w: $(\frac{\partial L}{\partial w} = -0.106)$
- Putting it all together gives us the gradient of the loss with respect to w, which is used to update w in the direction that reduces the loss.
- If we have a learning rate α, the weight update rule would be:

$$w_{\text{new}} = w - \alpha \cdot \frac{\partial L}{\partial w},$$

where w is the weight estimate in the previous update.

Exercise: Forward Propagation in a Simple Neural Network

- Objective: Calculate the output of a simple neural network with one input layer and one output neuron, given the input values, weights, and bias. Use the sigmoid function as the activation function for the output neuron.

Exercise: Forward Propagation in a Simple Neural Network

- Objective: Calculate the output of a simple neural network with one input layer and one output neuron, given the input values, weights, and bias. Use the sigmoid function as the activation function for the output neuron.
- Given:

Exercise: Forward Propagation in a Simple Neural Network

- Objective: Calculate the output of a simple neural network with one input layer and one output neuron, given the input values, weights, and bias. Use the sigmoid function as the activation function for the output neuron.
- Given:
  - Input values: $x_1 = 0.5$, $x_2 = 0.85$; $y = 1.2$

Exercise: Forward Propagation in a Simple Neural Network

- Objective: Calculate the output of a simple neural network with one input layer and one output neuron, given the input values, weights, and bias. Use the sigmoid function as the activation function for the output neuron.
- Given:
  - Input values: $x_1 = 0.5$, $x_2 = 0.85$; $y = 1.2$
  - Initial weights: $w_1 = 0.4$, $w_2 = 0.6$

Exercise: Forward Propagation in a Simple Neural Network

- Objective: Calculate the output of a simple neural network with one input layer and one output neuron, given the input values, weights, and bias. Use the sigmoid function as the activation function for the output neuron.
- Given:
  - Input values: $x_1 = 0.5$, $x_2 = 0.85$; $y = 1.2$
  - Initial weights: $w_1 = 0.4$, $w_2 = 0.6$
  - Bias: $b = 0.2$

Exercise: Forward Propagation in a Simple Neural Network

- Objective: Calculate the output of a simple neural network with one input layer and one output neuron, given the input values, weights, and bias. Use the sigmoid function as the activation function for the output neuron.
- Given:
  - Input values: $x_1 = 0.5$, $x_2 = 0.85$; $y = 1.2$
  - Initial weights: $w_1 = 0.4$, $w_2 = 0.6$
  - Bias: $b = 0.2$
  - Sigmoid function: $\sigma(z) = \frac{1}{1+e^{-z}}$

Exercise: Forward Propagation in a Simple Neural Network

- Objective: Calculate the output of a simple neural network with one input layer and one output neuron, given the input values, weights, and bias. Use the sigmoid function as the activation function for the output neuron.
- Given:
  - Input values: $x_1 = 0.5$, $x_2 = 0.85$; $y = 1.2$
  - Initial weights: $w_1 = 0.4$, $w_2 = 0.6$
  - Bias: $b = 0.2$
  - Sigmoid function: $\sigma(z) = \frac{1}{1+e^{-z}}$
  - Loss function: $(y - \hat{y})^2/2$

Exercise: Forward Propagation in a Simple Neural Network

- Objective: Calculate the output of a simple neural network with one input layer and one output neuron, given the input values, weights, and bias. Use the sigmoid function as the activation function for the output neuron.
- Given:
  - Input values: $x_1 = 0.5$, $x_2 = 0.85$; $y = 1.2$
  - Initial weights: $w_1 = 0.4$, $w_2 = 0.6$
  - Bias: $b = 0.2$
  - Sigmoid function: $\sigma(z) = \frac{1}{1+e^{-z}}$
  - Loss function: $(y - \hat{y})^2/2$
- What is the value of loss function?

Exercise: Forward Propagation in a Simple Neural Network

- Objective: Calculate the output of a simple neural network with one input layer and one output neuron, given the input values, weights, and bias. Use the sigmoid function as the activation function for the output neuron.
- Given:
  - Input values: $x_1 = 0.5$, $x_2 = 0.85$; $y = 1.2$
  - Initial weights: $w_1 = 0.4$, $w_2 = 0.6$
  - Bias: $b = 0.2$
  - Sigmoid function: $\sigma(z) = \frac{1}{1+e^{-z}}$
  - Loss function: $(y - \hat{y})^2/2$
- What is the value of loss function?
- Work out the update w_{next} after setting the learning rate $\alpha = 0.1$.

Exercise: Forward Propagation in a Simple Neural Network

- Solution: First, we calculate the weighted sum (z):
  $z = (0.5 \times 0.4) + (0.85 \times 0.6) + 0.2 = 0.91$

Exercise: Forward Propagation in a Simple Neural Network

- Solution: First, we calculate the weighted sum (z):
  $z = (0.5 \times 0.4) + (0.85 \times 0.6) + 0.2 = 0.91$
- Now, we apply the sigmoid function to z to get the output :
  $\hat{y} = \sigma(z) = \frac{1}{1+e^{-z}} = (1 + e^{-0.91})^{-1} = 0.713$

Exercise: Forward Propagation in a Simple Neural Network

- Solution: First, we calculate the weighted sum (z):
  $z = (0.5 \times 0.4) + (0.85 \times 0.6) + 0.2 = 0.91$
- Now, we apply the sigmoid function to z to get the output :
  $\hat{y} = \sigma(z) = \frac{1}{1+e^{-z}} = (1 + e^{-0.91})^{-1} = 0.713$
- Let's do the final calculation.

$$(1.2 - 0.713)^2/2 = 0.12.$$

Exercise: Forward Propagation in a Simple Neural Network

- Solution: The update rule for weights in a gradient descent optimization is given by: $w_{\text{new}} = w - \alpha \cdot \frac{\partial L}{\partial w}$

Exercise: Forward Propagation in a Simple Neural Network

- Solution: The update rule for weights in a gradient descent optimization is given by: $w_{\text{new}} = w - \alpha \cdot \frac{\partial L}{\partial w}$
- To do this, we first calculate the gradient of the loss function with respect to each weight. The gradient can be found using the chain rule as follows:

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial \widehat{y}} \cdot \frac{\partial \widehat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_i},$$

where

Exercise: Forward Propagation in a Simple Neural Network

- Solution: The update rule for weights in a gradient descent optimization is given by: $w_{\text{new}} = w - \alpha \cdot \frac{\partial L}{\partial w}$
- To do this, we first calculate the gradient of the loss function with respect to each weight. The gradient can be found using the chain rule as follows:

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial \widehat{y}} \cdot \frac{\partial \widehat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_i},$$

where

  - $\frac{\partial L}{\partial \widehat{y}} = \widehat{y} - y$,

Exercise: Forward Propagation in a Simple Neural Network

- Solution: The update rule for weights in a gradient descent optimization is given by: $w_{\text{new}} = w - \alpha \cdot \frac{\partial L}{\partial w}$
- To do this, we first calculate the gradient of the loss function with respect to each weight. The gradient can be found using the chain rule as follows:

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial \widehat{y}} \cdot \frac{\partial \widehat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_i},$$

where

  - $\frac{\partial L}{\partial \widehat{y}} = \widehat{y} - y,$
  - $\frac{\partial \widehat{y}}{\partial z} = \widehat{y} \cdot (1 - \widehat{y})$

Exercise: Forward Propagation in a Simple Neural Network

- Solution: The update rule for weights in a gradient descent optimization is given by: $w_{\text{new}} = w - \alpha \cdot \frac{\partial L}{\partial w}$
- To do this, we first calculate the gradient of the loss function with respect to each weight. The gradient can be found using the chain rule as follows:

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial \widehat{y}} \cdot \frac{\partial \widehat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_i},$$

where

  - $\frac{\partial L}{\partial \widehat{y}} = \widehat{y} - y,$
  - $\frac{\partial \widehat{y}}{\partial z} = \widehat{y} \cdot (1 - \widehat{y})$
  - $\frac{\partial z}{\partial w_i} = x_i$

Exercise: Forward Propagation in a Simple Neural Network

- Solution: The update rule for weights in a gradient descent optimization is given by: $w_{\text{new}} = w - \alpha \cdot \frac{\partial L}{\partial w}$
- To do this, we first calculate the gradient of the loss function with respect to each weight. The gradient can be found using the chain rule as follows:

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial \widehat{y}} \cdot \frac{\partial \widehat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_i},$$

  where
  - $\frac{\partial L}{\partial \widehat{y}} = \widehat{y} - y,$
  - $\frac{\partial \widehat{y}}{\partial z} = \widehat{y} \cdot (1 - \widehat{y})$
  - $\frac{\partial z}{\partial w_i} = x_i$
- After calculating these gradients, we'll update the weights by subtracting the product of the learning rate $\alpha = 0.1$ and the gradient from each weight. The updated weights are:

Exercise: Forward Propagation in a Simple Neural Network

- Solution: The update rule for weights in a gradient descent optimization is given by: $w_{\text{new}} = w - \alpha \cdot \frac{\partial L}{\partial w}$
- To do this, we first calculate the gradient of the loss function with respect to each weight. The gradient can be found using the chain rule as follows:

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial \widehat{y}} \cdot \frac{\partial \widehat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_i},$$

  where
  - $\frac{\partial L}{\partial \widehat{y}} = \widehat{y} - y,$
  - $\frac{\partial \widehat{y}}{\partial z} = \widehat{y} \cdot (1 - \widehat{y})$
  - $\frac{\partial z}{\partial w_i} = x_i$
- After calculating these gradients, we'll update the weights by subtracting the product of the learning rate $\alpha = 0.1$ and the gradient from each weight. The updated weights are:
- $w_1^{\text{next}} \approx 0.405$

Exercise: Forward Propagation in a Simple Neural Network

- Solution: The update rule for weights in a gradient descent optimization is given by: $w_{\text{new}} = w - \alpha \cdot \frac{\partial L}{\partial w}$
- To do this, we first calculate the gradient of the loss function with respect to each weight. The gradient can be found using the chain rule as follows:

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial \widehat{y}} \cdot \frac{\partial \widehat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_i},$$

where

  - $\frac{\partial L}{\partial \widehat{y}} = \widehat{y} - y$,
  - $\frac{\partial \widehat{y}}{\partial z} = \widehat{y} \cdot (1 - \widehat{y})$
  - $\frac{\partial z}{\partial w_i} = x_i$

- After calculating these gradients, we'll update the weights by subtracting the product of the learning rate $\alpha = 0.1$ and the gradient from each weight. The updated weights are:
- $w_1^{\text{next}} \approx 0.405$
- $w_2^{\text{next}} \approx 0.608$

Deep Neural Networks

- Deep Neural Networks (DNNs) refer to neural networks that contain multiple (often hundreds) layers of neurons between the input and output layers, enabling them to learn features at multiple levels of abstraction.

Deep Neural Networks

- Deep Neural Networks (DNNs) refer to neural networks that contain multiple (often hundreds) layers of neurons between the input and output layers, enabling them to learn features at multiple levels of abstraction.
- That is why they are called "deep" neural networks.

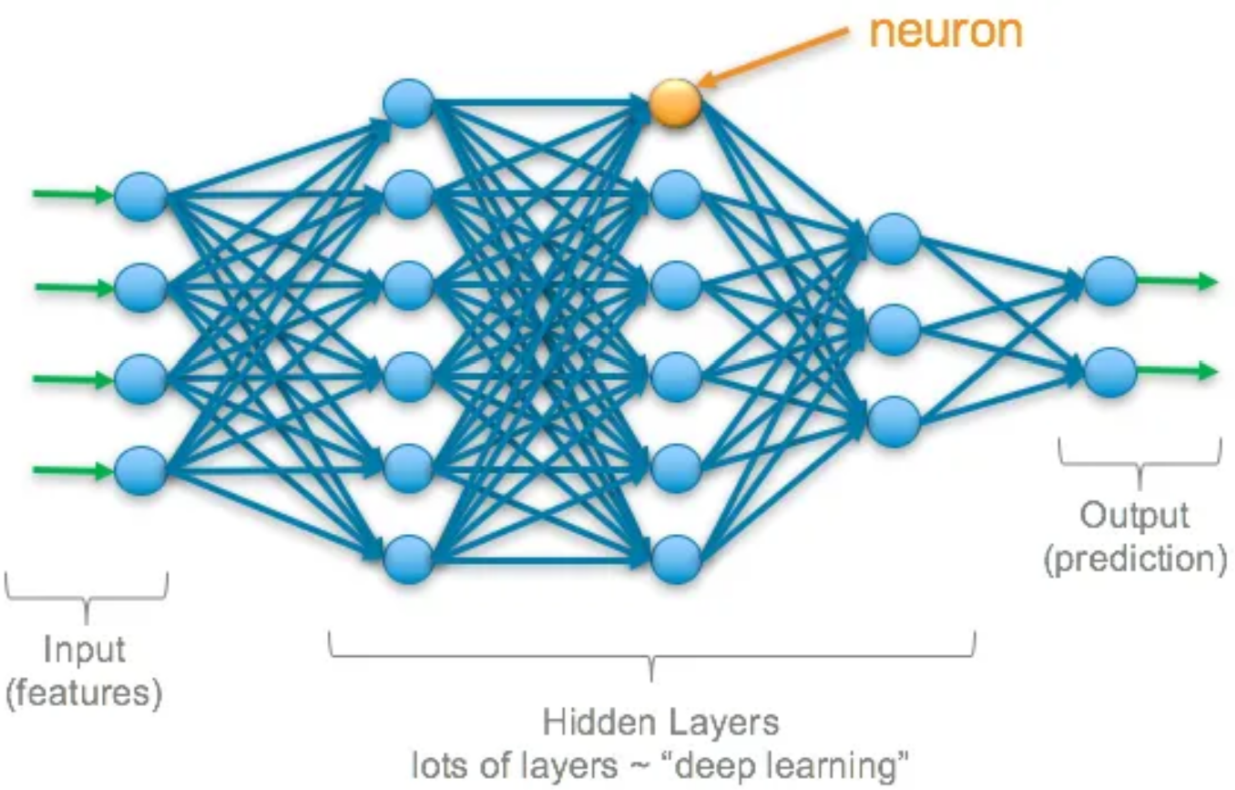
Deep Neural Networks

- Deep Neural Networks (DNNs) refer to neural networks that contain multiple (often hundreds) layers of neurons between the input and output layers, enabling them to learn features at multiple levels of abstraction.
- That is why they are called "deep" neural networks.
- This depth allows deep neural networks to handle very complex tasks, such as image recognition, natural language processing, and playing complex games, with remarkable accuracy.

Deep Neural Networks

- Deep Neural Networks (DNNs) refer to neural networks that contain multiple (often hundreds) layers of neurons between the input and output layers, enabling them to learn features at multiple levels of abstraction.
- That is why they are called "deep" neural networks.
- This depth allows deep neural networks to handle very complex tasks, such as image recognition, natural language processing, and playing complex games, with remarkable accuracy.
- Our previous examples are 1-hidden layer (fully connected) neural networks, i.e., shallow neural networks.

Layers in Deep Neural Networks



Layers in Deep Neural Networks

The architecture of a DNN is composed of several types of layers, each with a specific function:

- **Input Layer**: The first layer of the network that receives the raw input data. Each neuron in the input layer represents a feature of the input data.

Layers in Deep Neural Networks

The architecture of a DNN is composed of several types of layers, each with a specific function:

- **Input Layer**: The first layer of the network that receives the raw input data. Each neuron in the input layer represents a feature of the input data.
- Form of raw data: pixel in image data; tokenization in text data; embedding of tabular data.

Layers in Deep Neural Networks

- **Hidden Layers**: Layers between the input and output layers. There can be many hidden layers in a DNN, and they are where most of the computation takes place. Each hidden layer transforms its input data into a slightly more abstract and composite representation. The network can have different types of hidden layers, including:

Layers in Deep Neural Networks

- **Hidden Layers**: Layers between the input and output layers. There can be many hidden layers in a DNN, and they are where most of the computation takes place. Each hidden layer transforms its input data into a slightly more abstract and composite representation. The network can have different types of hidden layers, including:
  - **Dense (Fully Connected) Layers**: Every neuron in one layer is connected to every neuron in the next layer.

Layers in Deep Neural Networks

- **Hidden Layers**: Layers between the input and output layers. There can be many hidden layers in a DNN, and they are where most of the computation takes place. Each hidden layer transforms its input data into a slightly more abstract and composite representation. The network can have different types of hidden layers, including:
  - **Dense (Fully Connected) Layers**: Every neuron in one layer is connected to every neuron in the next layer.
  - **Convolutional Layers**: Primarily used in processing grid-like data such as images. These layers apply a convolution operation to the input, capturing the spatial and temporal dependencies in an image or time series data.

Layers in Deep Neural Networks

- **Hidden Layers**: Layers between the input and output layers. There can be many hidden layers in a DNN, and they are where most of the computation takes place. Each hidden layer transforms its input data into a slightly more abstract and composite representation. The network can have different types of hidden layers, including:
  - **Dense (Fully Connected) Layers**: Every neuron in one layer is connected to every neuron in the next layer.
  - **Convolutional Layers**: Primarily used in processing grid-like data such as images. These layers apply a convolution operation to the input, capturing the spatial and temporal dependencies in an image or time series data.
  - **Pooling Layers**: Used to reduce the dimensionality of the data, helping to decrease computational load and minimize overfitting. Pooling layers downsample the data by summarizing features in patches of the input data.

Layers in Deep Neural Networks

- **Hidden Layers**: Layers between the input and output layers. There can be many hidden layers in a DNN, and they are where most of the computation takes place. Each hidden layer transforms its input data into a slightly more abstract and composite representation. The network can have different types of hidden layers, including:
  - **Dense (Fully Connected) Layers**: Every neuron in one layer is connected to every neuron in the next layer.
  - **Convolutional Layers**: Primarily used in processing grid-like data such as images. These layers apply a convolution operation to the input, capturing the spatial and temporal dependencies in an image or time series data.
  - **Pooling Layers**: Used to reduce the dimensionality of the data, helping to decrease computational load and minimize overfitting. Pooling layers downsample the data by summarizing features in patches of the input data.
  - **Recurrent Layers**: Used for processing sequential data, where the output from previous steps is fed back into the model to predict the outcome of a sequence. This is particularly useful in language modeling and time series analysis.

Layers in Deep Neural Networks

- **Output Layer**: The final layer that produces the output of the model. The design of the output layer depends on the specific task (e.g., regression, classification). For classification tasks, the output layer often uses a softmax function to generate probabilities for each class.

What is the challenges of traditional neural networks ?

- **Sequence Data Processing**: Traditional neural networks, such as feedforward networks, assume that inputs are independent of each other. However, in many real-world applications like language translation or stock market prediction, the order of data points is crucial.

What is the challenges of traditional neural networks ?

- **Sequence Data Processing**: Traditional neural networks, such as feedforward networks, assume that inputs are independent of each other. However, in many real-world applications like language translation or stock market prediction, the order of data points is crucial.
- **Variable-Length Input and Output**: Traditional neural networks that require fixed-size inputs and outputs. However, the flexibility to deal with variable-length is particularly useful in applications like speech recognition, where the duration of input audio clips can vary widely.

What is the challenges of traditional neural networks ?

- **Sequence Data Processing**: Traditional neural networks, such as feedforward networks, assume that inputs are independent of each other. However, in many real-world applications like language translation or stock market prediction, the order of data points is crucial.
- **Variable-Length Input and Output**: Traditional neural networks that require fixed-size inputs and outputs. However, the flexibility to deal with variable-length is particularly useful in applications like speech recognition, where the duration of input audio clips can vary widely.
- How to improve the traditional neural networks ?

What is the challenges of traditional neural networks ?

- **Sequence Data Processing**: Traditional neural networks, such as feedforward networks, assume that inputs are independent of each other. However, in many real-world applications like language translation or stock market prediction, the order of data points is crucial.
- **Variable-Length Input and Output**: Traditional neural networks that require fixed-size inputs and outputs. However, the flexibility to deal with variable-length is particularly useful in applications like speech recognition, where the duration of input audio clips can vary widely.
- How to improve the traditional neural networks ?
- Use Recurrent Neural Networks!

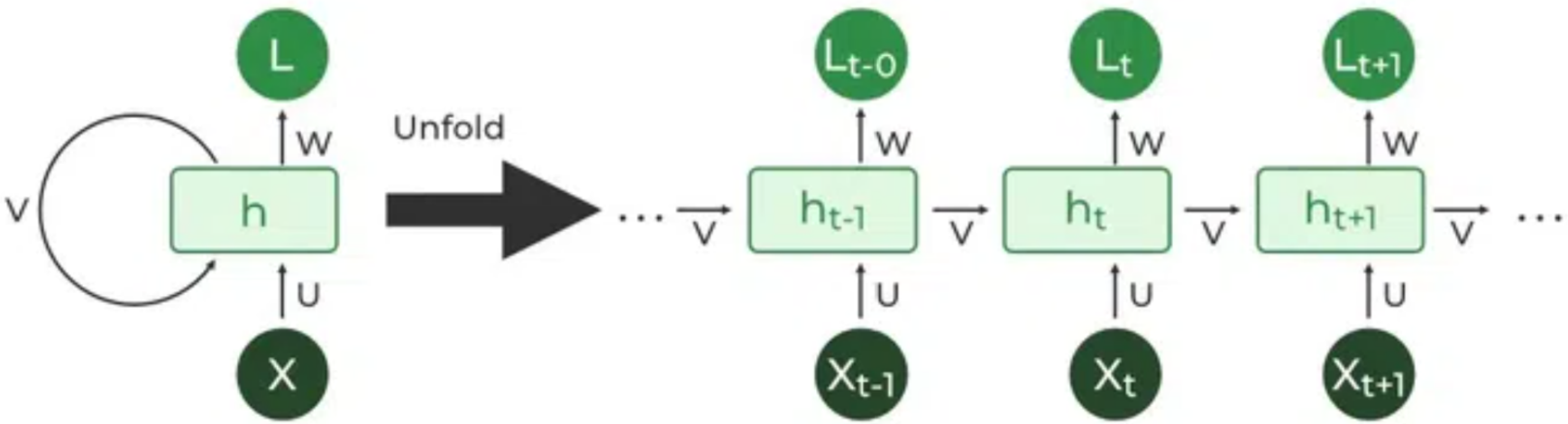
What is Recurrent Neural Network (RNN)?

- The main and most important feature of RNN is its Hidden state, which remembers some information about a sequence. The state is also referred to as Memory State since it remembers the previous input to the network.

What is Recurrent Neural Network (RNN)?

- The main and most important feature of RNN is its Hidden state, which remembers some information about a sequence. The state is also referred to as Memory State since it remembers the previous input to the network.
- It uses the same parameters for each input as it performs the same task on all the inputs or hidden layers to produce the output. This reduces the complexity of parameters, unlike other neural networks.

The schematic diagram of RNN



How does RNN work ?

- The Recurrent Neural Network consists of multiple fixed activation function units, one for each time step. Each unit has an internal state which is called the hidden state of the unit.

How does RNN work ?

- The Recurrent Neural Network consists of multiple fixed activation function units, one for each time step. Each unit has an internal state which is called the hidden state of the unit.
- This hidden state signifies the past knowledge that the network currently holds at a given time step. This hidden state is updated at every time step to signify the change in the knowledge of the network about the past.

How does RNN work ?

- The Recurrent Neural Network consists of multiple fixed activation function units, one for each time step. Each unit has an internal state which is called the hidden state of the unit.
- This hidden state signifies the past knowledge that the network currently holds at a given time step. This hidden state is updated at every time step to signify the change in the knowledge of the network about the past.
- The hidden state is updated using the following recurrence relation:

How does RNN work ?

- The Recurrent Neural Network consists of multiple fixed activation function units, one for each time step. Each unit has an internal state which is called the hidden state of the unit.
- This hidden state signifies the past knowledge that the network currently holds at a given time step. This hidden state is updated at every time step to signify the change in the knowledge of the network about the past.
- The hidden state is updated using the following recurrence relation:
  - $h_t = f(h_{t-1}, x_t)$

How does RNN work ?

- The Recurrent Neural Network consists of multiple fixed activation function units, one for each time step. Each unit has an internal state which is called the hidden state of the unit.
- This hidden state signifies the past knowledge that the network currently holds at a given time step. This hidden state is updated at every time step to signify the change in the knowledge of the network about the past.
- The hidden state is updated using the following recurrence relation:
  - $h_t = f(h_{t-1}, x_t)$
  - where h_t is the current state, h_{t-1} is the previous state, and x_t is the input state.

How does RNN work ?

- An example: activation function (tanh)

How does RNN work ?

- An example: activation function (tanh)
  - $h_t = tanh(W_{hh}h_{t-1} + W_{xh}x_t)$

How does RNN work ?

- An example: activation function (tanh)
  - $h_t = tanh(W_{hh}h_{t-1} + W_{xh}x_t)$
  - where W_{hh} is the weight at recurrent neuron and W_{xh} is the weight at input neuron

How does RNN work ?

- An example: activation function (tanh)
  - $h_t = tanh(W_{hh}h_{t-1} + W_{xh}x_t)$
  - where W_{hh} is the weight at recurrent neuron and W_{xh} is the weight at input neuron
- The formula for calculating output:

How does RNN work ?

- An example: activation function (tanh)
  - $h_t = tanh(W_{hh}h_{t-1} + W_{xh}x_t)$
  - where W_{hh} is the weight at recurrent neuron and W_{xh} is the weight at input neuron
- The formula for calculating output:
  - $y_t = W_{hy}h_t$

How does RNN work ?

- An example: activation function (tanh)
  - $h_t = tanh(W_{hh}h_{t-1} + W_{xh}x_t)$
  - where W_{hh} is the weight at recurrent neuron and W_{xh} is the weight at input neuron
- The formula for calculating output:
  - $y_t = W_{hy}h_t$
  - where y_t is the output and W_{hy} is the weight at output layer

How does RNN work ?

- An example: activation function (tanh)
  - $h_t = tanh(W_{hh}h_{t-1} + W_{xh}x_t)$
  - where W_{hh} is the weight at recurrent neuron and W_{xh} is the weight at input neuron
- The formula for calculating output:
  - $y_t = W_{hy}h_t$
  - where y_t is the output and W_{hy} is the weight at output layer
- These parameters are updated using Back propagation.

Advantages and Disadvantages of RNN

- **Advantages**

Advantages and Disadvantages of RNN

- **Advantages**
  - An RNN remembers each and every piece of information through time. It is useful in time series prediction only because of the feature to remember previous inputs as well. This is called Long Short Term Memory.

Advantages and Disadvantages of RNN

- **Advantages**
  - An RNN remembers each and every piece of information through time. It is useful in time series prediction only because of the feature to remember previous inputs as well. This is called Long Short Term Memory.
  - Recurrent neural networks are even used with convolutional layers to extend the effective pixel neighborhood.

Advantages and Disadvantages of RNN

- **Advantages**
  - An RNN remembers each and every piece of information through time. It is useful in time series prediction only because of the feature to remember previous inputs as well. This is called Long Short Term Memory.
  - Recurrent neural networks are even used with convolutional layers to extend the effective pixel neighborhood.
- **Disadvantages**

Advantages and Disadvantages of RNN

- **Advantages**
  - An RNN remembers each and every piece of information through time. It is useful in time series prediction only because of the feature to remember previous inputs as well. This is called Long Short Term Memory.
  - Recurrent neural networks are even used with convolutional layers to extend the effective pixel neighborhood.
- **Disadvantages**
  - Gradient vanishing and exploding problems.

Advantages and Disadvantages of RNN

- **Advantages**
  - An RNN remembers each and every piece of information through time. It is useful in time series prediction only because of the feature to remember previous inputs as well. This is called Long Short Term Memory.
  - Recurrent neural networks are even used with convolutional layers to extend the effective pixel neighborhood.
- **Disadvantages**
  - Gradient vanishing and exploding problems.
  - Training an RNN is a very difficult task.

Advantages and Disadvantages of RNN

- **Advantages**
  - An RNN remembers each and every piece of information through time. It is useful in time series prediction only because of the feature to remember previous inputs as well. This is called Long Short Term Memory.
  - Recurrent neural networks are even used with convolutional layers to extend the effective pixel neighborhood.
- **Disadvantages**
  - Gradient vanishing and exploding problems.
  - Training an RNN is a very difficult task.
  - It cannot process very long sequences if using tanh or relu as an activation function.

Advantages and Disadvantages of RNN

- How to deal with Long-Term Dependencies, Vanishing and Exploding Gradients in RNN ?

Advantages and Disadvantages of RNN

- How to deal with Long-Term Dependencies, Vanishing and Exploding Gradients in RNN ?
- Use LSTM !

What is LSTM?

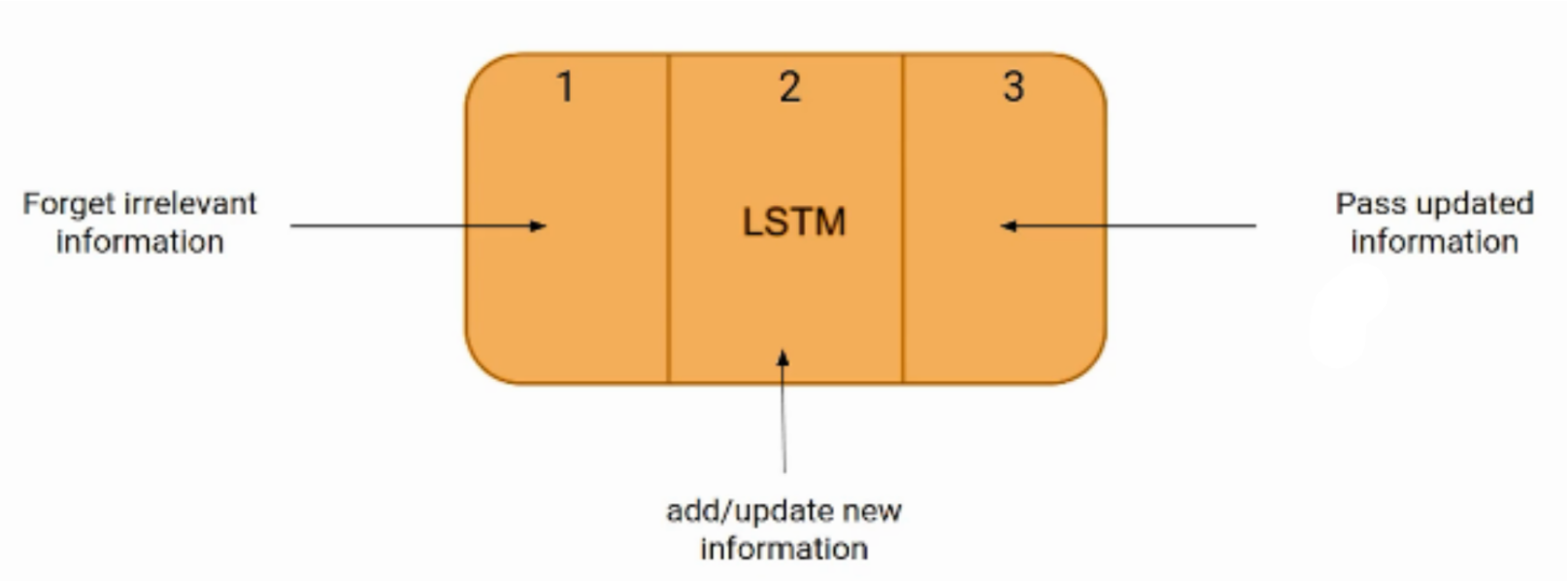
- LSTM (Long Short-Term Memory) is a recurrent neural network (RNN) architecture widely used in Deep Learning. It excels at capturing long-term dependencies, making it ideal for sequence prediction tasks.

What is LSTM?

- LSTM (Long Short-Term Memory) is a recurrent neural network (RNN) architecture widely used in Deep Learning. It excels at capturing long-term dependencies, making it ideal for sequence prediction tasks.
- Unlike traditional neural networks, LSTM incorporates feedback connections, allowing it to process entire sequences of data, not just individual data points. This makes it highly effective in understanding and predicting patterns in sequential data like time series, text, and speech.

LSTM architecture

- At a high level, LSTM works very much like an RNN cell. Here is the internal functioning of the LSTM network. The LSTM network architecture consists of three parts, as shown in the image below, and each part performs an individual function.



The Logic Behind LSTM

- The first part chooses whether the information coming from the previous timestamp is to be remembered or is irrelevant and can be forgotten.

The Logic Behind LSTM

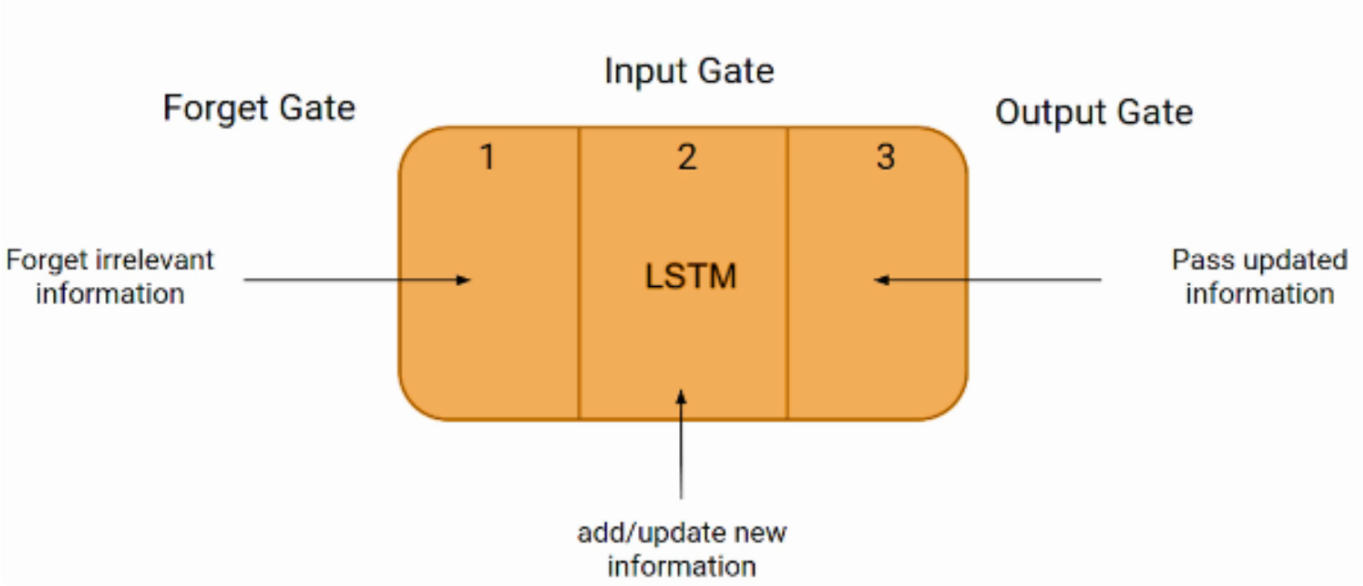
- The first part chooses whether the information coming from the previous timestamp is to be remembered or is irrelevant and can be forgotten.
- In the second part, the cell tries to learn new information from the input to this cell.

The Logic Behind LSTM

- The first part chooses whether the information coming from the previous timestamp is to be remembered or is irrelevant and can be forgotten.
- In the second part, the cell tries to learn new information from the input to this cell.
- At last, in the third part, the cell passes the updated information from the current timestamp to the next timestamp. This one cycle of LSTM is considered a single-time step.

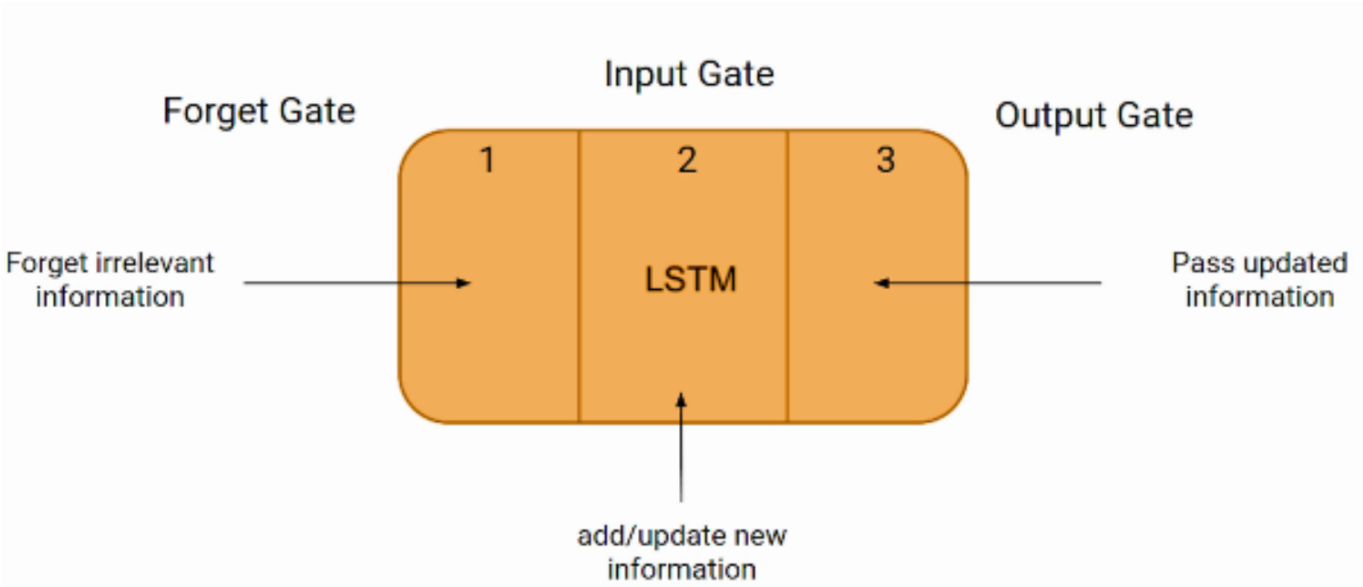
The Logic Behind LSTM

- These three parts of an LSTM unit are known as gates. They control the flow of information in and out of the memory cell or lstm cell.



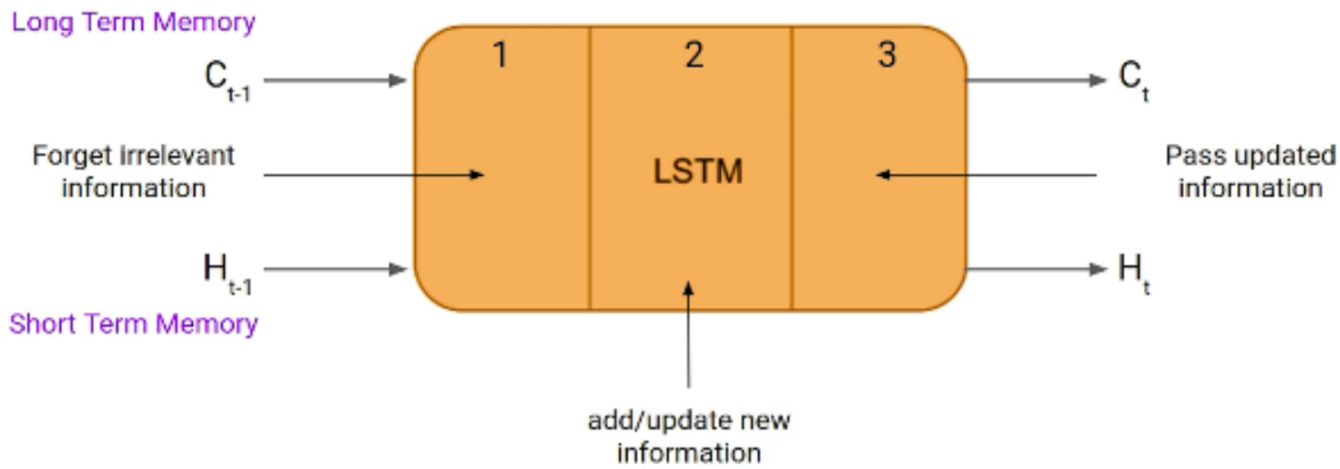
The Logic Behind LSTM

- These three parts of an LSTM unit are known as gates. They control the flow of information in and out of the memory cell or lstm cell.
- The first gate is called **Forget gate**, the second gate is known as the **Input gate**, and the last one is the **Output gate**.



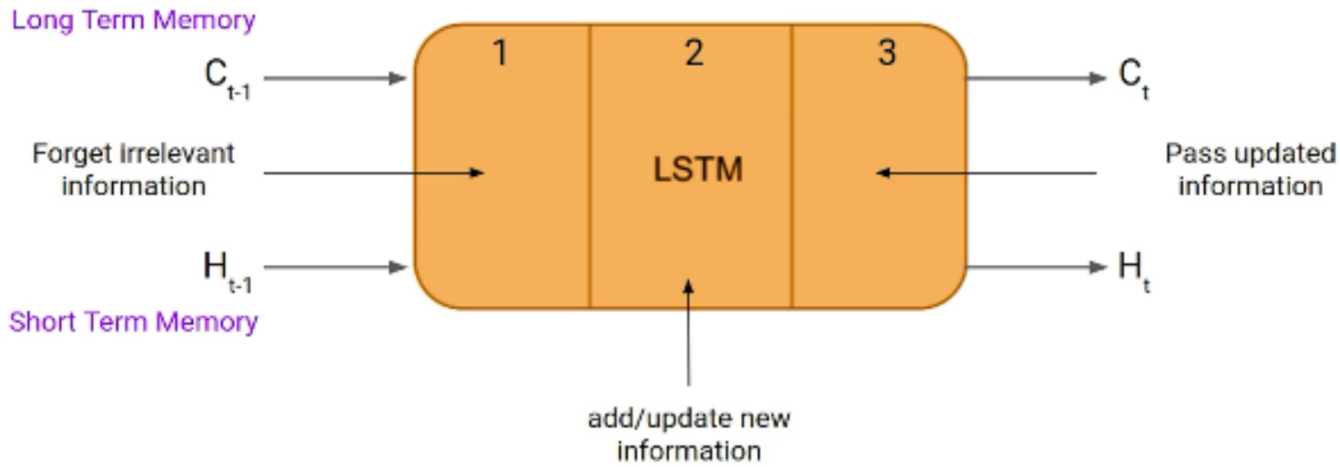
The Logic Behind LSTM

- An LSTM has a hidden state where H_{t-1} represents the hidden state of the previous timestamp and H_t is the hidden state of the current timestamp.



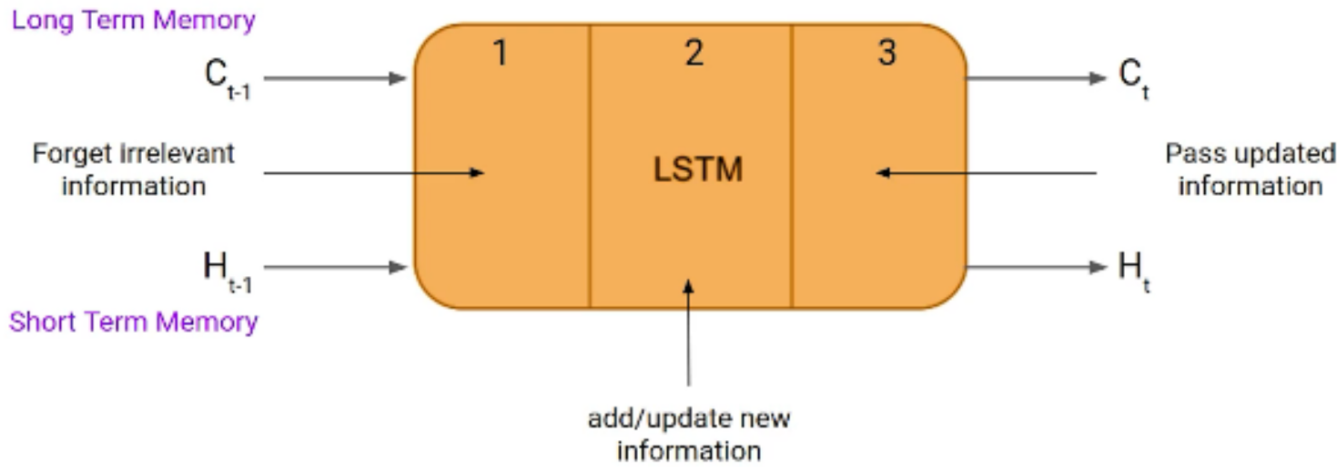
The Logic Behind LSTM

- An LSTM has a hidden state where H_{t-1} represents the hidden state of the previous timestamp and H_t is the hidden state of the current timestamp.
- Additionally, LSTM has a cell state represented by C_{t-1} and C_t for the previous and current timestamps, respectively.



The Logic Behind LSTM

- An LSTM has a hidden state where H_{t-1} represents the hidden state of the previous timestamp and H_t is the hidden state of the current timestamp.
- Additionally, LSTM has a cell state represented by C_{t-1} and C_t for the previous and current timestamps, respectively.
- Here the hidden state is known as Short term memory, and the cell state is known as Long term memory. Refer to the following image.



Forget gate

- The forget gate decides which information should be thrown away or kept. It looks at the current input x_t and the previous hidden state h_{t-1}, and outputs a number between 0 and 1 for each number in the cell state C_{t-1}. A value of 0 means "completely forget this" while a value of 1 means "completely retain this."

Forget gate

- The forget gate decides which information should be thrown away or kept. It looks at the current input x_t and the previous hidden state h_{t-1}, and outputs a number between 0 and 1 for each number in the cell state C_{t-1}. A value of 0 means "completely forget this" while a value of 1 means "completely retain this."
- The forget gate's decision is made using the following formula:

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

where W_f are the weights, b_f is the bias term for the forget gate, and σ denotes the sigmoid function, ensuring the output is between 0 and 1.

Remember (Input) gate

- The remember gate, often referred to as the input gate, decides what new information will be stored in the cell state.

Remember (Input) gate

- The remember gate, often referred to as the input gate, decides what new information will be stored in the cell state.
- It operates in two steps: First, a sigmoid layer decides which values will be updated, and then a tanh layer creates a vector of new candidate values, $\widetilde{C}_t$, that could be added to the state.

Remember (Input) gate

- The remember gate, often referred to as the input gate, decides what new information will be stored in the cell state.
- It operates in two steps: First, a sigmoid layer decides which values will be updated, and then a tanh layer creates a vector of new candidate values, $\widetilde{C}_t$, that could be added to the state.
- The formula for updating the cell state is:

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

$$\widetilde{C}_t = tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

where i_t is the output of the input gate, determining which values to update, and $\widetilde{C}_t$ is the vector of candidate values.

A more popular NN architecture to deal with sequence data ?

- Use Transformers !

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

"ImageNet Moment for Natural Language Processing"

Pretraining:
Download a lot of text from the internet

Train a giant Transformer model for language modeling

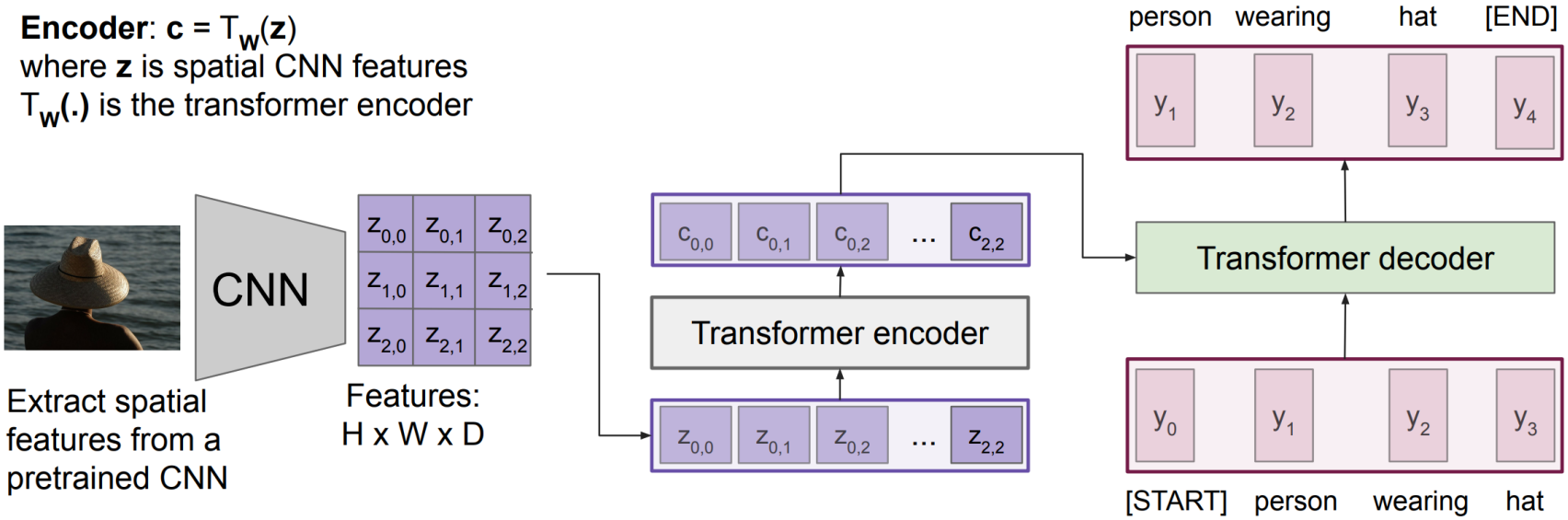
Finetuning:
Fine-tune the Transformer on your own NLP task

Image captioning using transformers

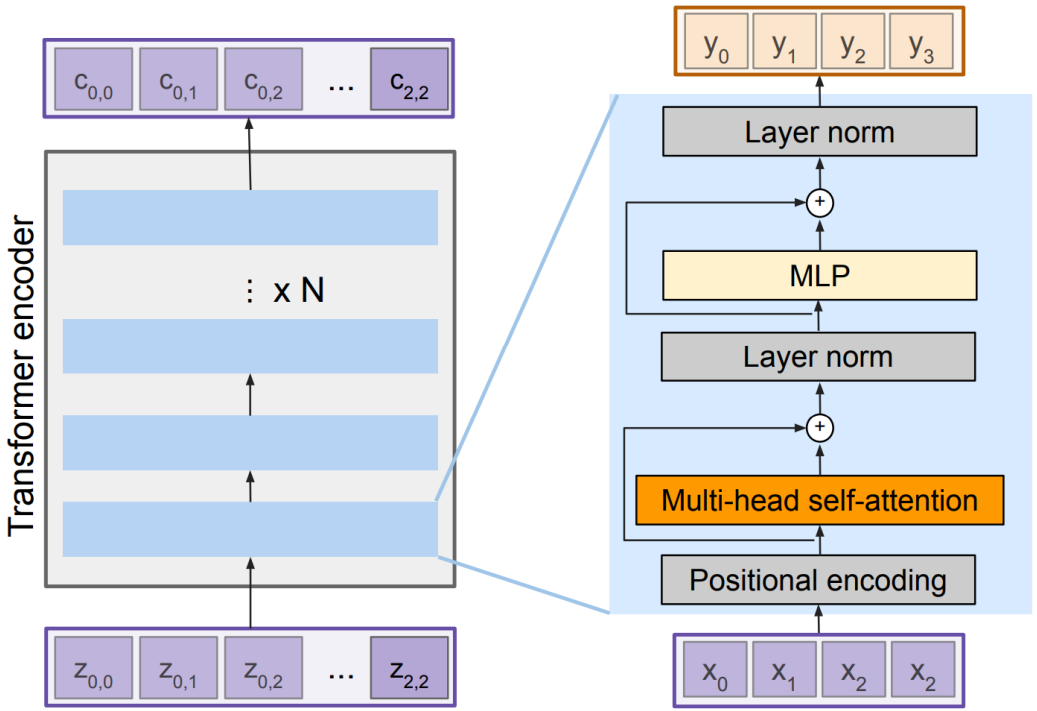
Input: Image **I**
Output: Sequence $\mathbf{y} = y_1, y_2, \ldots, y_T$

Encoder: $\mathbf{c} = T_W(\mathbf{z})$
where **z** is spatial CNN features
$T_W(.)$ is the transformer encoder

Decoder: $y_t = T_D(\mathbf{y}_{0:t-1}, \mathbf{c})$
where $T_D(.)$ is the transformer decoder



The Transformer encoder block



Transformer Encoder Block:

Inputs: Set of vectors **x**
Outputs: Set of vectors **y**

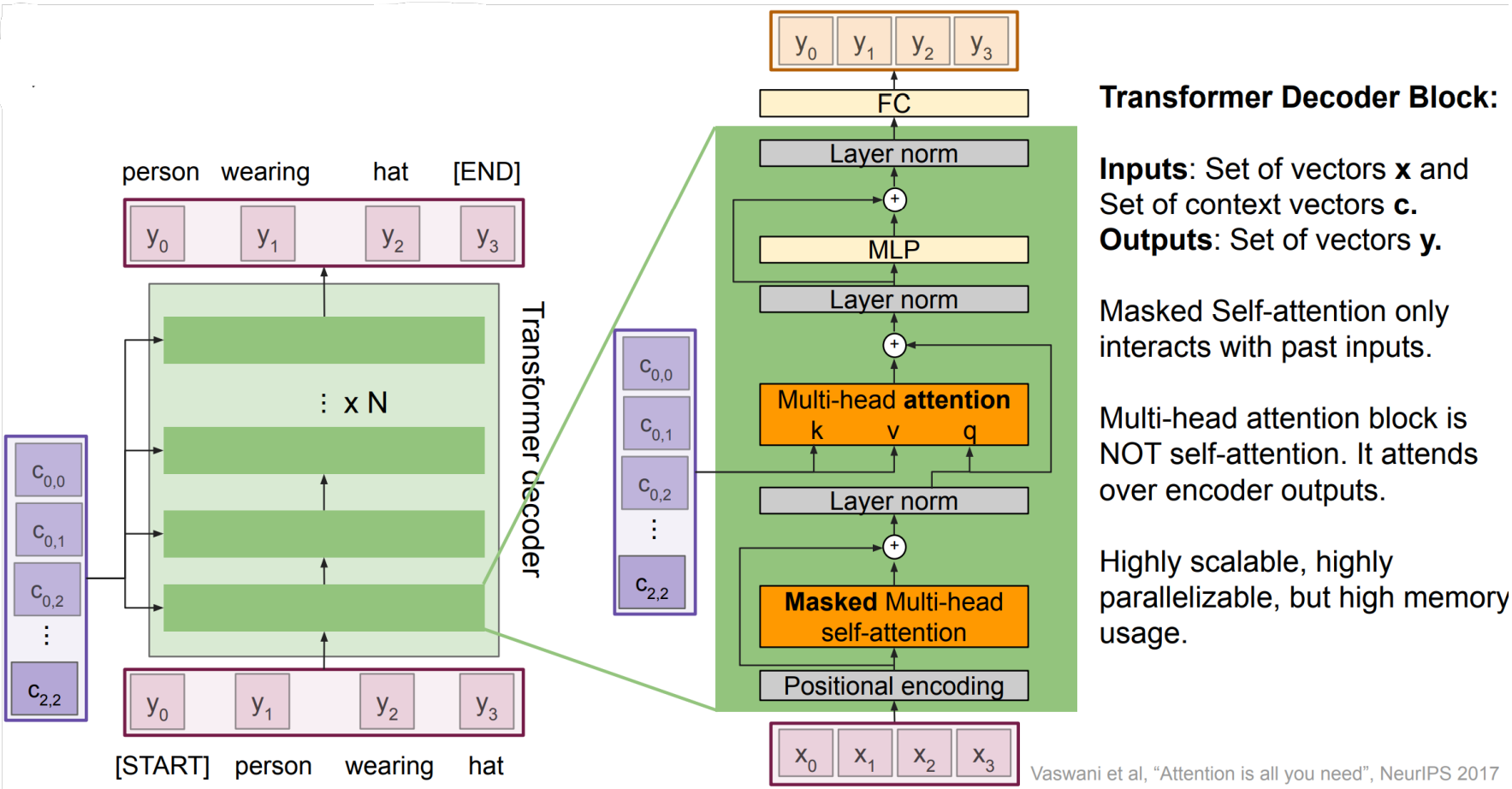
Self-attention is the only interaction between vectors.

Layer norm and MLP operate independently per vector.

Highly scalable, highly parallelizable, but high memory usage.

Vaswani et al, "Attention is all you need", NeurIPS 2017

The Transformer decoder block



Transformer Decoder Block:

Inputs: Set of vectors **x** and Set of context vectors **c.**
Outputs: Set of vectors **y.**

Masked Self-attention only interacts with past inputs.

Multi-head attention block is NOT self-attention. It attends over encoder outputs.

Highly scalable, highly parallelizable, but high memory usage.

Vaswani et al, "Attention is all you need", NeurIPS 2017

Image captioning using ONLY transformers

- Transformers from pixels to language