

# Deep Learning Foundations

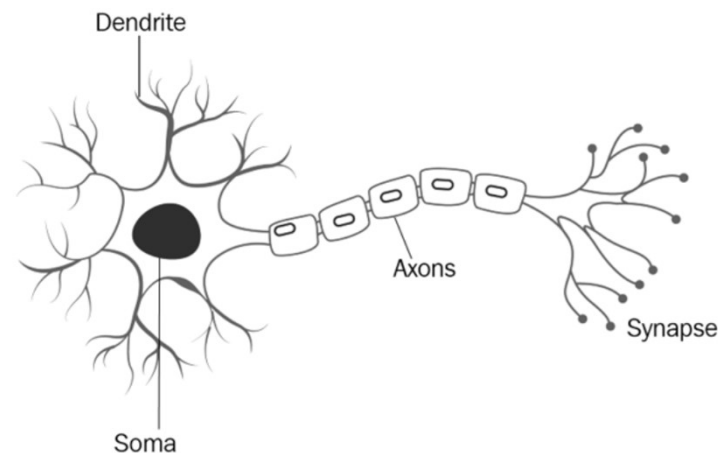
---

# Story So Far...

- We learned the basic reinforcement learning algorithms
  - dynamic programming
  - Monte Carlo methods
  - temporal difference learning
- Now we are about to move on to **Deep Reinforcement Learning (DRL)**, a method used for large and complex environments.
- DRL combines reinforcement learning with deep learning.
- Before going into DRL, we first look at the foundations of deep learning.
  - To those of you who already know and use deep learning, regard this as a refresher.

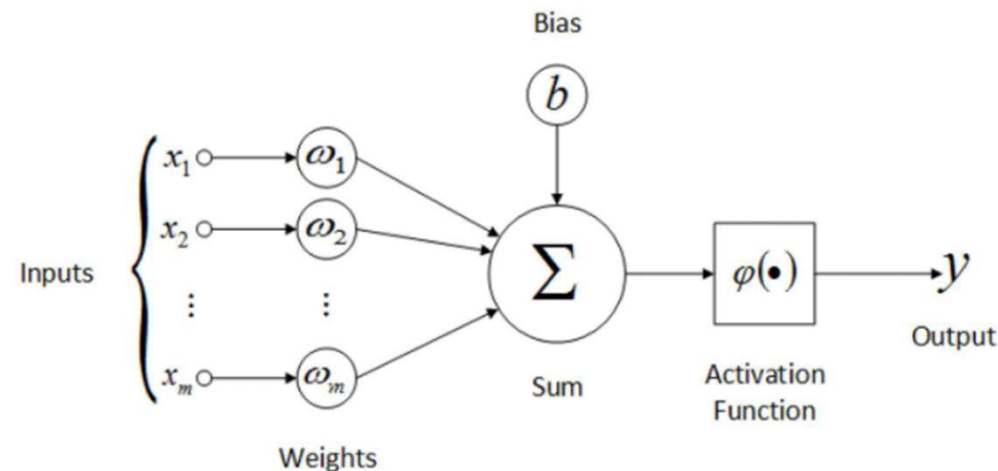
# Biological Neuron

- **Neurons** are the fundamental units of our brain and nervous system.
  - Our brain has approximately 100 billion neurons.
- Every neuron is connected to one another through a structure called a **synapse**.
  - A synapse receives input from the external environment via sensory organs.
  - A synapse also sends motor instructions to our muscles.
- A neuron can also receive inputs from other neurons through a branchlike structure called **dendrite**.
- The inputs are strengthened or weakened according to their importance, and they are summed together in the cell body called **soma**.
- From the cell body, the summed inputs are processed and move through **axons** and are sent to other neurons.



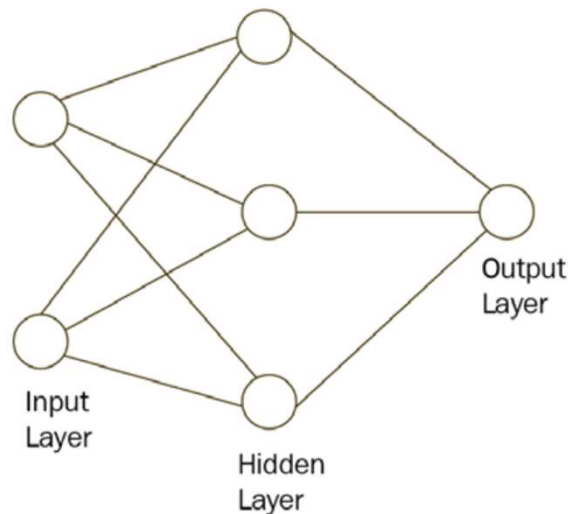
# Artificial Neuron

- A computational component modeled after a biological neuron
- An artificial neuron accepts one or multiple inputs  $x_i$ , and produces a single output  $y$ .
- The inputs are multiplied by their respective weights  $w_i$ .
  - All inputs are not equally important in calculating the output.
  - Weights are used to strengthen or weaken inputs.
- The weighted inputs are summed together, and a bias  $b$  is added.
- After that, we apply a non-linear function to the result. The function is called an **activation function** or a **transfer function**.



# Artificial Neural Network (ANN)

- To perform complex tasks, we need multiple neurons.
- The massive number of neurons in our brain are stacked in **layers** forming a network.
- Similarly, the artificial neurons are arranged in layers so that the information is passed from one layer to the other. This is called an artificial neural network (ANN).
- A typical ANN consists of the following layers
  - Input layer
  - Hidden layer(s)
  - Output layer



# Artificial Neural Network (ANN)

- Input layer
  - The input layer is where we feed input to the network.
  - The number of neurons in the input layer is the number of inputs.
  - No computation is performed in the input layer; it is just used for passing information from the outside world to the network.
- Hidden layer
  - Any layer between the input layer and the output layer is called a hidden layer.
  - A hidden layer is responsible for deriving complex relationships between input and output.
    - identifies patterns in the dataset
    - learns data representation
    - extracts features
  - We choose how many hidden layers will be in the model.
  - The network is called a **deep neural network** when we have many hidden layers.

# Artificial Neural Network (ANN)

- Output layer
  - The output layer produces output of the model.
  - The number of neurons in the output layer is based on the task we are trying to solve.
  - For a regression problem, we could have a single neuron in the output layer.
  - If we have a classification problem with five classes, the number of neurons in the output layer can be five.

# Activation Function

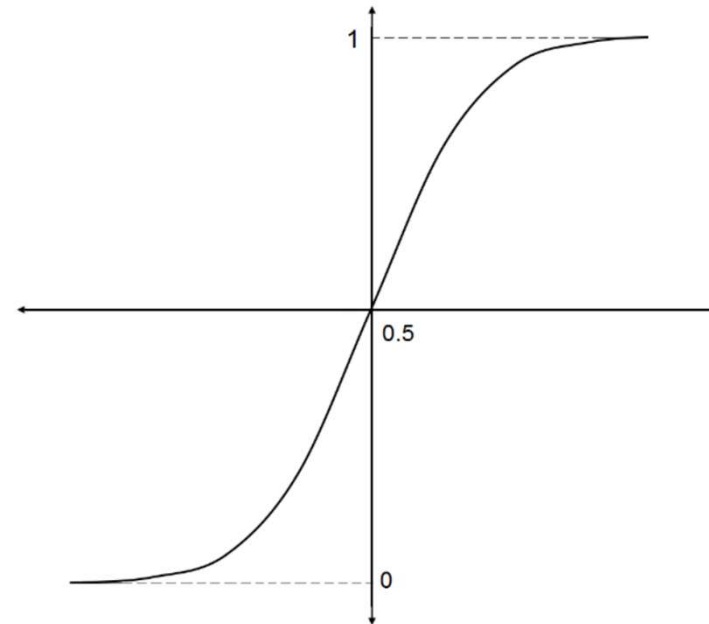
- A neuron adds all the weighted inputs and the bias, and then applies an activation function to calculate the output.
- The activation function adds **non-linearity** to the model.
- If we don't have activation function, the neural network becomes a linear model. A linear model is not effective in representing complex features.
- An activation function can be any function, but there are well-known and frequently used activation functions
  - sigmoid
  - tanh
  - ReLU
  - softmax



# Activation Function

- The sigmoid function
  - scales the value between 0 and 1
  - differentiable
  - monotonic
  - also known as logistic function
  - used for predicting the probability of output

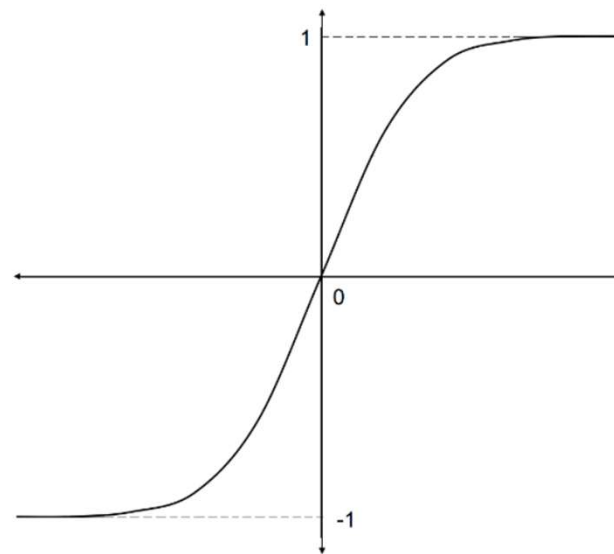
$$f(x) = \frac{1}{1 + e^{-x}}$$



# Activation Function

- The tanh function
  - hyperbolic tangent function
  - outputs a value between -1 and +1
  - centered at 0

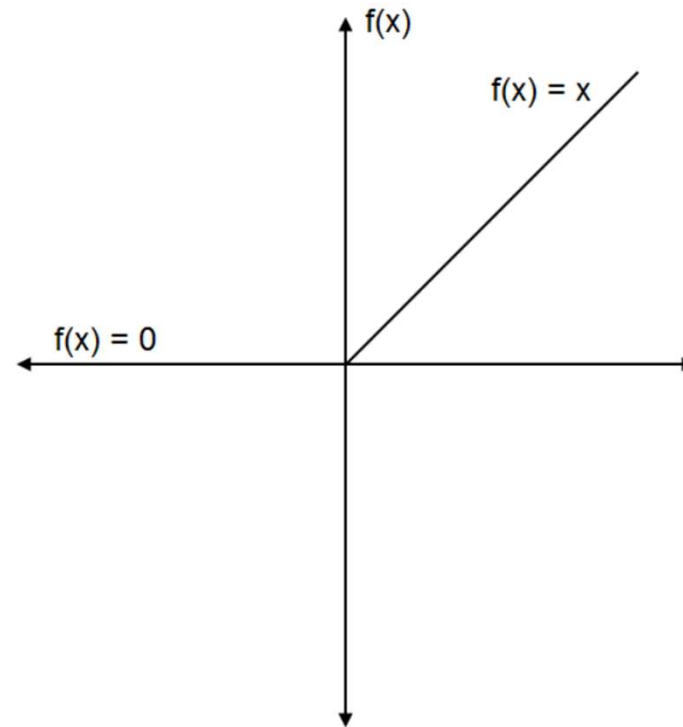
$$f(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$



# Activation Function

- The Rectified Linear Unit Function (ReLU)
  - outputs a value from 0 to infinity
  - a piecewise function

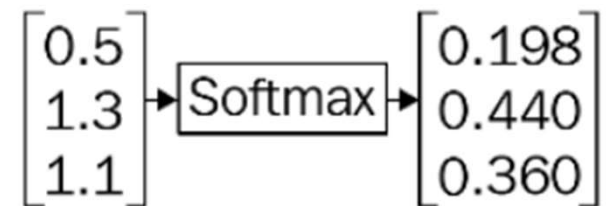
$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$



# Activation Function

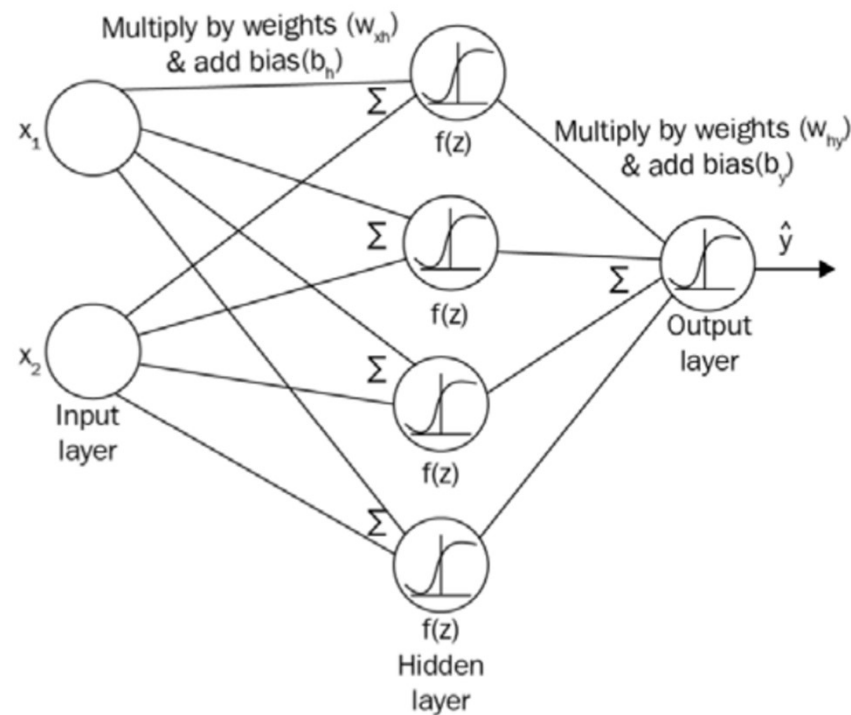
- The Softmax function
  - a generalization of the sigmoid function
  - usually applied to the final layer of the network while performing multi-class classification tasks
  - the sum of softmax values equals 1

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$



# Forward Propagation

- Suppose we have a network with one input layer, one hidden layer, and one output layer.
- We have two neurons in the input layer, four in the hidden layer, and one in the output layer.



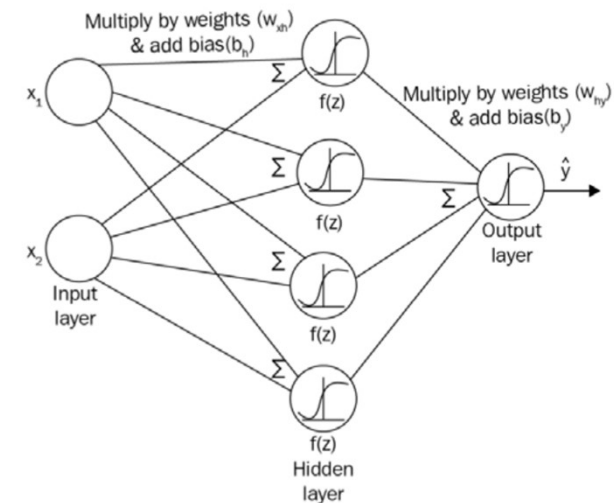
# Forward Propagation

- To move from input layer to hidden layer, we multiply the inputs by the weights and add the bias.

$$z_1 = XW_{xh} + b_h$$

- $W_{xh}$  is a weight matrix between the input and the hidden layer.
  - Its size is 2x4.
- $b_h$  is a bias vector of 4 elements.
- Then, we apply the activation function.

$$a_1 = \sigma(z_1)$$



# Forward Propagation

- Now we propagate from the hidden layer to the output layer.

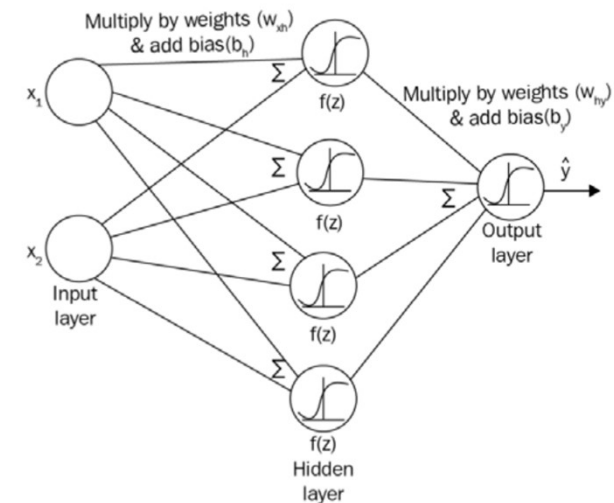
$$z_2 = a_1 W_{hy} + b_y$$

- $W_{hy}$  is a weight matrix between the hidden and the output layer.
  - Its size is 4x1.
- $b_y$  is a bias vector of 1 element.

- Then, we apply the activation function.

$$\hat{y} = \sigma(z_2)$$

- We have the output value.



# Forward Propagation

- The whole process

$$z_1 = XW_{xh} + b_h$$

$$a_1 = \sigma(z_1)$$

$$z_2 = a_1W_{hy} + b_y$$

$$\hat{y} = \sigma(z_2)$$

- In python code

```
def forward_prop(X):  
    z1 = np.dot(X, Wxh) + bh  
    a1 = sigmoid(z1)  
    z2 = np.dot(a1, Why) + by  
    y_hat = sigmoid(z2)  
  
    return y_hat
```



# The Cost Function

- In the neural network model we used, there were two weight matrices and two bias vectors:  $W_{xh}$ ,  $b_h$ ,  $W_{hy}$ ,  $b_y$ .
- They are called **model parameters**.
- When we first create the model, we do not know what values we should use for the parameters. So we just use **random values**.
- Because of that, when we propagate an input through the model, the output will be far from what we want.
- If we know the **"true" output (label)**, we can measure how far the predicted output (model output) is different from the true output, using a **cost function**. A cost function is also called a **loss function**.

# The Cost Function

- A popular cost function is the **mean squared error** (MSE) function.

$$J = \frac{1}{2n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

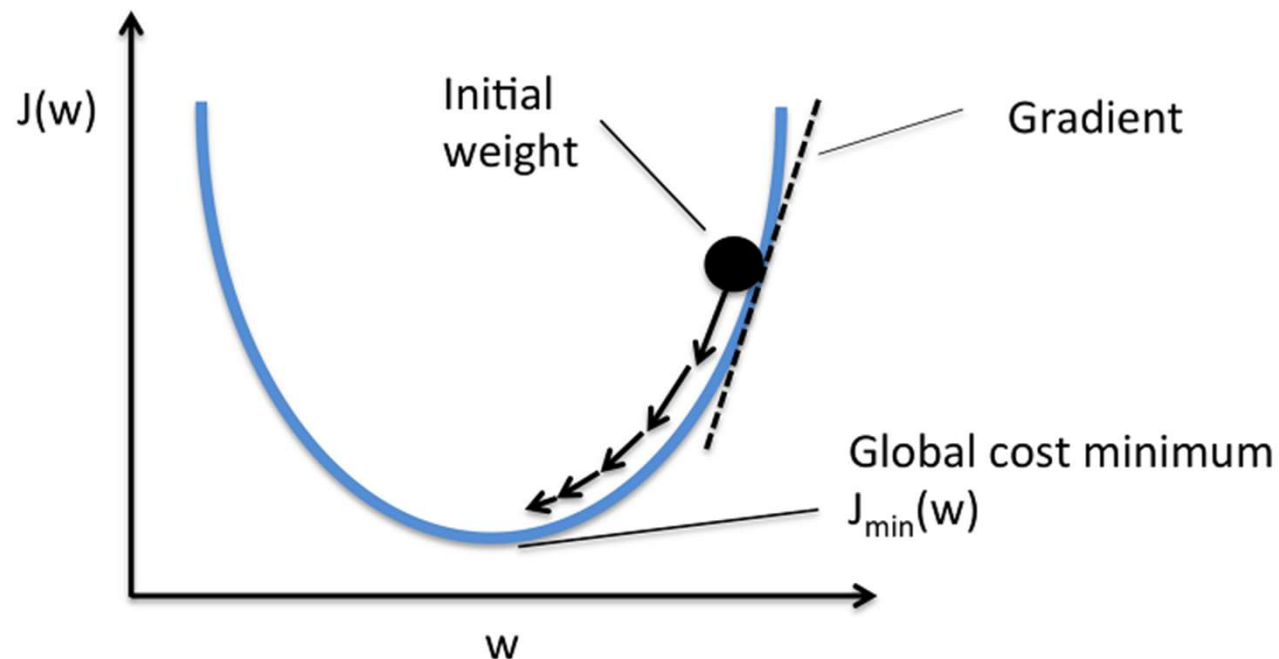
- $n$  is the number of training samples.
- $y_i$  is the true output.
- $\hat{y}_i$  is the predicted output (model output).
- If  $J$  is large, it means the model is not producing outputs close to their true outputs; the model is not performing well.
- If  $J$  is small, it means the model is performing well.

# Gradient Descent

- We want to iteratively **update the model parameters (weights and biases)** so that the **cost function is minimized**.
- We do this by applying a technique called **gradient descent**.

# Gradient Descent

- Let us consider a single scalar value  $w$ .
- We want to update  $w$  so that the cost function  $J$  is reduced.
- What we can do is calculate the **gradient of  $J$  at  $w$** .
- If the gradient is positive at  $w$ , we increase  $w$ .
- If the gradient is negative at  $w$ , we decrease  $w$ .
- Since gradient changes according to  $w$ , we change the value of  $w$  in steps.

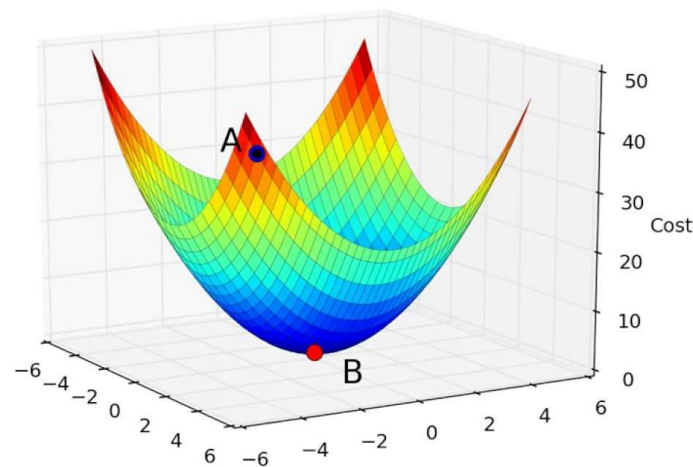


# Gradient Descent

- In a neural network models, we have vectors as parameters.
- We do similar things and calculate  $\frac{\partial J}{\partial W}$ . It gives us the **direction** in which we need to move.
- Then, we update the weights according to the following equation.

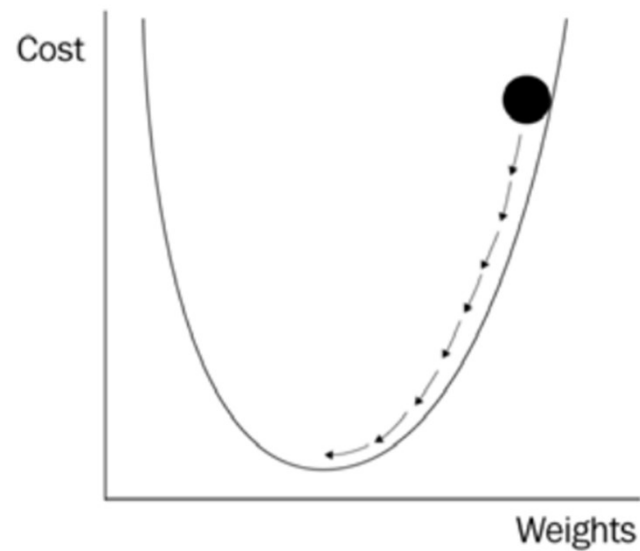
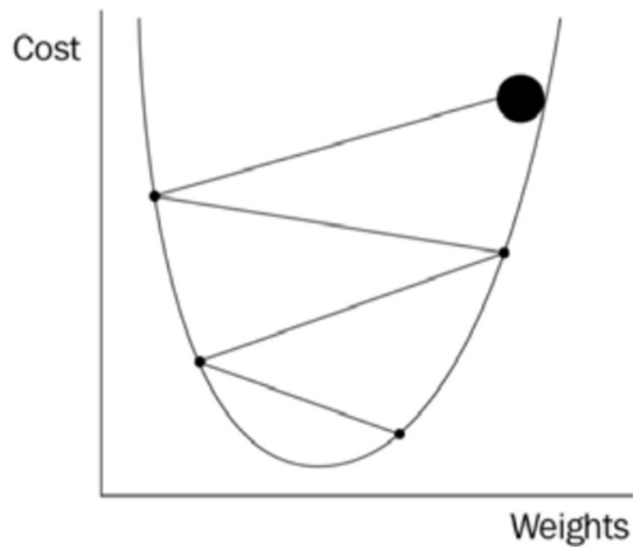
$$W = W - \alpha \frac{\partial J}{\partial W}$$

- $\alpha$  is the **learning rate**. It determines the step we move towards the direction.



# Gradient Descent

- Learning rate is a **hyper-parameter** which the operator chooses.
- If learning rate is too high, may fail to find the optimal point.
- If learning rate is too small, learning takes too much time.



# BackPropagation

- In order to update all the parameters, we need the gradients of  $J$  with respect to  $W_{xh}, b_h, W_{hy}, b_y$ .

- Recall the equations for forward propagation

$$z_1 = XW_{xh} + b_h$$

$$a_1 = \sigma(z_1)$$

$$z_2 = a_1W_{hy} + b_y$$

$$\hat{y} = \sigma(z_2)$$

- To calculate  $\frac{\partial J}{\partial W_{hy}}$ , we use the **chain rule**.

$$\frac{\partial J}{\partial W_{hy}} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_2} \cdot \frac{\partial z_2}{\partial W_{hy}}$$

- We calculate the three terms here and multiply them.

# BackPropagation

- Calculating  $\frac{\partial J}{\partial W_{hy}}, \frac{\partial J}{\partial b_y}$

$$\frac{\partial J}{\partial \hat{y}} = -(y - \hat{y}) \quad \frac{\partial \hat{y}}{\partial z_2} = \sigma'(z_2) \quad \frac{\partial z_2}{\partial W_{hy}} = a_1 \quad \frac{\partial z_2}{\partial b_y} = 1$$

- $\sigma'$  is a derivative of the sigmoid function.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \quad \sigma'(z) = \frac{e^z}{(1 + e^{-z})^2}$$

- Thus, substituting all terms in the chain rule equation, we get:

$$\frac{\partial J}{\partial W_{hy}} = -(y - \hat{y}) \cdot \sigma'(z_2) \cdot a_1$$

$$\frac{\partial J}{\partial b_y} = -(y - \hat{y}) \cdot \sigma'(z_2)$$



# BackPropagation

- Calculating  $\frac{\partial J}{\partial W_{xh}}, \frac{\partial J}{\partial b_h}$

$$\frac{dz_1}{dW_{xh}} = X$$

- Chain rule

$$\frac{\partial J}{\partial W_{xh}} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_2} \cdot \frac{\partial z_2}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial W_{xh}}$$

- We already know the first two terms. The last three terms are:

$$\frac{\partial z_2}{\partial a_1} = W_{hy} \quad \frac{\partial a_1}{\partial z_1} = \sigma'(z_1) \quad \frac{\partial z_1}{\partial W_{xh}} = x \quad \frac{\partial z_1}{\partial b_h} = 1$$

- Substituting all terms, we get:

$$\frac{\partial J}{\partial W_{xh}} = -(y - \hat{y}) \cdot \sigma'(z_2) \cdot W_{hy} \cdot \sigma'(z_1) \cdot x \quad \frac{\partial J}{\partial b_y} = -(y - \hat{y}) \cdot \sigma'(z_2) \cdot W_{hy} \cdot \sigma'(z_1)$$

# BackPropagation

- Once we calculate the gradients, we can update the parameters using gradient descent.

- $$W_{hy} = W_{hy} - \alpha \frac{\partial J}{\partial W_{hy}}$$

- $$b_y = b_y - \alpha \frac{\partial J}{\partial b_y}$$

- $$W_{xh} = W_{xh} - \alpha \frac{\partial J}{\partial W_{xh}}$$

- $$b_h = b_h - \alpha \frac{\partial J}{\partial b_h}$$

```
def backward_prop(y_hat, z1, a1, z2):  
    delta2 = np.multiply(-(y-y_hat),sigmoid_derivative(z2))  
    dJ_dWhy = np.dot(a1.T, delta2)  
  
    delta1 = np.dot(delta2,Why.T)*sigmoid_derivative(z1)  
    dJ_dWxh = np.dot(X.T, delta1)  
  
    Wxh = Wxh - alpha * dJ_dWhy  
    Why = Why - alpha * dJ_dWxh  
  
    return Wxh,Why
```

# Variants of Gradient Descent Methods

- Stochastic gradient descent (SGD)
  - Instead of getting the gradient from the whole dataset, we choose a small random subset of the dataset and calculate gradient.
  - Reduces computational burden, achieving faster iterations in trade for a lower convergence rate.
- Momentum
  - Uses a notion of momentum
  - remembers the previous update  $\Delta w$ , and determines the next update as a linear combination of the gradient and the previous update

$$\Delta w := \alpha \Delta w - \eta \nabla Q_i(w)$$

$$w := w + \Delta w$$

- tends to keep traveling in the same direction, preventing oscillations

# Variants of Gradient Descent Methods

- AdaGrad
  - Uses per-parameter learning rate
  - Increases learning rate for sparser parameters and decreases learning rate for ones that are less sparse
  - works well when data is sparse and sparse parameters are more informative.
- RMSProp
  - The problem of AdaGrad is that in the later stages, the learning rate becomes too small.
  - RMSProp revises the equation so that the learning does not stop
- Adam (Adaptive Moment Estimation)
  - A combination of momentum and RMSProp
  - Uses the concept of momentum, as well as the concept of adaptive learning rate

# Learning Process

- Prepare a training set
  - Do any necessary preprocessing
- Define a model
  - Choose the model architecture. E.g.) # of layers, # of neurons, activation function
  - Initialize parameters (weights and biases)
- Define the gradient descent method
  - Choose hyperparameters such as learning rate and momentum factor
- Choose a batch size for stochastic gradient descent
  - Divide the training set into batches of the given size
- Choose how many **epochs** we are going to run training.
  - One epoch is equal to one forward pass and one backward pass for all training samples.
- For each batch, we calculate the forward pass
  - Feed the batch samples into the network and get the output
- Then, calculate the backward pass and update the parameters
  - Perform backpropagation

## Example: Training a NN to do XOR operation [ex015]

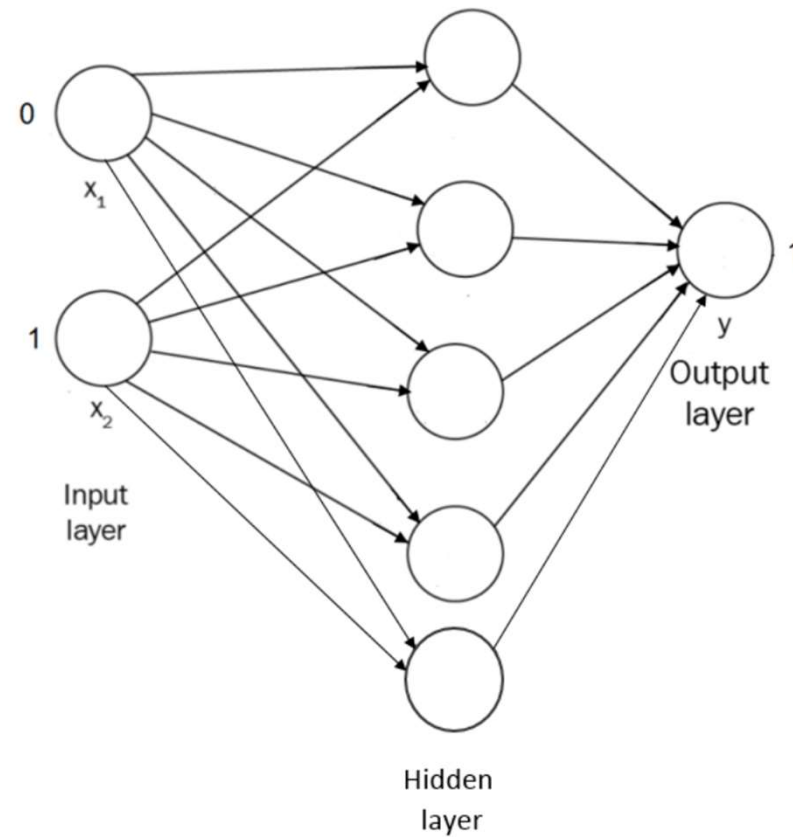
- Our goal is to train a neural network, so that it acts like an XOR gate.

| Input(x) |       | Output(y) |
|----------|-------|-----------|
| $x_1$    | $x_2$ | $y$       |
| 0        | 0     | 0         |
| 0        | 1     | 1         |
| 1        | 0     | 1         |
| 1        | 1     | 0         |

- We can see that the number of inputs is 2, and the number of outputs is 1.
- The four different samples will be our training set.

# Example: Training a NN to do XOR operation

- We will use a model with a single hidden layer.
- We will use sigmoid as our activation function.



# Example: Training a NN to do XOR operation

- Import libraries

```
import numpy as np
import matplotlib.pyplot as plt
```

- Dataset

```
X = np.array([[0,1], [1,0], [1,1], [0,0]])
y = np.array([[1], [1], [0], [0]])
```

- Model architecture

```
num_input = 2
num_hidden = 5
num_output = 1
```



# Example: Training a NN to do XOR operation

- Initialize model parameters

```
Wxh = np.random.randn(num_input, num_hidden)
bh = np.zeros((1, num_hidden))
Why = np.random.randn(num_hidden, num_output)
by = np.zeros((1, num_output))
```

- Define sigmoid function

```
def sigmoid(z):
    return 1 / (1+np.exp(-z))
```

- Define derivative of sigmoid

```
def sigmoid_derivative(z):
    return np.exp(-z)/((1+np.exp(-z))**2)
```

# Example: Training a NN to do XOR operation

- Define function for forward propagation

```
def forward_prop(x, Wxh, Why):  
    z1 = np.dot(x, Wxh) + bh  
    a1 = sigmoid(z1)  
    z2 = np.dot(a1, Why) + by  
    y_hat = sigmoid(z2)  
  
    return z1, a1, z2, y_hat
```

- Define function for backward propagation

```
def backward_prop(y_hat, z1, a1, z2):  
  
    delta2 = np.multiply(-(y-y_hat), sigmoid_derivative(z2))  
    dJ_dWhy = np.dot(a1.T, delta2)  
    dJ_dby = delta2  
    delta1 = np.dot(delta2, Why.T) * sigmoid_derivative(z1)  
    dJ_dWxh = np.dot(X.T, delta1)  
    dJ_dbh = delta1  
  
    return dJ_dWxh, dJ_dWhy, dJ_dbh, dJ_dby
```

# Example: Training a NN to do XOR operation

- Define cost function

```
def cost_function(y, y_hat):  
    J = 0.5*sum((y-y_hat)**2)  
    return J
```

- Code for training (updating model parameters)

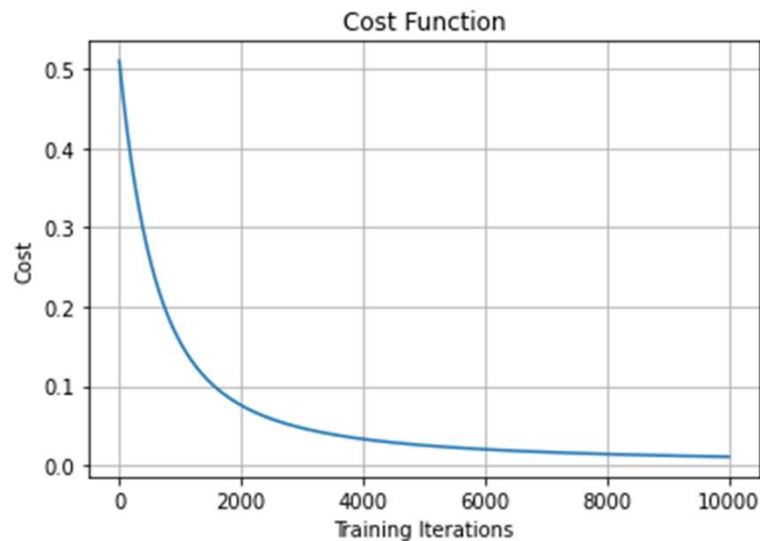
```
alpha = 0.01  
num_iterations = 10000  
  
cost = []  
for i in range(num_iterations):  
    z1, a1, z2, y_hat = forward_prop(X, Wxh, Why)  
    dJ_dWxh, dJ_dWhy, dJ_dbh, dJ_dby = backward_prop(y_hat, z1, a1, z2)  
  
    Wxh = Wxh - alpha * dJ_dWxh  
    Why = Why - alpha * dJ_dWhy  
    bh = bh - alpha * dJ_dbh  
    by = by - alpha * dJ_dby  
  
    c = cost_function(y, y_hat)  
    cost.append(c)
```

# Example: Training a NN to do XOR operation

- Plotting the cost function after each epoch

```
plt.grid()
plt.plot(range(num_iterations), cost)
plt.title('Cost Function')
plt.xlabel('Training Iterations')
plt.ylabel('Cost')
plt.show()
```

✓ 0.1s



```
_, _, _, pred = forward_prop(X, Wxh, Why)
print(pred)
```

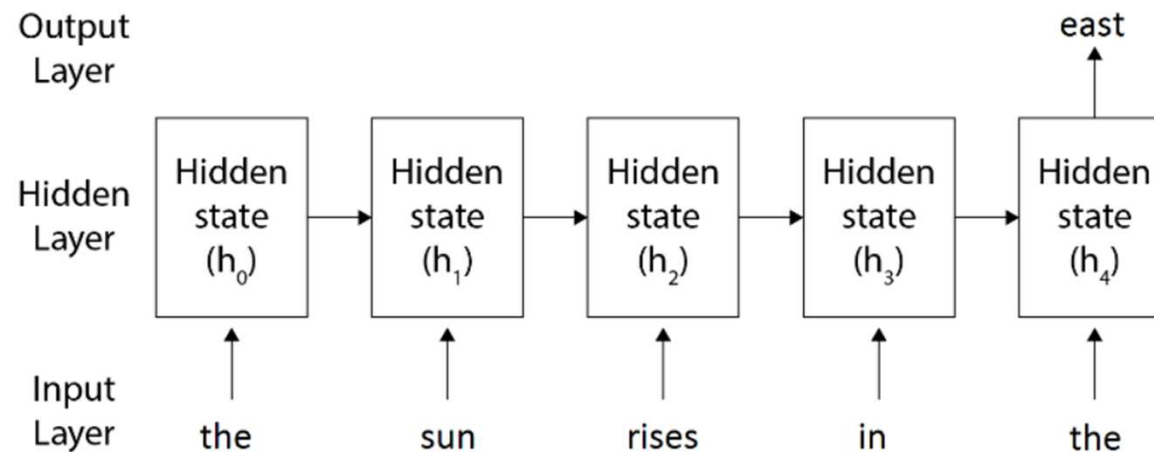
✓ 0.3s

```
[[0.9402146 ]
 [0.93216196]
 [0.06762825]
 [0.06723748]]
```

# Recurrent Neural Networks

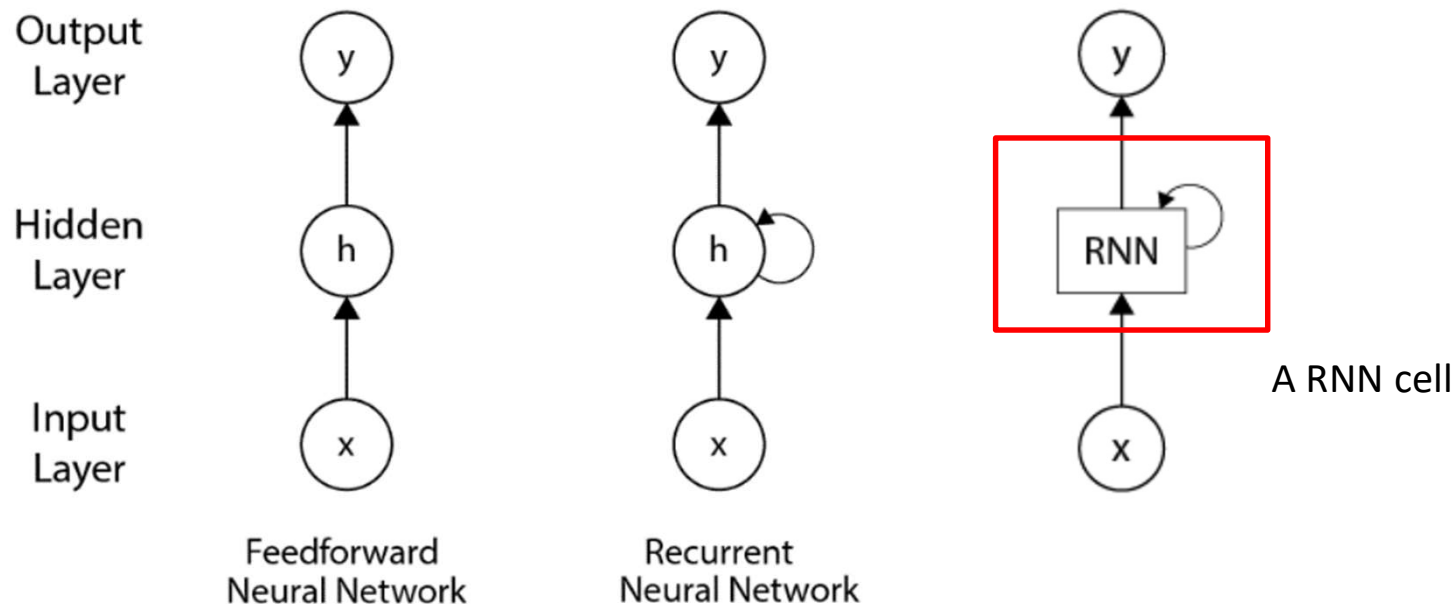
- A recurrent neural network (RNN) is a type of ANN commonly used for **sequential data or time series data**, such as the ones used in **speech recognition** or **natural language processing**.
- RNNs recognize data's sequential characteristics and use patterns to predict the next likely scenario.

*The sun rises in the \_\_\_\_.*



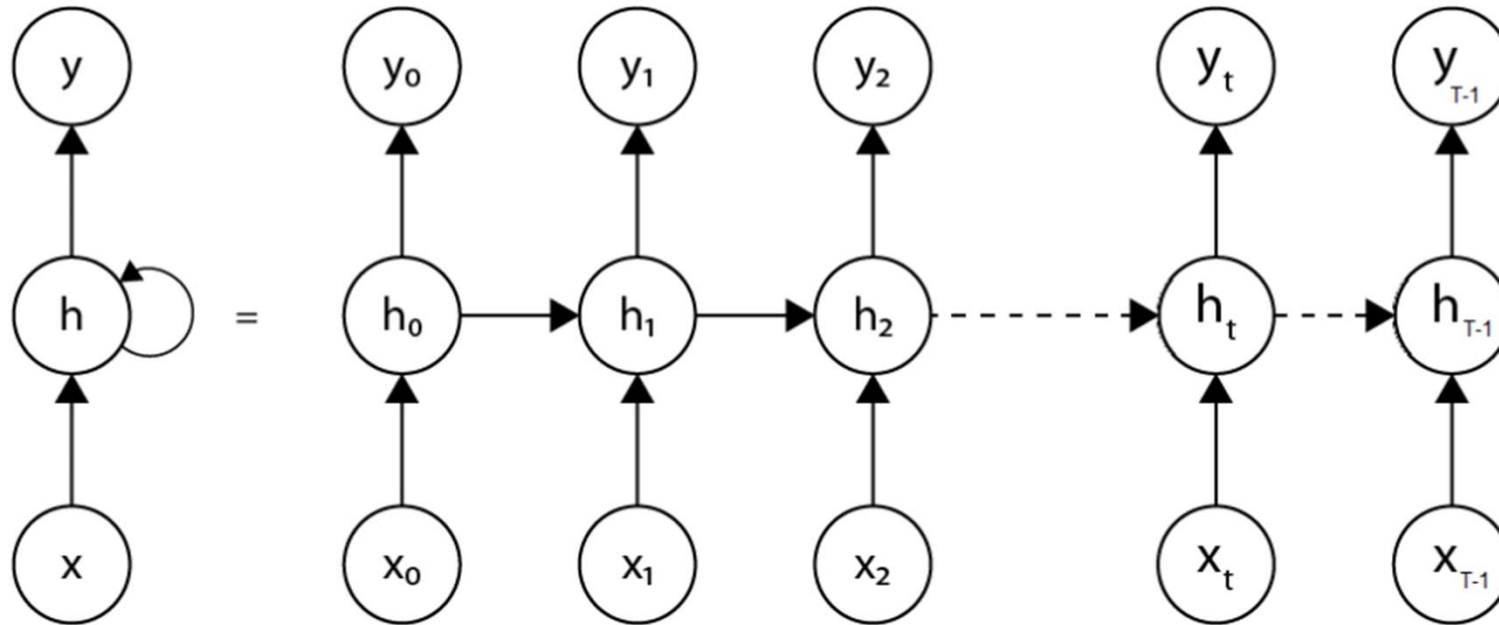
# RNN vs. Feed-Forward Networks

- In a **feed-forward network**, connections do not form a cycle. In other words, output of the previous input is not used in the processing of the current input.
- In a **recurrent network**, the output of the previous input is used in the processing of the current input. It is like having a **memory** of previous inputs.



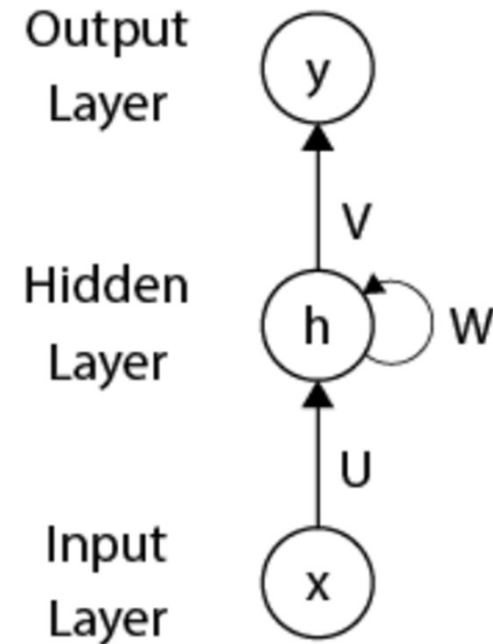
# Recurrent Neural Networks

- Suppose we have an input sequence of  $T$  words.
- Then, we have  $T$  layers, one for each word.



# Forward Propagation in an RNN

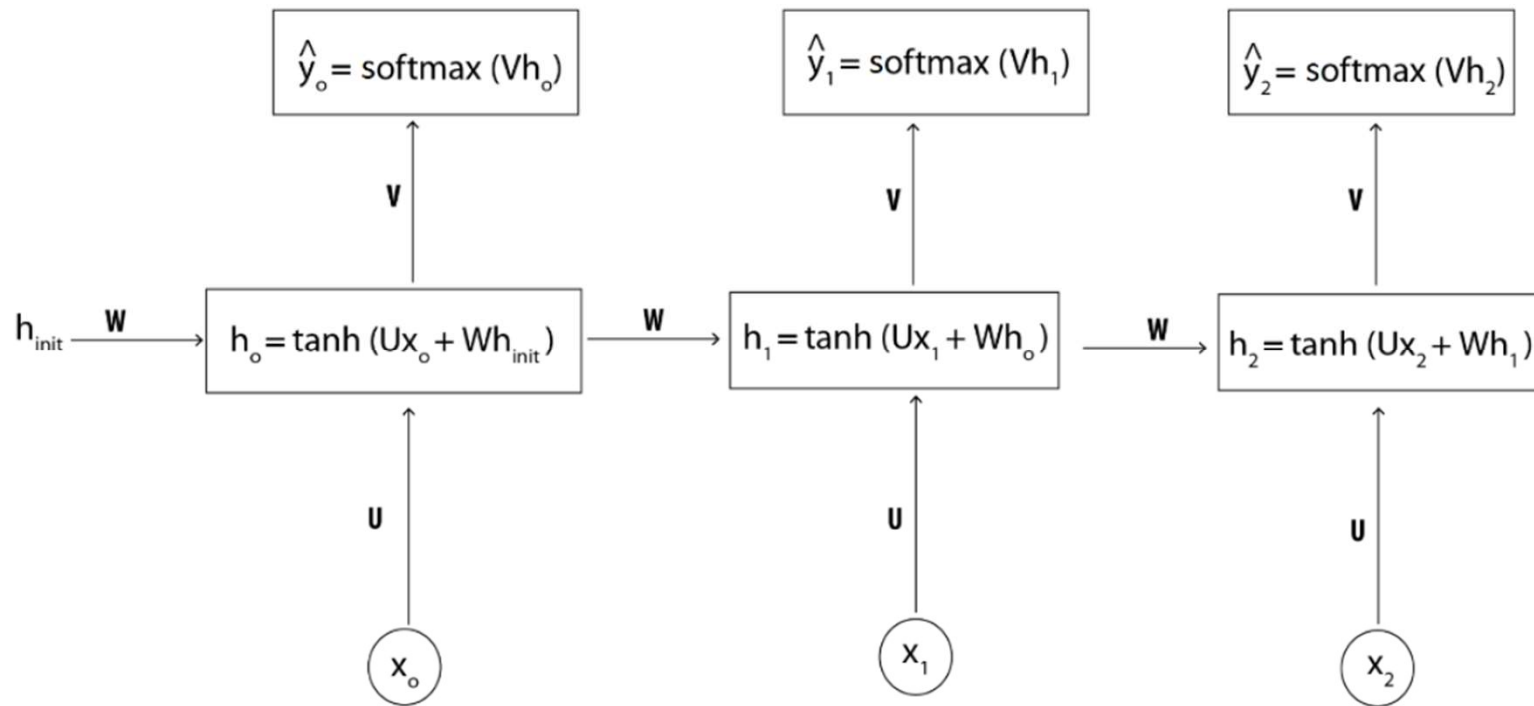
- Notations
  - $U$ : input to hidden layer weight matrix
  - $W$ : hidden to hidden layer weight matrix
  - $V$ : hidden to output layer weight matrix
- The hidden state  $h$  at time step  $t$ 
  - $h_t = \tanh(Ux_t + Wh_{t-1})$
  - The initial hidden state is set to random values
- The output at time step  $t$ 
  - $\hat{y}_t = \text{softmax}(Vh_t)$





# Forward Propagation in RNN

- Unrolled representation of RNN



# Backpropagation through Time

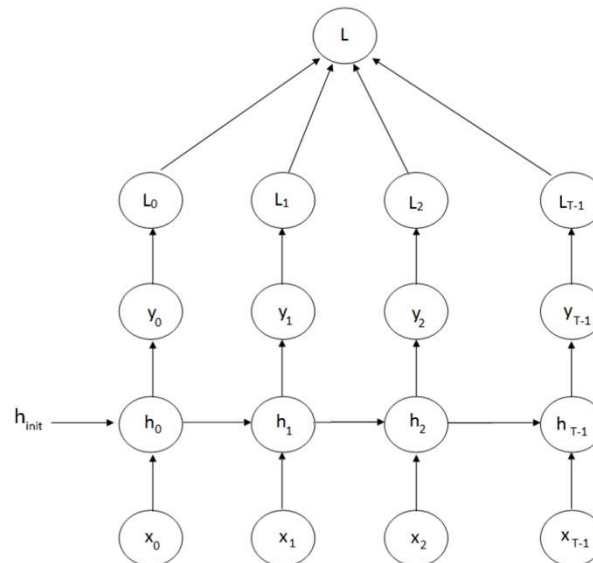
- In order to update the parameters, we first need to compute the loss function. Here we use the cross-entropy loss function.

$$L_t = -y_t \log(\hat{y}_t)$$

- cross-entropy is an alternative to mean squared error. CE is known to produce faster convergence compared to MSE.

- The final loss is a sum of the loss at all the time steps.

$$L = \sum_{j=0}^{T-1} L_j$$



# Backpropagation through Time

- Our goal is to update parameters so that the loss is minimized.
- Equation for updating the weights

$$U = U - \alpha \frac{\partial L}{\partial U} \quad W = W - \alpha \frac{\partial L}{\partial W} \quad V = V - \alpha \frac{\partial L}{\partial V}$$

- To update  $V$ , we use the following equation.
- For example, for the output at time  $t = 3$ ,

- $$\frac{\partial E_3}{\partial V} = \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial V}$$

# Backpropagation through Time

- Now, we need to update  $W$ . Using the chain rule, we can say:

$$\frac{\partial E_3}{\partial W} = \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial W}$$

- However, that is not all. We can also say,

$$\frac{\partial E_3}{\partial W} = \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial \bar{s}_2} \cdot \frac{\partial \bar{s}_2}{\partial W} \quad \frac{\partial E_3}{\partial W} = \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial \bar{s}_2} \cdot \frac{\partial \bar{s}_2}{\partial \bar{s}_1} \cdot \frac{\partial \bar{s}_1}{\partial W}$$

- So we add up everything.

$$\frac{\partial E_3}{\partial W} = \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial W} + \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial \bar{s}_2} \cdot \frac{\partial \bar{s}_2}{\partial W} + \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial \bar{s}_2} \cdot \frac{\partial \bar{s}_2}{\partial \bar{s}_1} \cdot \frac{\partial \bar{s}_1}{\partial W}$$

- Since the backpropagation propagates back in time, it is called **backpropagation through time (BPTT)**.

# Backpropagation through Time

- Say we are at  $t=10$ . How far should we go back in time when calculating the gradient for  $W$ ?
- As gradients are multiplied, its value becomes very small. This is called the **vanishing gradient problem**.
- So we can only backpropagate through a limited amount of time.

$$\frac{\partial E_3}{\partial W} = \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial \bar{s}_2} \cdot \frac{\partial \bar{s}_2}{\partial \bar{s}_1} \cdot \frac{\partial \bar{s}_1}{\partial W}$$

- Similar calculations apply to  $U$ .

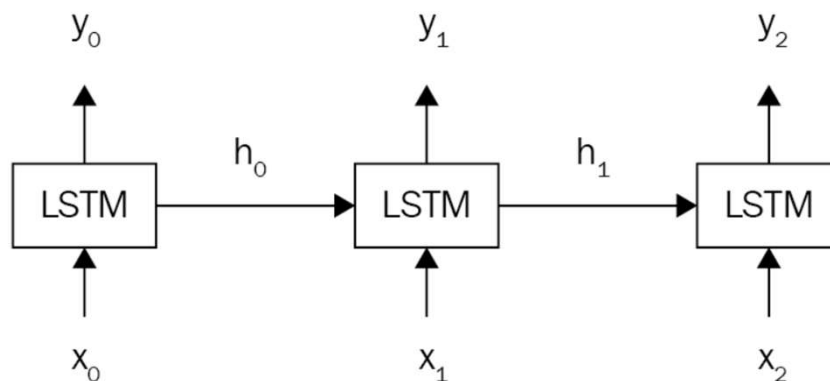
$$\frac{\partial E_3}{\partial U} = \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial U} + \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial \bar{s}_2} \cdot \frac{\partial \bar{s}_2}{\partial U} + \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial \bar{s}_2} \cdot \frac{\partial \bar{s}_2}{\partial \bar{s}_1} \cdot \frac{\partial \bar{s}_1}{\partial U}$$

# Long-Term Short-Memory (LSTM) Network

- The problem of RNN is that it cannot retain long sequences in the memory. Thus, it is not good at predicting the blank term in:

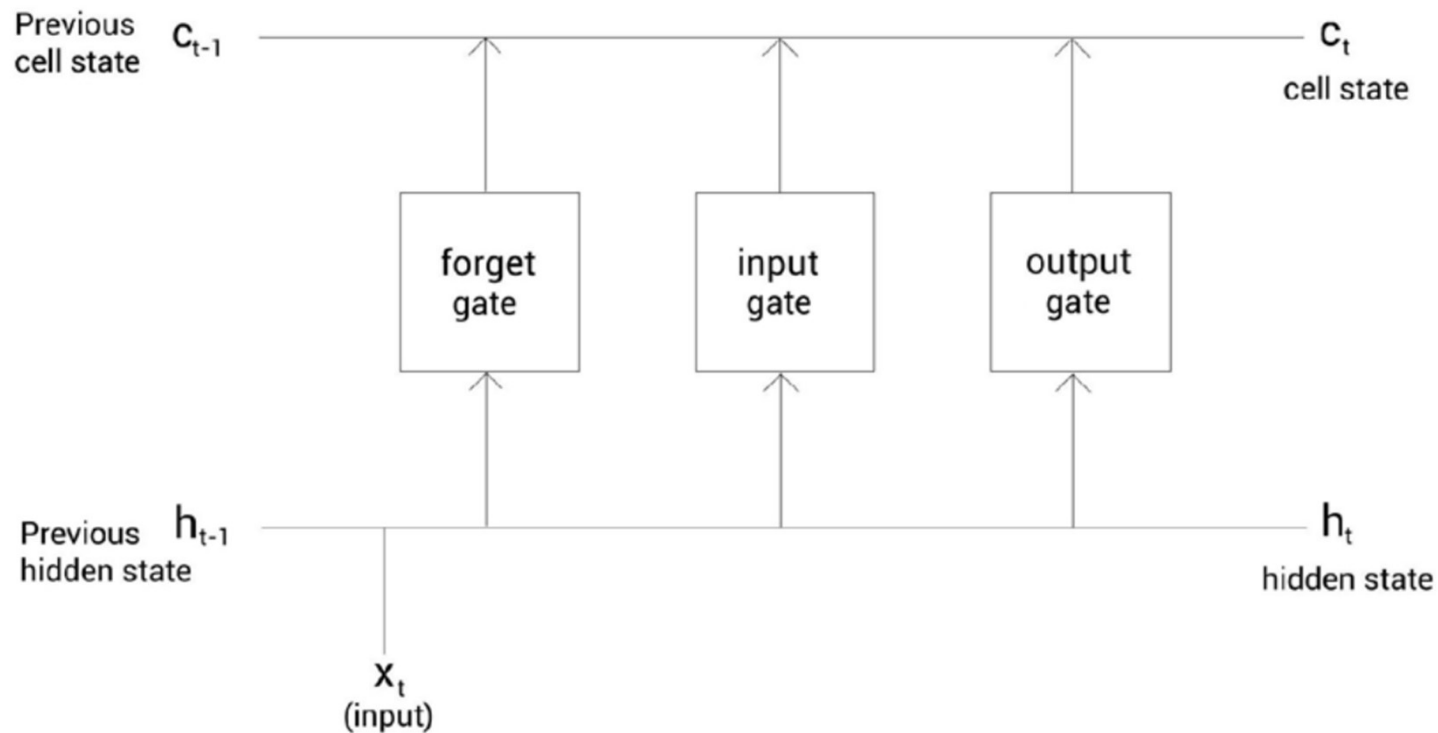
*Archie lived in China for 13 years. He loves listening to good music. He is a fan of comics.  
He is fluent in \_\_\_\_.*

- LSTM is a variant of RNN that can retain information in the memory as long as it is required.
- In LSTM network, the RNN cells are replaced with LSTM cells.



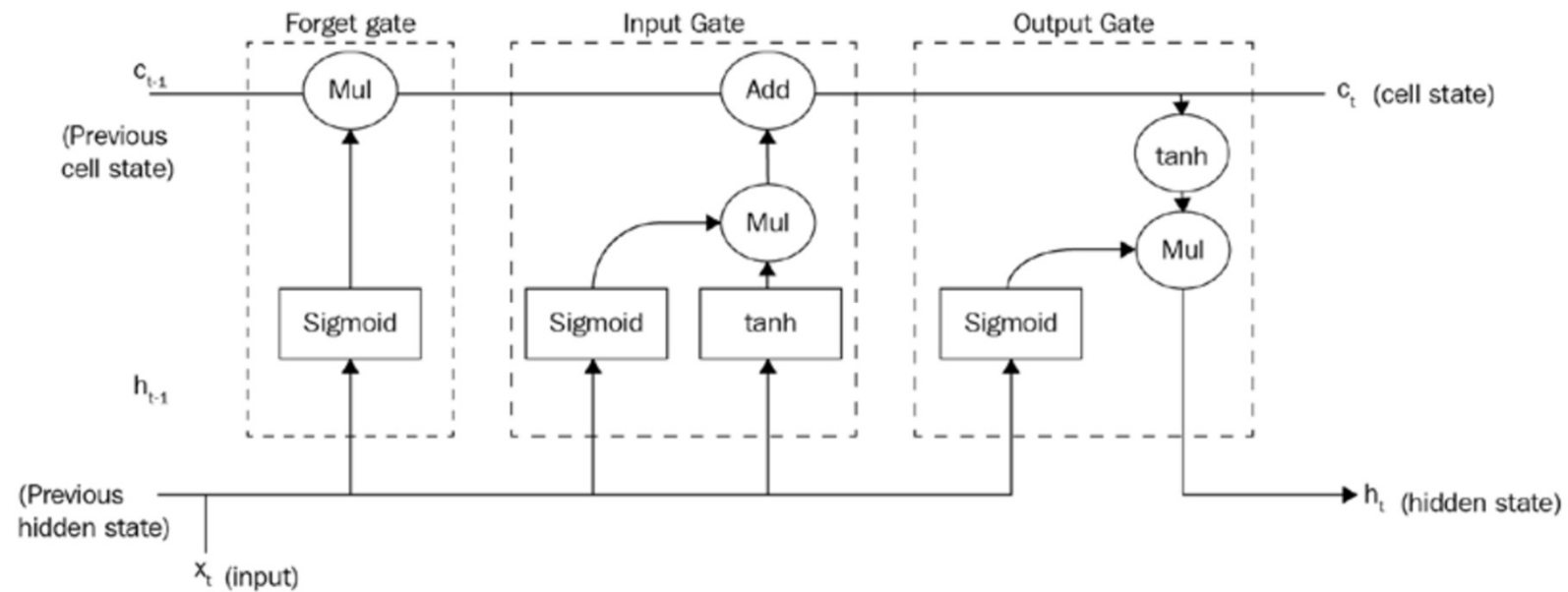
# Long-Term Short-Memory (LSTM) Network

- The LSTM cell



# Long-Term Short-Memory (LSTM) Network

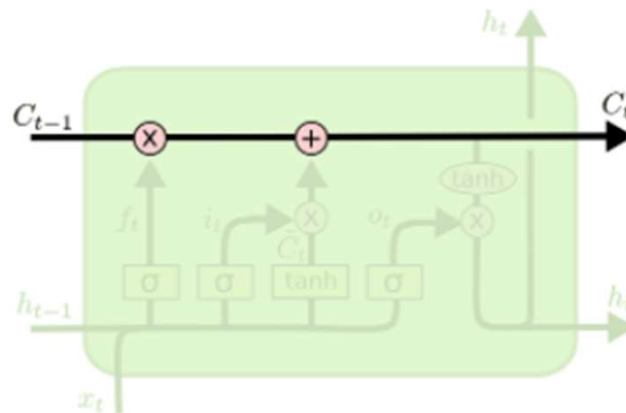
- Internals of LSTM gates





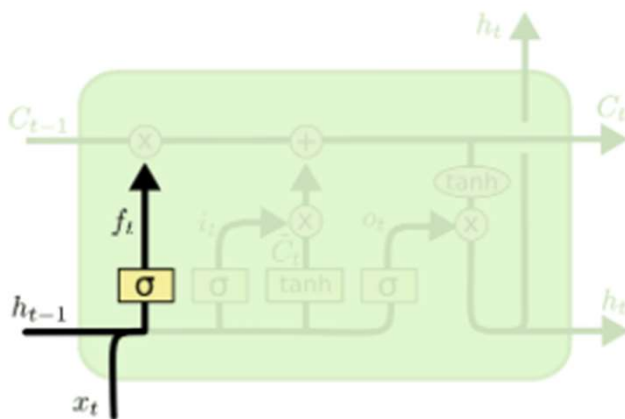
# Long-Term Short-Memory (LSTM) Network

- Cell state
  - LSTM maintains a cell state which acts as a long-term memory



# Long-Term Short-Memory (LSTM) Network

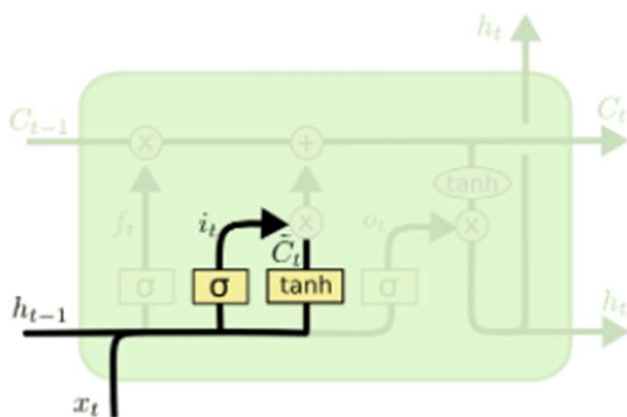
- Forget gate
  - A gate responsible for deciding what to discard from the previous cell state.
  - Suppose we are reading a sentence, and the previous subject was a male.
  - Now, the new input is another subject.
  - At this point, we would like to forget the gender of the previous subject.
  - If  $f_t$  is close to 0, it means "forget everything".
  - If  $f_t$  is close to 1, it means "remember everything".



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

# Long-Term Short-Memory (LSTM) Network

- Input gate
  - A gate responsible for deciding what information should be stored in the memory.
  - Since we have a new subject, we might want to remember the gender of the new subject.
  - The sigmoid line decides which information to update.
  - The tanh line calculates the new cell state  $\tilde{C}_t$ .

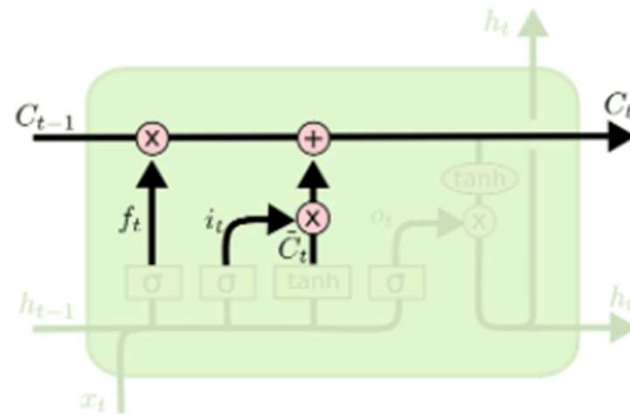


$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

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

# Long-Term Short-Memory (LSTM) Network

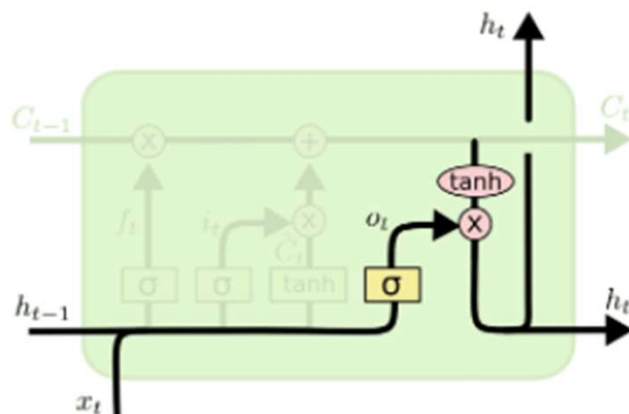
- Updating the cell state
  - We multiple  $f_t$  to the previous cell state  $C_{t-1}$ .
  - Then, we add the calculated cell state  $\tilde{C}_t$  multiplied by the weight  $i_t$  to get the new cell state  $C_t$ .



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

# Long-Term Short-Memory (LSTM) Network

- Output gate
  - A gate responsible for deciding the output of the cell
  - Since the input was a subject, the next word may be a verb.
  - Depending on what it remembers, it will output a proper verb.
  - $\tanh$  is applied to the new cell state  $C_t$ , and is multiplied by the value from the sigmoid line.



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

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

# Convolutional Neural Networks (CNN)

- CNN is a widely used model architecture for various computer vision tasks such as image recognition.
- Suppose we have an image containing a horse.



- When we feed the image to a model, we convert it to a matrix of pixel values.
- The dimension of the matrix is [width x height x channels]
- For color images, number of channels is 3 (RGB).
- We can use a simple ANN architecture for image recognition, but its performance is poor.
  - The model does not consider spatial structure of the image.

# Convolutional Neural Networks (CNN)

- In CNN, we use convolution layers and pooling layers to capture visual features from an image.
- Convolution layers
  - used to extract important features from the image
  - a convolution layer uses a **kernel** or a **filter**, which is used to perform element-wise multiplication on the input.
  - The size of the kernel is **pre-determined**.
  - The values in the kernel are **learnable parameters**.

|   |    |    |
|---|----|----|
| 0 | 13 | 13 |
| 7 | 7  | 7  |
| 9 | 11 | 11 |

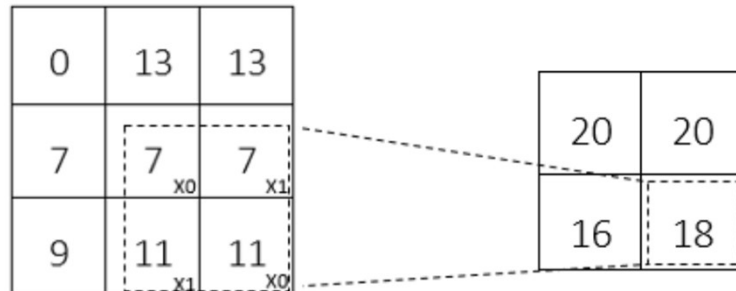
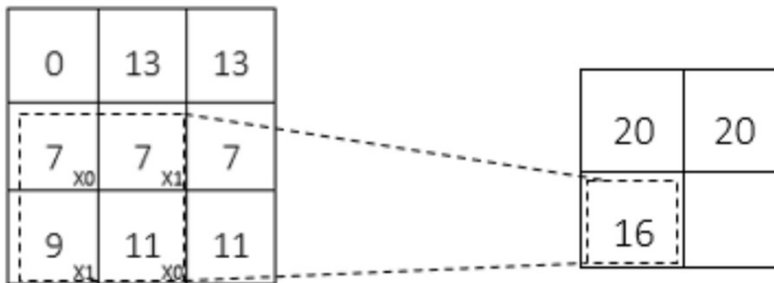
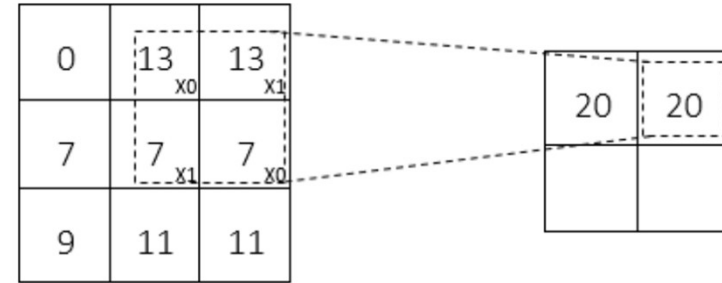
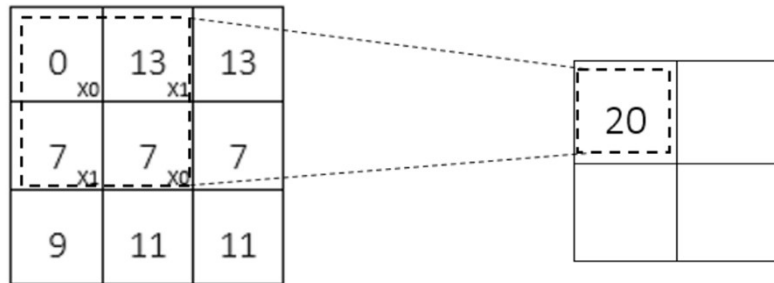
Input Matrix

|   |   |
|---|---|
| 0 | 1 |
| 1 | 0 |

Filter

# Convolutional Neural Networks (CNN)

- Convolution layers
  - When propagating through a convolution layer, we perform convolution operation.



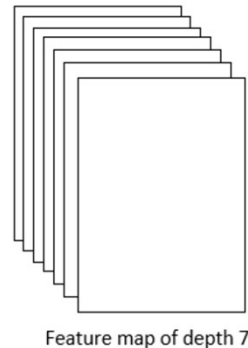


# Convolutional Neural Networks (CNN)

- Convolution layers
  - The output of a convolution layer is called a **feature map** or an **activation map**.
  - The feature map holds important features from the image.

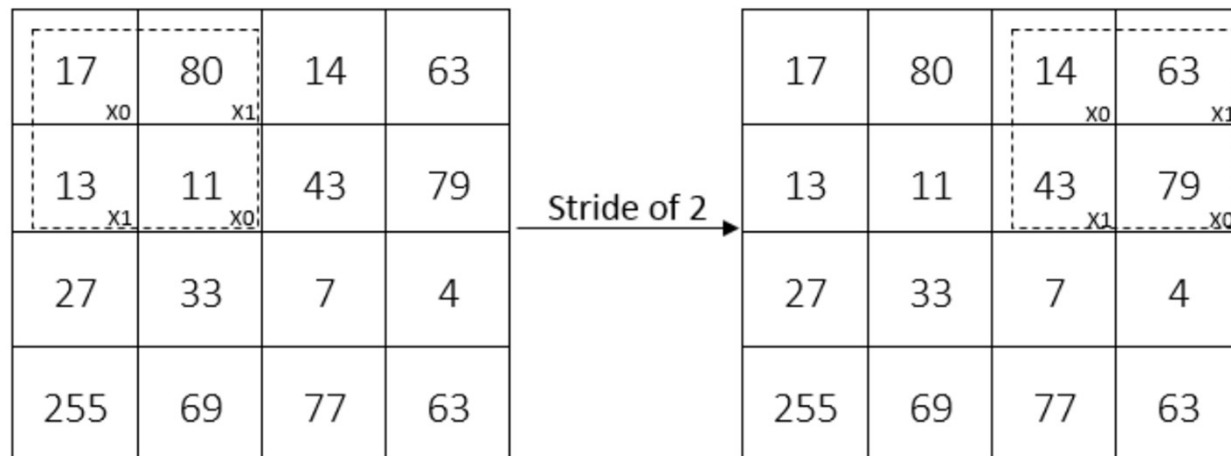


- Instead of calculating a single feature map, we can use multiple filters to create multiple feature maps.



# Convolutional Neural Networks (CNN)

- Strides
  - When doing convolution, the number of pixels we slide over the input matrix is called a **stride**.



# Convolutional Neural Networks (CNN)

- Padding
  - At the border of the image, the kernel may not fit into the image.

|     |    |    |    |  |
|-----|----|----|----|--|
| 17  | 80 | 14 | 63 |  |
| 13  | 11 | 43 | 79 |  |
| 27  | 33 | 7  | 4  |  |
| 255 | 89 | 77 | 63 |  |

Diagram illustrating a 4x4 input grid with a 2x2 kernel (dashed box) applied to the top-right corner. The kernel is labeled  $x_0$  and  $x_1$  for both rows and columns. The values 63 and 79 are shown in the top-right corner of the grid, corresponding to the kernel's position.

- We use **padding** to resolve this boundary case.

# Convolutional Neural Networks (CNN)

- Types of padding
  - zero padding: pad the borders with zero values
  - valid padding: if the kernel does not fit in a region, simply discard the region.

|     |    |    |    |   |
|-----|----|----|----|---|
| 17  | 80 | 14 | 63 | 0 |
| 13  | 11 | 43 | 79 | 0 |
| 27  | 33 | 7  | 4  |   |
| 255 | 89 | 77 | 63 |   |

Diagram illustrating zero padding. A 4x4 input grid is shown. A 2x2 dashed box highlights the top-right corner, with labels  $x_0$  and  $x_1$  indicating the padding values (0) added to the right and bottom of the original 2x2 region.

zero padding

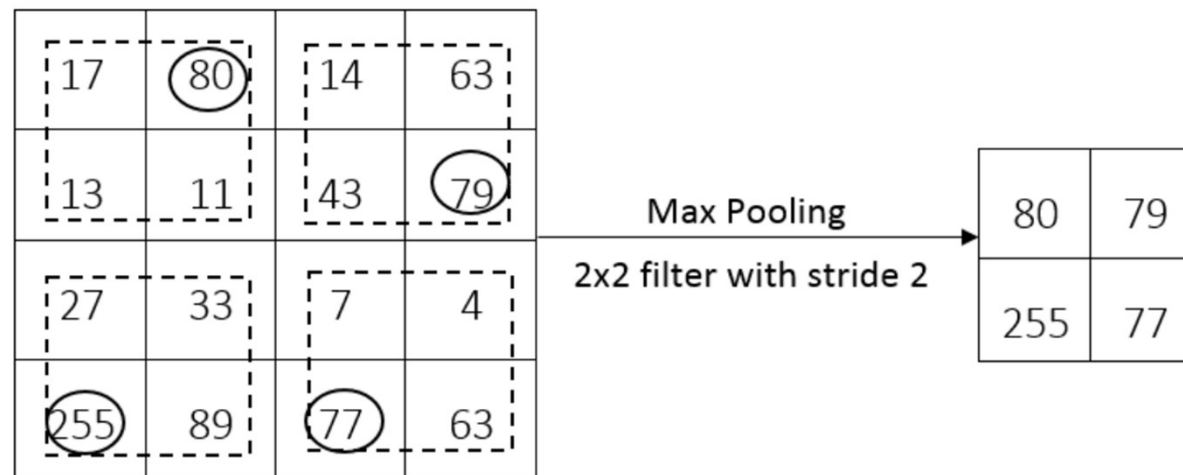
|     |    |    |    |  |
|-----|----|----|----|--|
| 17  | 80 | 14 | 63 |  |
| 13  | 11 | 43 | 79 |  |
| 27  | 33 | 7  | 4  |  |
| 255 | 89 | 77 | 63 |  |

Diagram illustrating valid padding. A 4x4 input grid is shown. A 2x2 dashed box highlights the top-right corner, with labels  $x_0$  and  $x_1$  indicating the padding values (0) added to the right and bottom of the original 2x2 region. The padding is not applied, and the region is discarded.

valid padding

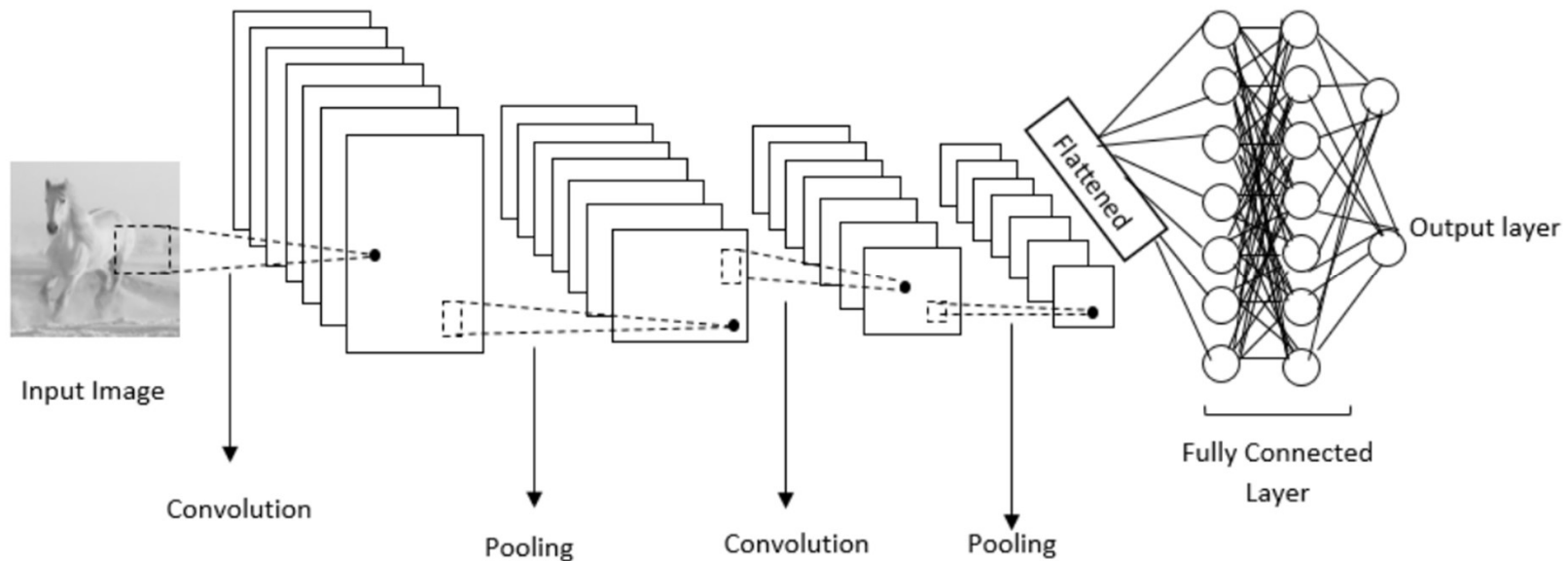
# Convolutional Neural Networks (CNN)

- Pooling layers
  - After convolution, we often use pooling layers to **downsample** an image to a smaller image.
  - In a max pooling, we choose the maximum value from a certain region and use it as a representative value for that region.
  - In an average pooling, we average the values from a region and use that value.
  - If we use a 2x2 pooling with stride 2, the resulting image is halved in both width and height.



# Convolutional Neural Networks (CNN)

- Fully-connected layers
  - The convolution and pooling layers extract features from the image and produce feature maps.
  - We still need to classify images based on the feature maps.
  - The fully-connected layers is used to classify images.
  - This layer is basically a layer in the ANN.



# Generative Adversarial Networks (GAN)

- Generative adversarial networks are networks used to generate new data points.
- It is used to generate realistic human faces, converting grayscale images to colored images, translating text descriptions to realistic images, and so on.



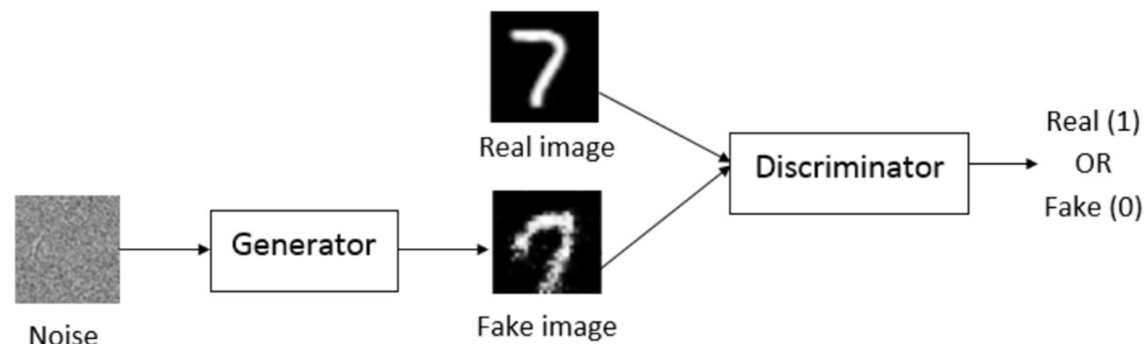
# Generative Adversarial Networks (GAN)

- Training a GAN is essentially playing a two-player game where one tries to defeat the other.
- There is a counterfeiter who tries to create fake money.
- There is a police who tries to find fake money.
- The goal of the counterfeiter is to create a very realistic fake money so that it cannot be differentiated from the real money.
- The goal of the police is to discriminate the fake money from the real money.



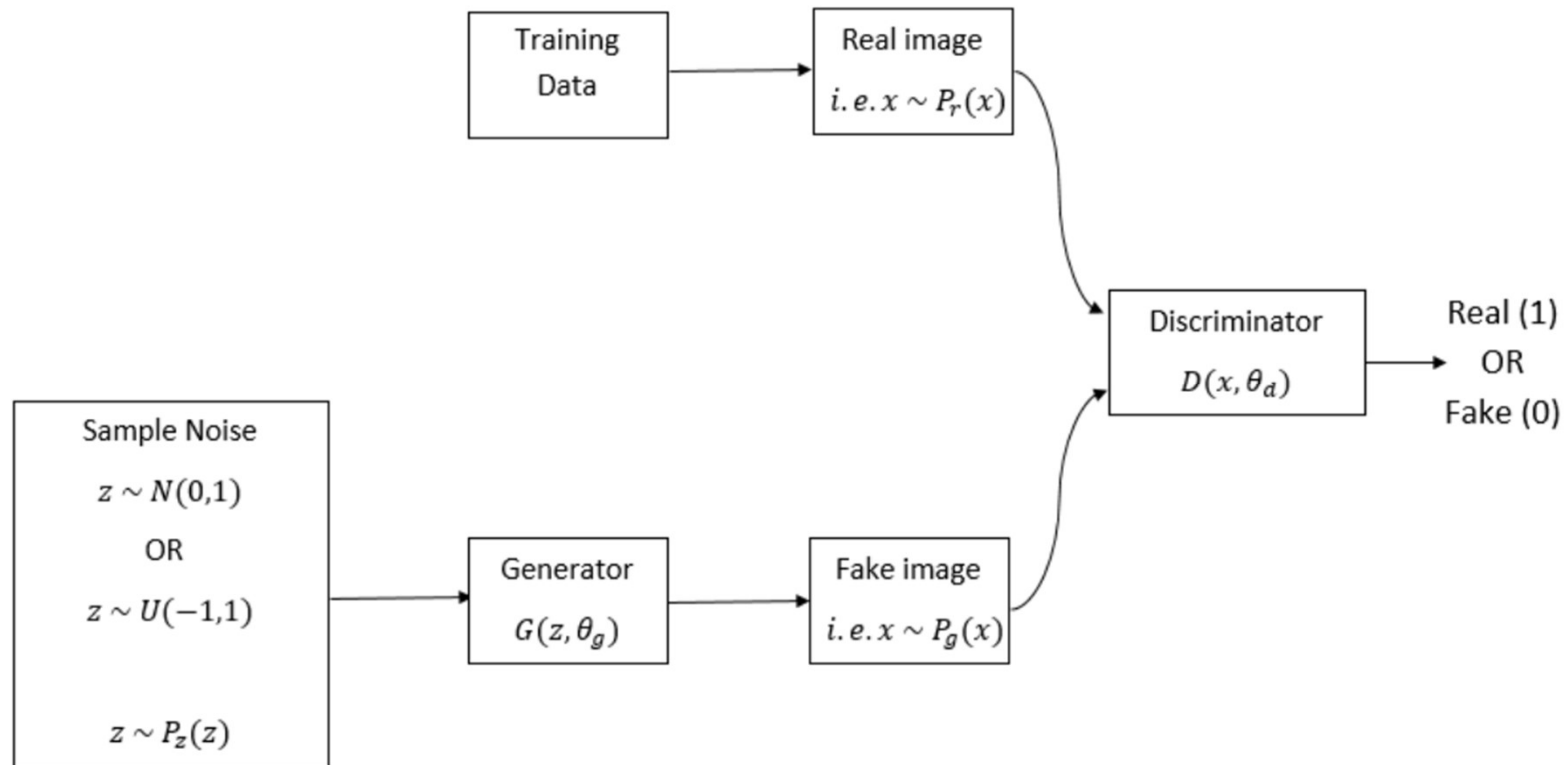
# Generative Adversarial Networks (GAN)

- A GAN consists of two components
  - Generator (counterfeiter)
  - Discriminator (police)
- Generator
  - The generator learns the **distribution of images** in the dataset.
  - Once it learns the distribution, it can convert a random noise into a new image of the learned distribution
- Discriminator
  - The discriminator performs a classification task with two classes: real and fake.
  - A "real" image is an original image in the dataset, and a "fake" image is one created by the generator.



# Generative Adversarial Networks (GAN)

- Architecture of a GAN



# Generative Adversarial Networks (GAN)

- Terms in the architecture
  - $G(z; \theta_g)$ : the generator
  - $z$ : input to the generator; a random noise
  - $P_z$ : distribution of the random noise
  - $N(0,1)$ : a Gaussian distribution with mean 0 and std 1.
  - $U(-1,1)$ : a Uniform distribution from -1 to 1.
  - $\theta_g$ : parameters of the generator network
  - $P_g$ : distribution of samples created by the generator
  - $P_r$ : distribution of the real dataset
  - $D(x; \theta_d)$ : the discriminator
  - $x$ : input to the discriminator; an image
  - $\theta_d$ : parameters of the discriminator network

# Generative Adversarial Networks (GAN)

- Loss functions
  - discriminator loss

$$\max_d L(D, G) = \mathbb{E}_{x \sim P_r(x)} [\log D(x; \theta_d)] + \mathbb{E}_{z \sim P_z(z)} \left[ \log \left( 1 - D(G(z; \theta_g); \theta_d) \right) \right]$$

- first term: log-likelihood that the discriminator will say "real" to the real images.
- second term: log-likelihood that the discriminator will say "fake" to the generated images.

- generator loss

$$\min_g L(D, G) = \mathbb{E}_{z \sim P_z(z)} \left[ \log \left( 1 - D(G(z; \theta_g); \theta_d) \right) \right]$$

- first term: log-likelihood that the discriminator will correctly determine the fake images.

# Generative Adversarial Networks (GAN)

- Total loss

$$\min_G \max_D L(D, G) = \mathbb{E}_{x \sim P_r(x)} [\log D(x)] + \mathbb{E}_{z \sim P_z(z)} [\log (1 - D(G(z)))]$$

- Our objective function is a min-max objective function.
- a maximization for the discriminator and a minimization for the generator

- Gradient Descent

- maximization on the discriminator

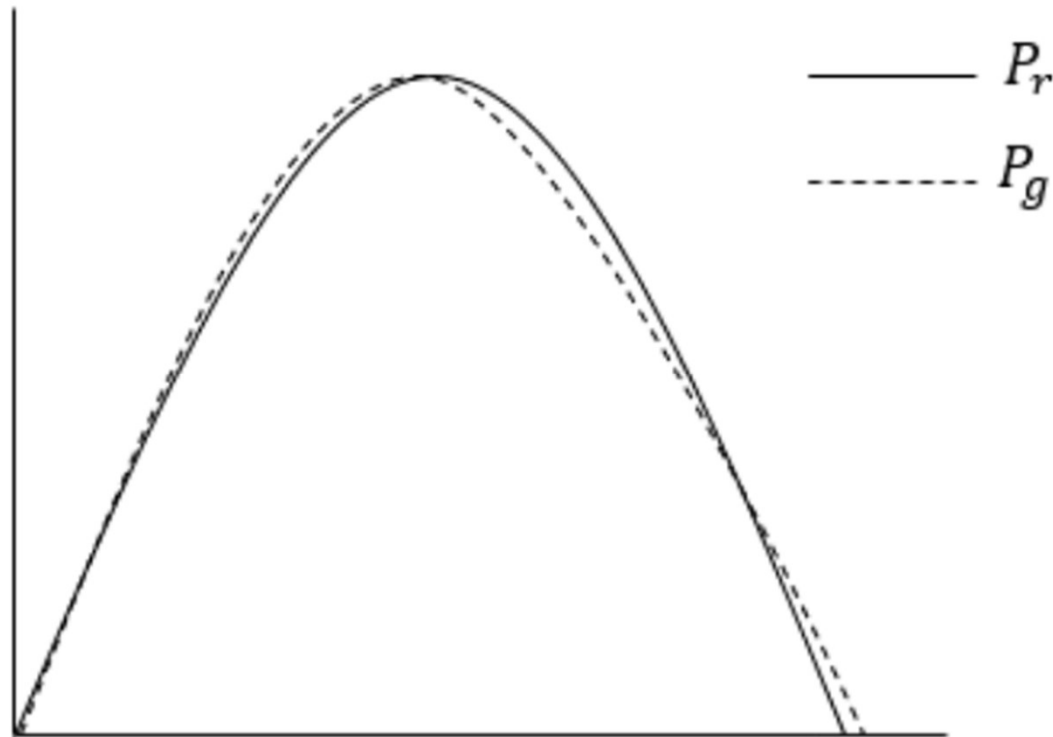
$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m [\log D(x^{(i)}) + \log (1 - D(G(z^{(i)})))]$$

- minimization on the generator

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log (1 - D(G(z^{(i)})))$$

# Generative Adversarial Networks (GAN)

- Goal of training
  - Generator learns the real data distribution, so that  $P_g$  and  $P_r$  become very similar.



## End of Chapter

- Do you understand the concept of deep learning as well as algorithms such as backpropagation?
- Do you understand various methods of deep learning such as RNN, CNN, and GAN?
- Can you write [ex015] yourself?

End of Class

---

Questions?

Email: [jso1@sogang.ac.kr](mailto:jso1@sogang.ac.kr)