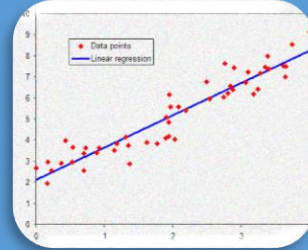


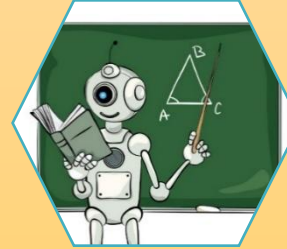
Road map!

- Module 1- Introduction to
- Module 2- Setting up Dee
- Module 3- Exponential S
- Module 4- ARIMA mode
- Module 5- Machine Learning
- **Module 6- Deep Neural Networks**
- Module 7- Deep Sequence Modeling (RNN, LSTM)
- Module 8- Prophet and Neural Prophet

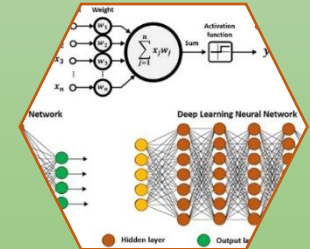
Econometrics



Machine Learning



Deep Learning

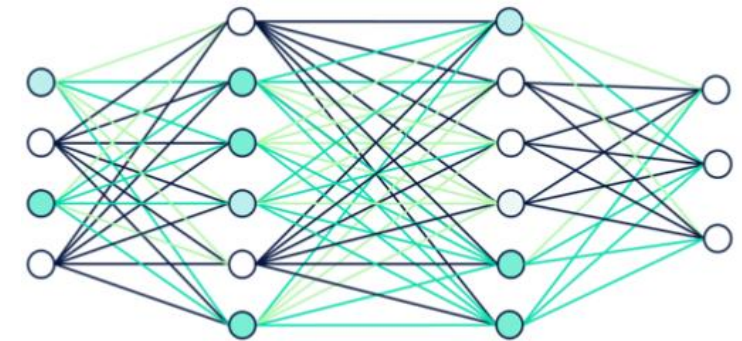
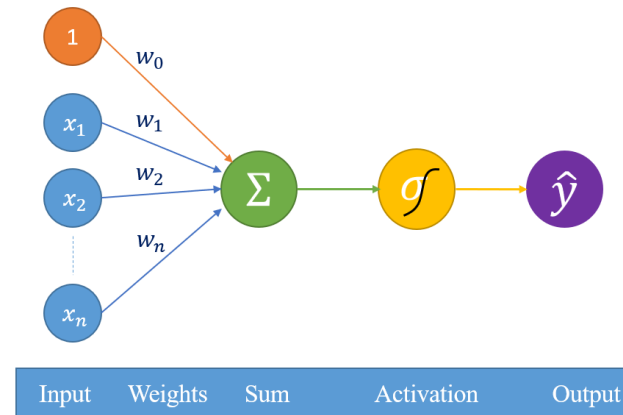
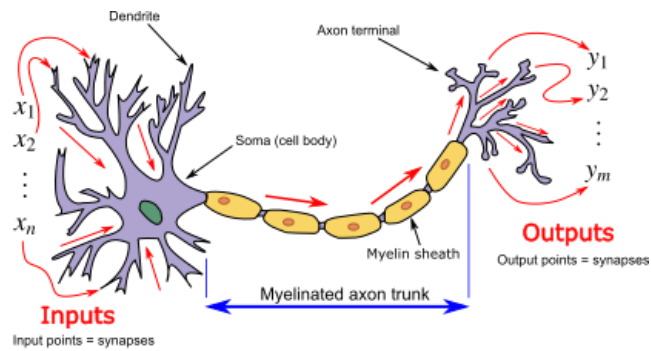


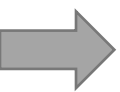
Deep Forecasting

Module 6 - Part I

Deep Neural Networks

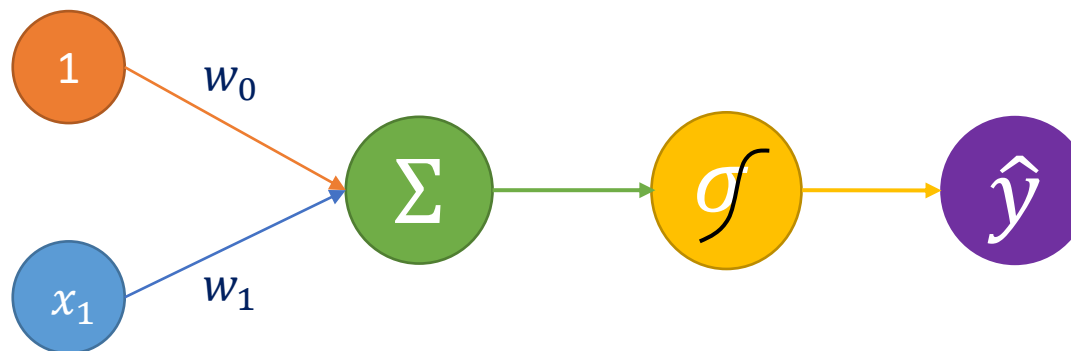
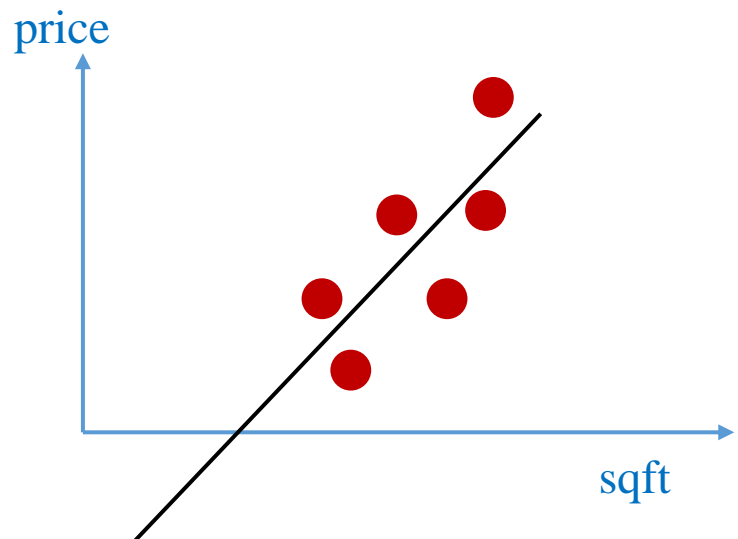
Basics





A simple example!

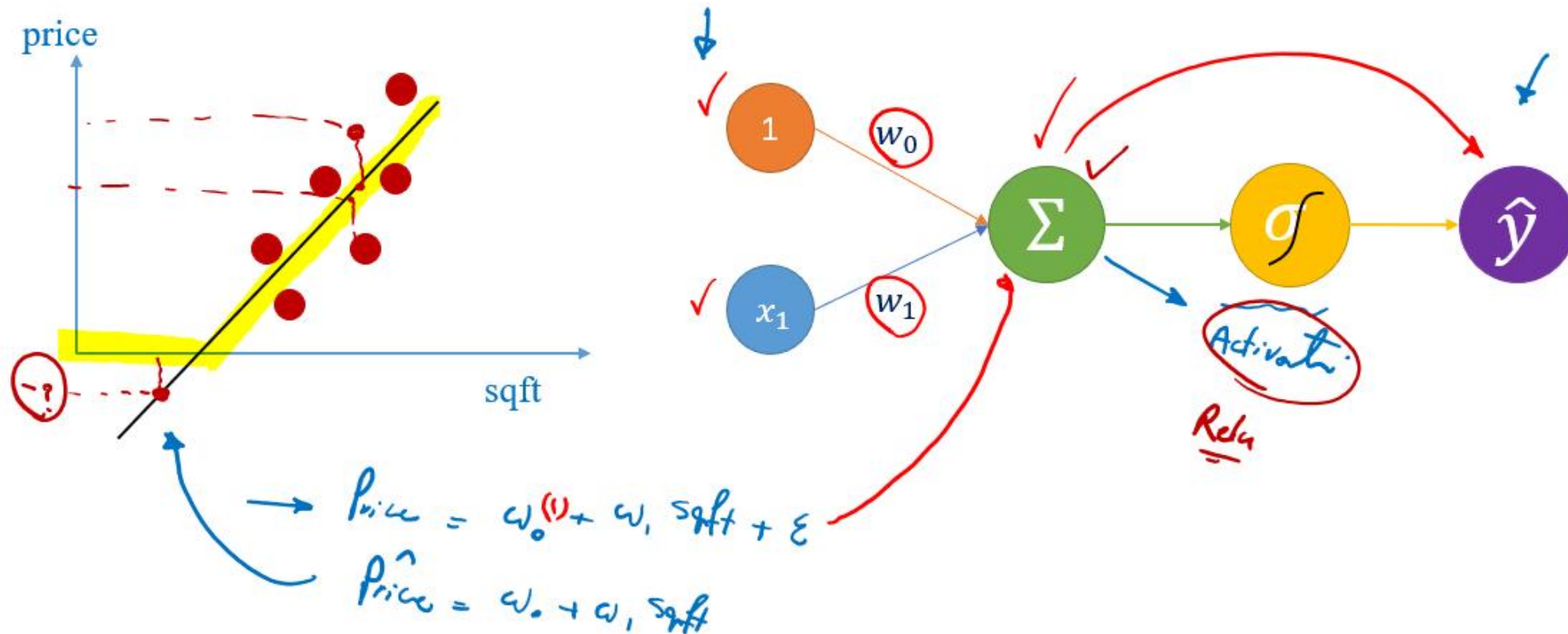
- Consider the housing price prediction! $price = f(sqft)$





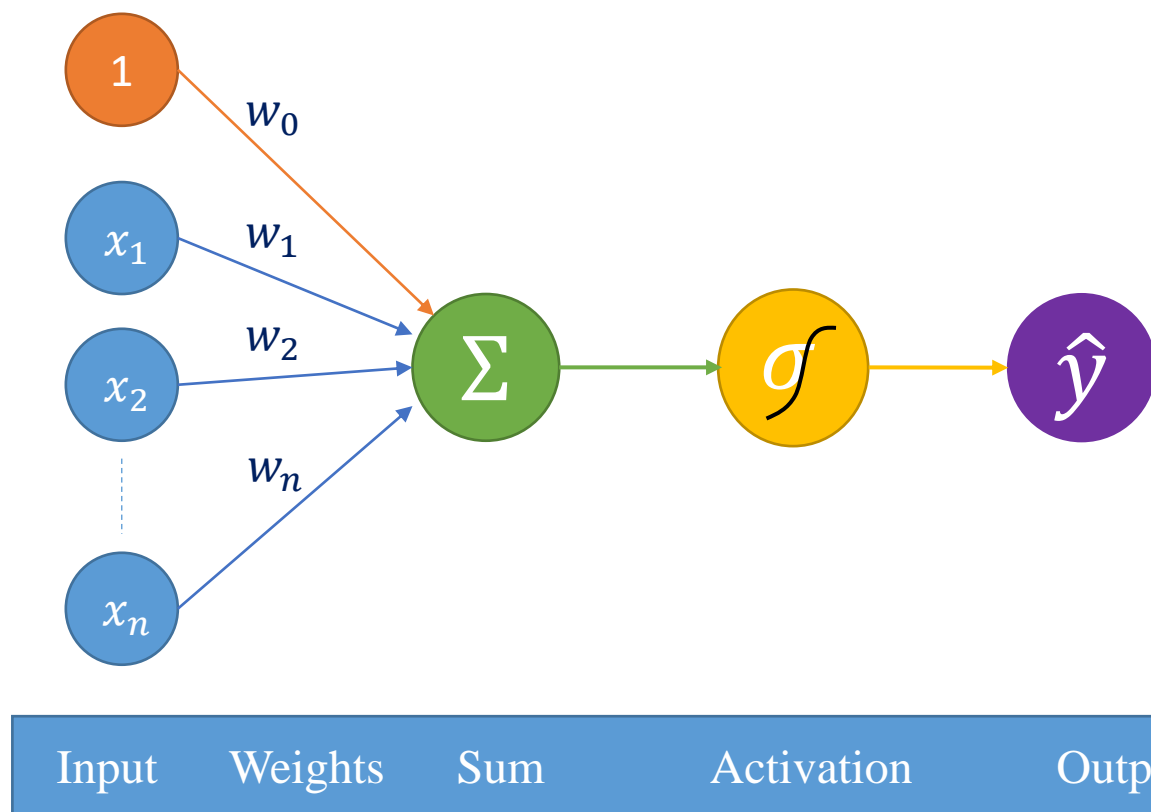
A simple example!

- Consider the housing price prediction! $\text{price} = f(\text{sqft}) + \varepsilon$



→ Forward Propagation

- Forward propagation is the process of calculating the output of a neural network, given an input.
- w_0 is the **bias** term which allows shifting Σ to the left or right.



➔ What are Neurons?

- Neurons, are the simplest elements or building blocks in a neural network. They are inspired by biological neurons that are found in the human brain.
- **What happens inside the neurons?**

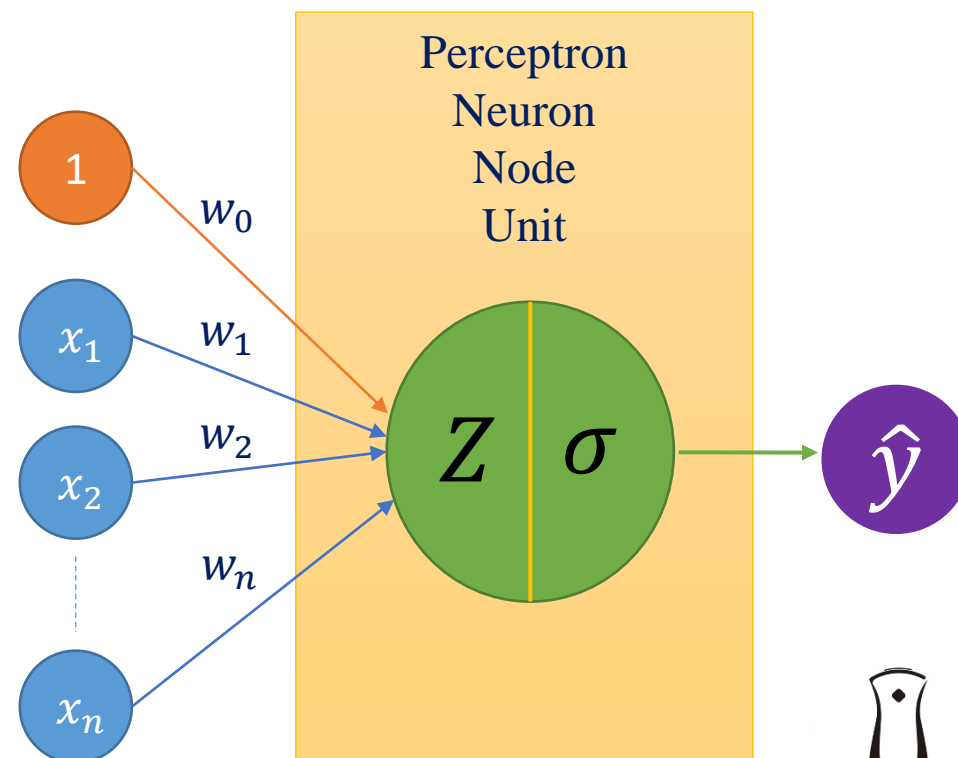
Activation function

Bias

$$\hat{y} = \sigma(w_0 + \sum_{i=1}^n w_i x_i)$$

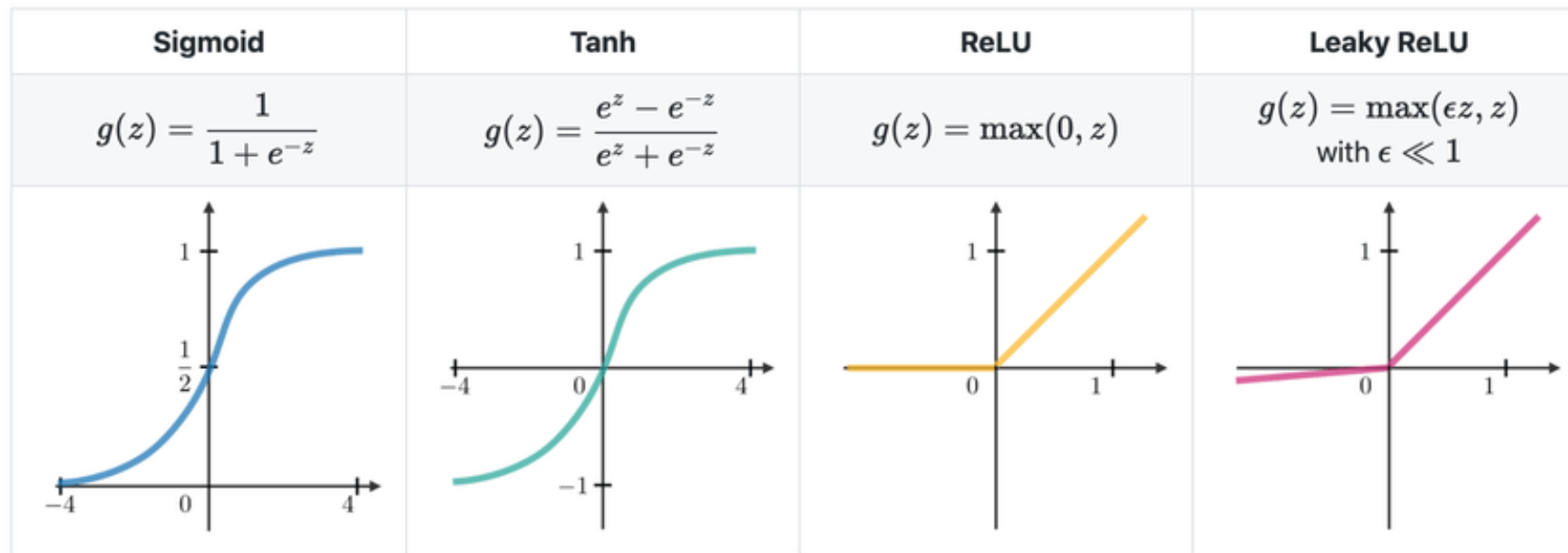
$$\hat{y} = \sigma(w_0 + W^T X)$$

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



→ Activation function

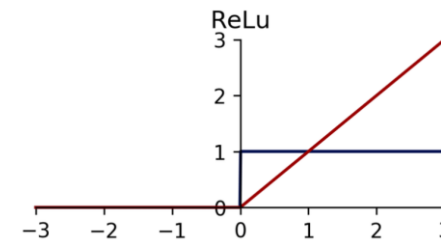
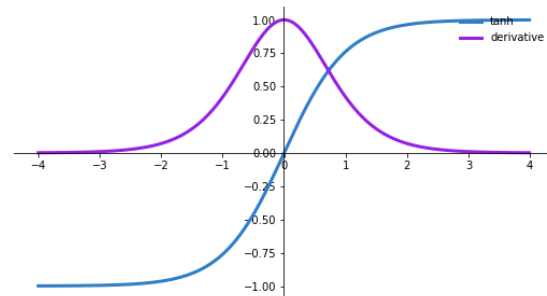
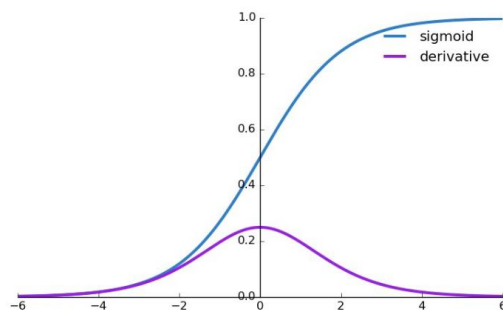
- An activation function is a mathematical function that is applied to the output of each neuron in a neural network
- Activation functions allow the network to learn **non-linearities** and **complex patterns** from the data and make accurate predictions



➔ Which activation function?

Again, there is no “right” answer! However:

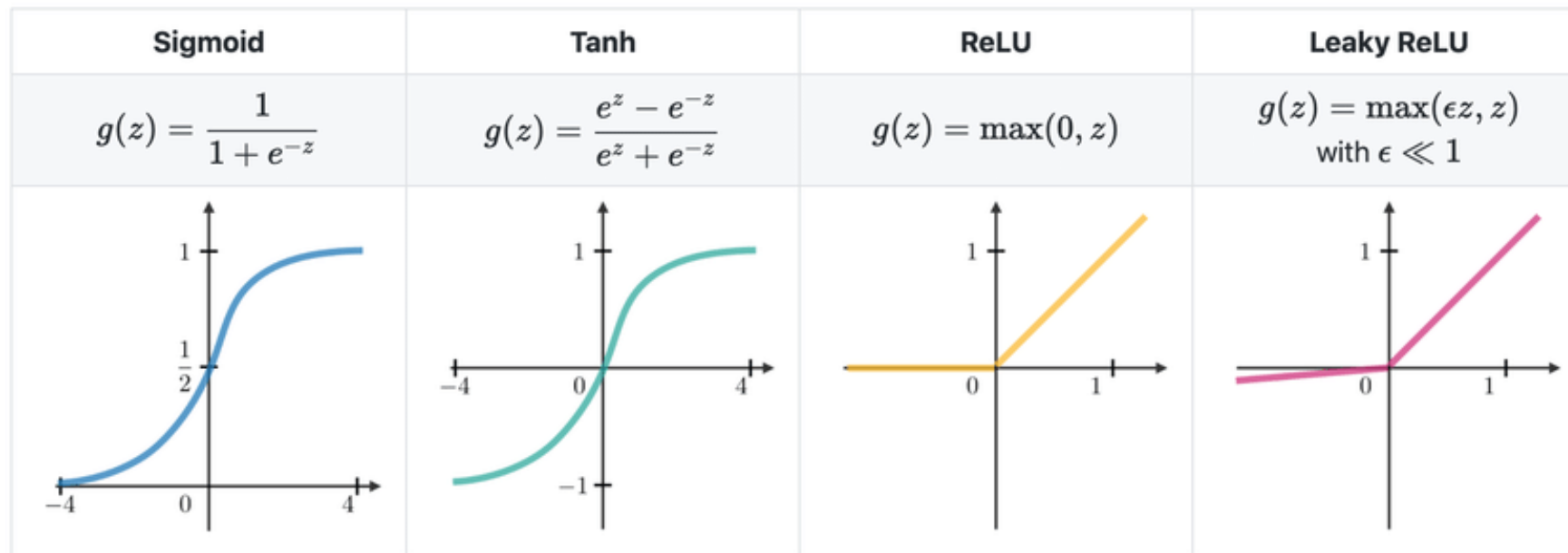
- For the hidden layers, almost always **tanh** is better than **sigmoid** because it center the data for the next layer.
- One downside of both sigmoid or tanh is that the gradient is very small for extreme values.

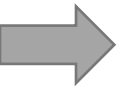


- In practice, RELU works **better/faster** than sigmoid or tanh for hidden layers.
- Leaky RELU might work better than RELU, but if we have enough number of hidden layers, RELU is just fine.
- Sigmoid is mostly used for output layer!

→ Activation function

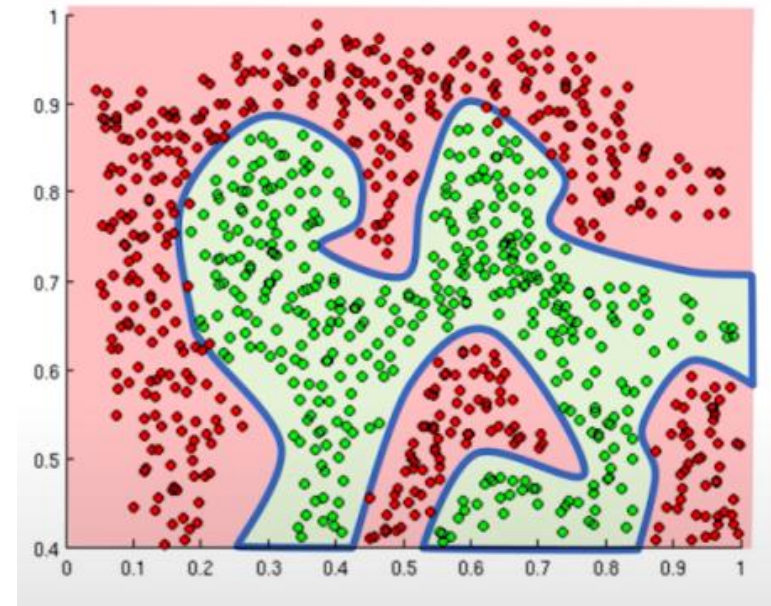
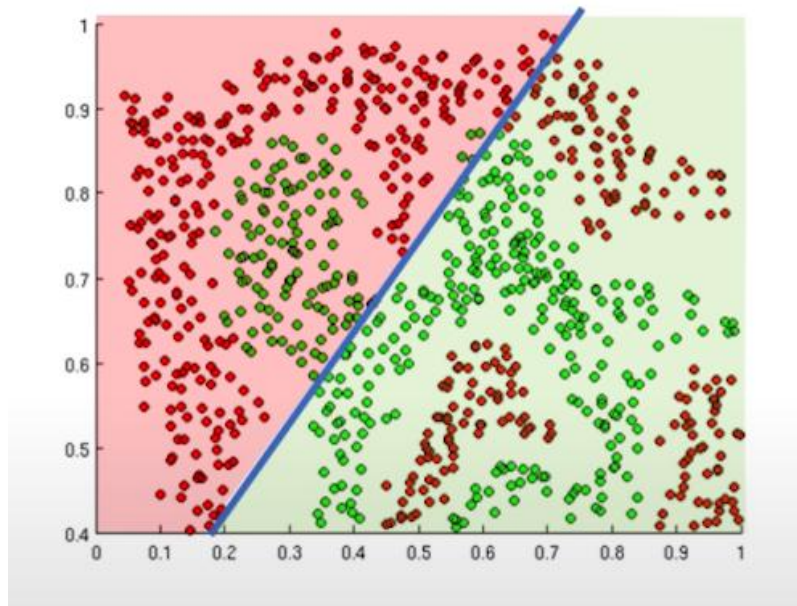
- An activation function is a mathematical function that is applied to the output of each neuron in a neural network
- Activation functions allow the network to learn **non-linearities** and **complex patterns** from the data and make accurate predictions

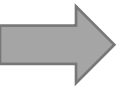




Why do we need non-linear activation functions?

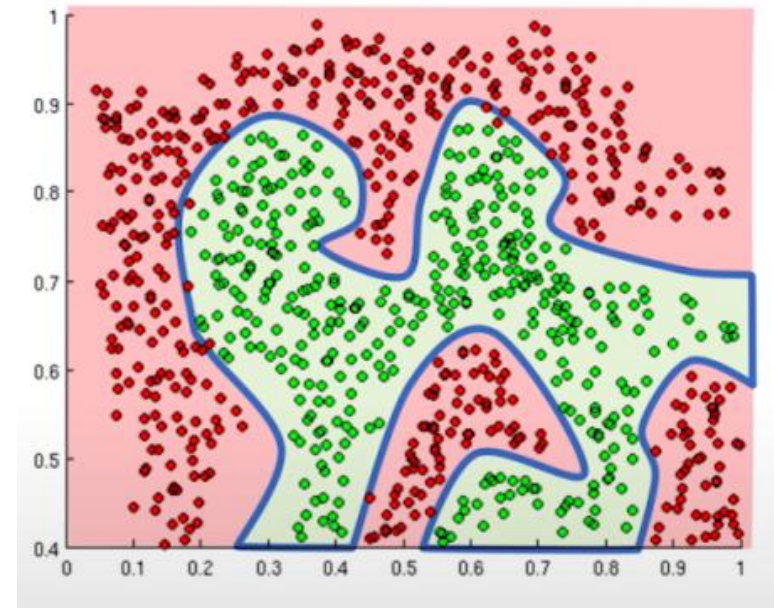
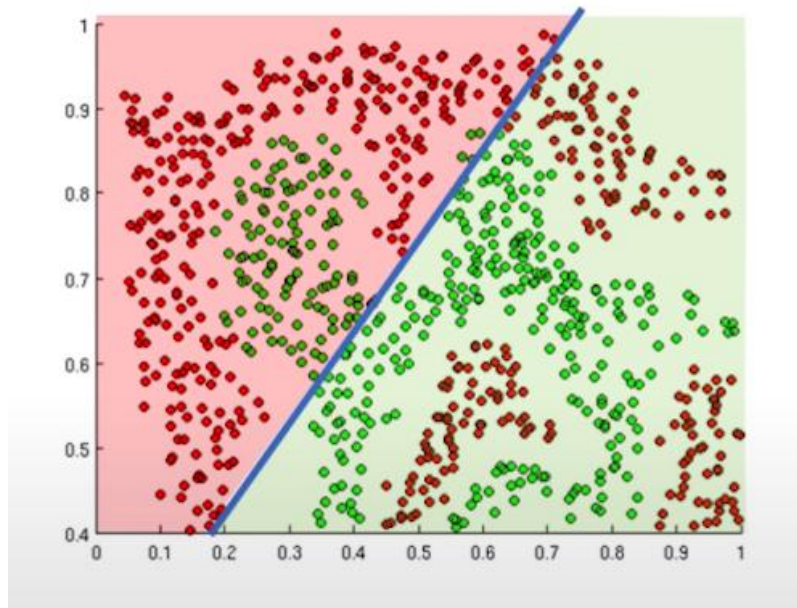
- If we use **linear** activation function $\sigma(z) = z = w_0 + W^T X$, then the NN will **always output a linear function of the inputs** regardless of the number of neurons or hidden layers used.
- Linear activation functions produce linear decision boundaries!
- Activation functions allow the network to approximate **complex patterns**!





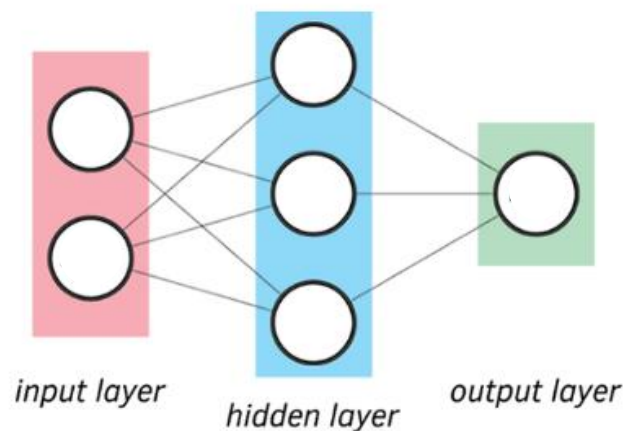
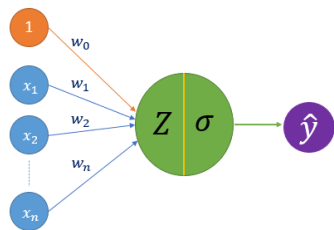
Why do we need non-linear activation functions?

- If we use **linear** activation function $\sigma(z) = z = w_0 + W^T X$, then the NN will **always output a linear function of the inputs** regardless of the number of neurons or hidden layers used.
- Linear activation functions produce linear decision boundaries!
- Activation functions allow the network to approximate **complex patterns**!



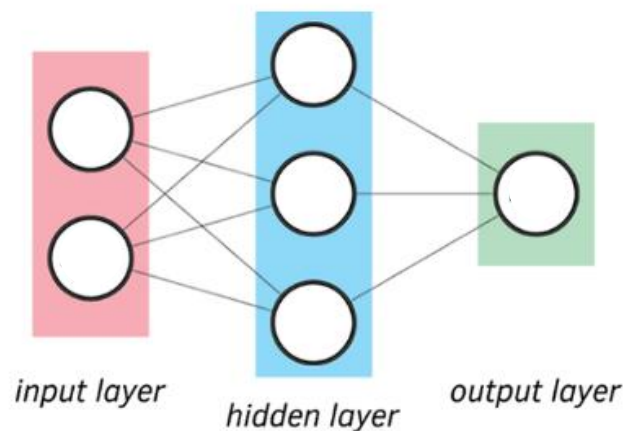
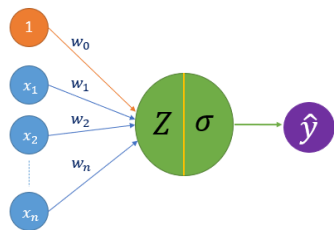
➔ Building a Neural Network

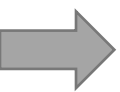
- A neural network is a type of machine learning model that is composed of **layers** of interconnected "**neurons**" which process and transmit information.
- Each neuron receives input from other neurons, processes it using a **nonlinear activation function**, and then transmits the output to other neurons in the **next layer**. The output of the **final layer** is the **prediction** made by the neural network.
- Because all the inputs are **densely** connected to all outputs, these layers are called **Dense** layers.
- The output layer can be either **single** output or **multiple** output.



➔ Building a Neural Network

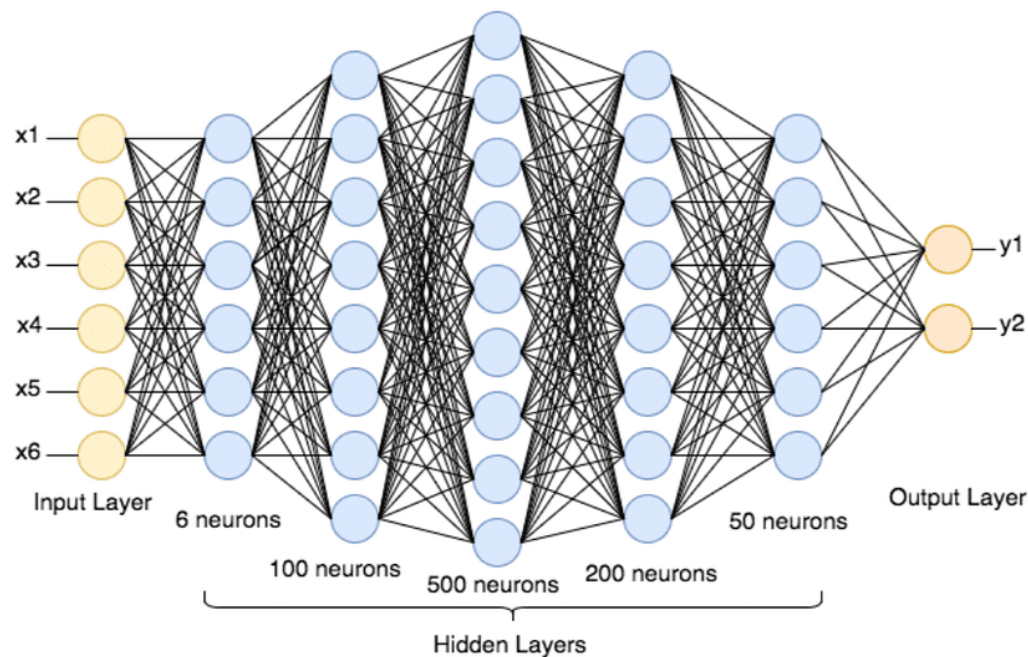
- A neural network is a type of machine learning model that is composed of **layers** of interconnected "**neurons**" which process and transmit information.
- Each neuron receives input from other neurons, processes it using a **nonlinear activation function**, and then transmits the output to other neurons in the **next layer**. The output of the **final layer** is the **prediction** made by the neural network.
- Because all the inputs are **densely** connected to all outputs, these layers are called **Dense** layers.
- The output layer can be either **single** output or **multiple** output.





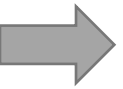
Building a Deep Neural Network

- Deep neural networks are neural networks **with a large number of layers**, typically consisting of multiple hidden layers.
- Deep neural networks can learn and **model very complex patterns** in data.
- DNNs have been successful in a wide range of applications, including image and speech recognition, natural language processing, and machine translation.



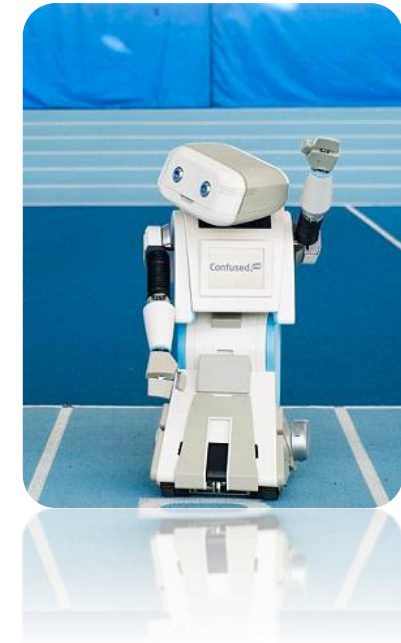
```
import tensorflow as tf

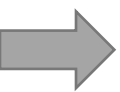
# Define the model architecture
model=tf.keras.Sequential([
    tf.keras.layers.Dense(6 , activation='relu'),
    tf.keras.layers.Dense(100, activation='relu'),
    tf.keras.layers.Dense(500, activation='relu'),
    tf.keras.layers.Dense(200, activation='relu'),
    tf.keras.layers.Dense(50 , activation='relu'),
    tf.keras.layers.Dense(2 , activation='sigmoid'),
])
```



Training a Neural Network, Backpropagation

- To train a neural network, we present it with many examples and **adjust the weights and biases** of the connections between neurons so that the network can accurately predict the output for each example and minimize **the loss function**.
- This process is known as **backpropagation**, and it is done using an optimization algorithm such as stochastic gradient descent.
- The **loss function** quantifies the distance between actuals and predictions. It provides **feedback** to the NN.
- Examples: MSE, Binary Cross Entropy, (Sparse) Categorical Cross Entropy, ...
- For the full list, visit <https://keras.io/api/losses/#available-losses>





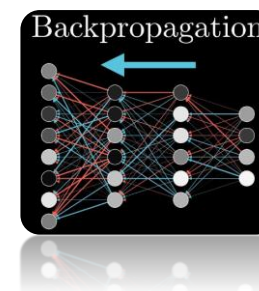
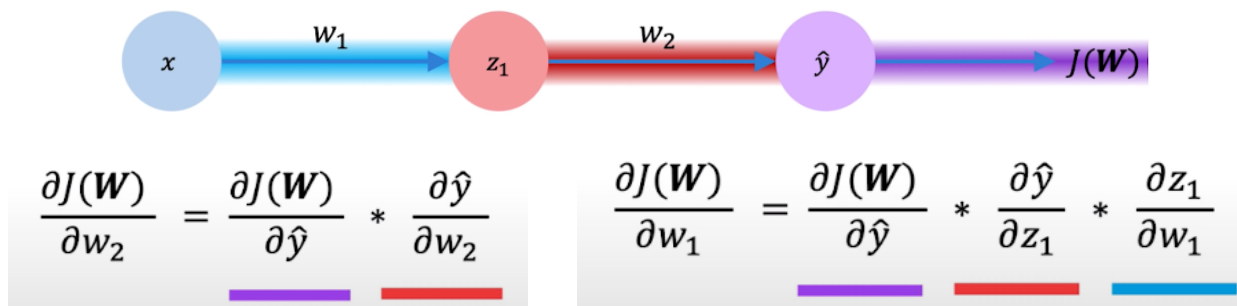
Backpropagation

- Backpropagation is a **supervised learning algorithm** that adjusts the weights and biases of the connections between neurons in the network to minimize **the loss function**.

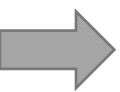
$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} \frac{1}{n} \sum_{i=1}^n \mathcal{L}(f(x^{(i)}; \mathbf{W}), y^{(i)})$$

- Steps:

- 1) Feed the input data through the neural network to compute the output.
- 2) Calculate the **loss** or **error** between the predicted output and the true output.
- 3) **Propagate the error back** through the network using the **chain rule** of calculus to calculate the gradient of the loss function with respect to the weights and biases.
- 4) **Update the weights** and biases using the gradient descent algorithm.
- 5) Repeat the process until the loss reaches a satisfactory level or a predetermined iterations.



Pedram Jahangiry

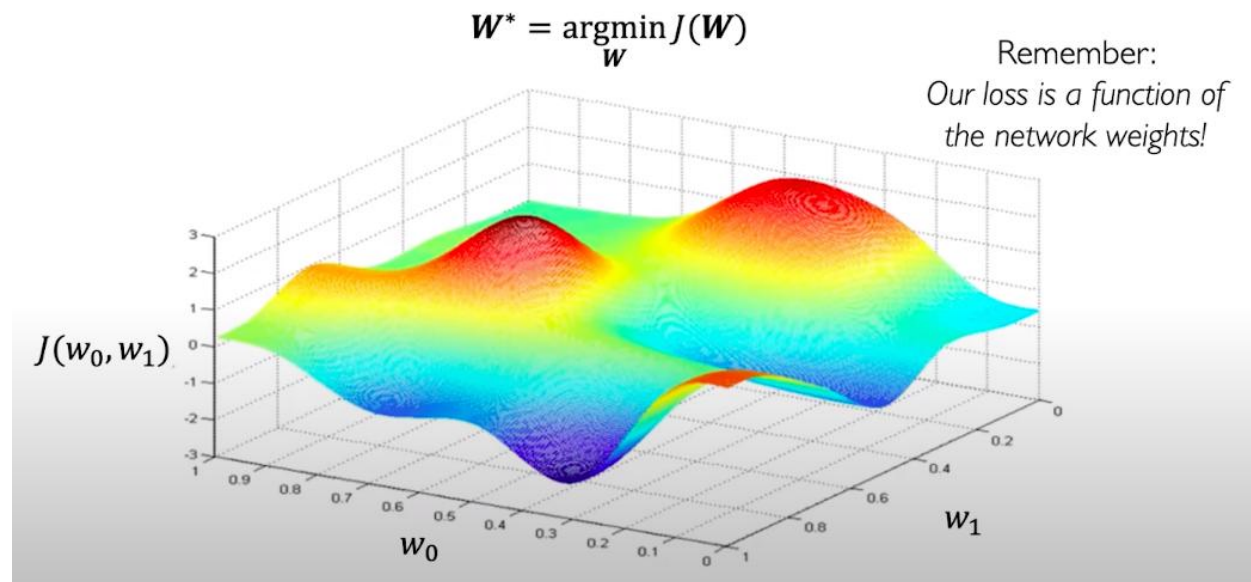


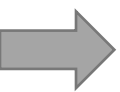
Loss optimization

$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} \frac{1}{n} \sum_{i=1}^n \mathcal{L}(f(x^{(i)}; \mathbf{W}), y^{(i)})$$

Gradient Descent

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
4. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
5. Return weights





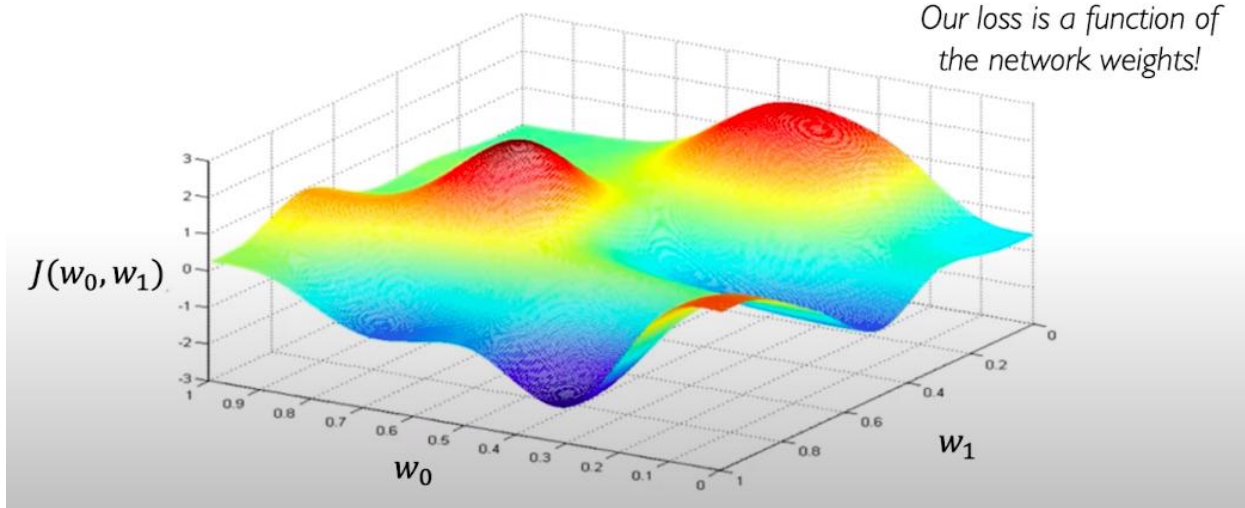
Loss optimization

$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} \frac{1}{n} \sum_{i=1}^n \mathcal{L}(f(x^{(i)}; \mathbf{W}), y^{(i)})$$

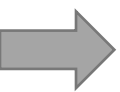
Stochastic GD

$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} J(\mathbf{W})$$

Remember:
Our loss is a function of
the network weights!



1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J_i(\mathbf{W})}{\partial \mathbf{W}}$
4. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
5. Return weights

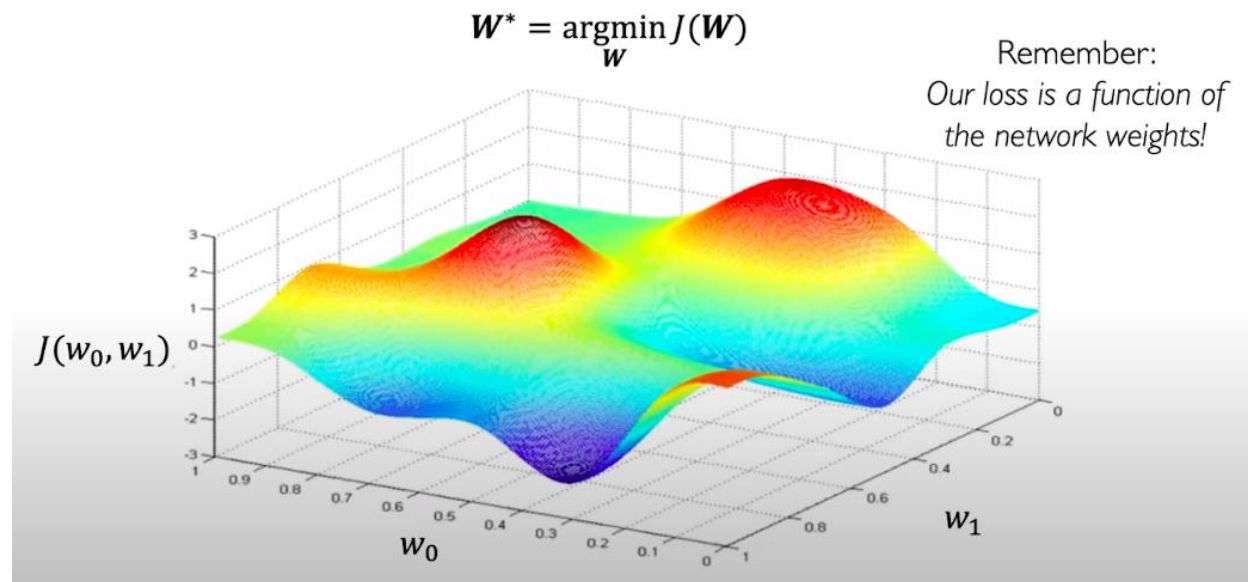


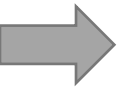
Loss optimization

$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} \frac{1}{n} \sum_{i=1}^n \mathcal{L}(f(x^{(i)}; \mathbf{W}), y^{(i)})$$

Mini-batch GD

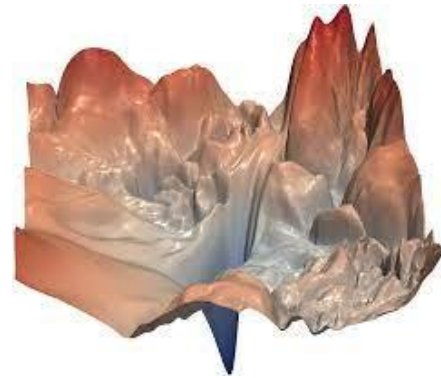
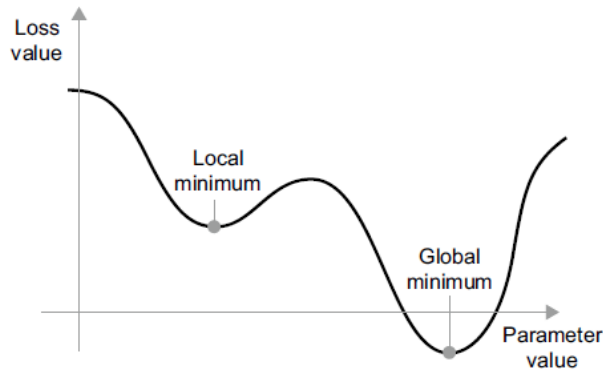
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} = \frac{1}{B} \sum_{k=1}^B \frac{\partial J_k(\mathbf{W})}{\partial \mathbf{W}}$
4. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
5. Return weights





DNN loss functions can be difficult to optimize!

- *Visualizing the loss landscape of neural nets*, Li et al, 2018



- **Solution:** Designing an adaptive learning rate that can **adapt** to the loss landscape. Rather than just looking at **the current gradient**, take into account the **previous weight updates**. This is called, **momentum**!
- Examples: Adam, Adadelata, Adagrad, RMSProp!

➔ Gradient Descent Algorithms

- Available optimizers in Keras:

- SGD
- RMSprop
- Adam
- AdamW
- Adadelta
- Adagrad
- Adamax
- Adafactor
- Nadam
- Ftrl



```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential()
model.add(layers.Dense(64, kernel_initializer='uniform', input_shape=(10,)))
model.add(layers.Activation('softmax'))

opt = keras.optimizers.Adam(learning_rate=0.01)
model.compile(loss='categorical_crossentropy', optimizer=opt)
```

```
lr_schedule = keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate=1e-2,
    decay_steps=10000,
    decay_rate=0.9)

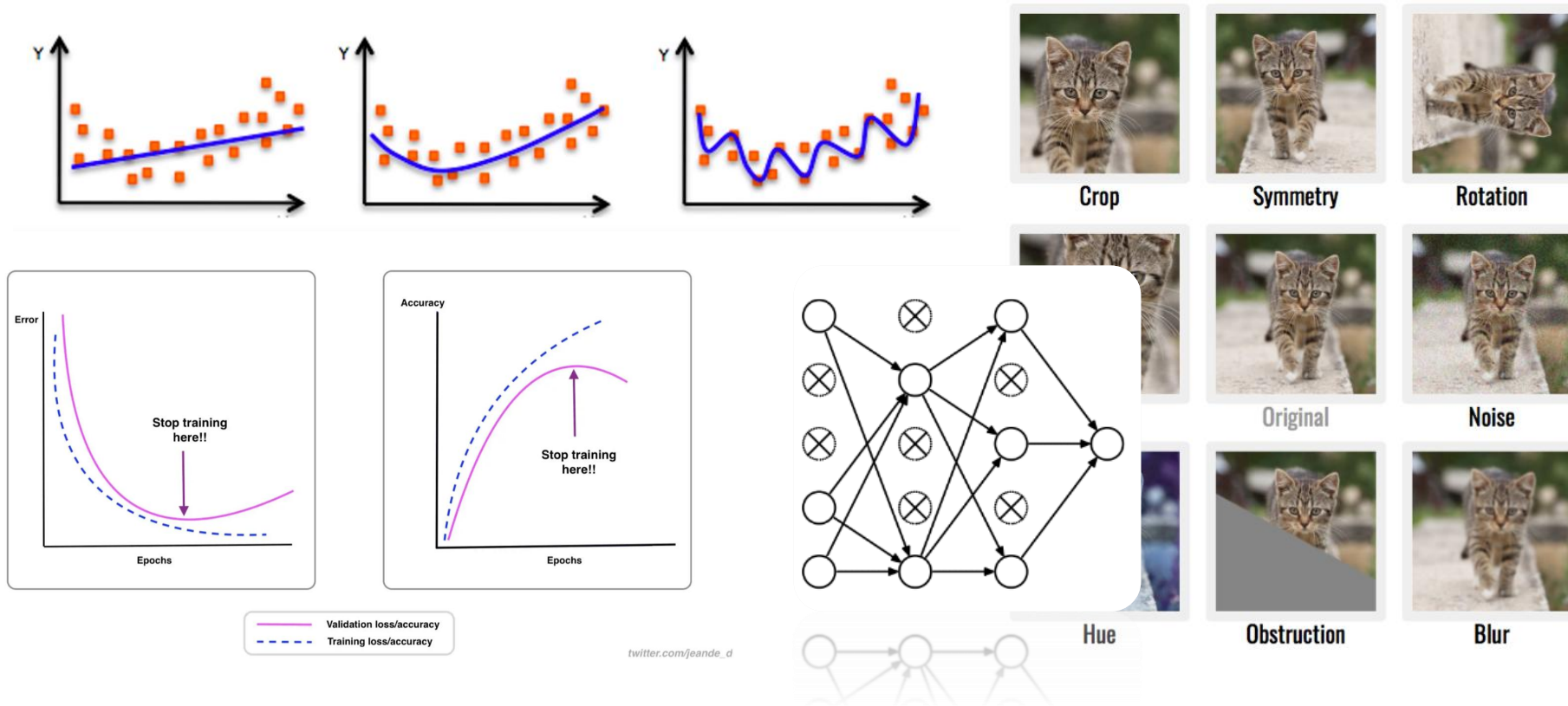
optimizer = keras.optimizers.SGD(learning_rate=lr_schedule)
```

- Which optimizer? There is no “right” answer.

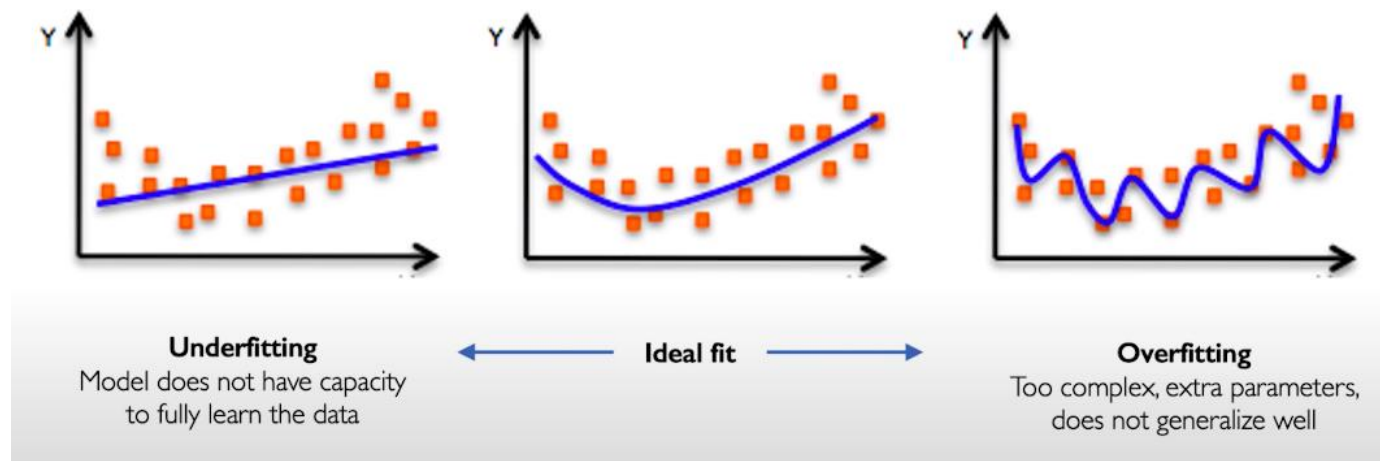
Module 6 - Part II

Deep Neural Networks

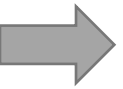
Regularization



➔ How to handle overfitting in DNN?



- Regularization refers to a set of techniques that are used to prevent overfitting by **discouraging complex models**.
- Regularization improve **generalization** of our model on **unseen data**.
- Methods include, L1, L2, drop out, Early stopping and Data augmentation.



Deep Learning Regularization techniques

L1 regularization: This technique adds a penalty term to the objective function that is proportional to the absolute value of the model weights. This results in a sparse model, with many weights being set to zero.

L2 regularization: This technique adds a penalty term to the objective function that is proportional to the square of the model weights. This results in a model with small, non-zero weights.

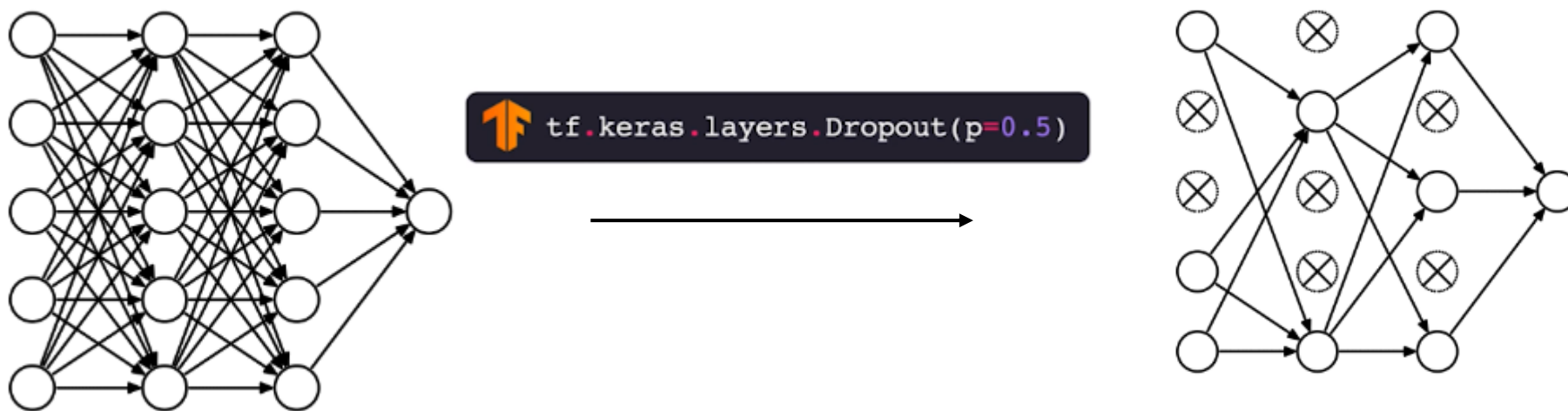
Dropout: This technique randomly sets a fraction of the model weights to zero during training, which helps to prevent overfitting

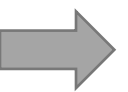
Early Stopping: This technique involves training the model until the performance on a validation set begins to degrade, and then stopping the training at that point. This helps to prevent the model from continuing to fit the training data too closely.

Data Augmentation: This technique involves generating additional training examples by applying random transformations to the existing training data. This can help to prevent overfitting by providing the model with a more diverse training set.

➔ Dropout Regularization

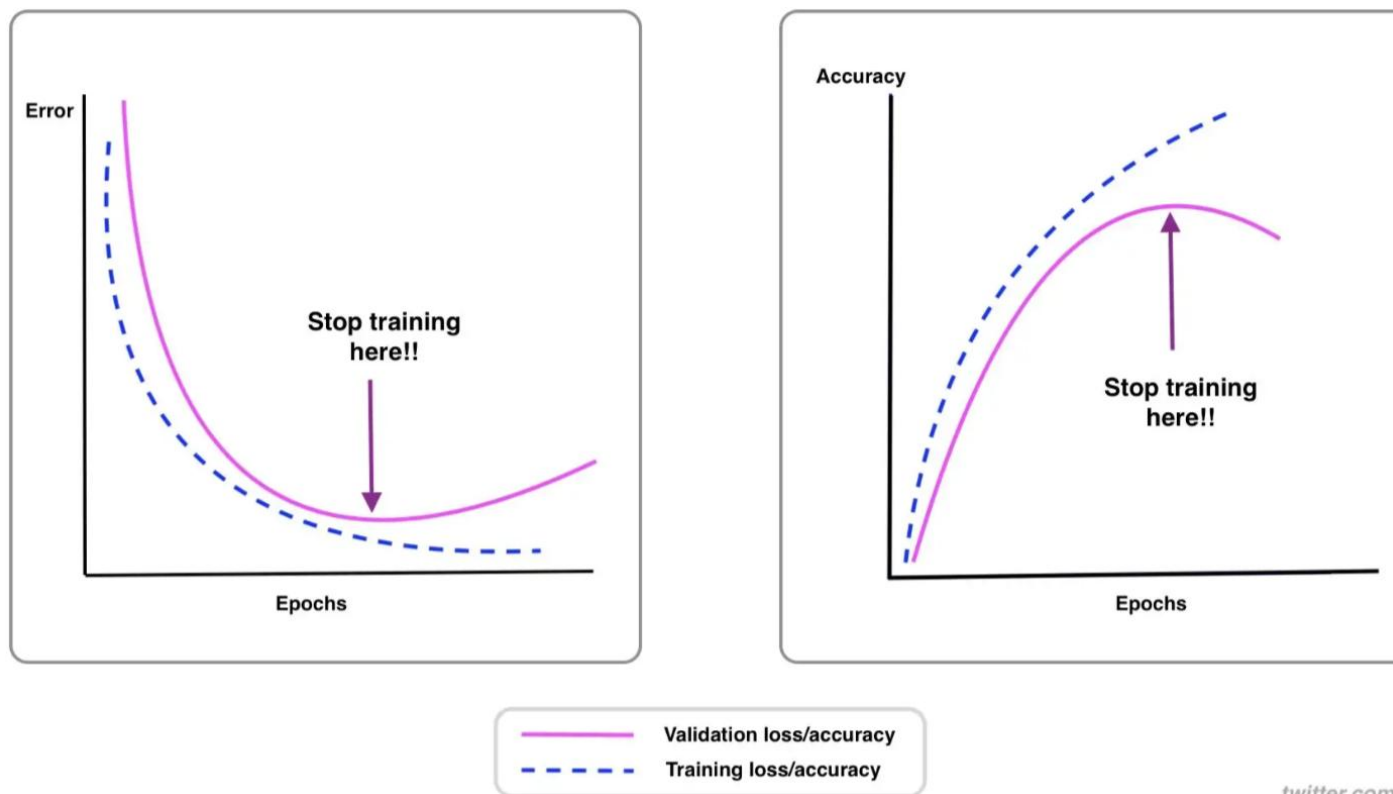
- Dropout is one of the most popular regularization techniques in deep learning.
- At each training iteration, dropout randomly chooses **different nodes to ignore**
- This prevents the network from **relying on any single neuron**.





Early Stopping

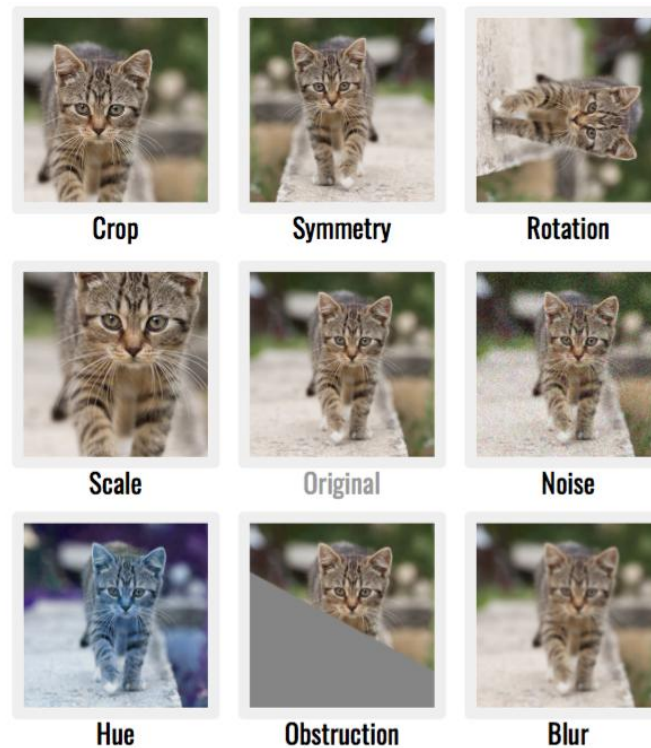
- Stop training when the performance on a validation set begins to degrade!

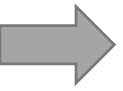


twitter.com/jeande_d

➔ Data Augmentation

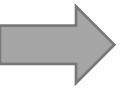
- Generating additional training examples by applying **random transformations** to the existing training data
- The goal of data augmentation is to improve the **generalizability** and robustness of machine learning models by providing them with **more diverse training data**.



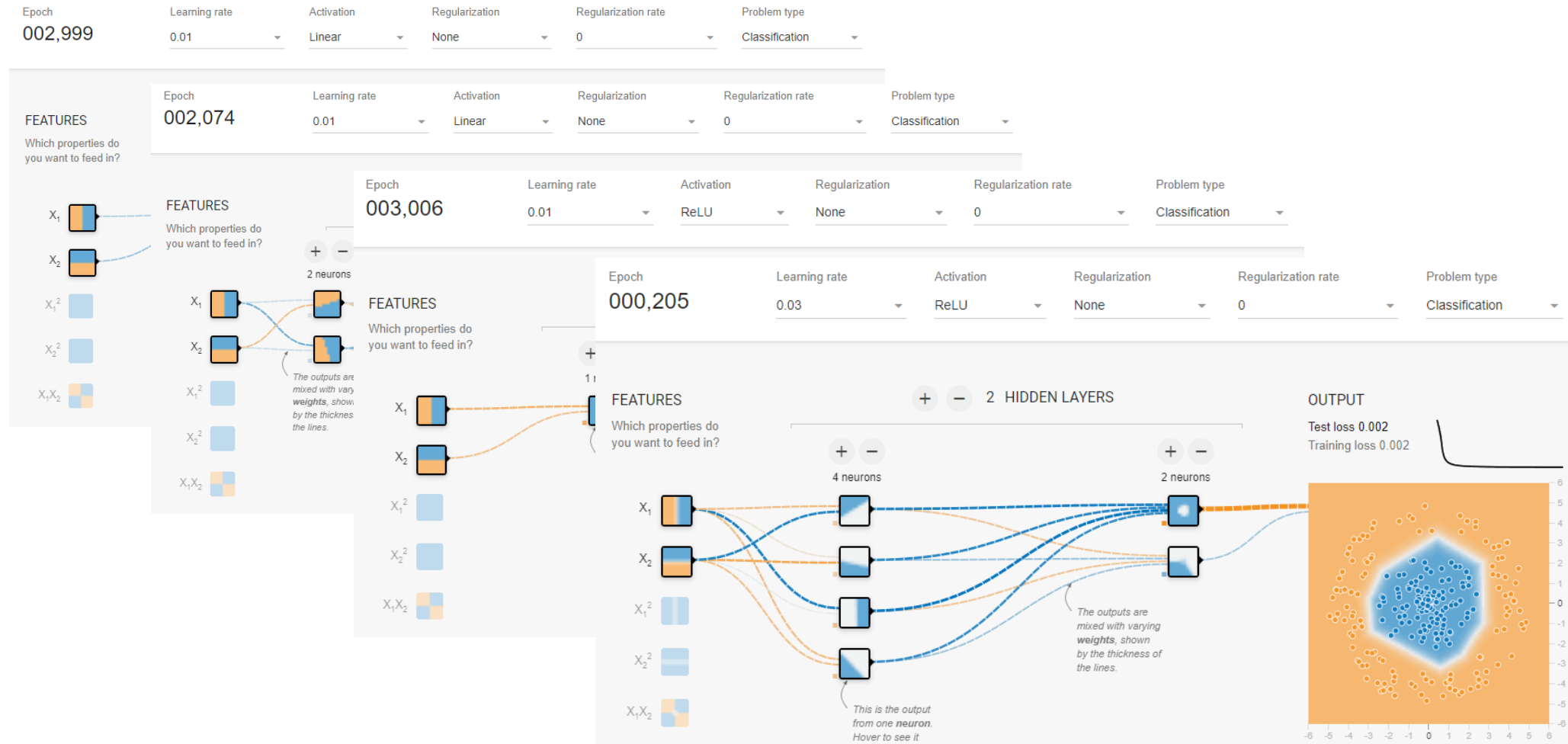


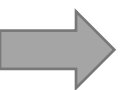
Data Augmentation for timeseries forecasting

- Data augmentation is a crucial technique in time series forecasting, enhancing model robustness and performance by **expanding the training dataset**.
- **Bootstrapping**: Resample the original time series data with replacement to create new samples, preserving inherent patterns:
 - Moving Block Bootstrap
 - Circular Block Bootstrap
 - Stationary Bootstrap
- **Generative Adversarial Networks** (GANs): Generate synthetic time series data that mimic the statistical properties of the original dataset.




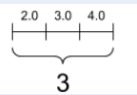
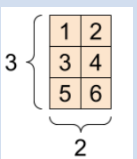
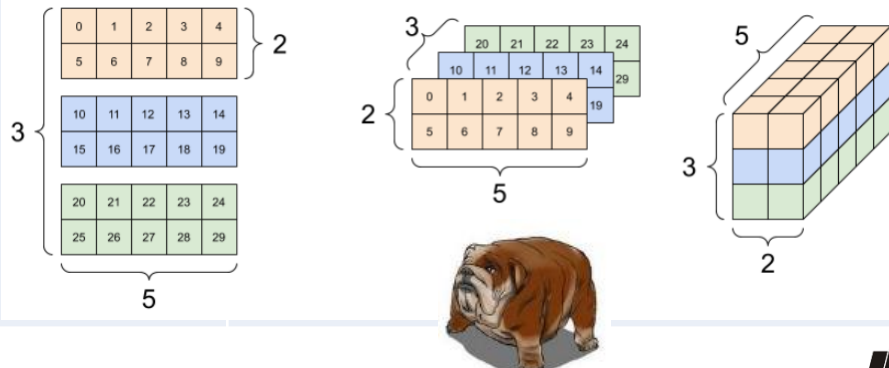
Neural Network Playground (TensorFlow)





What is a Tensor?

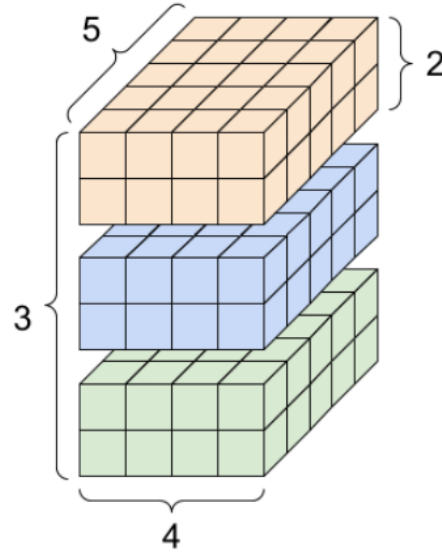
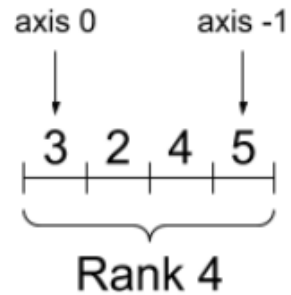
- Tensor: A **multi-dimensional** array (kind of like np.arrays)
- Rank**: Number of tensor axes , **Size**: The total number of items in the tensor

Tensor	tf code	shape	Example
Scalar (Rank-0 tensor)	<code>tf.constant(4)</code>	Shape = ()	
Vector (Rank-1 tensor)	<code>tf.constant([2.0, 3.0, 4.0])</code>	Shape = (3 ,)	
Matrix (Rank-2 tensor)	<code>tf.constant([[1, 2],[3, 4],[5, 6]])</code>	Shape = (3 , 2)	
Rank-3 tensor	<code>tf.constant([[0, 1, 2, 3, 4], [5, 6, 7, 8, 9], [10, 11, 12, 13, 14], [15, 16, 17, 18, 19], [20, 21, 22, 23, 24], [25, 26, 27, 28, 29]])</code>	Shape = (3, 2, 5)	

Higher dimension Tensors

```
rank_4_tensor = tf.zeros([3, 2, 4, 5])
```

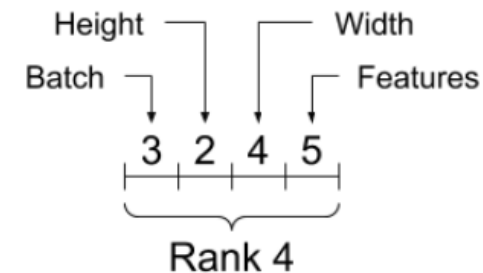
A rank-4 tensor, shape: [3, 2, 4, 5]



```
print("Type of every element:", rank_4_tensor.dtype)
print("Number of axes:", rank_4_tensor.ndim)
print("Shape of tensor:", rank_4_tensor.shape)
print("Elements along axis 0 of tensor:", rank_4_tensor.shape[0])
print("Elements along the last axis of tensor:", rank_4_tensor.shape[-1])
print("Total number of elements (3*2*4*5): ", tf.size(rank_4_tensor).numpy())
```

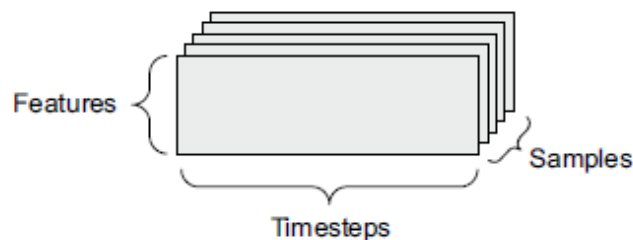
```
Type of every element: <dtype: 'float32'>
Number of axes: 4
Shape of tensor: (3, 2, 4, 5)
Elements along axis 0 of tensor: 3
Elements along the last axis of tensor: 5
Total number of elements (3*2*4*5): 120
```

- Often axes are ordered from **global to local**.
- The first axes is called the **batch axis** or batch dimension.



➔ Real-world examples of data tensors

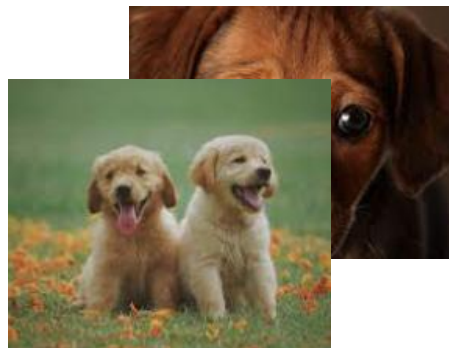
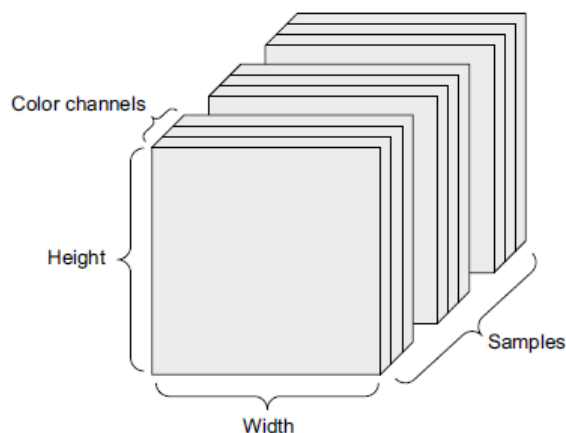
- The data we'll manipulate almost always fall into one of the following categories:
- ✓ **Vector data:** Rank-2 tensors of shape (samples, features), where each sample is a vector of numerical attributes (“features”)
- ✓ **Timeseries data or sequence data:** Rank-3 tensors of shape (samples, timesteps, features), where each sample is a sequence (of length timesteps) of feature vectors





Real-world examples of data tensors

- ✓ **Images:** Rank-4 tensors of shape (samples, height, width, channels), where each sample is a 2D grid of pixels, and each pixel is represented by a vector of values (“channels”)

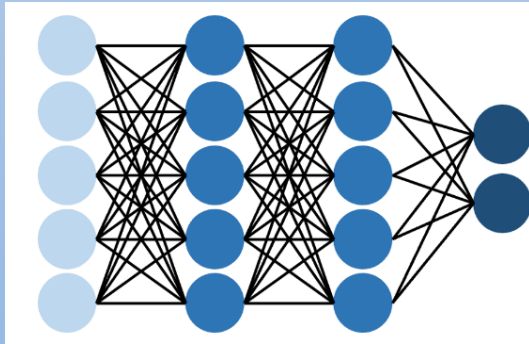


- ✓ **Video:** Rank-5 tensors of shape (samples, frames, height, width, channels), where each sample is a sequence (of length frames) of images.

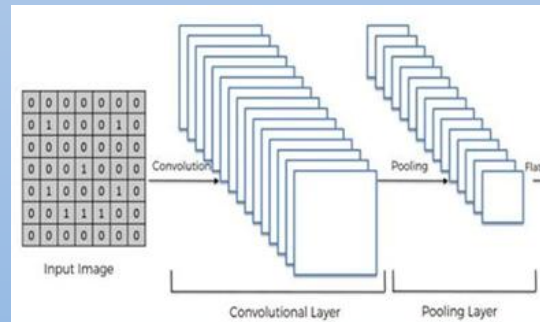
Example: What is the tensor shape for a 60-second, 144*256 video clip sampled at 4 frames per second?

Types of Neural Networks

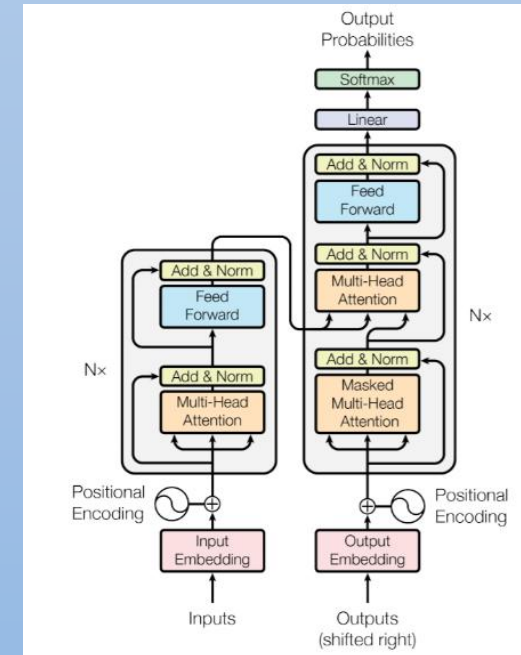
Standard NN



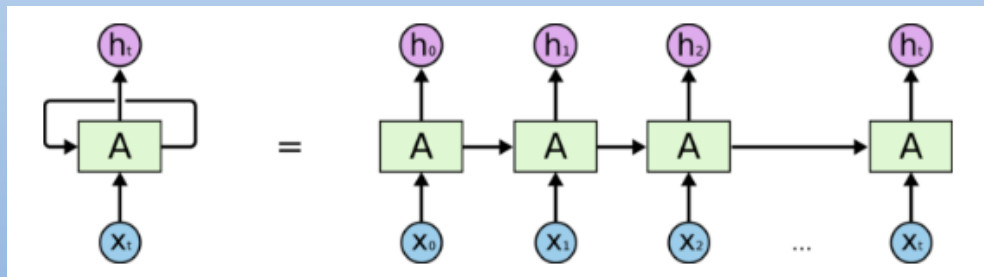
Convolutional NN

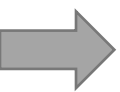


Transformers



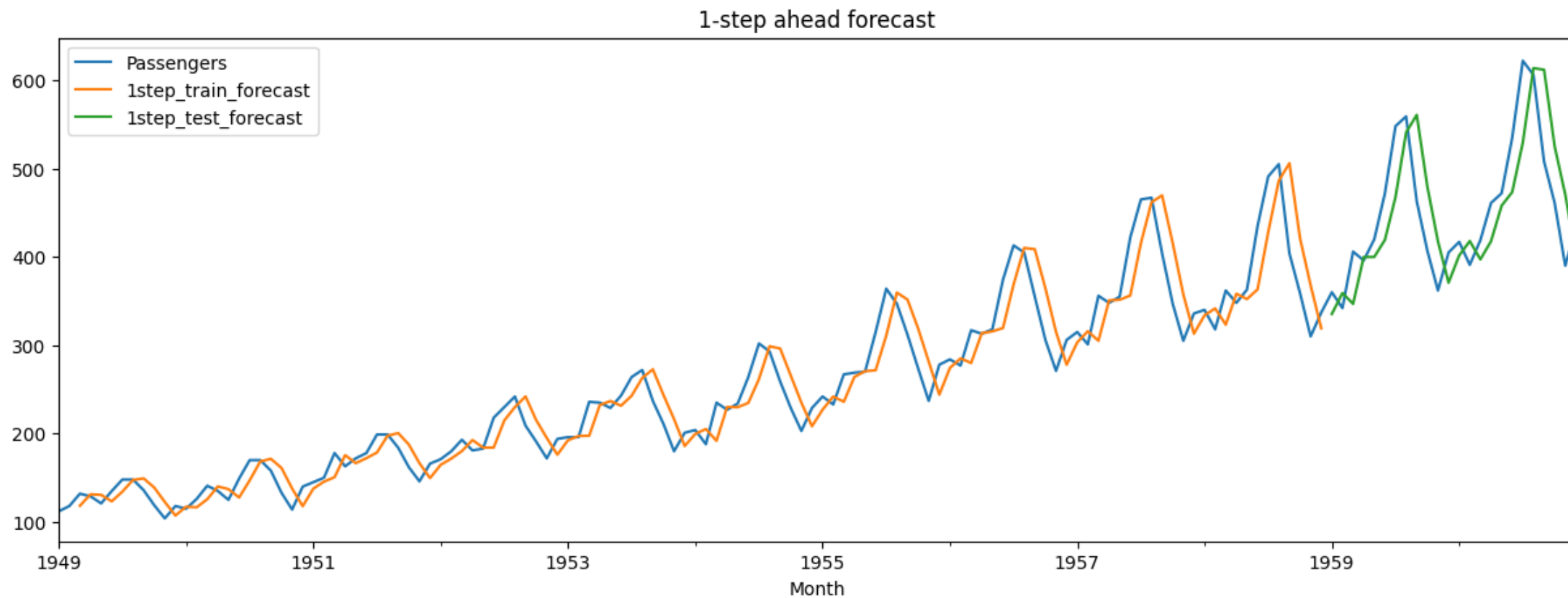
Recurrent NN

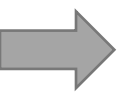




DNN using 2 lags

```
i = Input(shape=(Tx,))  
x = Dense(32, activation='relu')(i)  
x = Dense(16, activation='relu')(x)  
output = Dense(Ty, activation='linear')(x)  
model = Model(i, output)  
model.compile(loss='mse', optimizer='adam')
```

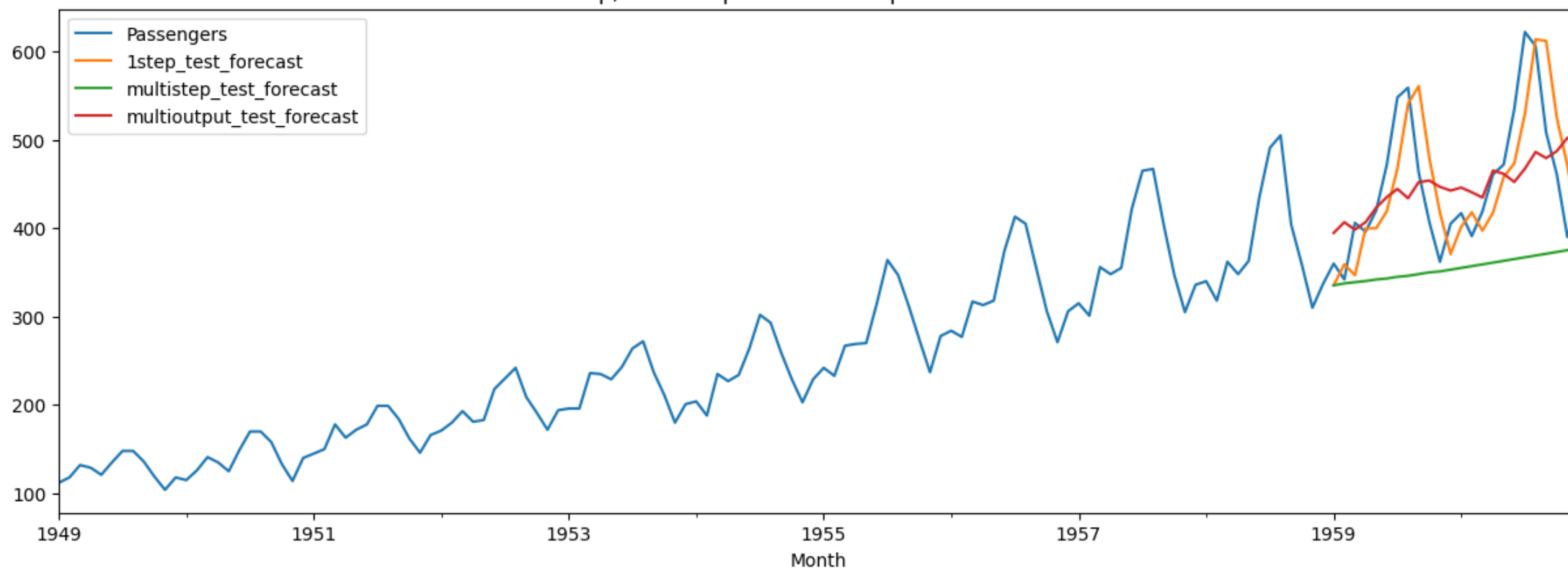




DNN using 2 lags

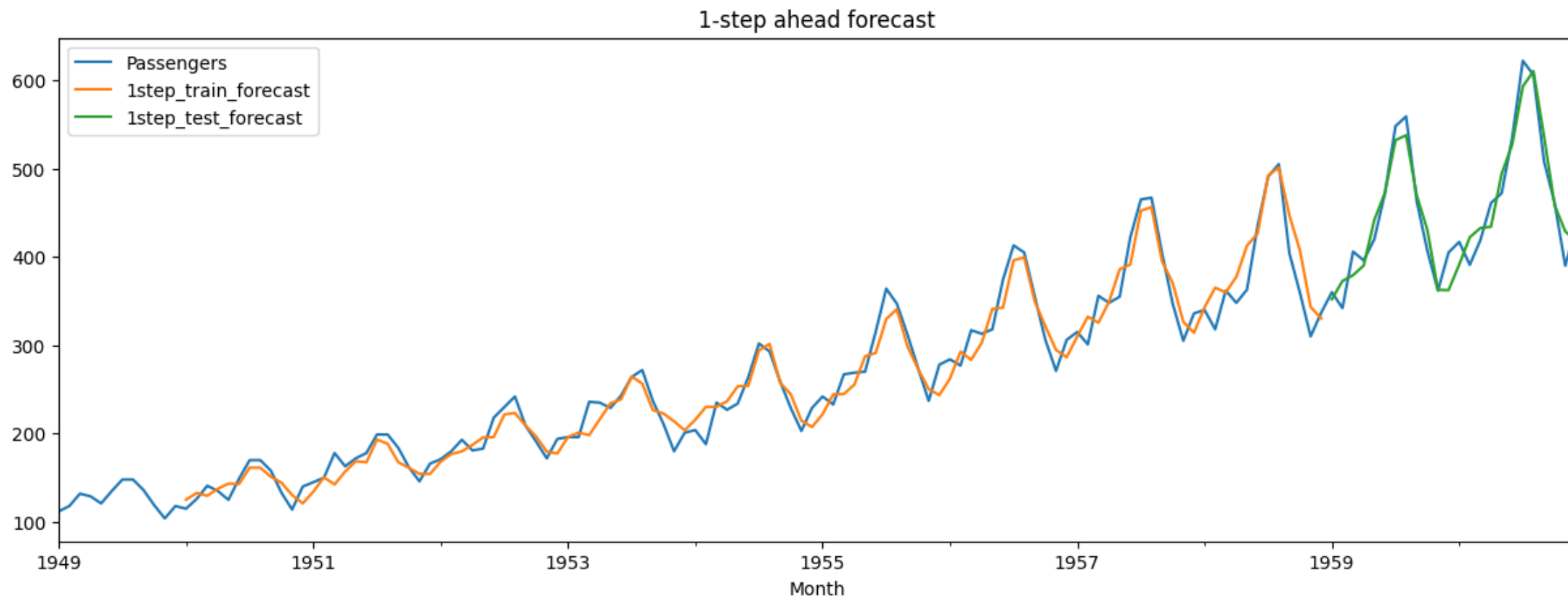
```
i = Input(shape=(Tx,))  
x = Dense(32, activation='relu')(i)  
x = Dense(16, activation='relu')(x)  
output = Dense(Ty, activation='linear')(x)  
model = Model(i, output)  
model.compile(loss='mse', optimizer='adam')
```

1-step, multi-step and multi-output ahead forecast



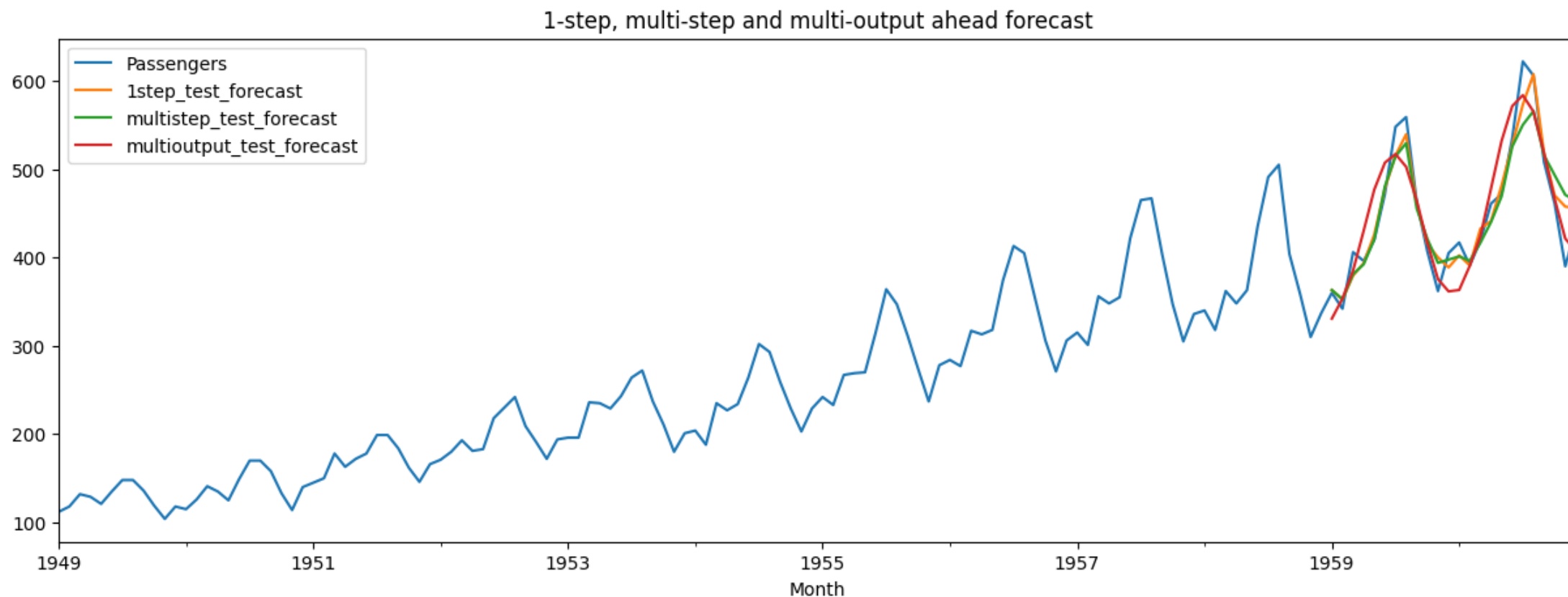
➔ DNN using 12 lags

```
i = Input(shape=(Tx,))
x = Dense(32, activation='relu')(i)
x = Dense(16, activation='relu')(x)
output = Dense(Ty, activation='linear')(x)
model = Model(i, output)
model.compile(loss='mse', optimizer='adam')
```



➔ DNN using 12 lags

```
i = Input(shape=(Tx,))
x = Dense(32, activation='relu')(i)
x = Dense(16, activation='relu')(x)
output = Dense(Ty, activation='linear')(x)
model = Model(i, output)
model.compile(loss='mse', optimizer='adam')
```



➔ Road map!

- ✓ Module 1- Introduction to Deep Forecasting
- ✓ Module 2- Setting up Deep Forecasting Environment
- ✓ Module 3- Exponential Smoothing
- ✓ Module 4- ARIMA models
- ✓ Module 5- Machine Learning for Time series Forecasting
- ✓ Module 6- Deep Neural Networks
- Module 7- Deep Sequence Modeling (RNN, LSTM)
- Module 8- Prophet and Neural Prophet

