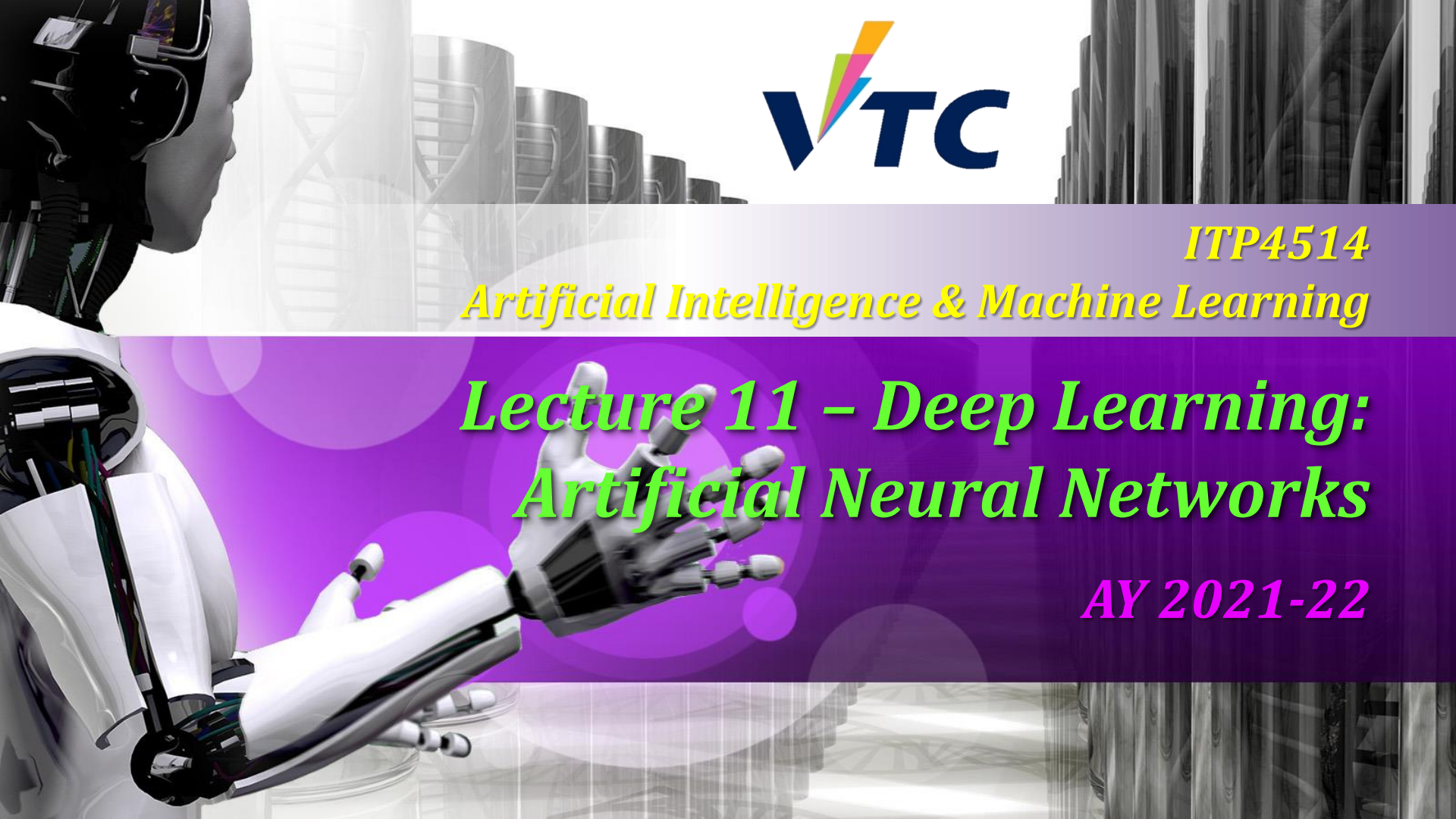


ITP4514

Artificial Intelligence & Machine Learning

Lecture 11 – Deep Learning: Artificial Neural Networks

AY 2021-22

Module Plan

1. AI: An Overview & Python Basics
2. Collection Data Types, NumPy & Pandas
3. AI Search Techniques
4. Logic Programming & CSP
5. Probabilistic Reasoning
6. ML – Classification
7. ML – Regression & Clustering
8. Computer Vision & AI Cloud Services
9. Natural Language Processing
10. Recommender Systems
11. ***DL – Basic ANN***
12. DL – CNN & RNN

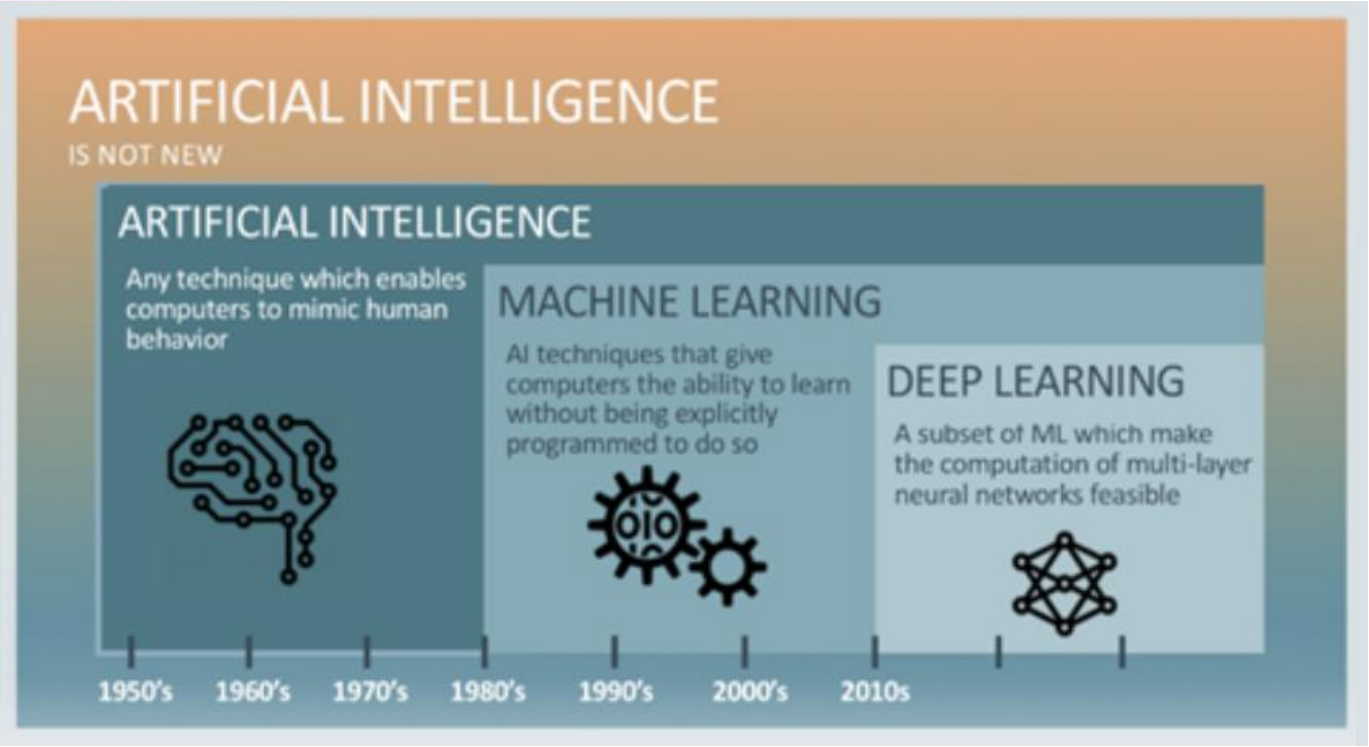
Lesson Outline

1. Deep Learning: An Overview
2. Deep Learning Basics
   - Activation Function
   - Normalizer
   - Optimizer
3. Deep Learning Modeling with Keras

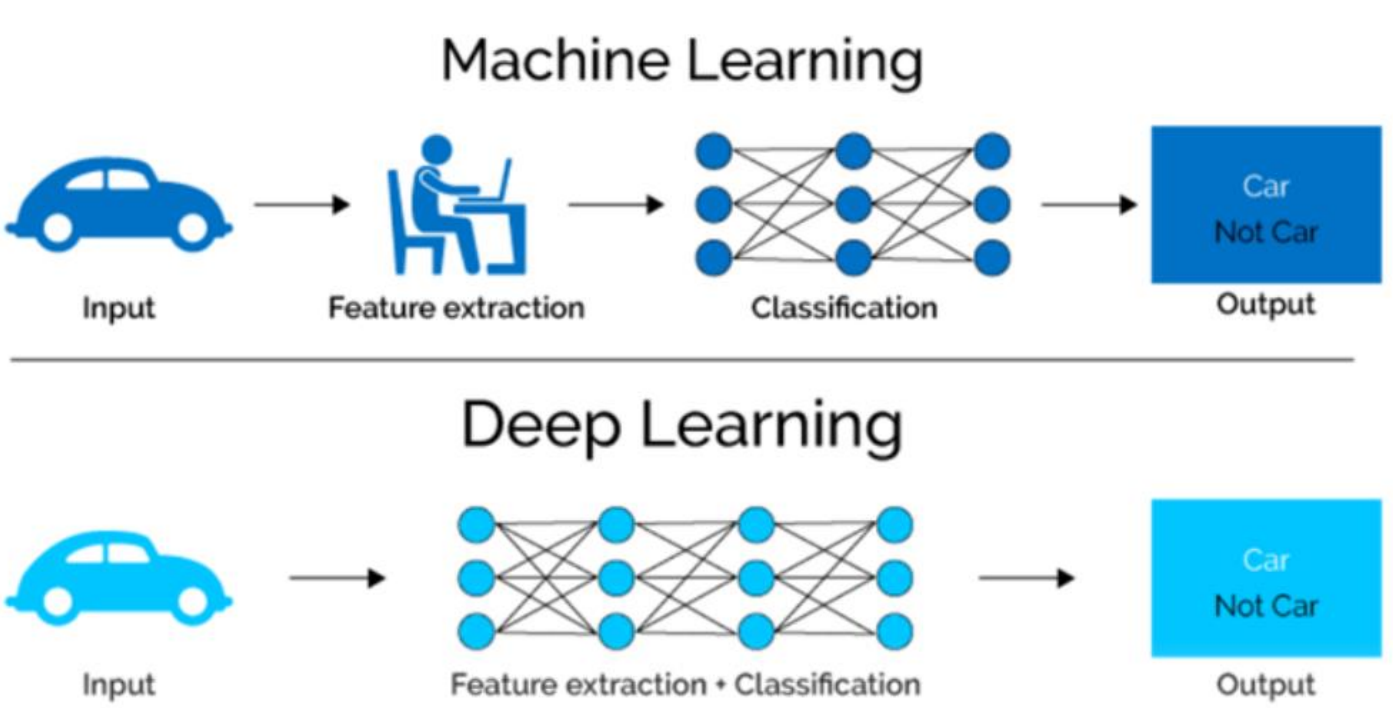
1. Deep Learning: An Overview

- *Deep learning* (also known as *deep structured learning*) is part of a broader family of machine learning methods based on *artificial neural networks* with representation learning.
- Learning can be supervised, semi-supervised or unsupervised.

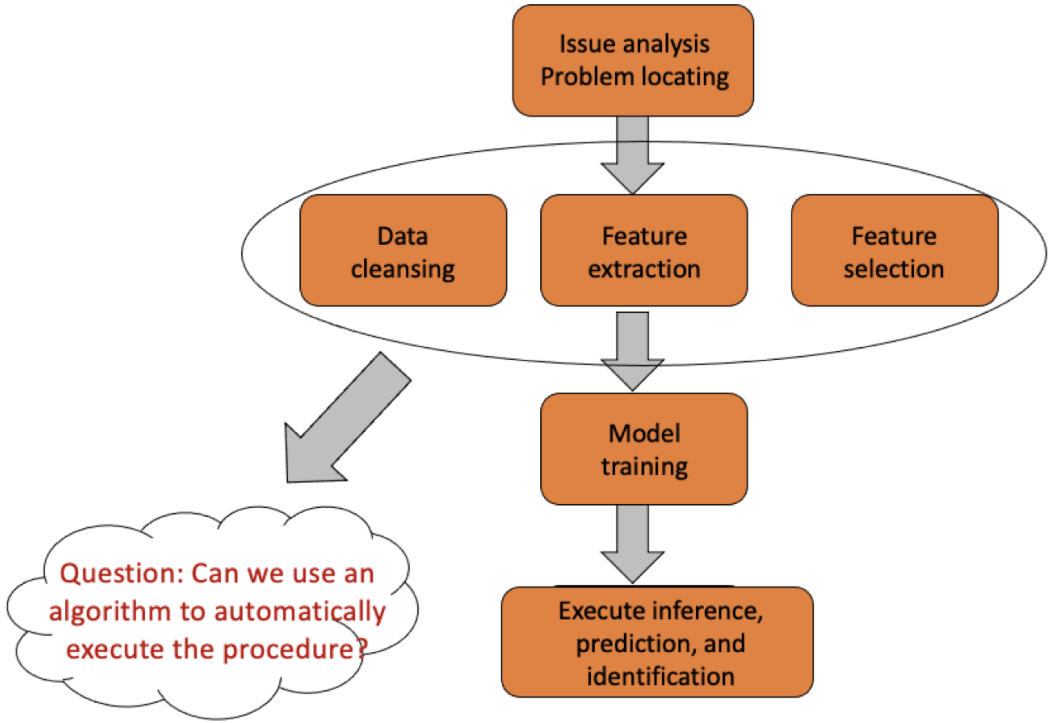
AI, Machine Learning and Deep Learning



Machine Learning vs Deep Learning

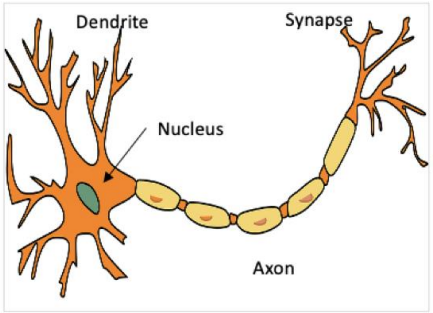


Traditional Machine Learning

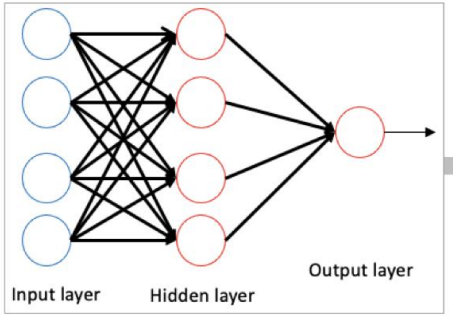


Deep Learning

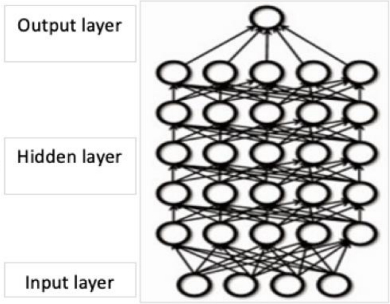
- The deep learning architecture is a deep neural network. "Deep" in "deep learning" refers to the number of layers of the neural network.



Human neural network



Perceptron



Deep neural network

Artificial Neural Networks

- A neural network can be simply expressed as an information processing system designed *to imitate the human brain structure and functions* based on its source, features, and explanations.
- Artificial neural network (neural network): Formed by *artificial neurons* connected to each other, the neural network extracts and simplifies the human brain's microstructure and functions.
- It is an important approach to simulate human intelligence and reflect several basic features of human brain functions, such as *concurrent information processing*, *learning, association*, *model classification*, and *memory*.

Artificial Neural Networks (2)

Artificial Neural Networks (ANNs) are inspired by the *human brain*:

- Massively parallel, distributed system, made up of simple processing units (neurons).
- Synaptic connection strengths among neurons are used to store the acquired knowledge.
- Knowledge is acquired by the network from its environment through a learning process.

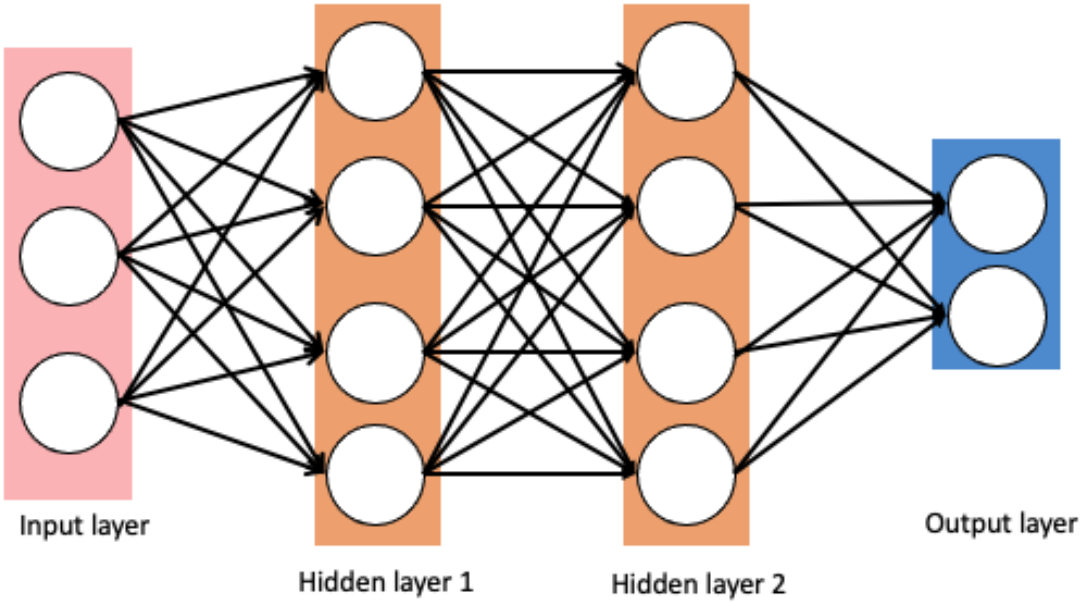
Properties of ANNs

- *Learning from examples* – labeled or unlabeled.
- *Adaptivity* – changing the connection strengths to learn things.
- *Non-linearity* – the non-linear activation functions are essential.
- *Fault tolerance* – if one of the neurons or connections is damaged, the whole network still works quite well.

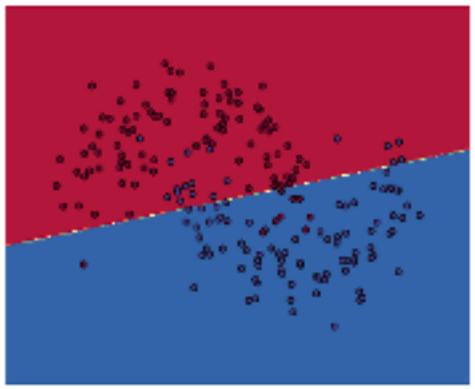
Properties of ANNs (2)

- Thus, they might be better alternatives than classical solutions for problems characterized by:
  - high dimensionality, noisy, imprecise or imperfect data; *and*
  - a lack of a clearly stated mathematical solution or algorithm.

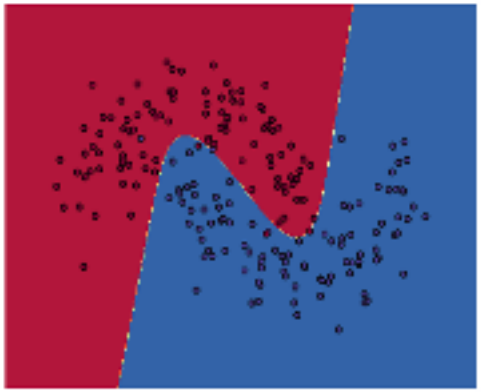
Feedforward Neural Network



Impacts of Hidden Layers on A Neural Network



0 hidden layers



3 hidden layers

20 hidden layers

Training a NN Model

- To build a neural network model, you need to consider
  - Task for NN (Regression? Classification?)
  - Input and output dimension
  - number of hidden layers
  - number of neutrons for each layer
  - Activation functions (Sigmoid, Tanh, ReLU, Softmax)
  - Cost Function (RSS / Cross-entropy)
  - Normalization
  - Optimizer
  - Common problems of ANN

Residual Sum of Squares or Ordinary Least Square

Packages for ANN

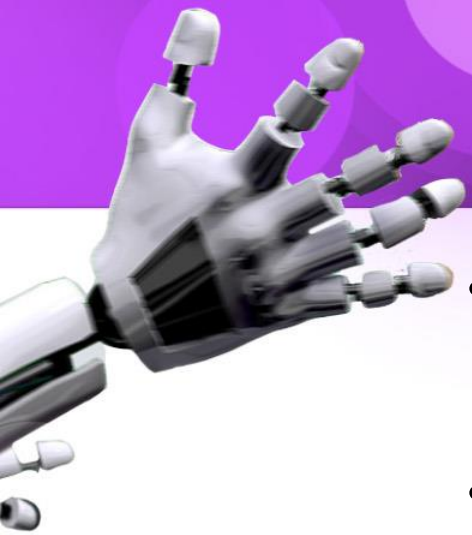
- You don't need to create an ANN from scratch, instead, we will use DL framework to complete the tasks:
  - TensorFlow (tf) – a free and open-source software library for machine learning.

    import tensorflow as tf
  - Keras – high-level API built on tensorflow that could help programmers build DL model easier.

    from tensorflow import keras
- Generally, Keras could solve common DL problems and tf provides more customizable options.
- DL frameworks also need to work with libraries like panda, numpy for data manipulation.

2. Deep Learning Basics

- Activation Functions
- Normalizer
- Optimizer



2.1 Activation Functions

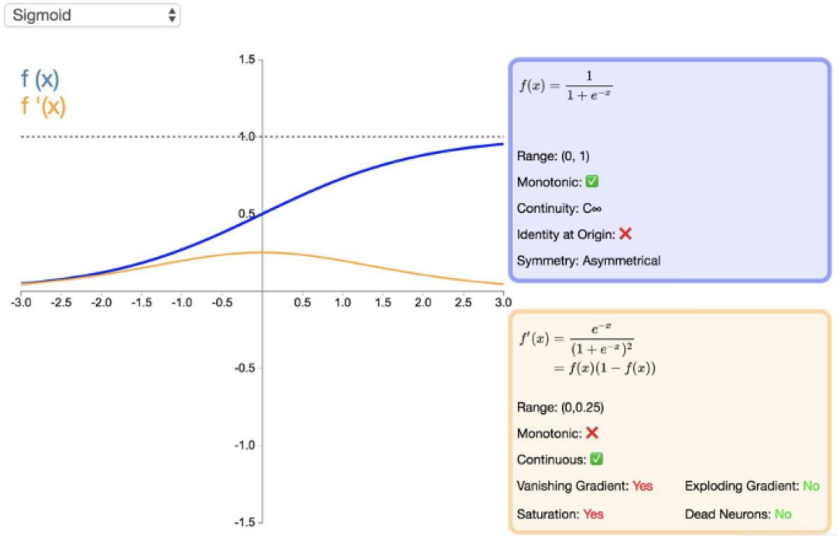
- Activation functions are important for the neural network model to learn and understand complex non-linear functions.
- They allow introduction of non-linear features to the network.
- Without activation functions, output signals are only simple linear functions.
- The complexity of linear functions is limited, and the capability of learning complex function mappings from data is low.

Common Activation Functions

1. *Sigmoid*
2. *Tanh*
3. *ReLU*
4. *Softmax*

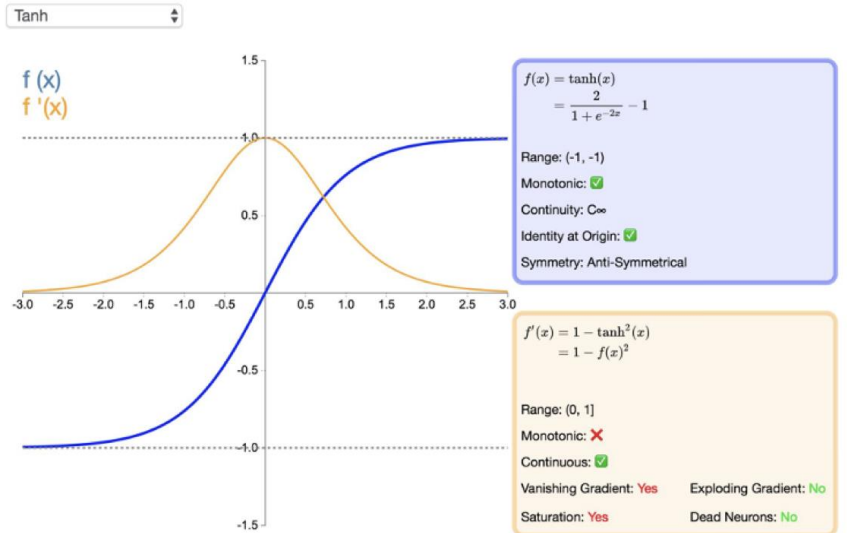
Sigmoid

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



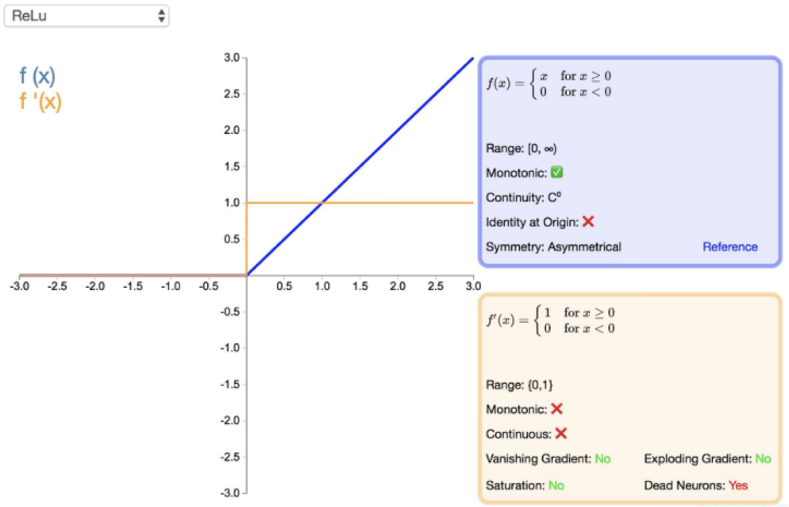
Tanh

$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



Rectified Linear Unit (ReLU)

$$y = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases}$$



Softmax

- Softmax function:

$$\sigma(\mathrm{z})_j = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

- The Softmax function is used to map a K-dimensional vector of arbitrary real values to another K-dimensional vector of real values, where each vector element is in the interval (0, 1). All the elements add up to 1.
- The Softmax function is often used as the output layer of a multiclass classification task.

2.2 Normalizer

- ***Regularization*** is an important and effective technology to reduce generalization errors in machine learning.
- It is especially useful for deep learning models that tend to be over-fit due to a large number of parameters.

Normalizer

- Researchers have proposed many effective technologies to prevent *over-fitting*, including:
  - Adding constraints to parameters, such as $L1$ and $L2$ norms
  - Expanding the training set, such as adding noise and transforming data
  - Dropout
  - Early stopping

Penality Parameters

- Many regularization methods restrict the learning capability of models by adding a penalty parameter $\Omega(\theta)$ to the objective function J. Assume that the target function after regularization is $\tilde{J}$.

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha\Omega(\theta),$$

- Where $\alpha\epsilon[0, \infty)$ is a hyperparameter that weights the relative contribution of the norm penalty term Ω and the standard objective function $J(X; \theta)$. If α is set to 0, no regularization is performed. The penalty in regularization increases with α.

L1 Regularization

- Add L_1 norm constraint to model parameters, that is,

$$\tilde{J}(w; X, y) = J(w; X, y) + \alpha \|w\|_1,$$

- If a gradient method is used to resolve the value, the parameter gradient is

$$\nabla \tilde{J}(w) = \propto sign(w) + \nabla J(w).$$

L2 Regularziation

- Add norm penalty term L_2 to prevent overfitting.

$$\tilde{J}(w; X, y) = J(w; X, y) + \frac{1}{2}\alpha\|w\|_2^2,$$

- A parameter optimization method can be inferred using an optimization technology (such as a gradient method):

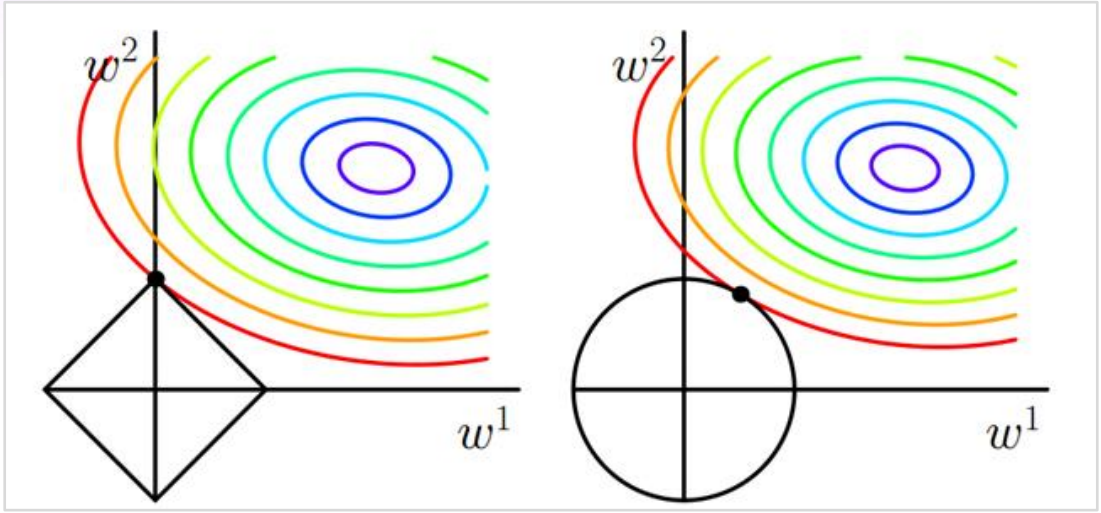
$$w = (1 - \varepsilon\alpha)\omega - \varepsilon\nabla J(w),$$

- where ε is the learning rate. Compared with a common gradient optimization formula, this formula multiplies the parameter by a reduction factor.

L1 vs L2

- The major differences between L_2 and L_1:
  - According to the preceding analysis, L_1 can generate a more sparse model than L_2. When the value of parameter w is small, L_1 regularization can directly reduce the parameter value to 0, which can be used for feature selection.
  - From the perspective of probability, many norm constraints are equivalent to adding prior probability distribution to parameters. In L_2 regularization, the parameter value complies with the Gaussian distribution rule. In L_1 regularization, the parameter value complies with the Laplace distribution rule.

L1 vs L2



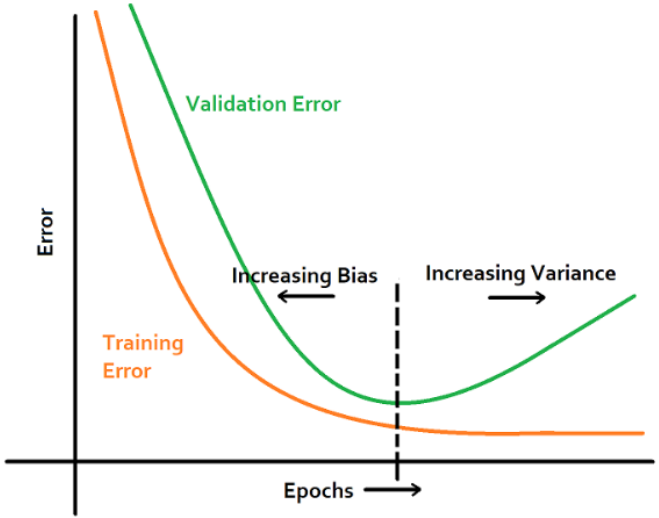
Dropout

- ***Dropout*** is a common and simple regularization method, which has been widely used since 2014.

- Dropout randomly discards some inputs during the training process. In this case, the parameters corresponding to the discarded inputs are not updated.

Early Stopping

- A test on data of the validation set can be inserted during the training. When the data loss of the verification set increases, perform early stopping.



2.3 Optimizer

- ***Optimizers*** tie together the loss function and model parameters by updating the model in response to the output of the loss function.
- In simpler terms, optimizers shape and mold your model into its most accurate possible form by futzing with the weights.
- The loss function is the guide to the terrain, telling the optimizer when it's moving in the right or wrong direction.

Optimizer

- Purposes of the algorithm optimization include but are not limited to:
  - Accelerating algorithm convergence.
  - Preventing or jumping out of local extreme values.
  - Simplifying manual parameter setting, especially the learning rate (LR).
- *Common optimizers*: common GD optimizer, momentum optimizer, Nesterov, AdaGrad, AdaDelta, RMSProp, Adam, AdaMax, and Nadam.

3. Deep Learning Modeling with Keras

- Keras Neural Network Models
- Our First Neural Network
- Basic Neural Network
- Multi-Layer Perceptron Overview
- Multi-Layer Perceptron on Digit Classification

Keras Neural Network Models

- The focus of the Keras library is a *model*.
- The simplest model is defined in the *Sequential* class which is a linear stack of *Layers*.
- You can create a Sequential model and define all of the layers in the *constructor*, for example:

```
from keras.models import Sequential
model = Sequential(...)
```

Keras Neural Network Models (2)

- A more useful way is to create a Sequential model and add your layers in the order of the computation you wish to perform.
- For example:

```
from keras.models import Sequential
model = Sequential()
model.add(...)
model.add(...)
model.add(...)
```

Keras Neural Network Models (3)

A. Model Inputs

- The first layer in your model must specify the *shape* of the input.
- This is the number of input attributes and is defined by the *input_dim* argument. This argument expects an integer.
- For example, you can define input in terms of 8 inputs for a *Dense* type layer as follows:

```
Dense(16, input_dim=8)
```

Keras Neural Network Models (4)

B. Model Layers

- Layers of different type are a few properties in common, specifically their method of *weight initialization* and *activation functions*.

Keras Modeling
A. Model Inputs
B. Model Layers
C. Model Compilation
D. Model Training
E. Model Prediction

1. Weight Initialization

- The type of initialization used for a layer is specified in the *kernel_initializer* and *bias_initializer* arguments.

Keras Neural Network Models (5)

- Some common types of layer initialization include:
  - **random_uniform**: Weights are initialized to small uniformly random values between 0 and 0.05.
  - **random_normal**: Weights are initialized to small Gaussian random values (zero mean and standard deviation of 0.05).
  - **zeros**: All weights are set to zero values.

```
Dense(units=64, kernel_initializer='random_normal', bias_initializer='zeros')
```

- You can see a full list of initialization techniques supported on the Usage of initializations page.

Keras Neural Network Models (6)

2. Activation Function

- Keras supports a range of standard neuron activation function, such as: *softmax*, *relu*, *tanh* and *sigmoid*.
- You typically specify the type of activation function used by a layer in the activation argument, which takes a string value.
- You can see a full list of activation functions supported by Keras on the Usage of activations page.
- Interestingly, you can also create an *Activation* object and add it directly to your model after your layer to apply that activation to the output of the Layer.

relu : max(x,0)

sigmoid : $\begin{cases} \approx 0, x < -5 \\ \approx 1, x > 5 \end{cases}$

softmax : range (0,1) & sum to 1

Keras Neural Network Models (7)

3. Layer Types

- There are a large number of core Layer types for standard neural networks.
- Some common and useful layer types you can choose from are:
  - **Dense**: Fully connected layer and the most common type of layer used on multi-layer perceptron models.
  - **Dropout**: Apply dropout to the model, setting a fraction of inputs to zero in an effort to reduce over fitting.
  - **Merge**: Combine the inputs from multiple models into a single model.

You can learn about the full list of core Keras layers on the Core Layers page.

Keras Neural Network Models (8)

C. Model Compilation

- Once you have defined your model, it needs to be compiled.
- This creates the efficient structures used by the underlying backend (Theano or TensorFlow) in order to efficiently execute your model during training.
- You compile your model using the **compile()** function and it accepts three important attributes:
  - Model optimizer.
  - Loss function.
  - Metrics.

```
model.compile(optimizer=, loss=, metrics=)
```

Keras Modeling
A. Model Inputs
B. Model Layers
C. Model Compilation
D. Model Training
E. Model Prediction

Keras Neural Network Models (9)

1. Model Optimizers

- The optimizer is the search technique used *to update weights* in your model.
- You can create an optimizer object and pass it to the compile function via the optimizer argument.
- This allows you to configure the optimization procedure with it's own arguments, such as learning rate.
- For example:

```
sgd = SGD(...)
model.compile(optimizer=sgd)
```

Keras Neural Network Models (10)

1. Model Optimizers (cont.)

- You can also use the default parameters of the optimizer by specifying the name of the optimizer to the *optimizer* argument. For example:

```
model.compile(optimizer='sgd')
```

- Some popular gradient descent optimizers you might like to choose from include:
  - **SGD**: stochastic gradient descent, with support for momentum.
  - **Adam**: Adaptive Moment Estimation (Adam) that also uses adaptive learning rates.

You can learn about all of the optimizers supported by Keras on the Usage of optimizers page.

Keras Neural Network Models (11)

2. Model Loss Functions

- The *loss function*, also called the objective function is the evaluation of the model used by the optimizer to navigate the weight space.
- You can specify the name of the loss function to use to the compile function by the loss argument. Some common examples include:
  - **mse**: for mean squared error.
  - **binary_crossentropy**: for binary logarithmic loss (logloss).
  - **categorical_crossentropy**: for multi-class logarithmic loss (logloss).

You can learn more about the loss functions supported by Keras on the Usage of objectives page.

Keras Neural Network Models (12)

3. Model Metrics

- Metrics are evaluated by the model during training.
- Common metric is *accuracy*.
- See Metrics for more choices.

Keras Neural Network Models (13)

D. Model Training

- The model is trained on NumPy arrays using the fit() function, for example

  model.fit(X, y, epochs=, batch_size=)

- Training both specifies the number of epochs to train on and the batch size.
  - Epochs (epochs) is the number of times that the model is exposed to the training dataset.
  - Batch Size (batch_size) is the number of training instances shown to the model before a weight update is performed.

Keras Modeling
A. Model Inputs
B. Model Layers
C. Model Compilation
D. Model Training
E. Model Prediction

Keras Neural Network Models (14)

E. Model Prediction

- Once you have trained your model, you can use it to make predictions on test data or new data.
- There are a number of different output types you can calculate from your trained model, each calculated using a different function call on your model object.
- For example:
  - **model.evaluate()**: To calculate the loss values for input data.
  - **model.predict()**: To generate network output for input data.

Keras Modeling
A. Model Inputs
B. Model Layers
C. Model Compilation
D. Model Training
E. Model Prediction

Keras Neural Network Models (15)

E. Model Prediction (cont.)

- **model.predict_classes()**: To generate class outputs for input data.
- **model.predict_proba()**: To generate class probabilities for input data.

- For example, on a classification problem you will use the **predict_classes()** function to make predictions for test data or new data instances.

Our First Neural Network

- We are implementing our first neural network in Keras.
- This model is a *single-layer perceptron*, which models a logical AND gate:

A	B	AND
0	0	0
0	1	0
1	0	0
1	1	1

Our First Neural Network (2)

- Dependencies:

First_NN.ipynb

```
import numpy as np
from keras.models import Sequential
from keras.layers.core import Dense
```

- Dataset:

```
X = np.array([[0,0],[0,1],[1,0],[1,1]], "float32")
y = np.array([[0],[0],[0],[1]], "float32")
```

Our First Neural Network (3)

One Layer

- Single-Layer Perceptron:

```
model = Sequential()
model.add(Dense(1, input_dim=2,
                activation='sigmoid'))

model.compile(optimizer='adam',
              loss='mean_squared_error',
              metrics=['accuracy'])
model.fit(x=X, y=y, epochs=300, verbose=1)
print(model.predict(X).round())
```

*** Note the results reach at most 75% accuracy only.*

Our First Neural Network (4)

- Multi-Layer Perceptron:

```
model = Sequential()
model.add(Dense(4, input_dim=2,
                activation='relu'))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='adam',
              loss='mean_squared_error',
              metrics=['accuracy'])
model.fit(x=X, y=y, epochs=300, verbose=1)
print(model.predict(X).round())
```

Two Layers

*** Note the results possibly reach 100% accuracy.*

Basic Neural Network

Steps of Deep Learning Project:

1. *Preprocess and load data* – We need to process it before feeding to the neural network. In this step, we will also visualize data which will help us to gain insight into the data.
2. *Define model* – Now we need a neural network model. This means we need to specify the number of hidden layers in the neural network and their size, the input and output size.

Basic Neural Network (2)

3. *Loss and optimizer* – Now we need to define the loss function according to our task. We also need to specify the optimizer to use with learning rate and other hyperparameters of the optimizer.
4. *Fit model* – This is the training step of the neural network. Here we need to define the number of epochs for which we need to train the neural network.

Basic Neural Network (3)

Data processing:

- We will use simple data of *mobile price range classifier*.
- The dataset consists of *20 features* and we need to predict the price range in which phone lies.
- These ranges are divided into *4 classes*.
- The features of our dataset include:

  'battery_power', 'blue', 'clock_speed', 'dual_sim', 'fc', 'four_g', 'int_memory', 'm_dep', 'mobile_wt', 'n_cores', 'pc', 'px_height', 'px_width', 'ram', 'sc_h', 'sc_w', 'talk_time', 'three_g', 'touch_screen', 'wifi'

Basic Neural Network (4)

Data processing (cont.)

Basic_NN.ipynb

```
import warnings
warnings.filterwarnings('ignore')
```

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

# Dataset import
dataset = pd.read_csv('Datasets/mobile.csv')
dataset.head()
```

	battery_power	blue	clock_speed	dual_sim	fc	four_g	...	px_height	px_width
0	842	0	2.2	0	1	0	...	20	756
1	1021	1	0.5	1	0	1	...	905	1988
2	563	1	0.5	1	2	1	...	1263	1716

Basic Neural Network (5)

```
# Changing pandas dataframe to numpy array
X = dataset.iloc[:,:20].values
y = dataset.iloc[:,20:21].values

# Normalizing the data
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X = sc.fit_transform(X)
print('Normalized data:')
print(X[0])

# One hot encode
from sklearn.preprocessing import OneHotEncoder
ohe = OneHotEncoder()
y = ohe.fit_transform(y).toarray()
print('One hot encoded array:')
print(y[0:5])

# Train test split of model
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size = 0.1,random_state = 0)
```

Basic Neural Network (6)

Building Neural Network:

- *Keras* is a simple tool for constructing a neural network.
- It is a high-level framework based on *tensorflow*, *theano* or *cntk* backends.
- In our dataset, the input is of 20 values and output is of 4 values.
- So the *input* and *output layer* is of 20 and 4 dimensions respectively.

Basic Neural Network (7)

Building Neural Network (cont.)

```
# Dependencies
import keras
from keras.models import Sequential
from keras.layers import Dense

# Neural network
model = Sequential()
model.add(Dense(16, input_dim=20, activation='relu'))
model.add(Dense(12, activation='relu'))
model.add(Dense(4, activation='softmax'))
```

Basic Neural Network (8)

Building Neural Network (cont.)

- In our neural network, we are using *two hidden layers* of 16 and 12 dimension.
- Sequential specifies to Keras that we are creating model sequentially and the output of each layer we add is input to the next layer we specify.
- model.add is used to add a layer to our neural network. We need to specify as an argument what type of layer we want. The Dense is used to specify the *fully connected layer*.

Basic Neural Network (9)

Building Neural Network (cont.)

- the output layer takes different activation functions and for the case of *multiclass* classification, it is *softmax*.
- '*relu*' (Rectified Linear Unit): Returns element-wise *max(x, 0)*.

Basic Neural Network (10)

Building Neural Network (cont.)

- Now we specify the *loss function* and the *optimizer*.
- It is done using *compile* function in Keras.

```
model.compile(loss='categorical_crossentropy',
optimizer='adam', metrics=['accuracy'])
```

- *Categorical_crossentropy* specifies that we have multiple classes. The optimizer is *Adam*.
- Metrics is used to specify the way we want to judge the performance of our neural network. Here we have specified it to *accuracy*.

Training Our Model

- Training step is simple in Keras.
- model.fit is used to train it.

```
history = model.fit(X_train, y_train, epochs=100,
batch_size=64)
```

- Here we need to specify the input data -> X_train, labels -> y_train, *number of epochs (iterations)*, and *batch size*.
- It returns the *history of model training*.
- History consists of *model accuracy* and *losses* after each epoch.

Training Our Model (2)

- Here is the history of the training:

```
Epoch 1/100
1800/1800 [==============================] - 2s 834us/step - loss: 1.4317 - accuracy: 0.2894
Epoch 2/100
1800/1800 [==============================] - 0s 40us/step - loss: 1.3783 - accuracy: 0.3328
. . .
Epoch 99/100
1800/1800 [==============================] - 0s 39us/step - loss: 0.0470 - accuracy: 0.9928
Epoch 100/100
1800/1800 [==============================] - 0s 38us/step - loss: 0.0451 - accuracy: 0.9933
```

Training Our Model (3)

- Check the model's performance on test data:

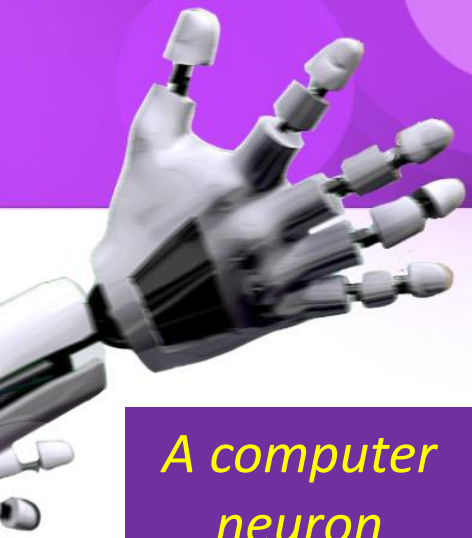
```
y_pred = model.predict(X_test)

# Converting predictions to label
pred = list()
for i in range(len(y_pred)):
    pred.append(np.argmax(y_pred[i]))
```

```
# Converting one hot encoded test label to label
test = list()
for i in range(len(y_test)):
    test.append(np.argmax(y_test[i]))
```

```
from sklearn.metrics import accuracy_score
a = accuracy_score(pred, test)
print('Accuracy is:', a*100)
```

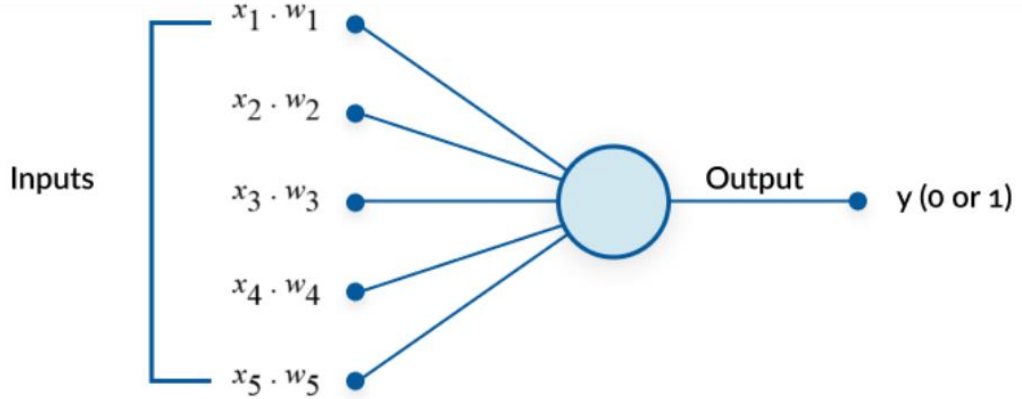
```
Accuracy is: 94.5
```

Multi-Layer Perceptron Overview



A computer neuron

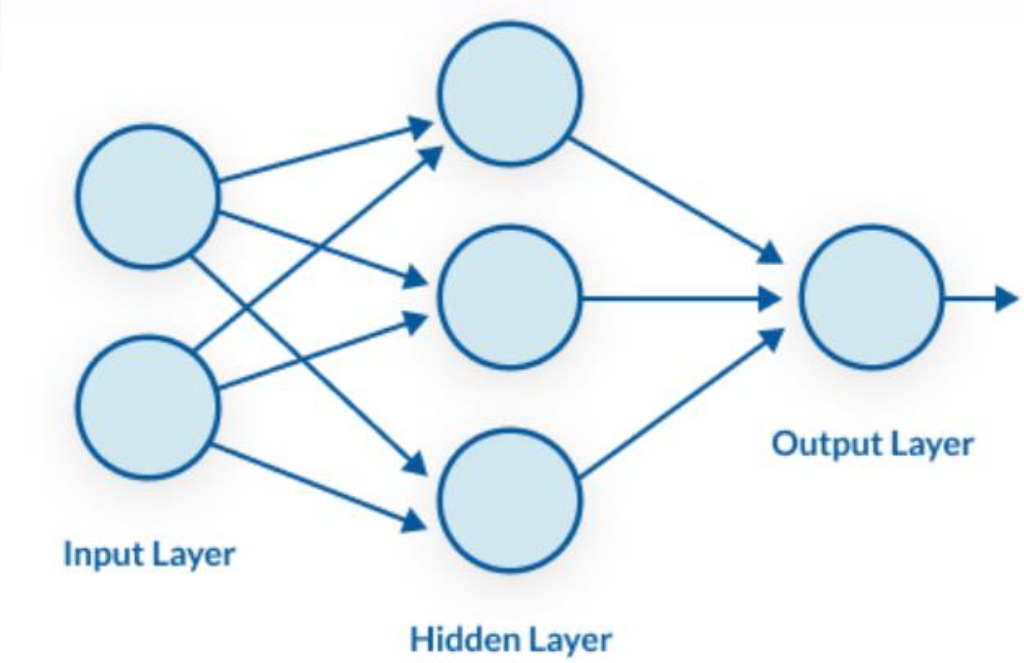
- A perceptron is a simple *binary classification* algorithm.
- It helps to divide a set of input signals into two parts—"yes" and "no".



Multi-Layer Perceptron Overview (2)

- A *multilayer perceptron (MLP)* is a perceptron that teams up with additional perceptrons, stacked in several layers, to solve complex problems.
- The diagram on the next slide shows an *MLP with three layers*.
- Each perceptron in the first layer on the left (the *input layer*), sends outputs to all the perceptrons in the second layer (the *hidden layer*), and all perceptrons in the second layer send outputs to the final layer on the right (the *output layer*).

Multi-Layer Perceptron Overview (3)



Multi-Layer Perceptron on Digit Classification

MNIST Dataset

- Every MNIST data point has two parts: an image of a *handwritten digit (28×28)* and a corresponding *label (0-9)*.
- We'll call the images "X" and the labels "y".
- The *training set* and *test set* contain 6000 and 1000 data points respectively.

Multi-Layer Perceptron on Digit Classification (2)

MLP_MNIST.ipynb

```
# Suppress all warnings
import warnings
warnings.filterwarnings('ignore')

# Import dependencies
import numpy as np
import pandas as pd
from keras.utils import np_utils
np.random.seed(10)

# Load data
train_data = pd.read_csv('Datasets/mnist_train.csv')
test_data = pd.read_csv('Datasets/mnist_test.csv')

X_train = train_data.iloc[:,1:].values.astype('float32')
y_train = train_data.iloc[:,0].values
X_test = test_data.iloc[:,1:].values.astype('float32')
y_test = test_data.iloc[:,0].values
```

Prepare Data

```
Using TensorFlow backend.
```

Multi-Layer Perceptron on Digit Classification (3)

```
X_train_normalized = X_train / 255
X_test_normalized = X_test / 255
```

```
y_train_OneHot = np_utils.to_categorical(y_train)
y_test_OneHot = np_utils.to_categorical(y_test)
```

Prepare Data

```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()
model.add(Dense(units=256,
                input_dim=784,
                kernel_initializer='normal',
                activation='relu'))
model.add(Dense(units=10,
                kernel_initializer='normal',
                activation='softmax'))
```

Build Model

```
print(model.summary())
```

Multi-Layer Perceptron on Digit Classification (4)

Train Model

```
model.compile(loss='categorical_crossentropy',
              optimizer='adam', metrics=['accuracy'])

train_history = model.fit(x=X_train_normalized,
                          y=y_train_OneHot, validation_split=0.2,
                          epochs=10, batch_size=200, verbose=2)
```

Visualization

```
import matplotlib.pyplot as plt

def show_train_history(train_history,train,validation):
    plt.plot(train_history.history[train])
    plt.plot(train_history.history[validation])
    plt.title('Train History')
    plt.ylabel(train)
    plt.xlabel('Epoch')
    plt.legend(['train', 'validation'], loc='upper left')
    plt.show()
```

```
show_train_history(train_history, 'accuracy', 'val_accuracy')
```

Multi-Layer Perceptron on Digit Classification (5)

Check Accuracy

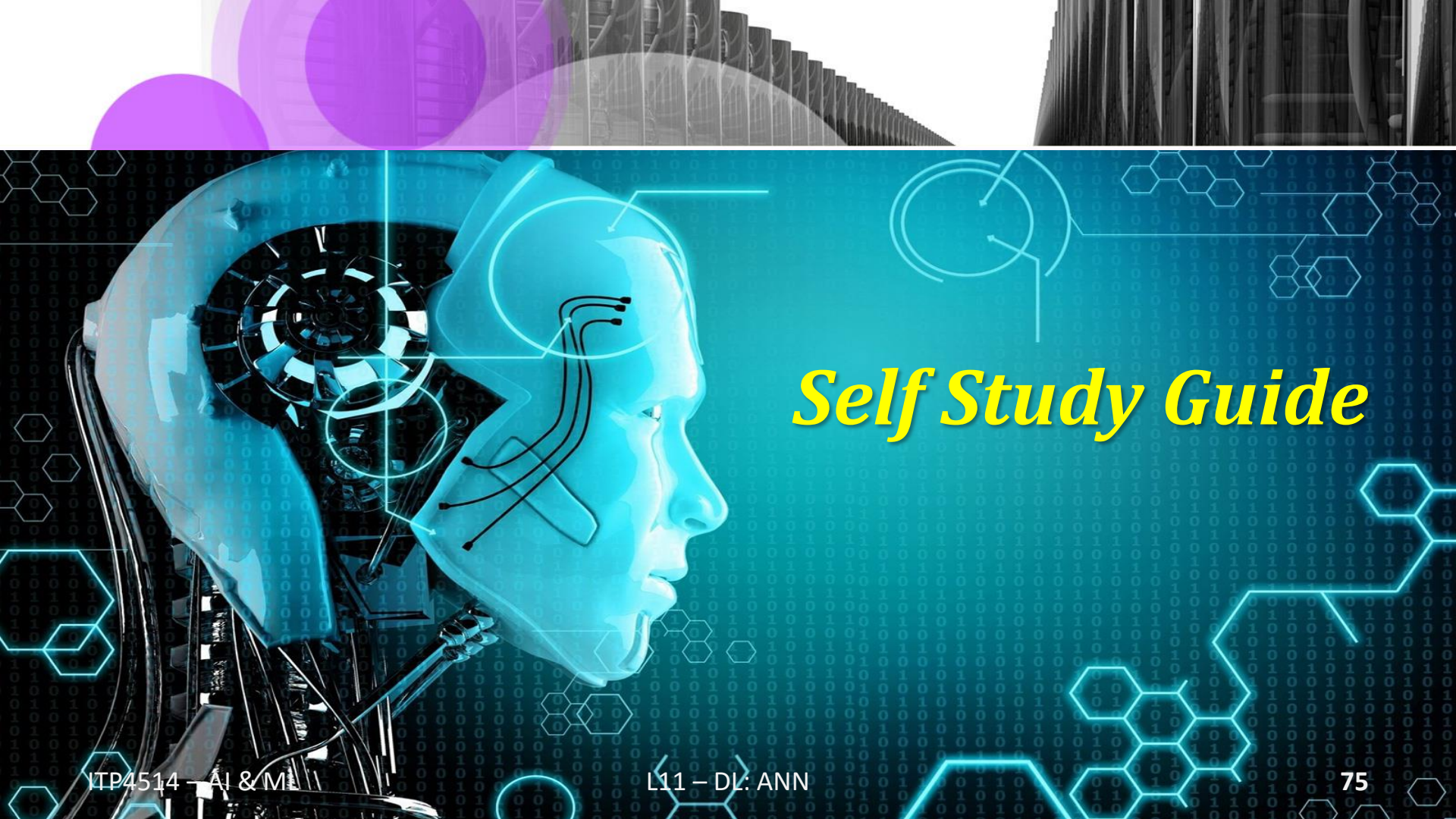
```
scores = model.evaluate(X_test_normalized, y_test_OneHot)
print()
print('accuracy=', scores[1])
```

```
10000/10000 [==============================] - 1s 55us/step

accuracy= 0.9761999845504761
```

Prediction

```
prediction = model.predict_classes(X_test)
prediction
```

```
array([7, 2, 1, ..., 4, 5, 6], dtype=int64)
```


Self Study Guide

Self Study Guide

References

- ✓ Chollet, F. (2018). *Deep Learning with Python* (1st ed.). Shelter Island, NY: Manning.
- ✓ Buduma, N. (2017). *Fundamentals of Deep Learning: Designing Next-Generation Machine Intelligence Algorithms* (1st ed.). Sebastopol, CA: O'Reilly.

Useful Sites

- ✓ https://www.javatpoint.com/artificial-neural-network
- ✓ https://medium.com/@sprhlabs/understanding-deep-learning-dnn-rnn-lstm-cnn-and-r-cnn-6602ed94dbff