UNIT-2

Introduction to Keras

- Keras is a high-level neural networks library, written in Python and capable of running on top of either Tensor Flow or Theano.
- It was developed with a focus on enabling fast experimentation.
- The core data structure of Keras is a **model**, a way to organize layers.
- The main type of model is the Sequential model, a linear stack of layers.
- For more complex architectures, use the Keras functional API.
- Here's the Sequential model is given as:

```
from keras.models import Sequential

model = Sequential()
```

Benefits of Keras

- Keras leverages various optimization techniques to make high level neural network API easier and more performant.
- It supports the following features –
  - Consistent, simple and extensible API.
  - Minimal structure - easy to achieve the result without any frills.
  - It supports multiple platforms and backends.
  - It is user friendly framework which runs on both CPU and GPU.
  - Highly scalability of computation.

Installation of Keras

- Step 1: Install Python
- Step 2: Now, Open the Command Prompt
- Step 3: Now, type ‘pip’ in Command Prompt: It will help you to check whether Python is installed or not.
- Step 4: Write ‘pip install tensorflow’ in Command Prompt: Being the fact that Keras runs on the top of Keras.
- Step 5: Write ‘pip install keras’ on Command Prompt

Overview of Deep learning

- Deep learning is an evolving subfield of machine learning. Deep learning involves analyzing the input in layer by layer manner, where each layer progressively extracts higher level information about the input.
- Let us take a simple scenario of analyzing an image. Let us assume that your input image is divided up into a rectangular grid of pixels.
  - Now, the first layer extracts the pixels.
  - The second layer understands the edges in the image.
  - The Next layer constructs nodes from the edges.
  - Then, the next would find branches from the nodes.
  - Finally, the output layer will detect the full object.
- Here, the feature extraction process goes from the output of one layer into the input of the next subsequent layer.
- By using this approach, we can process huge amount of features, which makes deep learning a very powerful tool.
- Deep learning algorithms are also useful for the analysis of unstructured data.

Convolutional Neural Network (CNN)

- Convolutional neural network is one of the most popular ANN.
- It is widely used in the fields of image and video recognition.
- It is based on the concept of convolution, a mathematical concept.
- It is almost similar to multi-layer perceptron except it contains series of convolution layer and pooling layer before the fully connected hidden neuron layer.
- It has three important layers:
  - Convolution layer: It is the primary building block and perform computational tasks based on convolution function.
  - Pooling layer: It is arranged next to convolution layer and is used to reduce the size of inputs by removing unnecessary information so computation can be performed faster.
  - Fully connected layer: It is arranged to next to series of convolution and pooling layer and classify input into various categories.

Recurrent Neural Network (RNN)

- Recurrent Neural Networks (RNN) are useful to address the flaw in other ANN models.
- Well, Most of the ANN doesn't remember the steps from previous situations and learned to make decisions based on context in training.
- Meanwhile, RNN stores the past information and all its decisions are taken from what it has learnt from the past.
- This approach is mainly useful in image classification.
- In this case bidirectional RNN is helpful to learn from the past and predict the future.
- For example, we have handwritten samples in multiple inputs.
  - Suppose, we have confusion in one input then we need to check again other inputs to recognize the correct context which takes the decision from the past.

Keras Layers and Models

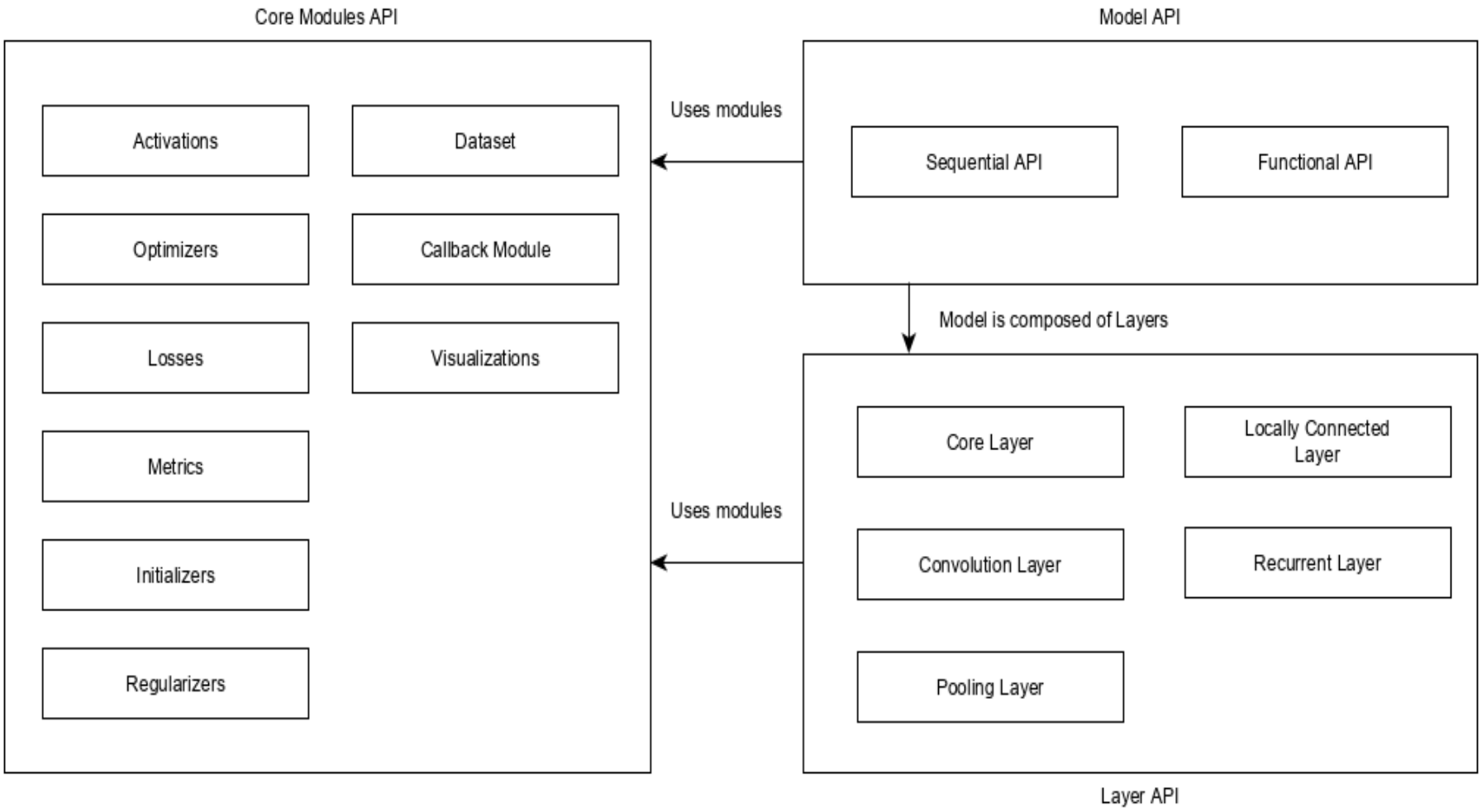
Architecture of Keras

- Keras Application programming interface (API) can be divided into three main categories:
  - Model
  - Layer
  - Core Modules

Cont.

- In Keras, every ANN is represented by Keras Models.
- In turn, every Keras Model is composition of Keras Layers and represents ANN layers like input, hidden layer, output layers, convolution layer, pooling layer, etc.
- Keras model and layer access Keras modules for activation function (transformation), loss function (difference between predicted and actual), regularization function (prevent overfitting and underfitting), etc.
- Using Keras model, Keras Layer, and Keras modules, any ANN algorithm (CNN, RNN, etc.,) can be represented in a simple and efficient manner.

Keras_model


Core Modules API
Activations
Dataset
Optimizers
Callback Module
Losses
Visualizations
Metrics
Initializers
Regularizers
Uses modules
Uses modules
Model API
Sequential API
Functional API
Model is composed of Layers
Core Layer
Locally Connected Layer
Convolution Layer
Recurrent Layer
Pooling Layer
Layer API

Model

- Sequential Model - Sequential model is basically a linear composition of Keras Layers. Sequential model is easy, minimal as well as has the ability to represent nearly all available neural networks.
- Functional API: Functional API is basically used to create complex models.

The Sequential model

- A Sequential model is appropriate for **a plain stack of layers** where each layer has **exactly one input tensor and one output tensor**.

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

```
# Define Sequential model with 3 layers
model = keras.Sequential(
    [
        layers.Dense(2, activation="relu", name="layer1"),
        layers.Dense(3, activation="relu", name="layer2"),
        layers.Dense(4, name="layer3"),
    ]
)
# Call model on a test input
x = tf.ones((3, 3))
y = model(x)
```

- Equivalent to:

```
# Create 3 layers
layer1 = layers.Dense(2, activation="relu", name="layer1")
layer2 = layers.Dense(3, activation="relu", name="layer2")
layer3 = layers.Dense(4, name="layer3")

# Call layers on a test input
x = tf.ones((3, 3))
y = layer3(layer2(layer1(x)))
```

- A Sequential model is **not appropriate** when:
  - Your model has multiple inputs or multiple outputs
  - Any of your layers has multiple inputs or multiple outputs
  - You need to do layer sharing
  - You want non-linear topology

- There's also a corresponding pop() method to remove layers: a Sequential model behaves very much like a list of layers.

```
model.pop()
print(len(model.layers))  # 2
```

- Sequential constructor accepts a name argument, just like any layer or model in Keras. This is useful to annotate TensorBoard graphs with semantically meaningful names.

```
model = keras.Sequential(name="my_sequential")
model.add(layers.Dense(2, activation="relu", name="layer1"))
model.add(layers.Dense(3, activation="relu", name="layer2"))
model.add(layers.Dense(4, name="layer3"))
```

Specifying the input shape in advance

- Generally, all layers in Keras need to know the shape of their inputs in order to be able to create their weights. So when you create a layer like this, initially, it has no weights:

```
layer = layers.Dense(3)
layer.weights  # Empty
```

- It creates its weights the first time it is called on an input, since the shape of the weights depends on the shape of the inputs:

```
# Call layer on a test input
x = tf.ones((1, 4))
y = layer(x)
layer.weights  # Now it has weights, of shape (4, 3) and (3,)
```

- Naturally, this also applies to Sequential models. When you instantiate a Sequential model without an input shape, it isn't "built": it has no weights (and calling model.weights results in an error stating just this). The weights are created when the model first sees some input data:

```
model = keras.Sequential(
    [
        layers.Dense(2, activation="relu"),
        layers.Dense(3, activation="relu"),
        layers.Dense(4),
    ]
)  # No weights at this stage!

# At this point, you can't do this:
# model.weights

# You also can't do this:
# model.summary()

# Call the model on a test input
x = tf.ones((1, 4))
y = model(x)
print("Number of weights after calling the model:", len(model.weights))  # 6
```

- Once a model is "built", you can call its summary() method to display its contents:

- model.summary()

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_7 (Dense)              multiple                  10
_________________________________________________________________
dense_8 (Dense)              multiple                  9
_________________________________________________________________
dense_9 (Dense)              multiple                  16
=================================================================
Total params: 35
Trainable params: 35
Non-trainable params: 0
```

- However, it can be very useful when building a Sequential model incrementally to be able to display the summary of the model so far, including the current output shape. In this case, you should start your model by passing an Input object to your model, so that it knows its input shape from the start:

```
model = keras.Sequential()
model.add(keras.Input(shape=(4,)))
model.add(layers.Dense(2, activation="relu"))

model.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_10 (Dense)             (None, 2)                 10
=================================================================
Total params: 10
Trainable params: 10
Non-trainable params: 0
_________________________________________________________________
```

- Note that the Input object is not displayed as part of model.layers, since it isn't a layer:
  - model.layers
- A simple alternative is to just pass an input_shape argument to your first layer:

```
model = keras.Sequential()
model.add(layers.Dense(2, activation="relu", input_shape=(4,)))

model.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_11 (Dense)             (None, 2)                 10
=================================================================
Total params: 10
Trainable params: 10
Non-trainable params: 0
_________________________________________________________________
```

- Models built with a predefined input shape like this always have weights (even before seeing any data) and always have a defined output shape.
- In general, it's a recommended best practice to always specify the input shape of a Sequential model in advance if you know what it is.

A common debugging workflow: add() + summary()

- When building a new Sequential architecture, it's useful to incrementally stack layers with add() and frequently print model summaries. For instance, this enables you to monitor how a stack of Conv2D and MaxPooling2D layers is downsampling image feature maps:

```
model = keras.Sequential()
model.add(keras.Input(shape=(250, 250, 3)))  # 250x250 RGB images
model.add(layers.Conv2D(32, 5, strides=2, activation="relu"))
model.add(layers.Conv2D(32, 3, activation="relu"))
model.add(layers.MaxPooling2D(3))

# Can you guess what the current output shape is at this point? Probably not.
# Let's just print it:
model.summary()

# The answer was: (40, 40, 32), so we can keep downsampling...

model.add(layers.Conv2D(32, 3, activation="relu"))
model.add(layers.Conv2D(32, 3, activation="relu"))
model.add(layers.MaxPooling2D(3))
model.add(layers.Conv2D(32, 3, activation="relu"))
model.add(layers.Conv2D(32, 3, activation="relu"))
model.add(layers.MaxPooling2D(2))

# And now?
model.summary()

# Now that we have 4x4 feature maps, time to apply global max pooling.
model.add(layers.GlobalMaxPooling2D())

# Finally, we add a classification layer.
model.add(layers.Dense(10))
```

What to do once you have a model

- Once your model architecture is ready, you will want to:
  - Train your model, evaluate it, and run inference.
  - Save your model to disk and restore it.
  - Speed up model training by leveraging multiple GPUs.

Feature extraction with a Sequential model

- Once a Sequential model has been built, it behaves like a Functional API model. This means that every layer has an input and output attribute. These attributes can be used to do neat things, like quickly creating a model that extracts the outputs of all intermediate layers in a Sequential model:

```
initial_model = keras.Sequential(
    [
        keras.Input(shape=(250, 250, 3)),
        layers.Conv2D(32, 5, strides=2, activation="relu"),
        layers.Conv2D(32, 3, activation="relu"),
        layers.Conv2D(32, 3, activation="relu"),
    ]
)
feature_extractor = keras.Model(
    inputs=initial_model.inputs,
    outputs=[layer.output for layer in initial_model.layers],
)

# Call feature extractor on test input.
x = tf.ones((1, 250, 250, 3))
features = feature_extractor(x)
```

- Here's a similar example that only extract features from one layer:

```
initial_model = keras.Sequential(
    [
        keras.Input(shape=(250, 250, 3)),
        layers.Conv2D(32, 5, strides=2, activation="relu"),
        layers.Conv2D(32, 3, activation="relu", name="my_intermediate_layer"),
        layers.Conv2D(32, 3, activation="relu"),
    ]
)
feature_extractor = keras.Model(
    inputs=initial_model.inputs,
    outputs=initial_model.get_layer(name="my_intermediate_layer").output,
)
# Call feature extractor on test input.
x = tf.ones((1, 250, 250, 3))
features = feature_extractor(x)
```

Transfer learning & fine-tuning

- **Transfer learning** consists of taking features learned on one problem, and leveraging them on a new, similar problem.
- For instance, features from a model that has learned to identify racoons may be useful to kick-start a model meant to identify tanukis.
- Transfer learning is usually done for tasks where your dataset has too little data to train a full-scale model from scratch.
- The most common incarnation of transfer learning in the context of deep learning is the following worfklow:
  - Take layers from a previously trained model.
  - Freeze them, so as to avoid destroying any of the information they contain during future training rounds.
  - Add some new, trainable layers on top of the frozen layers. They will learn to turn the old features into predictions on a new dataset.
  - Train the new layers on your dataset.
- A last, optional step, is **fine-tuning**, which consists of unfreezing the entire model you obtained above (or part of it), and re-training it on the new data with a very low learning rate.
- This can potentially achieve meaningful improvements, by incrementally adapting the pretrained features to the new data.

Freezing layers: understanding the trainable attribute

- Layers & models have three weight attributes:
  - weights is the list of all weights variables of the layer.
  - trainable_weights is the list of those that are meant to be updated (via gradient descent) to minimize the loss during training.
  - non_trainable_weights is the list of those that aren't meant to be trained. Typically they are updated by the model during the forward pass.

The typical transfer-learning workflow

- This leads us to how a typical transfer learning workflow can be implemented in Keras:
  - Instantiate a base model and load pre-trained weights into it.
  - Freeze all layers in the base model by setting trainable = False.
  - Create a new model on top of the output of one (or several) layers from the base model.
  - Train your new model on your new dataset.
- Note that an alternative, more lightweight workflow could also be:
  - Instantiate a base model and load pre-trained weights into it.
  - Run your new dataset through it and record the output of one (or several) layers from the base model. This is called **feature extraction**.
  - Use that output as input data for a new, smaller model.
- A key advantage of that second workflow is that you only run the base model once one your data, rather than once per epoch of training. So it's a lot faster & cheaper.
- An issue with that second workflow, though, is that it doesn't allow you to dynamically modify the input data of your new model during training, which is required when doing data augmentation, for instance.

Transfer learning with a Sequential model

- First, let's say that you have a Sequential model, and you want to freeze all layers except the last one. In this case, you would simply iterate over model.layers and set layer.trainable = False on each layer, except the last one.

```
model = keras.Sequential([
    keras.Input(shape=(784))
    layers.Dense(32, activation='relu'),
    layers.Dense(32, activation='relu'),
    layers.Dense(32, activation='relu'),
    layers.Dense(10),
])

# Presumably you would want to first load pre-trained weights.
model.load_weights(...)

# Freeze all layers except the last one.
for layer in model.layers[:-1]:
  layer.trainable = False

# Recompile and train (this will only update the weights of the last layer).
model.compile(...)
model.fit(...)
```

- Another common blueprint is to use a Sequential model to stack a pre-trained model and some freshly initialized classification layers.

```
# Load a convolutional base with pre-trained weights
base_model = keras.applications.Xception(
    weights='imagenet',
    include_top=False,
    pooling='avg')

# Freeze the base model
base_model.trainable = False

# Use a Sequential model to add a trainable classifier on top
model = keras.Sequential([
    base_model,
    layers.Dense(1000),
])

# Compile & train
model.compile(...)
model.fit(...)
```

Image Classification: Keras

Basic classification: Classify images of clothing

- Trains a neural network model to classify images of clothing, like sneakers and shirts.

```
# TensorFlow and tf.keras
import tensorflow as tf
from tensorflow import keras

# Helper libraries
import numpy as np
import matplotlib.pyplot as plt

print(tf.__version__)
```

- **Import the Fashion MNIST dataset**

```
fashion_mnist = keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
```

- The images are 28x28 NumPy arrays, with pixel values ranging from 0 to 255. The *labels* are an array of integers, ranging from 0 to 9. These correspond to the *class* of clothing the image represents:

Label	Class
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

- Each image is mapped to a single label. Since the *class names* are not included with the dataset, store them here to use later when plotting the images:

```
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

- **Explore the data**:
  - Let's explore the format of the dataset before training the model. The following shows there are 60,000 images in the training set, with each image represented as 28 x 28 pixels:

```
train_images.shape
```

```
(60000, 28, 28)
```

  - Likewise, there are 60,000 labels in the training set:

```
len(train_labels)
```

```
60000
```

  - Each label is an integer between 0 and 9:

```
train_labels
```

```
array([9, 0, 0, ..., 3, 0, 5], dtype=uint8)
```

  - There are 10,000 images in the test set. Again, each image is represented as 28 x 28 pixels:

```
test_images.shape
```

```
(10000, 28, 28)
```

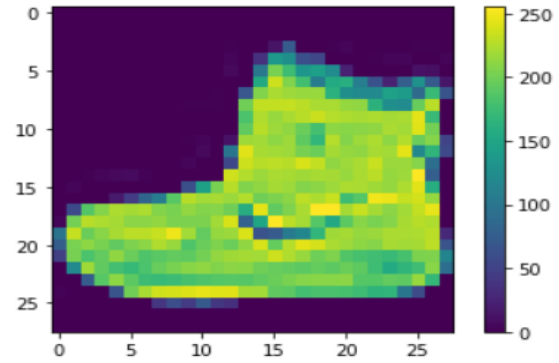
  - And the test set contains 10,000 images labels:

```
len(test_labels)
```

Preprocess the data

- The data must be pre-processed before training the network. If you inspect the first image in the training set, you will see that the pixel values fall in the range of 0 to 255:

```
plt.figure()
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
plt.show()
```

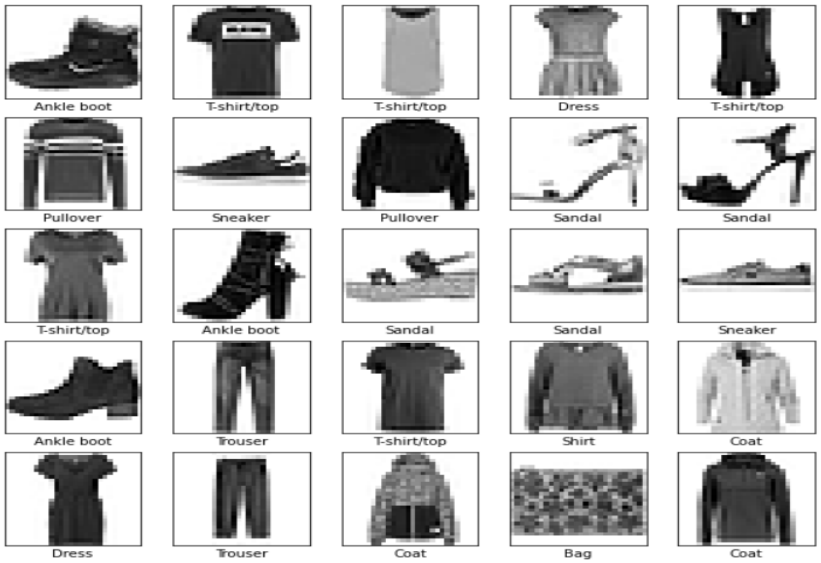


- Scale these values to a range of 0 to 1 before feeding them to the neural network model. To do so, divide the values by 255. It's important that the *training set* and the *testing set* be preprocessed in the same way:

```
train_images = train_images / 255.0

test_images = test_images / 255.0
```

- To verify that the data is in the correct format and that you're ready to build and train the network, let's display the first 25 images from the *training set* and display the class name below each image.

```
plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.show()
```



Build the model

- Building the neural network requires configuring the layers of the model, then compiling the model.
- The basic building block of a neural network is the *layer*. Layers extract representations from the data fed into them. Hopefully, these representations are meaningful for the problem at hand.
- Most of deep learning consists of chaining together simple layers. Most layers, such as tf.keras.layers.Dense, have parameters that are learned during training.

```
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dense(10)
])
```

- The first layer in this network, tf.keras.layers.Flatten, transforms the format of the images from a two-dimensional array (of 28 by 28 pixels) to a one-dimensional array (of 28 * 28 = 784 pixels).
- Think of this layer as unstacking rows of pixels in the image and lining them up.
- This layer has no parameters to learn; it only reformats the data.
- After the pixels are flattened, the network consists of a sequence of two tf.keras.layers.Dense layers.
- These are densely connected, or fully connected, neural layers.
- The first Dense layer has 128 nodes (or neurons).
- The second (and last) layer returns a logits array with length of 10.
- Each node contains a score that indicates the current image belongs to one of the 10 classes.

Compile the model

- Before the model is ready for training, it needs a few more settings. These are added during the model's *compile* step:
  - *Loss function* —This measures how accurate the model is during training. You want to minimize this function to "steer" the model in the right direction.
  - *Optimizer* —This is how the model is updated based on the data it sees and its loss function.
  - *Metrics* —Used to monitor the training and testing steps. The following example uses *accuracy*, the fraction of the images that are correctly classified.

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

Many built-in optimizers, losses, and metrics are available

- In general, you won't have to create from scratch your own losses, metrics, or optimizers, because what you need is likely already part of the Keras API:
- Optimizers:
  - SGD(), RMSprop, Adam()
- Losses:
  - MeanSquaredError(), KLDivergence(), CosineSimilarity()
- Metrics:
  - AUC(), Precision(), Recall()

Train the model

- Training the neural network model requires the following steps:
  - Feed the training data to the model. In this example, the training data is in the train_images and train_labels arrays.
  - The model learns to associate images and labels.
  - You ask the model to make predictions about a test set—in this example, the test_images array.
  - Verify that the predictions match the labels from the test_labels array.
- **Feed the model**
  - To start training, call the model.fit method—so called because it "fits" the model to the training data:

```
model.fit(train_images, train_labels, epochs=10)
```

Evaluate accuracy

- Next, compare how the model performs on the test dataset:

```
test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)

print('\nTest accuracy:', test_acc)
```

```
313/313 - 0s - loss: 0.3272 - accuracy: 0.8841

Test accuracy: 0.8841000199317932
```

- It turns out that the accuracy on the test dataset is a little less than the accuracy on the training dataset.
- This gap between training accuracy and test accuracy represents *overfitting*.
- Overfitting happens when a machine learning model performs worse on new, previously unseen inputs than it does on the training data.
- An overfitted model "memorizes" the noise and details in the training dataset to a point where it negatively impacts the performance of the model on the new data.

Make predictions

- With the model trained, you can use it to make predictions about some images.
- The model's linear outputs, logits. Attach a softmax layer to convert the logits to probabilities, which are easier to interpret.

```
probability_model = tf.keras.Sequential([model,
                                         tf.keras.layers.Softmax()])
```

```
predictions = probability_model.predict(test_images)
```

- Here, the model has predicted the label for each image in the testing set. Let's take a look at the first prediction:

```
predictions[0]
```

```
array([2.5733272e-07, 1.3352397e-09, 3.0229703e-09, 2.1916051e-09,
       9.3775823e-09, 1.2524406e-03, 3.0483918e-07, 6.3156984e-03,
       3.5332143e-08, 9.9243128e-01], dtype=float32)
```

- A prediction is an array of 10 numbers. They represent the model's "confidence" that the image corresponds to each of the 10 different articles of clothing. You can see which label has the highest confidence value:

```
np.argmax(predictions[0])
```

```
9
```

- So, the model is most confident that this image is an ankle boot, or class_names[9]. Examining the test label shows that this classification is correct:

```
test_labels[0]
```

```
9
```

Now, to look at the full set of 10 class predictions.

```
def plot_image(i, predictions_array, true_label, img):
  true_label, img = true_label[i], img[i]
  plt.grid(False)
  plt.xticks([])
  plt.yticks([])

  plt.imshow(img, cmap=plt.cm.binary)

  predicted_label = np.argmax(predictions_array)
  if predicted_label == true_label:
    color = 'blue'
  else:
    color = 'red'

  plt.xlabel("{} {:2.0f}% ({})".format(class_names[predicted_label],
                                100*np.max(predictions_array),
                                class_names[true_label]),
                                color=color)

def plot_value_array(i, predictions_array, true_label):
  true_label = true_label[i]
  plt.grid(False)
  plt.xticks(range(10))
  plt.yticks([])
  thisplot = plt.bar(range(10), predictions_array, color="#777777")
  plt.ylim([0, 1])
  predicted_label = np.argmax(predictions_array)

  thisplot[predicted_label].set_color('red')
  thisplot[true_label].set_color('blue')
```

Verify predictions

- With the model trained, you can use it to make predictions about some images.
- Let's look at the 0th image, predictions, and prediction array. Correct prediction labels are blue and incorrect prediction labels are red. The number gives the percentage (out of 100) for the predicted label

```
i = 0
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions[i], test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions[i],  test_labels)
plt.show()
```

```
i = 12
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions[i], test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions[i],  test_labels)
plt.show()
```

Use the trained model

- Finally, use the trained model to make a prediction about a single image.

```
# Grab an image from the test dataset.
img = test_images[1]

print(img.shape)
```

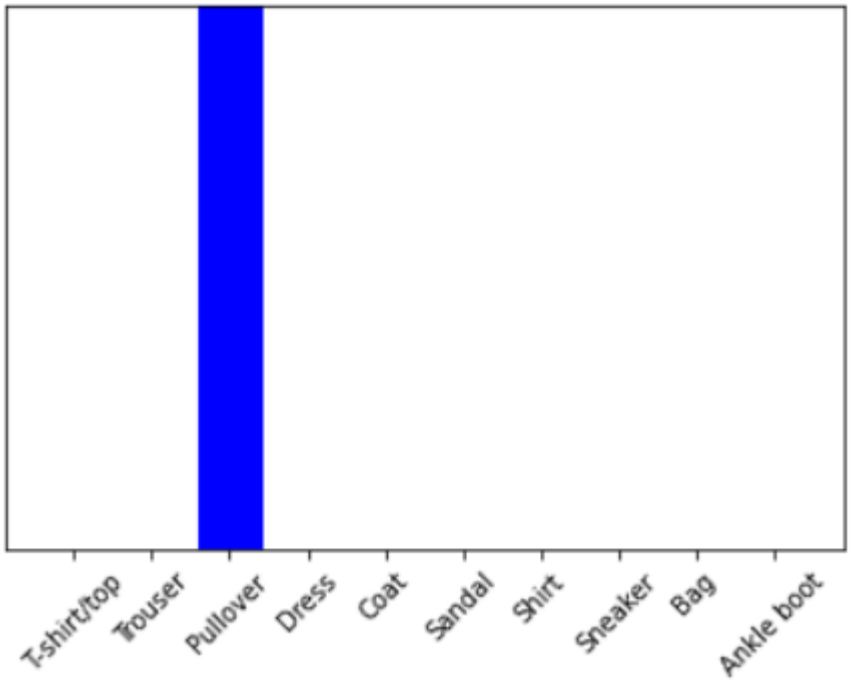
- tf.keras models are optimized to make predictions on a *batch*, or collection, of examples at once. Accordingly, even though you're using a single image, you need to add it to a list:

```
# Add the image to a batch where it's the only member.
img = (np.expand_dims(img,0))

print(img.shape)
```

- Now predict the correct label for this image:

```
predictions_single = probability_model.predict(img)

print(predictions_single)
```

```
plot_value_array(1, predictions_single[0], test_labels)
_ = plt.xticks(range(10), class_names, rotation=45)
```



- keras.Model.predict returns a list of lists—one list for each image in the batch of data. Grab the predictions for our (only) image in the batch:

```
np.argmax(predictions_single[0])
```

```
2
```

- And the model predicts a label as expected.

Explore the following dataset

"https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz"

Explore MNIST handwritten Images

Text classification in keras

```
import tensorflow.keras as tf
import matplotlib.pyplot as plt
import numpy as np

mnist = tf.datasets.mnist

(xtrain,ytrain),(xtest,ytest) = mnist.load_data()

plt.imshow(xtrain[59999],cmap='gray')
plt.show()

ytrain[59999]
```

```
# Create neural network
model = tf.models.Sequential()    ### empty neural network
model.add(tf.layers.Flatten())      ### input layer
model.add(tf.layers.Dense(784,activation="relu"))    ## hidde
n layer
model.add(tf.layers.Dense(10,activation="softmax"))    ### ou
tput layer
model.compile(loss='sparse_categorical_crossentropy',optimiz
er='adam',metrics=["accuracy"])
```

```
### Scale the data
xtrain = xtrain/255
xtest = xtest/255

## Train the model
model.fit(xtrain,ytrain,epochs=20)

ypred = model.predict(xtest)

ytest[23]

ypred[23]

ypred[23].argmax()

ytest[44]

ypred[44].argmax()

model.evaluate(xtest,ytest, verbose=2)
```

```
import cv2
img = cv2.imread("/content/2.png",0)
img = cv2.bitwise_not(img)
img = cv2.resize(img,(28,28))
img = img/255
plt.imshow(img,cmap='gray')
plt.show()


img.shape

model.predict(np.array([[img]])).argmax()
```

Implementing linear regression

1. **load the required libraries.**

```
from sklearn.datasets import load_boston
from keras.models import Sequential
from keras.layers import Dense, Conv1D, Flatten
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
```

2. Preparing the data

We can use the Boston housing dataset as target regression data. First, we'll load the dataset and check the data dimensions of both x and y.

```
boston = load_boston()
x, y = boston.data, boston.target
print(x.shape)
```

An x data has two dimensions that are the number of rows and columns. Here, we need to add the third dimension that will be the number of the single input row. In our example, it becomes 1 that is [13, 1]. We'll reshape the x data accordingly.

```
x = x.reshape(x.shape[0], x.shape[1], 1)
print(x.shape)
(506, 13, 1)
```

Next, we'll split the data into the train and test parts.

```
xtrain, xtest, ytrain, ytest=train_test_split(x, y, test_size=0.15)
```

Next, we'll split the data into the train and test parts.

```
xtrain, xtest, ytrain, ytest=train_test_split(x, y, test_size=0.15)
```

3. Defining and fitting the model

We'll define the Keras sequential model and add a one-dimensional convolutional layer. Input shape becomes as it is defined above (13,1). We'll add Flatten and Dense layers and compile it with optimizers.

```
model = Sequential()
model.add(Conv1D(32, 2, activation="relu", input_shape=(13, 1)))
model.add(Flatten())
model.add(Dense(64, activation="relu"))
model.add(Dense(1))
model.compile(loss="mse", optimizer="adam")
model.summary()
```

Next, we'll fit the model with train data.

```
model.fit(xtrain, ytrain,
batch_size=12,epochs=200, verbose=0)
```

4. Predicting and visualizing the results

Now we can predict the test data with the trained model.

```
ypred = model.predict(xtest)
```

We can evaluate the model, check the mean squared error rate (MSE) of the predicted result, and visualize the result in a plot.

```
print(model.evaluate(xtrain, ytrain))
print("MSE: %.4f" % mean_squared_error(ytest, ypred))

x_ax = range(len(ypred))
plt.scatter(x_ax, ytest, s=5, color="blue", label="original")
plt.plot(x_ax, ypred, lw=0.8, color="red", label="predicted")
plt.legend()
plt.show()
```

Overfitting and Underfitting

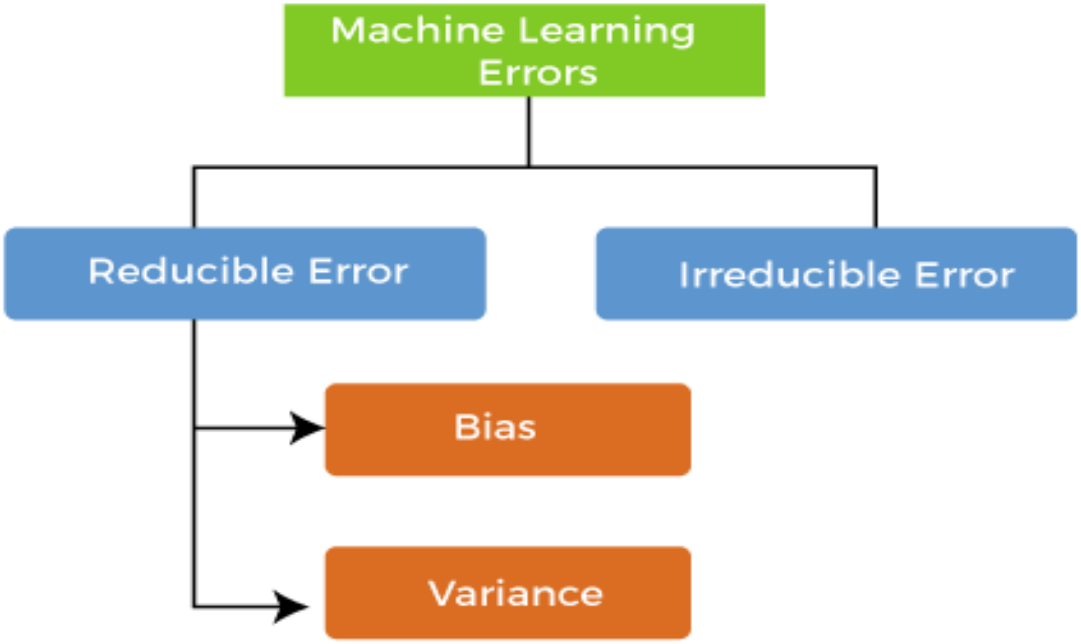
- Machine learning is a branch of Artificial Intelligence, which allows machines to perform data analysis and make predictions.
- However, if the machine learning model is not accurate, it can make predictions errors, and these prediction errors are usually known as **Bias** and **Variance**.
- In machine learning, these errors will always be present as there is always a slight difference between the model predictions and actual predictions. The main aim of ML/data science analysts is to reduce these errors in order to get more accurate results.

Errors in Machine Learning

- In machine learning, an error is a measure of how accurately an algorithm can make predictions for the previously unknown dataset.
- On the basis of these errors, the machine learning model is selected that can perform best on the particular dataset. There are mainly two types of errors in machine learning, which are:

Reducible Errors: These errors can be reduced to improve the model accuracy. Such errors can further be classified into bias and Variance.

Irreducible Errors: These errors will always be present in the model.



Bias

- In general, a machine learning model analyses the data, find patterns in it and make predictions. While training, the model learns these patterns in the dataset and applies them to test data for prediction.
- **Bias:** Assumptions made by a model to make a function easier to learn. It is actually the error rate of the training data. When the error rate has a high value, we call it High Bias and when the error rate has a low value, we call it low Bias.
- It can be defined as an inability of machine learning algorithms such as Linear Regression to capture the true relationship between the data points. Each algorithm begins with some amount of bias because bias occurs from assumptions in the model, which makes the target function simple to learn.

A model has either:

- **Low Bias:** A low bias model will make fewer assumptions about the form of the target function.
- **High Bias:** A model with a high bias makes more assumptions, and the model becomes unable to capture the important features of our dataset. **A high bias model also cannot perform well on new data.**

- Some examples of machine learning algorithms with low bias **are Decision Trees, k-Nearest Neighbors and Support Vector Machines**.
- At the same time, an algorithm with high bias is **Linear Regression, Linear Discriminant Analysis and Logistic Regression.**

Ways to reduce High Bias:

- High bias mainly occurs due to a much simple model. Below are some ways to reduce the high bias:
- Increase the input features as the model is under-fitted.
- Decrease the regularization term.
- Use more complex models, such as including some polynomial features.

Variance Error

- The variance would specify the amount of variation in the prediction if the different training data was used.
- **Variance:** The difference between the error rate of training data and testing data is called variance. If the difference is high then it's called high variance and when the difference of errors is low then it's called low variance. Usually, we want to make a low variance for generalized our model.
- Ideally, a model should not vary too much from one training dataset to another, which means the algorithm should be good in understanding the hidden mapping between inputs and output variables.

Variance errors are either of **low variance or high variance.**

- **Low variance** means there is a small variation in the prediction of the target function with changes in the training data set.
- At the same time, **High variance** shows a large variation in the prediction of the target function with changes in the training dataset.

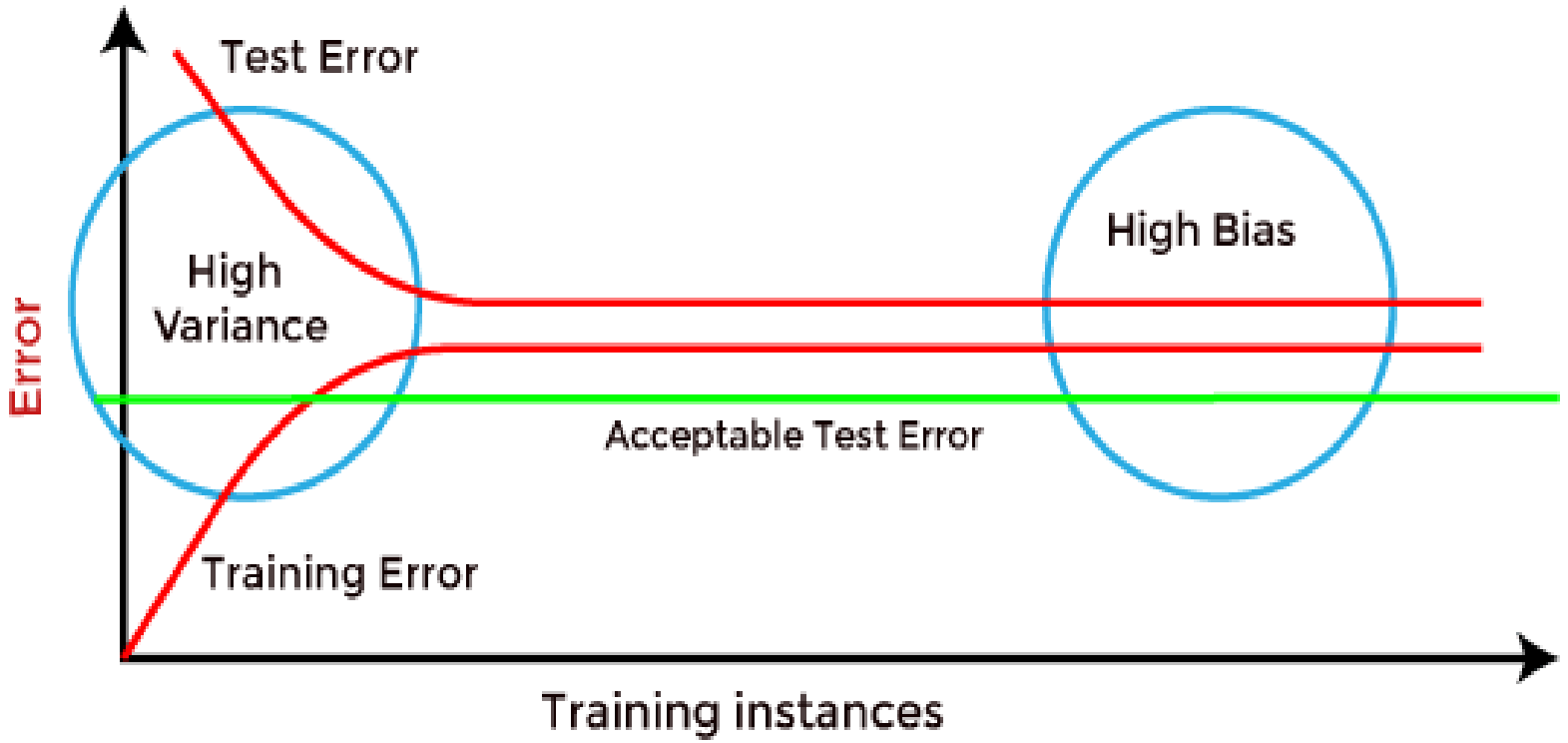
- A model that shows high variance learns a lot and perform well with the training dataset, and does not generalize well with the unseen dataset. As a result, such a model gives good results with the training dataset but shows high error rates on the test dataset.
- Since, with high variance, the model learns too much from the dataset, it leads to overfitting of the model. A model with high variance has the below problems:
- A high variance model leads to overfitting.
- Increase model complexities.
- Some examples of machine learning algorithms with low variance are, **Linear Regression, Logistic Regression, and Linear discriminant analysis**.
- At the same time, algorithms with high variance are **decision tree, Support Vector Machine, and K-nearest neighbors.**

Ways to Reduce High Variance:

- Reduce the input features or number of parameters as a model is over-fitted.
- Do not use a much complex model.
- Increase the training data.
- Increase the Regularization term.

- **Low-Bias, Low-Variance:** The combination of low bias and low variance shows an ideal machine learning model. However, it is not possible practically.
- **Low-Bias, High-Variance:** With low bias and high variance, model predictions are inconsistent and accurate on average. This case occurs when the model learns with a large number of parameters and hence leads to an **over-fitting**
- **High-Bias, Low-Variance:** With High bias and low variance, predictions are consistent but inaccurate on average. This case occurs when a model does not learn well with the training dataset or uses few numbers of the parameter. It leads to **under-fitting** problems in the model.
- **High-Bias, High-Variance:** With high bias and high variance, predictions are inconsistent and also inaccurate on average.

- High variance can be identified if the model has: **Low training error and high test error.**
- High Bias can be identified if the model has: **High training error and the test error is almost similar to training error.**



Underfitting

- A statistical model or a machine learning algorithm is said to have under-fitting when it cannot capture the underlying trend of the data, i.e., it only performs well on training data but performs poorly on testing data.
- Underfitting destroys the accuracy of our machine learning model. Its occurrence simply means that our model or the algorithm does not fit the data well enough. It usually happens when we have fewer data to build an accurate model and also when we try to build a linear model with fewer non-linear data.
- In such cases, the rules of the machine learning model are too easy and flexible to be applied to such minimal data and therefore the model will probably make a lot of wrong predictions. Underfitting can be avoided by using more data and also reducing the features by feature selection.

- Underfitting refers to a model that can neither performs well on the training data nor generalize to new data.

Reasons for Underfitting:

- High bias and low variance
- The size of the training dataset used is not enough.
- The model is too simple.
- Training data is not cleaned and also contains noise in it.

Techniques to reduce underfitting:

- Increase model complexity
- Increase the number of features, performing feature engineering
- Remove noise from the data.
- Increase the number of epochs or increase the duration of training to get better results.

Overfitting

- A statistical model is said to be overfitted when the model does not make accurate predictions on testing data. When a model gets trained with so much data, it starts learning from the noise and inaccurate data entries in our data set. And when testing with test data results in High variance. Then the model does not categorize the data correctly, because of too many details and noise.
- The causes of overfitting are the non-parametric and non-linear methods because these types of machine learning algorithms have more freedom in building the model based on the dataset and therefore they can really build unrealistic models.
- A solution to avoid overfitting is using a linear algorithm if we have linear data or using the parameters like the maximal depth if we are using decision trees.

Reasons for Overfitting are as follows:

- High variance and low bias
- The model is too complex
- The size of the training data

Techniques to reduce overfitting:

- Increase training data.
- Reduce model complexity.
- Early stopping during the training phase (have an eye over the loss over the training period as soon as loss begins to increase stop training).
- Ridge Regularization and Lasso Regularization
- Use dropout for neural networks to tackle overfitting.

Good Fit in a Statistical Model

- Ideally, the case when the model makes the predictions with 0 error, is said to have a *good fit* on the data.
- This situation is achievable at a spot between overfitting and underfitting.
- In order to understand it, we will have to look at the performance of our model with the passage of time, while it is learning from the training dataset.

Keras load/save model

There are three ways in which the model can be saved:

- Saving everything into a single file, usually it is in the Keras H5 format (An H5 file is a data file saved in the Hierarchical Data Format (HDF). It contains multidimensional arrays of scientific data).
- Saving the architecture as only a JSON or YAML file.
- Saving the weight values only. This is generally used when training the model so that they may be used again. These weights can be used in a similar representation of a different model.

- HDF5 file stands for **Hierarchical Data Format 5**. It is an open-source file which comes in handy to store large amount of data. As the name suggests, it stores data in a hierarchical structure within a single file.
- JSON (JavaScript Object Notation) is a popular data format used for representing structured data. It's common to transmit and receive data between a server and web application in JSON format. In Python, JSON exists as a string. For example: p = '{"name": "Bob", "languages": ["Python", "Java"]}'
- YAML (YAML Ain't Markup Language) is a human-readable data-serialization language. It is commonly used for configuration files, but it is also used in data storage (e.g. debugging output) or transmission (e.g. document headers).

Method 1

Saving the model

In order to save the model, a user can use the save() function specified in the Keras library. A user will specify the name and location of the file with .h5 extension. If you do not specify the path it will, by default, save it in the folder where the model is executed. Look at the syntax below:

```
model.save("myModel.h5")
```

The function save() saves the following features:

The architecture of the model.

- Weights of the model.
- Training configuration.
- State of the optimizer.

Loading the model

In order to load the model, a user can use the load_model() function specified in the keras.models library. The user will first have to import this function from the relevant library in order to use it. The user will specify the name and location of the file with an .h5 extension. Take a look at the syntax below:

```
from keras.models import load_model
model_new = load_model("myModel.h5")
```

Method 2

Saving only the architecture of the model

The architecture is saved as a JSON file or a YAML file. Look at the syntax below:

As a JSON:

```
from keras.models import model_from_json
save_model_json = model.to_json()
with open("myModel.json", "w") as json_file:
    json_file.write(save_model_json )
```

As a YAML:

```
from keras.models import model_from_yaml
model_yaml = model.to_yaml()
with open("myModel.yaml", "w") as yaml_file:
    yaml_file.write(model_yaml)
```

Loading the saved architecture of the model

As a JSON:

```
json_file = open('myModel.json', 'r')
loaded_model_json = json_file.read()
json_file.close()
new_model =
model_from_json(loaded_model_json)
```

As a YAML:

```
yaml_file = open('myModel.yaml', 'r')
loaded_model_yaml = yaml_file.read()
yaml_file.close()
new_model =
model_from_yaml(loaded_model_yaml)
```

Method 3

Saving only the weights of the model

model.save_weights("myModel.h5")

Loading the saved weights of the model

new_model.load_weights("myModel.h5")

Hyper parameter tuning

Model Design

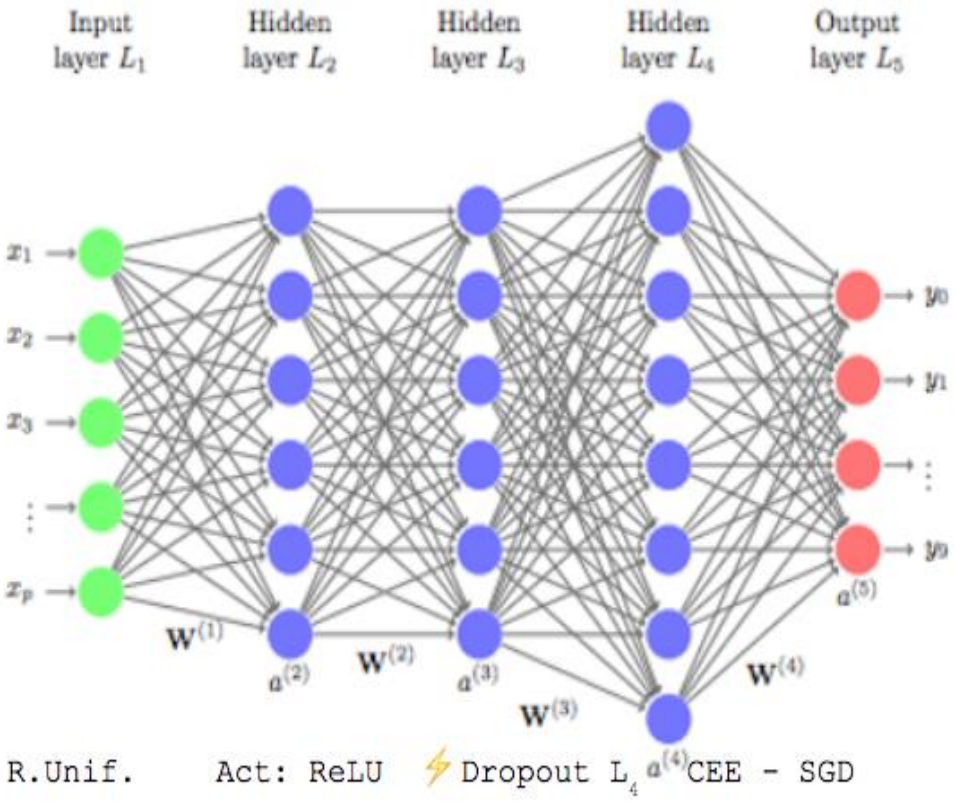
- Weight init.: Random Uniform
- Act.: ReLU
- Loss: CEE
- # Hidden Layers: 3
- # Units per layer {p, p+1, p+1, p+3, 10}
- Optimizer: SGD
- Dropout layer: L_4

Hyperparameters

- Learning rate
- Dropout Rate
- Batch size

Model Parameters

- $W^{(1)} \Rightarrow W^{(4)}$



Hyperparameters | **Parameters** | **Score**

n_layers = 3
n_neurons = 512
learning_rate = 0.1

Weights optimization

85%

n_layers = 3
n_neurons = 1024
learning_rate = 0.01

Weights optimization

80%

n_layers = 5
n_neurons = 256
learning_rate = 0.1

Weights optimization

92%

A hyperparameter is a machine learning parameter whose value is chosen before a learning algorithm is trained.

Examples of hyperparameters in machine learning include:

- Model architecture.
- Learning rate.
- Number of epochs.
- Number of branches in a decision tree.
- Number of clusters in a clustering algorithm.

Optimal Hyperparameters: Hyperparameters control the over-fitting and under-fitting of the model. Optimal hyperparameters often differ for different datasets. To get the best hyperparameters the following steps are followed:

1. For each proposed hyperparameter setting the model is evaluated

2. The hyperparameters that give the best model are selected.

Hyperparameters Search: Grid search picks out a grid of hyperparameter values and evaluates all of them. Guesswork is necessary to specify the min and max values for each hyperparameter. Random search randomly values a random sample of points on the grid. It is more efficient than grid search. Smart hyperparameter tuning picks a few hyperparameter settings, evaluates the validation matrices, adjusts the hyperparameters, and re-evaluates the validation matrices. Examples of smart hyper-parameter are Spearmint (hyperparameter optimization using Gaussian processes) and Hyperopt (hyperparameter optimization using Tree-based estimators).

Models can have many hyperparameters and finding the best combination of parameters can be treated as a search problem. The two best strategies for Hyperparameter tuning are:

- GridSearchCV
- RandomizedSearchCV

GridSearchCV: In GridSearchCV approach, the machine learning model is evaluated for a range of hyperparameter values. This approach is called GridSearchCV, because it searches for the best set of hyperparameters from a grid of hyperparameters values.

For example, if we want to set two hyperparameters C and Alpha of the Logistic Regression Classifier model, with different sets of values. The grid search technique will construct many versions of the model with all possible combinations of hyperparameters and will return the best one.

As in the image, for C = [0.1, 0.2, 0.3, 0.4, 0.5] and Alpha = [0.1, 0.2, 0.3, 0.4]. For a combination of C=0.3 and Alpha=0.2, the performance score comes out to be 0.726(Highest), therefore it is selected.

The following code illustrates how to use GridSearchCV

```
# Necessary imports
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV

# Creating the hyperparameter grid
c_space = np.logspace(-5, 8, 15)
param_grid = {'C': c_space}

# Instantiating logistic regression classifier
logreg = LogisticRegression()

# Instantiating the GridSearchCV object
logreg_cv = GridSearchCV(logreg, param_grid, cv = 5)

logreg_cv.fit(X, y)

# Print the tuned parameters and score
print("Tuned Logistic Regression Parameters: {}".format(logreg_cv.best_params_))
print("Best score is {}".format(logreg_cv.best_score_))
```

Drawback: GridSearchCV go through all the intermediate combinations of hyperparameters which makes grid search computationally very expensive.

RandomizedSearchCV

RandomizedSearchCV solves the drawbacks of GridSearchCV, as it goes through only a fixed number of hyperparameter settings. It moves within the grid in a random fashion to find the best set of hyperparameters. This approach reduces unnecessary computation.

The following code illustrates how to use RandomizedSearchCV

```
# Necessary imports
from scipy.stats import randint
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import RandomizedSearchCV

# Creating the hyperparameter grid
param_dist = {"max_depth": [3, None],
              "max_features": randint(1, 9),
              "min_samples_leaf": randint(1, 9),
              "criterion": ["gini", "entropy"]}

# Instantiating Decision Tree classifier
tree = DecisionTreeClassifier()
```

```
# Instantiating RandomizedSearchCV object
tree_cv = RandomizedSearchCV(tree, param_dist, cv = 5)

tree_cv.fit(X, y)

# Print the tuned parameters and score
print("Tuned Decision Tree Parameters: {}".format(tree_cv.best_params_))
print("Best score is {}".format(tree_cv.best_score_))
```

Thank you