***DEEP LEARNING COMPUTATIONS***



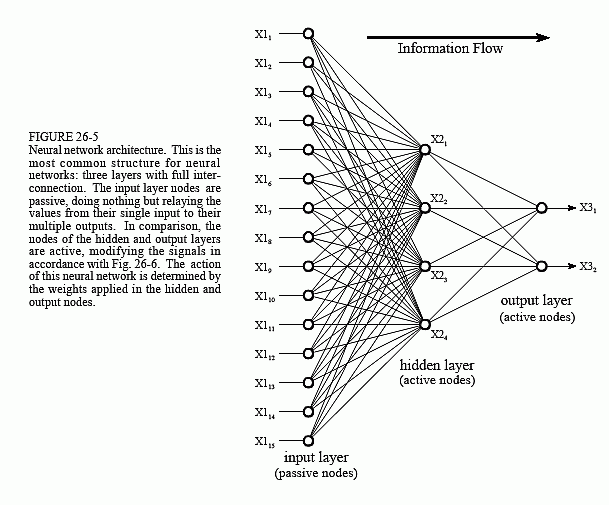
At a very basic level, **deep learning** is a machine learning technique. It teaches a computer to filter inputs through layers to learn how to predict and classify information. Observations can be in the form of images, text, or sound.

The inspiration for deep learning is the way that the human brain filters information. Its purpose is to mimic how the human brain works to create some real magic.

## **How do artificial neural networks learn?**

There are two different approaches to get a program to do what you want. First, there is the specifically guided and hard-programmed approach. In this approach, you tell the program exactly what you want it to do. Then there are **neural networks**. In neural networks, you tell your network the inputs and what you want for the outputs, and let it learn on its own. By allowing the network to learn on its own, we can avoid the necessity of entering in all the rules. For a neural network, you can create the architecture and then let it go and learn. Once it is trained up, you can give it a new image and it will be able to distinguish output.

**Layers in a Neural Network**



This neural network is formed in three layers, called the input layer, hidden layer, and output layer. Each layer consists of one or more nodes, represented in this diagram by the small circles. The lines between the nodes indicate the flow of information from one node to the next. In this neural network, the information flows only from the input to the output (that is, from left-to-right). Other types of neural networks have more intricate connections, such as feedback paths.

1. **SEQUENTIAL LAYER**

The Keras deep learning library helps to develop the neural network models fast and easy. There are two ways to create Keras model such as sequential and functional.

The sequential API develop the model layer-by-layer like a linear stack of layers. It seems to be very easy to build a network. But the sequential API has few limitations that it does not allow us to build models that share layers or have multiple inputs or outputs.

The functional API is an alternative way to build a neural network. It provides more flexibility to develop a very complex network with multiple inputs or outputs as well as a model that can share layers.

A close up of a logo

Description automatically generated

**Defining a Sequential model with three 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)

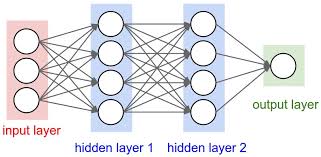
1. **DENSE LAYER**

The name suggests that layers are fully connected (dense) by the neurons in a network layer. Each neuron in a layer receives an input from all the neurons present in the previous layer—thus, they’re densely connected.

In other words, the dense layer is a fully connected layer, meaning all the neurons in a layer are connected to those in the next layer. A densely connected layer provides learning features from all the combinations of the features of the previous layer, whereas a convolutional layer relies on consistent features with a small repetitive field.

# Create a `Sequential` model and add a Dense layer as the first layer.  
model =

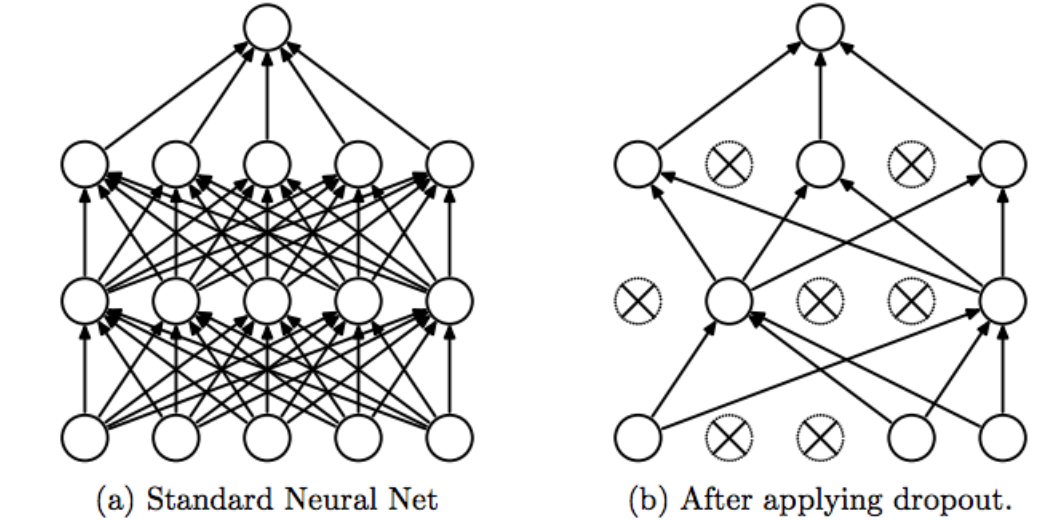
tf.keras.models.Sequential()  
model.add(tf.keras.Input(shape=(16,)))  
model.add(tf.keras.layers.Dense(32,activation='relu'))  
# Now the model will take as input arrays of shape (None, 16)  
#and output arrays of shape (None, 32).  
#Note that after the first layer, you don't need to specify  
# the size of the input anymore:  
model.add(tf.keras.layers.Dense(32))  
model.output\_shape



1. **DROPOUT LAYER**

Simply put, dropout refers to ignoring units (i.e. neurons) during the training phase of certain set of neurons which is chosen at random. By “ignoring”, means these units are not considered during a particular forward or backward pass. More technically, at each training stage, individual nodes are either dropped out of the net with probability 1-p or kept with probability p, so that a reduced network is left; incoming and outgoing edges to a dropped-out node are also removed.

A fully connected layer occupies most of the parameters, and hence, neurons develop co-dependency amongst each other during training which curbs the individual power of each neuron leading to over-fitting of training data. Hence dropout layers are used to prevent overfitting of the data.



# dropout in the input layer with weight constraint example

def create\_model():

# create model

model = Sequential()

model.add(Dropout(0.2, input\_shape=(60,)))

model.add(Dense(60,activation='relu', kernel\_constraint=maxnorm(3)))

model.add(Dense(30,activation='relu', kernel\_constraint=maxnorm(3)))

model.add(Dense(1, activation='sigmoid'))

# Compile model

sgd = SGD(lr=0.1, momentum=0.9)

model.compile(loss='binary\_crossentropy',optimizer=sgd,metrics=['accuracy'])

return model

1. **CONVOLUTION LAYER**

Convolutional layers are the major building blocks used in convolutional neural networks. A convolution is the simple application of a filter to an input that results in an activation. Repeated application of the same filter to an input results in a map of activations called a feature map, indicating the locations and strength of a detected feature in an input, such as an image.

The innovation of convolutional neural networks is the ability to automatically learn a large number of filters in parallel specific to a training dataset under the constraints of a specific predictive modeling problem, such as image classification. The result is highly specific features that can be detected anywhere on input images. Example in code are :

>>> # The inputs are 28x28x28 volumes with a single channel, and the

>>> # batch size is 4

>>> input\_shape =(4, 28, 28, 28, 1)

>>> x = tf.random.normal(input\_shape)

>>> y = tf.keras.layers.Conv3D(

... 2, 3, activation='relu', input\_shape=input\_shape[1:])(x)

>>> print(y.shape)

(4, 26, 26, 26, 2)

Diagram

Description automatically generated

1. **POOLING LAYER**

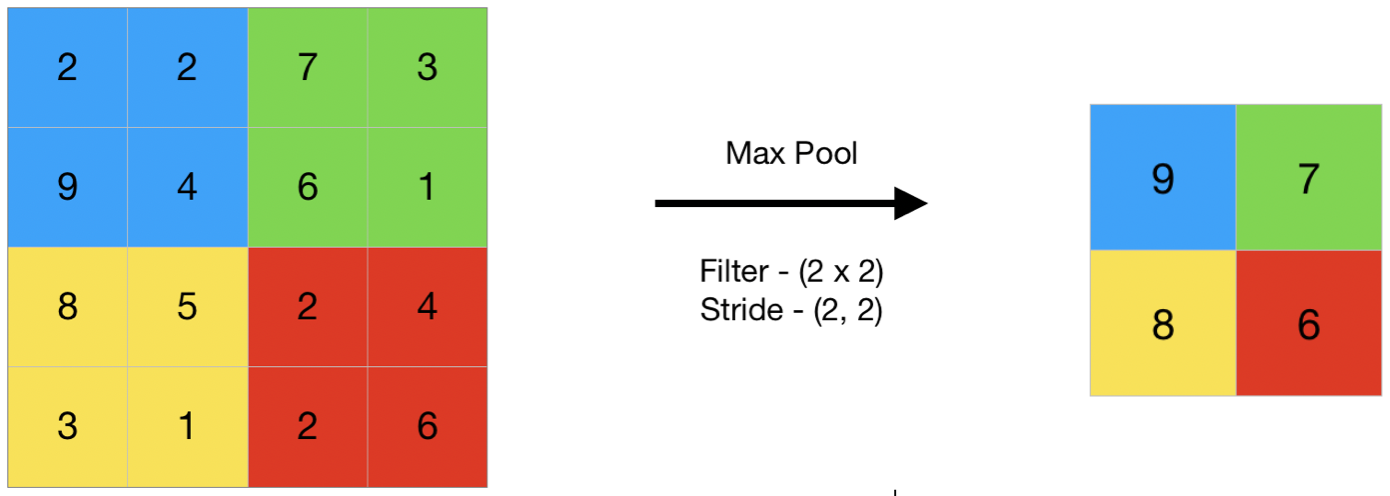
Pooling layers are used to reduce the dimensions of the feature maps. Thus, it reduces the number of parameters to learn and the amount of computation performed in the network.

The pooling operation involves sliding a two-dimensional filter over each channel of feature map and summarizing the features lying within the region covered by the filter. The pooling layer summarizes the features present in a region of the feature map generated by a convolution layer. So, further operations are performed on summarized features instead of precisely positioned features generated by the convolution layer. This makes the model more robust to variations in the position of the features in the input image.

**Types of Pooling Layers**

* **Max Pooling**

Max pooling is a pooling operation that selects the maximum element from the region of the feature map covered by the filter. Thus, the output after max-pooling layer would be a feature map containing the most prominent features of the previous feature map.



**Performing Max pooling using Keras**

import numpy as np

from keras.models import Sequential

from keras.layers import MaxPooling2D

# define input image

image = np.array([[2, 2, 7, 3],

[9, 4, 6, 1],

[8, 5, 2, 4],

[3, 1, 2, 6]])

image = image.reshape(1, 4, 4, 1)

# define model containing just a single max pooling layer

model = Sequential( [MaxPooling2D(pool\_size = 2, strides = 2)])

# generate pooled output

output = model.predict(image)

# print output image

output = np.squeeze(output)

print(output)

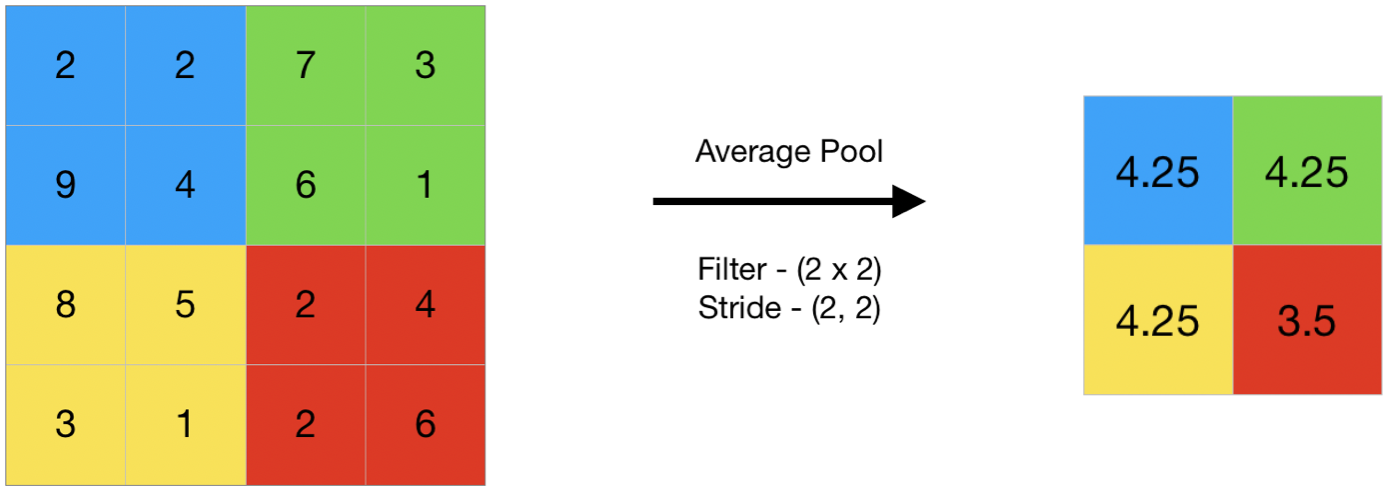
**Output:**

[[9. 7.]

[8. 6.]]

* **Average Pooling**

Average pooling computes the average of the elements present in the region of feature map covered by the filter. Thus, while max pooling gives the most prominent feature in a particular patch of the feature map, average pooling gives the average of features present in a patch.



**Performing Average pooling using Keras**

import numpy as np

from keras.models import Sequential

from keras.layers import AveragePooling2D

# define input image

image = np.array([[2, 2, 7, 3],

[9, 4, 6, 1],

[8, 5, 2, 4],

[3, 1, 2, 6]])

image = image.reshape(1, 4, 4, 1)

# define model containing just a single average pooling layer

model = Sequential(

[AveragePooling2D(pool\_size = 2, strides = 2)])

# generate pooled output

output = model.predict(image)

# print output image

output = np.squeeze(output)

print(output)

**Output**:

[[4.25 4.25]

[4.25 3.5 ]]

* **Global Pooling**

Global pooling reduces each channel in the feature map to a single value. Thus, an nh x nw x nc feature map is reduced to 1 x 1 x nc feature map. This is equivalent to using a filter of dimensions nh x nw i.e. the dimensions of the feature map.

Further, it can be either global max pooling or global average pooling.

Performing Global Pooling using Keras

import numpy as np

from keras.models import Sequential

from keras.layers import GlobalMaxPooling2D

from keras.layers import GlobalAveragePooling2D

# define input image

image = np.array([[2, 2, 7, 3],

[9, 4, 6, 1],

[8, 5, 2, 4],

[3, 1, 2, 6]])

image = image.reshape(1, 4, 4, 1)

# defne gm\_model containing just a single global-max pooling layer

gm\_model = Sequential(

[GlobalMaxPooling2D()])

# define ga\_model containing just a single global-average pooling layer

ga\_model = Sequential(

[GlobalAveragePooling2D()])

# generate pooled output

gm\_output = gm\_model.predict(image)

ga\_output = ga\_model.predict(image)

# print output image

gm\_output = np.squeeze(gm\_output)

ga\_output = np.squeeze(ga\_output)

print("gm\_output: ", gm\_output)

print("ga\_output: ", ga\_output)

**Output:**

gm\_output: 9.0

ga\_output: 4.0625

1. **EMBEDDING LAYER**

Keras offers an Embedding layer that can be used for neural networks on text data. It requires that the input data be integer encoded, so that each word is represented by a unique integer. The Embedding layer is initialized with random weights and will learn an embedding for all the words in the training dataset.

It is a flexible layer that can be used in a variety of ways, such as:

* It can be used alone to learn a word embedding that can be saved and used in another model later.
* It can be used as part of a deep learning model where the embedding is learned along with the model itself.
* It can be used to load a pre-trained word embedding model, a type of transfer learning.
* The Embedding layer is defined as the first hidden layer of a network.

It must specify 3 arguments:

**input\_dim**: This is the size of the vocabulary in the text data. For example, if your data is integer encoded to values between 0-10, then the size of the vocabulary would be 11 words.

**output\_dim**: This is the size of the vector space in which words will be embedded. It defines the size of the output vectors from this layer for each word. For example, it could be 32 or 100 or even larger. Test different values for your problem.

**input\_length:** This is the length of input sequences, as you would define for any input layer of a Keras model. For example, if all your input documents are comprised of 1000 words, this would be 1000.

Diagram

Description automatically generated

1. **LSTM LAYER (Long Short Term Memory)**

LSTM networks are an extension of recurrent neural networks (RNNs) mainly introduced to handle situations where RNNs fail. RNN is a network that works on the present input by taking into consideration the previous output (feedback) and storing in its memory for a short period of time (short-term memory).

Out of its various applications, the most popular ones are in the fields of speech processing, and music composition. Nevertheless, there are drawbacks to RNNs. First, it fails to store information for a longer period. At times, a reference to certain information stored quite a long time ago is required to predict the current output. But RNNs are incapable of handling such “long-term dependencies”. Second, there is no finer control over which part of the context needs to be carried forward and how much of the past needs to be ‘forgotten’. Other issues with RNNs are exploding and vanishing gradients which occur during the training process of a network through backtracking.

Thus, Long Short-Term Memory (LSTM) was brought into the picture. It has been so designed that the vanishing gradient problem is almost completely removed, while the training model is left unaltered. Long time lags in certain problems are bridged using LSTMs where they also handle noise, distributed representations, and continuous values. With LSTMs, there is no need to keep a finite number of states from beforehand as required in the hidden Markov model (HMM). LSTMs provide us with a large range of parameters such as learning rates, and input and output biases. Hence, no need for fine adjustments. The complexity to update each weight is reduced to O (1) with LSTMs, similar to that of Back Propagation Through Time (BPTT), which is an advantage.

**Creating the Keras LSTM Structure**

model = Sequential()

model.add(Embedding(vocabulary, hidden\_size, input\_length=num\_steps))

model.add(LSTM(hidden\_size, return\_sequences=True))

model.add(LSTM(hidden\_size, return\_sequences=True))

if use\_dropout:

model.add(Dropout(0.5))

model.add(TimeDistributed(Dense(vocabulary)))

model.add(Activation('softmax'))

**Diagram

Description automatically generated**

1. **CUSTOM LAYER**

If Deep Learning Toolbox does not provide the layer you require for your classification or regression problem, then you can define your own custom layer.

To define a custom deep learning layer through the following steps:

* Name the layer – give the layer a name so that it can be used in MATLAB®.
* Declare the layer properties – specify the properties of the layer and which parameters are learned during training.
* Create a constructor function (optional) – specify how to construct the layer and initialize its properties. If you do not specify a constructor function, then at creation, the software initializes the Name, Description, and Type properties with [] and sets the number of layer inputs and outputs to 1.
* Create forward functions – specify how data passes forward through the layer (forward propagation) at prediction time and at training time.
* Create a backward function (optional) – specify the derivatives of the loss with respect to the input data and the learnable parameters (backward propagation). If you do not specify a backward function, then the forward functions must support dlarray objects.

**Code for creating a customized layer**

from keras import backend as K from keras.layers import Layer

class MyCustomLayer(Layer):

def \_\_init\_\_(self, output\_dim, \*\*kwargs):

self.output\_dim = output\_dim

super(MyCustomLayer, self).\_\_init\_\_(\*\*kwargs)

def build(self, input\_shape): self.kernel =

self.add\_weight(name = 'kernel',

shape = (input\_shape[1], self.output\_dim),

initializer = 'normal', trainable = True)

super(MyCustomLayer, self).build(input\_shape) #

Be sure to call this at the end

def call(self, input\_data): return K.dot(input\_data, self.kernel)

def compute\_output\_shape(self,input\_shape):return(input\_shape[0], self.output\_dim)