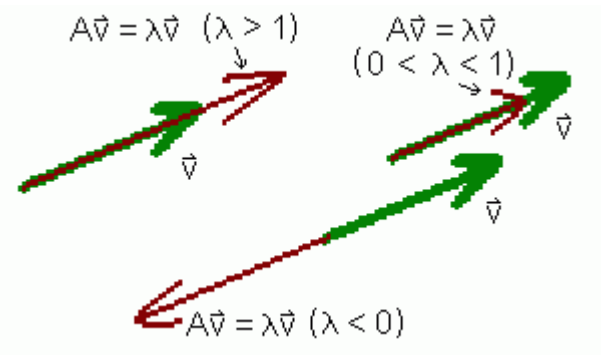
**INDEX**

**PRACTICAL 1**

**MATRIX MULTIPLICATION, EIGEN VECTORS, EIGENVALUE COMPUTATION USING TENSORFLOW**

Eigenvalues and eigenvectors play a prominent role in the study of ordinary differential equations and in many applications in the physical sciences. Expect to see them come up in a variety of contexts!

**Definitions**




Let AA be an n×nn×n matrix. The number λλ is an **eigenvalue** of AA if there exists a non-zero vector vv such that

$$Av=\lambda v.Av=\lambda v.$$

In this case, vector vv is called an **eigenvector** of AA corresponding to λλ.

**CODE:**

```
import tensorflow as tf
print("Matrix Multiplication Demo")
x=tf.constant([1,2,3,4,5,6],shape=[2,3])
print(x)
y=tf.constant([7,8,9,10,11,12],shape=[3,2])
print(y)
z=tf.matmul(x,y)
```

```
print("Product:",z)

e_matrix_A=tf.random.uniform([2,2],minval=3,maxval=10,dtype=tf.float32,name="matrixA")

print("Matrix A:\n{}\n\n".format(e_matrix_A))

eigen_values_A,eigen_vectors_A=tf.linalg.eigh(e_matrix_A)

print("Eigen Vectors:\n{}\n\nEigen values:\n{}\n".format(eigen_vectors_A, eigen_values_A))
```

**OUTPUT:**

```
Matrix Multiplication Demo
tf.Tensor(
[[1 2 3]
 [4 5 6]], shape=(2, 3), dtype=int32)
tf.Tensor(
[[ 7  8]
 [ 9 10]
 [11 12]], shape=(3, 2), dtype=int32)
Product: tf.Tensor(
[[ 58  64]
 [139 154]], shape=(2, 2), dtype=int32)
Matrix A:
[[6.275464  3.9095778]
 [7.5652266 7.9653716]]


Eigen Vectors:
[[-0.7453184  -0.66670865]
 [ 0.66670865 -0.7453184 ]]

Eigen values:
[-0.4918485 14.732683 ]
```

**PRACTICAL 2**
**DEEP FORWARD NETWORK FOR XOR**

**Deep feedforward networks**, also often called **feedforward neural networks**, or **multilayer perceptron's** (MLPs), are the quintessential deep learning models. The goal of a feedforward network is to approximate some function f*. For example, for a classifier, y = f*(**x**) maps an input **x** to a category **y**. A feedforward network defines a mapping **y** = f (**x**; **θ**) and learns the value of the parameters **θ** that result in the best function approximation.

These models are called feedforward because information flows through the function being evaluated from **x**, through the intermediate computations used to define f, and finally to the output y. There are no feedback connections in which outputs of the model are fed back into itself.

XOR Truth Table:

| Inputs | | Output |
|---|---|---|
| A | B | X |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**CODE:**

```
import numpy as np
from keras.layers import Dense
from keras.models import Sequential
model=Sequential()
model.add(Dense(units=2,activation='relu',input_dim=2))
```

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

model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
print(model.summary())
print(model.get_weights())

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

model.fit(X,Y,epochs=1000,batch_size=4)

print(model.get_weights())
print(model.predict(X,batch_size=4))
```

**OUTPUT:**

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_6 (Dense)              (None, 2)                 6
_________________________________________________________________
dense_7 (Dense)              (None, 1)                 3
=================================================================
Total params: 9
Trainable params: 9
Non-trainable params: 0
_________________________________________________________________
None
[array([[-0.97247744,  1.0488704 ],
       [ 0.0012579 , -0.09331477]], dtype=float32), array([0., 0.], dtype=float32), array([[-0.887708 ],
       [-1.3425305]], dtype=float32), array([0.], dtype=float32)]
Epoch 1/1000
1/1 [==============================] - 0s 364ms/step - loss: 0.8146 - accuracy: 0.5000
Epoch 2/1000
1/1 [==============================] - 0s 7ms/step - loss: 0.8137 - accuracy: 0.5000
Epoch 3/1000
1/1 [==============================] - 0s 6ms/step - loss: 0.8129 - accuracy: 0.5000
Epoch 4/1000
1/1 [==============================] - 0s 7ms/step - loss: 0.8122 - accuracy: 0.5000
Epoch 5/1000
```

.

.

.

```
Epoch 993/1000
1/1 [==============================] - 0s 16ms/step - loss: 0.5863 - accuracy: 0.7500
Epoch 994/1000
1/1 [==============================] - 0s 15ms/step - loss: 0.5863 - accuracy: 0.7500
Epoch 995/1000
1/1 [==============================] - 0s 14ms/step - loss: 0.5862 - accuracy: 0.7500
Epoch 996/1000
1/1 [==============================] - 0s 6ms/step - loss: 0.5861 - accuracy: 0.7500
Epoch 997/1000
1/1 [==============================] - 0s 7ms/step - loss: 0.5860 - accuracy: 0.7500
Epoch 998/1000
1/1 [==============================] - 0s 5ms/step - loss: 0.5858 - accuracy: 0.7500
Epoch 999/1000
1/1 [==============================] - 0s 14ms/step - loss: 0.5857 - accuracy: 0.7500
Epoch 1000/1000
1/1 [==============================] - 0s 8ms/step - loss: 0.5857 - accuracy: 0.7500
[array([[-0.97247744,  0.6868896 ],
       [-0.00474744,  0.68647397]], dtype=float32), array([-0.00600533, -0.6870204 ], dtype=float32), array([[-0.8818216],
       [-1.5626694]], dtype=float32), array([0.2367169], dtype=float32)]
[[0.5589044]
 [0.5589044]
 [0.5589044]
 [0.3024179]]
```

**PRACTICAL 3A**
**CLASSIFICATION USING DNN**

Classification neural networks used for feature categorization are **very similar to fault-diagnosis networks**, except that they only allow one output response for any input pattern, instead of allowing multiple faults to occur for a given set of operating conditions.
The classification network selects the category based on which output response has the highest output value.

**Problem statement:**
#The given dataset comprises health information about diabetic women patients. We need to create a deep feed forward network that will classify women suffering from diabetes mellitus as 1.

**CODE:**

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

dataset=loadtxt('/content/sample_data/pima-indians-diabetes.csv',delimiter=',')
dataset
```

```
array([[  6.   , 148.   ,  72.   , ...,   0.627,  50.   ,   1.   ],
       [  1.   ,  85.   ,  66.   , ...,   0.351,  31.   ,   0.   ],
       [  8.   , 183.   ,  64.   , ...,   0.672,  32.   ,   1.   ],
       ...,
       [  5.   , 121.   ,  72.   , ...,   0.245,  30.   ,   0.   ],
       [  1.   , 126.   ,  60.   , ...,   0.349,  47.   ,   1.   ],
       [  1.   ,  93.   ,  70.   , ...,   0.315,  23.   ,   0.   ]])
```

```
X=dataset[:,0:8]
Y=dataset[:,8]
X
```

```
array([[  6.   , 148.   ,  72.   , ...,  33.6  ,   0.627,  50.   ],
       [  1.   ,  85.   ,  66.   , ...,  26.6  ,   0.351,  31.   ],
       [  8.   , 183.   ,  64.   , ...,  23.3  ,   0.672,  32.   ],
       ...,
       [  5.   , 121.   ,  72.   , ...,  26.2  ,   0.245,  30.   ],
       [  1.   , 126.   ,  60.   , ...,  30.1  ,   0.349,  47.   ],
       [  1.   ,  93.   ,  70.   , ...,  30.4  ,   0.315,  23.   ]])
```

Y

```
array([1., 0., 1., 0., 1., 0., 1., 0., 1., 1., 0., 1., 0., 1., 1., 1., 1.,
       1., 0., 1., 0., 0., 1., 1., 1., 1., 1., 0., 0., 0., 0., 1., 0., 0.,
       0., 0., 0., 1., 1., 1., 0., 0., 0., 1., 0., 1., 0., 0., 1., 0., 0.,
       0., 0., 1., 0., 0., 1., 0., 0., 0., 0., 1., 0., 0., 1., 0., 1., 0.,
       0., 0., 1., 0., 1., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 1.,
       0., 0., 0., 1., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 1., 1., 0.,
       0., 0., 0., 0., 0., 0., 0., 1., 1., 1., 0., 0., 1., 1., 1., 0., 0.,
       0., 1., 0., 0., 0., 1., 1., 0., 0., 1., 1., 1., 1., 1., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 1.,
       0., 1., 1., 0., 0., 0., 1., 0., 0., 0., 0., 1., 1., 0., 0., 0., 0.,
       1., 1., 0., 0., 0., 1., 0., 1., 0., 1., 0., 0., 0., 0., 1., 1., 1.,
       1., 1., 1., 0., 0., 1., 1., 0., 1., 0., 1., 1., 1., 0., 0., 0., 0.,
       0., 0., 1., 1., 0., 1., 0., 0., 0., 1., 1., 1., 1., 0., 1., 1., 1.,
       1., 0., 0., 0., 0., 0., 1., 0., 0., 1., 1., 0., 0., 0., 1., 1., 1.,
       1., 0., 0., 0., 1., 1., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 1.,
       1., 0., 0., 0., 1., 0., 1., 0., 0., 1., 0., 1., 0., 0., 1., 1., 0.,
       0., 0., 0., 0., 1., 0., 0., 0., 1., 0., 0., 1., 1., 0., 0., 1., 0.,
       0., 0., 1., 1., 1., 0., 0., 1., 0., 1., 0., 0., 1., 0., 1., 0., 0.,
       1., 0., 1., 1., 0., 0., 1., 0., 1., 0., 0., 1., 0., 1., 0., 1., 1.,
       1., 0., 0., 1., 0., 1., 0., 0., 0., 1., 0., 0., 0., 0., 1., 1., 1.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 1., 1.,
       1., 0., 1., 0., 0., 1., 1., 0., 1., 0., 0., 1., 0., 0., 0., 1., 0.,
       1., 0., 1., 0., 1., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 1., 0.,
       0., 1., 1., 1., 0., 0., 1., 0., 0., 1., 0., 0., 0., 1., 0., 0., 1.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0.,
       1., 0., 0., 0., 1., 0., 0., 0., 1., 1., 0., 0., 0., 0., 0., 0., 0.,
       1., 0., 0., 0., 0., 1., 0., 0., 0., 1., 0., 0., 0., 1., 0., 0., 0.,
       1., 0., 0., 0., 0., 1., 1., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 1., 1., 1., 1., 0.,
       0., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1.,
       1., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 1.,
       0., 1., 1., 0., 0., 0., 0., 1., 0., 1., 0., 1., 0., 1., 0., 0., 0.,
       1., 0., 0., 1., 0., 0., 0., 0., 1., 1., 0., 1., 0., 0., 0., 0., 1.,
       1., 0., 1., 0., 0., 0., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 1., 0., 0., 0., 0., 1., 0., 0., 1., 0., 0., 0., 1., 0., 0., 0.,
       1., 1., 1., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 1., 0., 1., 1.,
       1., 1., 0., 1., 1., 0., 0., 0., 0., 0., 0., 0., 1., 1., 0., 1., 0.,
       0., 1., 0., 1., 0., 0., 0., 0., 1., 0., 0., 1., 0., 1., 0., 1., 1.,
       0., 0., 0., 0., 1., 1., 0., 0., 0., 1., 0., 1., 1., 0., 0., 1., 0.,
       0., 1., 1., 0., 0., 1., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 1.,
       1., 1., 0., 0., 0., 0., 0., 0., 1., 1., 0., 0., 1., 0., 0., 1., 0.,
       1., 1., 1., 0., 0., 1., 1., 1., 0., 1., 0., 1., 0., 1., 0., 0., 0.,
       0., 1., 0.])
```

#Creating model
model = Sequential()
model.add(Dense(12, input_dim=8, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

#Compiling and fitting model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(X, Y, epochs=150, batch_size=10)

```
Epoch 1/150
77/77 [==============================] - 1s 2ms/step - loss: 43.7715 - accuracy: 0.3490
Epoch 2/150
77/77 [==============================] - 0s 2ms/step - loss: 10.4616 - accuracy: 0.3529
Epoch 3/150
77/77 [==============================] - 0s 2ms/step - loss: 3.2980 - accuracy: 0.3932
Epoch 4/150
77/77 [==============================] - 0s 2ms/step - loss: 1.0124 - accuracy: 0.6328
Epoch 5/150
77/77 [==============================] - 0s 2ms/step - loss: 0.8380 - accuracy: 0.6628
Epoch 6/150
77/77 [==============================] - 0s 2ms/step - loss: 0.7541 - accuracy: 0.6654
Epoch 7/150
77/77 [==============================] - 0s 1ms/step - loss: 0.7034 - accuracy: 0.6706
Epoch 8/150
77/77 [==============================] - 0s 1ms/step - loss: 0.6750 - accuracy: 0.6693
Epoch 9/150
77/77 [==============================] - 0s 1ms/step - loss: 0.6556 - accuracy: 0.6680
Epoch 10/150
77/77 [==============================] - 0s 2ms/step - loss: 0.6416 - accuracy: 0.6771
```

```
Epoch 140/150
77/77 [==============================] - 0s 2ms/step - loss: 0.5902 - accuracy: 0.7031
Epoch 141/150
77/77 [==============================] - 0s 1ms/step - loss: 0.5914 - accuracy: 0.7057
Epoch 142/150
77/77 [==============================] - 0s 1ms/step - loss: 0.5918 - accuracy: 0.7005
Epoch 143/150
77/77 [==============================] - 0s 1ms/step - loss: 0.5983 - accuracy: 0.7070
Epoch 144/150
77/77 [==============================] - 0s 2ms/step - loss: 0.5908 - accuracy: 0.7122
Epoch 145/150
77/77 [==============================] - 0s 2ms/step - loss: 0.5935 - accuracy: 0.7161
Epoch 146/150
77/77 [==============================] - 0s 1ms/step - loss: 0.5911 - accuracy: 0.7044
Epoch 147/150
77/77 [==============================] - 0s 2ms/step - loss: 0.5962 - accuracy: 0.7044
Epoch 148/150
77/77 [==============================] - 0s 1ms/step - loss: 0.5930 - accuracy: 0.7005
Epoch 149/150
77/77 [==============================] - 0s 2ms/step - loss: 0.5889 - accuracy: 0.7109
Epoch 150/150
77/77 [==============================] - 0s 2ms/step - loss: 0.5912 - accuracy: 0.7057
<keras.callbacks.History at 0x7fc4b722d0d0>
```

```
model.fit(X, Y, epochs=150, batch_size=10)
predictions = model.predict(X)
rounded = [round(x[0]) for x in predictions]
print(rounded)
scores = model.evaluate(X, Y)
print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

```
Epoch 143/150
77/77 [==============================] - 0s 1ms/step - loss: 0.4995 - accuracy: 0.7682
Epoch 144/150
77/77 [==============================] - 0s 2ms/step - loss: 0.4980 - accuracy: 0.7565
Epoch 145/150
77/77 [==============================] - 0s 2ms/step - loss: 0.4997 - accuracy: 0.7500
Epoch 146/150
77/77 [==============================] - 0s 1ms/step - loss: 0.5022 - accuracy: 0.7656
Epoch 147/150
77/77 [==============================] - 0s 2ms/step - loss: 0.4947 - accuracy: 0.7630
Epoch 148/150
77/77 [==============================] - 0s 2ms/step - loss: 0.4985 - accuracy: 0.7604
Epoch 149/150
77/77 [==============================] - 0s 1ms/step - loss: 0.4945 - accuracy: 0.7656
Epoch 150/150
77/77 [==============================] - 0s 2ms/step - loss: 0.4978 - accuracy: 0.7682
WARNING:tensorflow:5 out of the last 5 calls to <function Model.make_predict_function.<locals>.predict_function at 0x7fc4b52209e0> triggered tf.function retracing. Tracing is expensi
[1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0
24/24 [==============================] - 0s 1ms/step - loss: 0.4847 - accuracy: 0.7695

accuracy: 76.95%
```

**PRACTICAL 3B**
**BINARY CLASSIFICATION USING MLP**

Multilayer Perceptron falls under the category of feedforward algorithms, because inputs are combined with the initial weights in a weighted sum and subjected to the activation function, just like in the Perceptron. But the difference is that each linear combination is propagated to the next layer. Each layer is *feeding* the next one with the result of their computation, their internal representation of the data. This goes all the way through the hidden layers to the output layer.

Binary classification, which looks at an input and predicts which of two possible classes it belongs to. Practical uses include sentiment analysis, spam detection, and credit-card fraud detection. Such models are trained with datasets labelled with 1s and 0s representing the two classes, employ popular learning algorithms such as logistic regression and Naïve Bayes, and are frequently built with libraries such as Scikit-learn.

**CODE:**

```
# mlp for binary classification
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
# load the dataset
path = '/content/sample_data/lonospear.csv'
df = read_csv(path, header=None)
# split into input and output columns
X, y = df.values[:, :-1], df.values[:, -1]
# ensure all data are floating point values
X = X.astype('float32')
# encode strings to integer
y = LabelEncoder().fit_transform(y)
# split into train and test datasets
```

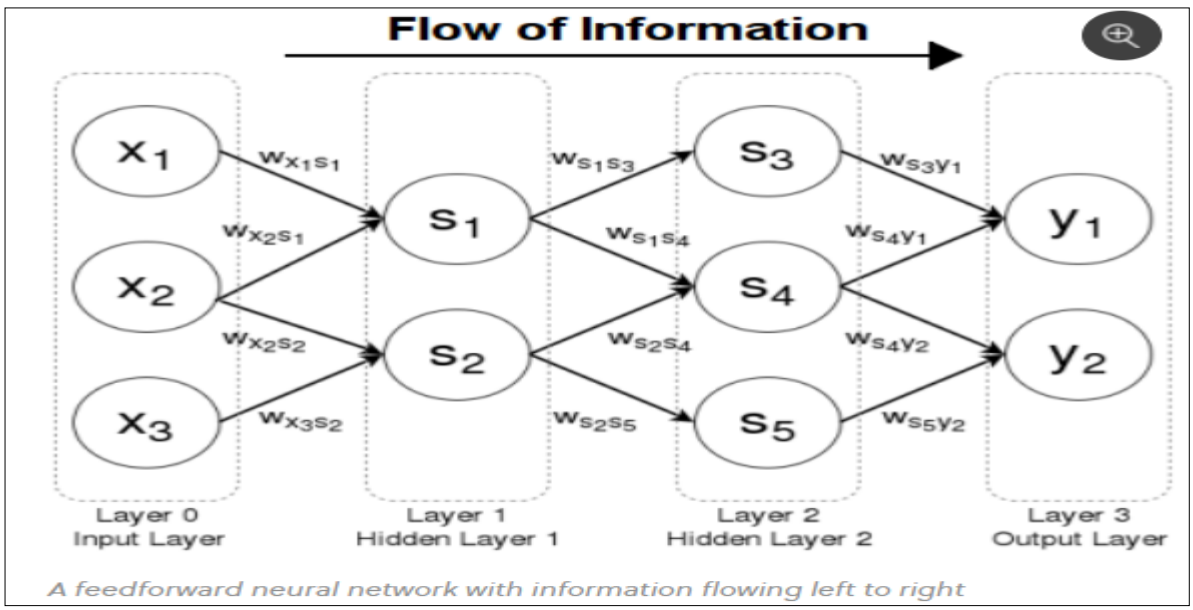
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33)
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
# determine the number of input features
n_features = X_train.shape[1]
# define model
model = Sequential()
model.add(Dense(10, activation='relu', kernel_initializer='he_normal', input_shape=(n_features,)))
model.add(Dense(8, activation='relu', kernel_initializer='he_normal'))
model.add(Dense(1, activation='sigmoid'))
# compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
# fit the model
model.fit(X_train, y_train, epochs=150, batch_size=32, verbose=0)
# evaluate the model
loss, acc = model.evaluate(X_test, y_test, verbose=0)
print('Test Accuracy: %.3f' % acc)
# make a prediction
row = [1,0,0.99539,-0.05889,0.85243,0.02306,0.83398,-0.37708,1,0.03760,0.85243,-
0.17755,0.59755,-0.44945,0.60536,-0.38223,0.84356,-0.38542,0.58212,-0.32192,0.56971,-
0.29674,0.36946,-0.47357,0.56811,-0.51171,0.41078,-0.46168,0.21266,-0.34090,0.42267,-
0.54487,0.18641,-0.45300]
yhat = model.predict([row])
print('Predicted: %.3f' % yhat)
```

**OUTPUT:**

```
(235, 34) (116, 34) (235,) (116,)
Test Accuracy: 0.922
Predicted: 0.990
```

**PRACTICAL 4A**
**FEED FORWARD NN**

**Feedforward neural networks** are artificial neural networks where the connections between units do not form a cycle. Feedforward neural networks were the first type of artificial neural network invented and are simpler than their counterpart, recurrent neural networks. They are called *feedforward* because information only travels forward in the network (no loops), first through the input nodes, then through the hidden nodes (if present), and finally through the output nodes. Feedforward neural networks are primarily used for supervised learning in cases where the data to be learned is neither sequential nor time-dependent. That is, feedforward neural networks compute a function ff on fixed size input xx such that f(x) \approx. yf(x)≈y for training pairs (x, y) (x, y). On the other hand, recurrent neural networks learn sequential data, computing gg on variable length input X_k = \{x_1, \dots, x_k\}Xk={x1,…,xk} such that g(X_k) \approxy_kg(Xk)≈yk for training pairs (X_n, Y_n)(Xn,Yn) for all 1 \le k \le n1≤k≤n.



A feedforward neural network with information flowing left to right

**CODE:**

```
from keras.models import Sequential
from keras.layers import Dense
from sklearn.datasets import make_blobs
from sklearn.preprocessing import MinMaxScaler
```

```
import numpy as np
X,Y=make_blobs(n_samples=100,centers=2,n_features=2,random_state=1)
scalar=MinMaxScaler()
scalar.fit(X)
X=scalar.transform(X)
model=Sequential()
model.add(Dense(4,input_dim=2,activation='relu'))
model.add(Dense(4,activation='relu'))
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam')
model.fit(X,Y,epochs=500)
```

```
Epoch 492/500
4/4 [==============================] - 0s 3ms/step - loss: 0.1101
Epoch 493/500
4/4 [==============================] - 0s 5ms/step - loss: 0.1098
Epoch 494/500
4/4 [==============================] - 0s 5ms/step - loss: 0.1095
Epoch 495/500
4/4 [==============================] - 0s 3ms/step - loss: 0.1091
Epoch 496/500
4/4 [==============================] - 0s 3ms/step - loss: 0.1089
Epoch 497/500
4/4 [==============================] - 0s 5ms/step - loss: 0.1085
Epoch 498/500
4/4 [==============================] - 0s 3ms/step - loss: 0.1082
Epoch 499/500
4/4 [==============================] - 0s 3ms/step - loss: 0.1079
Epoch 500/500
4/4 [==============================] - 0s 2ms/step - loss: 0.1077
<keras.callbacks.History at 0x7f46a337a190>
```

```
Xnew,Yreal=make_blobs(n_samples=5,centers=2,n_features=2,random_state=1)
Xnew=scalar.transform(Xnew)
Ynew=model.predict(Xnew)
Ynew=np.round(Ynew).astype(int)
for i in range(len(Xnew)):
  print("X=%s,Predicted=%d,Desired=%s"%(Xnew[i],Ynew[i],Yreal[i]))
```

```
X=[0.26814469 0.0290577 ],Predicted=1,Desired=1
X=[0.17582555 0.16948267],Predicted=1,Desired=1
X=[0.89337759 0.65864154],Predicted=0,Desired=0
X=[0.96440204 0.77809405],Predicted=0,Desired=0
X=[0.78082614 0.75391697],Predicted=0,Desired=0
```

**PRACTICAL 4B**
**PREDICTING THE PROBABILITY OF THE CLASS**

**CODE:**

```
from keras.models import Sequential
from keras.layers import Dense
from sklearn.datasets import make_blobs
from sklearn.preprocessing import MinMaxScaler
X,Y=make_blobs(n_samples=100,centers=2,n_features=2,random_state=1)
scalar=MinMaxScaler()
scalar.fit(X)
X=scalar.transform(X)
model=Sequential()
model.add(Dense(4,input_dim=2,activation='relu'))
model.add(Dense(4,activation='relu'))
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam')
model.fit(X,Y,epochs=500)
```

```
Epoch 494/500
4/4 [==============================] - 0s 3ms/step - loss: 0.0917
Epoch 495/500
4/4 [==============================] - 0s 4ms/step - loss: 0.0915
Epoch 496/500
4/4 [==============================] - 0s 4ms/step - loss: 0.0913
Epoch 497/500
4/4 [==============================] - 0s 3ms/step - loss: 0.0911
Epoch 498/500
4/4 [==============================] - 0s 4ms/step - loss: 0.0909
Epoch 499/500
4/4 [==============================] - 0s 4ms/step - loss: 0.0907
Epoch 500/500
4/4 [==============================] - 0s 5ms/step - loss: 0.0905
<keras.callbacks.History at 0x7f46a32dcb10>
```

```
Xnew,Yreal=make_blobs(n_samples=3,centers=2,n_features=2,random_state=1)
Xnew=scalar.transform(Xnew)
Yclass=model.predict(Xnew)
Ynew=model.predict(Xnew)
for i in range(len(Xnew)):
```

print("X=%s,Predicted_probability=%s,Predicted_class=%s"%(Xnew[i],Ynew[i],Yclass[i]))

```
X=[0.89337759 0.65864154],Predicted_probability=[0.00300109],Predicted_class=[0.00300109]
X=[0.29097707 0.12978982],Predicted_probability=[0.8361509],Predicted_class=[0.8361509]
X=[0.78082614 0.75391697],Predicted_probability=[0.00391737],Predicted_class=[0.00391737]
```

**PRACTICAL 5A**
**CNN FOR CIFAR10 IMAGES**

A **neural network** in which at least one layer is a **convolutional layer**. A typical convolutional neural network consists of some combination of the following layers:

- **convolutional layers**
- **pooling layers**
- **dense layers**

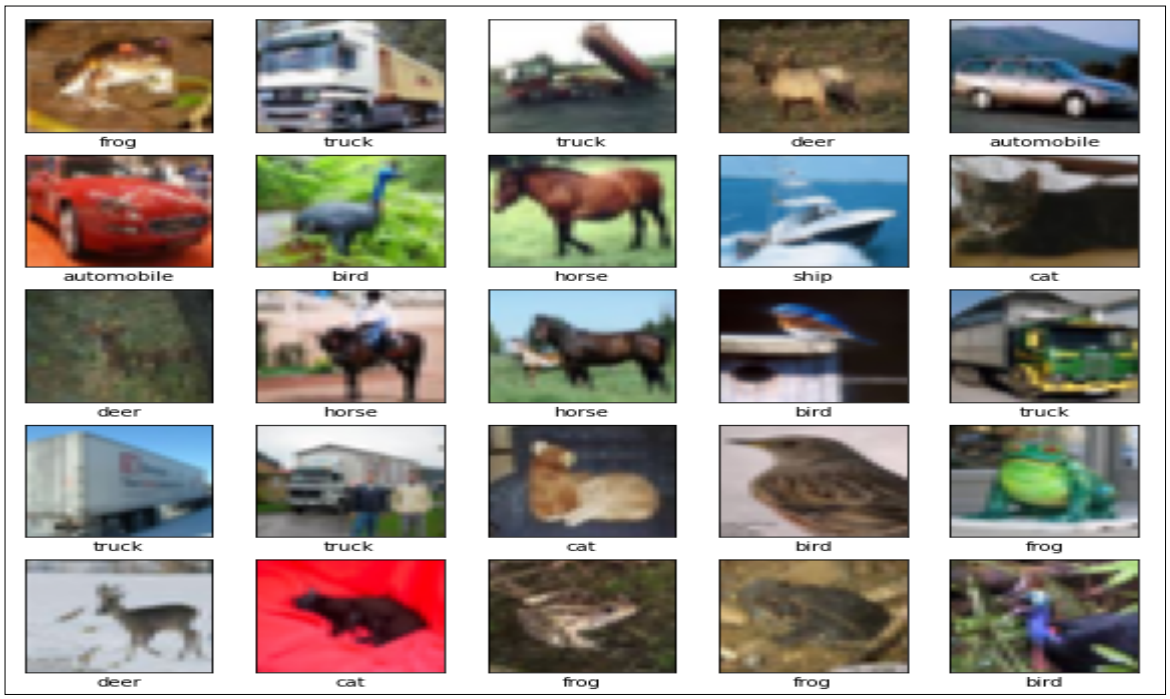
Convolutional neural networks have had great success in certain kinds of problems, such as image recognition.

**CODE:**

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()
# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images / 255.0
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170500096/170498071 [==============================] - 2s 0us/step
170508288/170498071 [==============================] - 2s 0us/step
```

```
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']
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])
    # The CIFAR labels happen to be arrays,
    # which is why you need the extra index
    plt.xlabel(class_names[train_labels[i][0]])
plt.show()
```




```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 30, 30, 32)        896

 max_pooling2d (MaxPooling2D  (None, 15, 15, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 13, 13, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 6, 6, 64)         0
 2D)

 conv2d_2 (Conv2D)           (None, 4, 4, 64)          36928

=================================================================
Total params: 56,320
Trainable params: 56,320
Non-trainable params: 0
_________________________________________________________________
```

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10))
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 30, 30, 32)        896

 max_pooling2d (MaxPooling2D  (None, 15, 15, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 13, 13, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 6, 6, 64)         0
 2D)

 conv2d_2 (Conv2D)           (None, 4, 4, 64)          36928

 flatten (Flatten)           (None, 1024)              0

 dense (Dense)               (None, 64)                65600

 dense_1 (Dense)             (None, 10)                650

=================================================================
Total params: 122,570
Trainable params: 122,570
Non-trainable params: 0
_________________________________________________________________
```

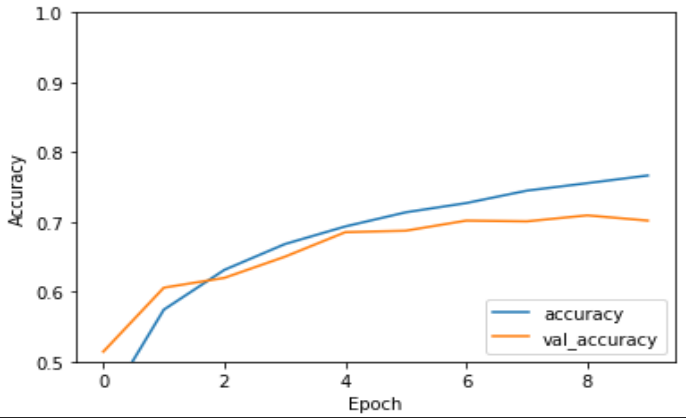
**this step will take time to execute –5 to 8 minutes**

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

history = model.fit(train_images, train_labels, epochs=10,
            validation_data=(test_images, test_labels))
```

```
Epoch 1/10
1563/1563 [==============================] - 67s 43ms/step - loss: 1.5530 - accuracy: 0.4310 - val_loss: 1.3419 - val_accuracy: 0.5141
Epoch 2/10
1563/1563 [==============================] - 65s 41ms/step - loss: 1.1935 - accuracy: 0.5743 - val_loss: 1.1104 - val_accuracy: 0.6058
Epoch 3/10
1563/1563 [==============================] - 65s 41ms/step - loss: 1.0420 - accuracy: 0.6313 - val_loss: 1.0676 - val_accuracy: 0.6198
Epoch 4/10
1563/1563 [==============================] - 65s 41ms/step - loss: 0.9424 - accuracy: 0.6682 - val_loss: 1.0080 - val_accuracy: 0.6501
Epoch 5/10
1563/1563 [==============================] - 64s 41ms/step - loss: 0.8729 - accuracy: 0.6935 - val_loss: 0.9047 - val_accuracy: 0.6853
Epoch 6/10
1563/1563 [==============================] - 64s 41ms/step - loss: 0.8154 - accuracy: 0.7138 - val_loss: 0.9179 - val_accuracy: 0.6874
Epoch 7/10
1563/1563 [==============================] - 63s 40ms/step - loss: 0.7757 - accuracy: 0.7270 - val_loss: 0.8705 - val_accuracy: 0.7019
Epoch 8/10
1563/1563 [==============================] - 65s 41ms/step - loss: 0.7277 - accuracy: 0.7448 - val_loss: 0.8626 - val_accuracy: 0.7008
Epoch 9/10
1563/1563 [==============================] - 64s 41ms/step - loss: 0.6931 - accuracy: 0.7555 - val_loss: 0.8563 - val_accuracy: 0.7094
Epoch 10/10
1563/1563 [==============================] - 67s 43ms/step - loss: 0.6618 - accuracy: 0.7665 - val_loss: 0.8888 - val_accuracy: 0.7019
```

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label ='val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.5,1])
plt.legend(loc='lower right')

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

```
313/313 - 3s - loss: 0.8888 - accuracy: 0.7019 - 3s/epoch - 11ms/step
```




print(test_acc)

```
0.7019000053405762
```

**PRACTICAL 5B**
**IMAGE CLASSIFICATION**

We will classify images of flowers. It creates an image classifier using a tf.keras.Sequential model, and loads data using tf.keras.utils.image_dataset_from_directory. You will gain practical experience with the following concepts:

- Efficiently loading a dataset off disk.
- Identifying overfitting and applying techniques to mitigate it, including data augmentation and dropout.

1. Examine and understand data
2. Build an input pipeline
3. Build the model
4. Train the model
5. Test the model
6. Improve the model and repeat the process

**CODE:**

```
import matplotlib.pyplot as plt
import numpy as np
import os
import PIL
import tensorflow as tf

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

This tutorial uses a dataset of about 3,700 photos of flowers. The dataset contains five sub-directories, one per class:

```
flower_photo/
  daisy/
  dandelion/
  roses/
  sunflowers/
  tulips/
```

```
import pathlib
dataset_url = "https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz"
data_dir = tf.keras.utils.get_file('flower_photos', origin=dataset_url, untar=True)
data_dir = pathlib.Path(data_dir)
```

```
Downloading data from https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz
228818944/228813984 [==============================] - 2s 0us/step
228827136/228813984 [==============================] - 2s 0us/step
```

```
image_count = len(list(data_dir.glob('*/*.jpg')))
print(image_count)
```

```
3670
```

```
roses = list(data_dir.glob('roses/*'))
PIL.Image.open(str(roses[0]))
```

```
PIL.Image.open(str(roses[1]))
```

```
tulips = list(data_dir.glob('tulips/*'))
```

PIL.Image.open(str(tulips[0]))

```
batch_size = 32
img_height = 180
img_width = 180
train_ds = tf.keras.utils.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="training",
  seed=123,
  image_size=(img_height, img_width),
  batch_size=batch_size)
```

```
Found 3670 files belonging to 5 classes.
Using 2936 files for training.
```

```
val_ds = tf.keras.utils.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="validation",
```

```
  seed=123,
  image_size=(img_height, img_width),
  batch_size=batch_size)
```

```
Found 3670 files belonging to 5 classes.
Using 734 files for validation.
```

```
class_names = train_ds.class_names
print(class_names)
```

```
['daisy', 'dandelion', 'roses', 'sunflowers', 'tulips']
```

```
importmatplotlib.pyplotasplt

plt.figure(figsize=(10,10))
for images, labels intrain_ds.take(1):
  foriin range(9):
    ax=plt.subplot(3,3,i+1)
    plt.imshow(images[i].numpy().astype("uint8"))
    plt.title(class_names[labels[i]])
    plt.axis("off")
```


```
forimage_batch,labels_batchintrain_ds:
  print(image_batch.shape)
  print(labels_batch.shape)
```

```
  break
```

```
(32, 180, 180, 3)
(32,)
```

```
AUTOTUNE = tf.data.AUTOTUNE

train_ds = train_ds.cache().shuffle(1000).prefetch(buffer_size=AUTOTUNE)
val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)
normalization_layer = layers.Rescaling(1./255)

normalized_ds = train_ds.map(lambda x, y: (normalization_layer(x), y))
image_batch, labels_batch = next(iter(normalized_ds))
first_image = image_batch[0]
# Notice the pixel values are now in `[0,1]`.
print(np.min(first_image), np.max(first_image))
```

```
0.0 0.96902645
```

```
num_classes = 5

model = Sequential([
  layers.Rescaling(1./255, input_shape=(img_height, img_width, 3)),
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(num_classes)
])

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

model.summary()
```

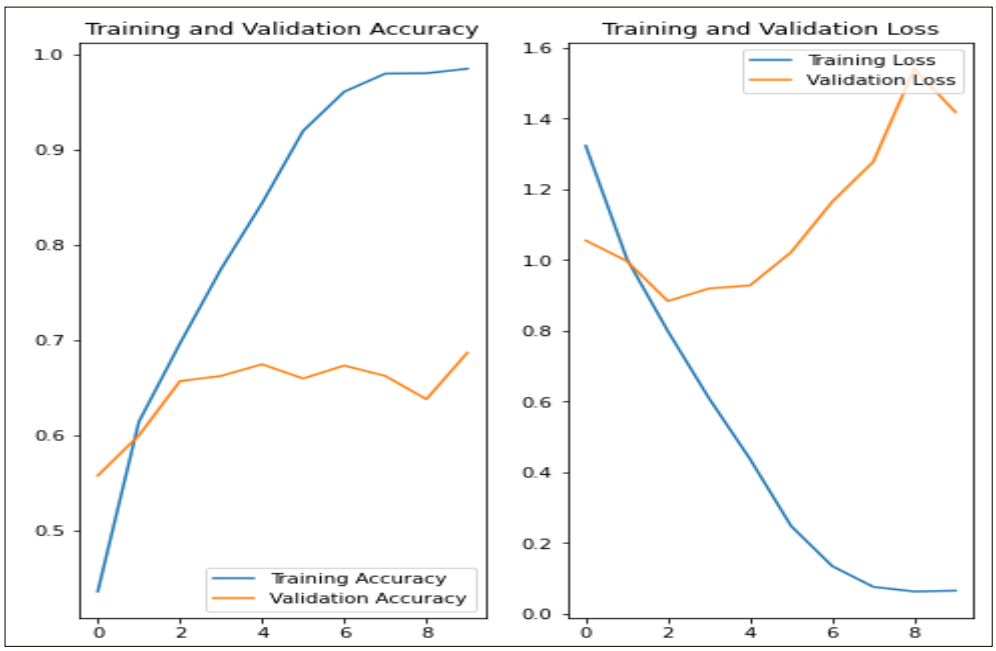
```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 rescaling_1 (Rescaling)     (None, 180, 180, 3)       0

 conv2d (Conv2D)             (None, 180, 180, 16)      448

 max_pooling2d (MaxPooling2D  (None, 90, 90, 16)       0
 )

 conv2d_1 (Conv2D)           (None, 90, 90, 32)        4640

 max_pooling2d_1 (MaxPooling  (None, 45, 45, 32)       0
 2D)

 conv2d_2 (Conv2D)           (None, 45, 45, 64)        18496

 max_pooling2d_2 (MaxPooling  (None, 22, 22, 64)       0
 2D)

 flatten (Flatten)           (None, 30976)             0

 dense (Dense)               (None, 128)               3965056

 dense_1 (Dense)             (None, 5)                 645

=================================================================
Total params: 3,989,285
Trainable params: 3,989,285
Non-trainable params: 0
_________________________________________________________________
```

```
epochs=10
history = model.fit(
  train_ds,
  validation_data=val_ds,
  epochs=epochs
)
```

```
Epoch 1/10
92/92 [==============================] - 15s 60ms/step - loss: 1.3226 - accuracy: 0.4353 - val_loss: 1.0551 - val_accuracy: 0.5572
Epoch 2/10
92/92 [==============================] - 4s 42ms/step - loss: 1.0034 - accuracy: 0.6138 - val_loss: 0.9969 - val_accuracy: 0.5995
Epoch 3/10
92/92 [==============================] - 4s 41ms/step - loss: 0.7988 - accuracy: 0.6962 - val_loss: 0.8837 - val_accuracy: 0.6567
Epoch 4/10
92/92 [==============================] - 4s 41ms/step - loss: 0.6094 - accuracy: 0.7745 - val_loss: 0.9194 - val_accuracy: 0.6621
Epoch 5/10
92/92 [==============================] - 4s 41ms/step - loss: 0.4375 - accuracy: 0.8440 - val_loss: 0.9280 - val_accuracy: 0.6744
Epoch 6/10
92/92 [==============================] - 4s 40ms/step - loss: 0.2482 - accuracy: 0.9200 - val_loss: 1.0218 - val_accuracy: 0.6594
Epoch 7/10
92/92 [==============================] - 4s 41ms/step - loss: 0.1350 - accuracy: 0.9612 - val_loss: 1.1642 - val_accuracy: 0.6730
Epoch 8/10
92/92 [==============================] - 4s 41ms/step - loss: 0.0763 - accuracy: 0.9802 - val_loss: 1.2773 - val_accuracy: 0.6621
Epoch 9/10
92/92 [==============================] - 4s 41ms/step - loss: 0.0632 - accuracy: 0.9806 - val_loss: 1.5403 - val_accuracy: 0.6376
Epoch 10/10
92/92 [==============================] - 4s 41ms/step - loss: 0.0656 - accuracy: 0.9854 - val_loss: 1.4175 - val_accuracy: 0.6866
```

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(epochs)
```

```
plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```




```
data_augmentation = keras.Sequential(
  [
    layers.RandomFlip("horizontal",
                      input_shape=(img_height,
                                   img_width,
                                   3)),
    layers.RandomRotation(0.1),
    layers.RandomZoom(0.1),
  ]
)
plt.figure(figsize=(10, 10))
for images, _ in train_ds.take(1):
  for i in range(9):
    augmented_images = data_augmentation(images)
    ax = plt.subplot(3, 3, i + 1)
    plt.imshow(augmented_images[0].numpy().astype("uint8"))
```

```
  plt.axis("off")
```

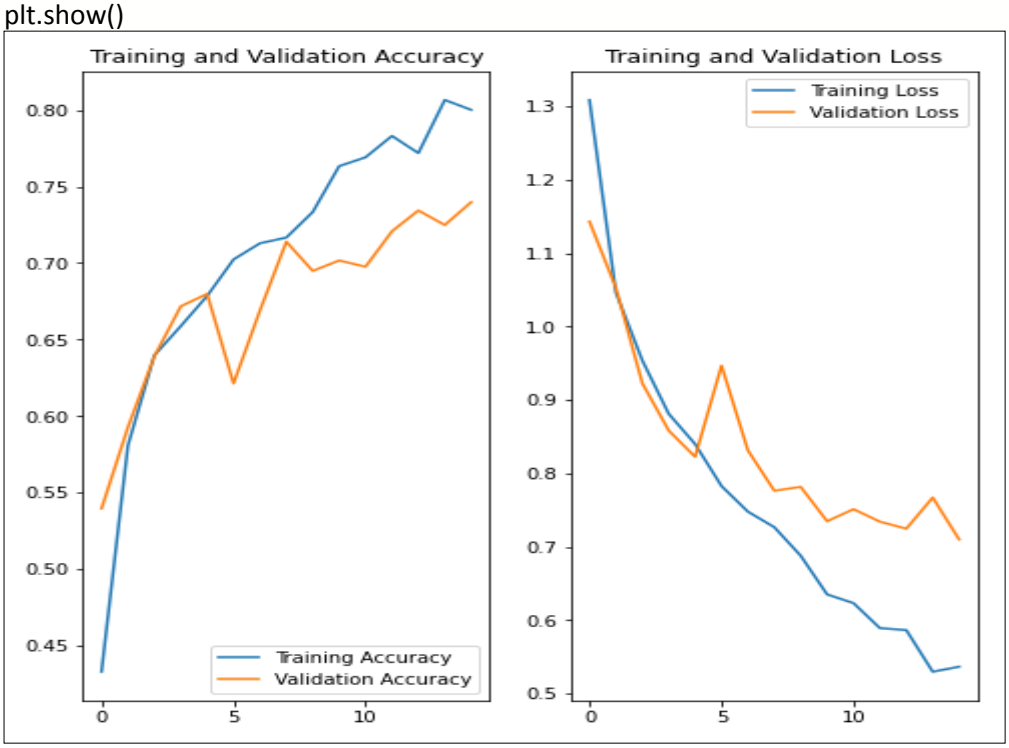
```
model = Sequential([
  data_augmentation,
  layers.Rescaling(1./255),
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Dropout(0.2),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(num_classes)
])
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 sequential_1 (Sequential)   (None, 180, 180, 3)       0

 rescaling_2 (Rescaling)     (None, 180, 180, 3)       0

 conv2d_3 (Conv2D)           (None, 180, 180, 16)      448

 max_pooling2d_3 (MaxPooling  (None, 90, 90, 16)       0
 2D)

 conv2d_4 (Conv2D)           (None, 90, 90, 32)        4640

 max_pooling2d_4 (MaxPooling  (None, 45, 45, 32)       0
 2D)

 conv2d_5 (Conv2D)           (None, 45, 45, 64)        18496

 max_pooling2d_5 (MaxPooling  (None, 22, 22, 64)       0
 2D)

 dropout (Dropout)           (None, 22, 22, 64)        0

 flatten_1 (Flatten)         (None, 30976)             0

 dense_2 (Dense)             (None, 128)               3965056

 dense_3 (Dense)             (None, 5)                 645

=================================================================
Total params: 3,989,285
Trainable params: 3,989,285
Non-trainable params: 0
_________________________________________________________________
```

```
epochs = 15
history = model.fit(
  train_ds,
  validation_data=val_ds,
  epochs=epochs
)
```

```
Epoch 1/15
92/92 [==============================] - 6s 50ms/step - loss: 1.3089 - accuracy: 0.4329 - val_loss: 1.1432 - val_accuracy: 0.5395
Epoch 2/15
92/92 [==============================] - 4s 46ms/step - loss: 1.0464 - accuracy: 0.5807 - val_loss: 1.0529 - val_accuracy: 0.5926
Epoch 3/15
92/92 [==============================] - 4s 47ms/step - loss: 0.9540 - accuracy: 0.6396 - val_loss: 0.9229 - val_accuracy: 0.6390
Epoch 4/15
92/92 [==============================] - 4s 47ms/step - loss: 0.8812 - accuracy: 0.6587 - val_loss: 0.8583 - val_accuracy: 0.6717
Epoch 5/15
92/92 [==============================] - 4s 47ms/step - loss: 0.8399 - accuracy: 0.6785 - val_loss: 0.8225 - val_accuracy: 0.6798
Epoch 6/15
92/92 [==============================] - 4s 47ms/step - loss: 0.7826 - accuracy: 0.7023 - val_loss: 0.9469 - val_accuracy: 0.6213
Epoch 7/15
92/92 [==============================] - 4s 47ms/step - loss: 0.7478 - accuracy: 0.7129 - val_loss: 0.8313 - val_accuracy: 0.6689
Epoch 8/15
92/92 [==============================] - 4s 47ms/step - loss: 0.7266 - accuracy: 0.7166 - val_loss: 0.7763 - val_accuracy: 0.7139
Epoch 9/15
92/92 [==============================] - 4s 46ms/step - loss: 0.6877 - accuracy: 0.7333 - val_loss: 0.7816 - val_accuracy: 0.6948
Epoch 10/15
92/92 [==============================] - 4s 47ms/step - loss: 0.6350 - accuracy: 0.7633 - val_loss: 0.7347 - val_accuracy: 0.7016
Epoch 11/15
92/92 [==============================] - 4s 47ms/step - loss: 0.6231 - accuracy: 0.7691 - val_loss: 0.7511 - val_accuracy: 0.6975
Epoch 12/15
92/92 [==============================] - 4s 46ms/step - loss: 0.5890 - accuracy: 0.7830 - val_loss: 0.7341 - val_accuracy: 0.7207
Epoch 13/15
92/92 [==============================] - 4s 47ms/step - loss: 0.5864 - accuracy: 0.7718 - val_loss: 0.7245 - val_accuracy: 0.7343
Epoch 14/15
92/92 [==============================] - 4s 47ms/step - loss: 0.5296 - accuracy: 0.8065 - val_loss: 0.7671 - val_accuracy: 0.7248
Epoch 15/15
92/92 [==============================] - 4s 47ms/step - loss: 0.5364 - accuracy: 0.8001 - val_loss: 0.7099 - val_accuracy: 0.7398
```

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(epochs)

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```




```
sunflower_url = "https://storage.googleapis.com/download.tensorflow.org/example_images/592px-Red_sunflower.jpg"
sunflower_path = tf.keras.utils.get_file('Red_sunflower', origin=sunflower_url)

img = tf.keras.utils.load_img(
    sunflower_path, target_size=(img_height, img_width)
```

```
)
img_array = tf.keras.utils.img_to_array(img)
img_array = tf.expand_dims(img_array, 0) # Create a batch

predictions = model.predict(img_array)
score = tf.nn.softmax(predictions[0])

print(
    "This image most likely belongs to {} with a {:.2f} percent confidence."
    .format(class_names[np.argmax(score)], 100 * np.max(score))
)
```

```
Downloading data from https://storage.googleapis.com/download.tensorflow.org/example_images/592px-Red_sunflower.jpg
122880/117948 [==============================] - 0s 0us/step
131072/117948 [==============================] - 0s 0us/step
This image most likely belongs to sunflowers with a 98.71 percent confidence.
```

**PRACTICAL 5C**
**DATA AUGMENTATION**

Data augmentation is a strategy that enables practitioners to significantly increase the diversity of data available for training models, without actually collecting new data. Data augmentation techniques such as cropping, padding, and horizontal flipping are commonly used to train large neural networks.

**CODE:**

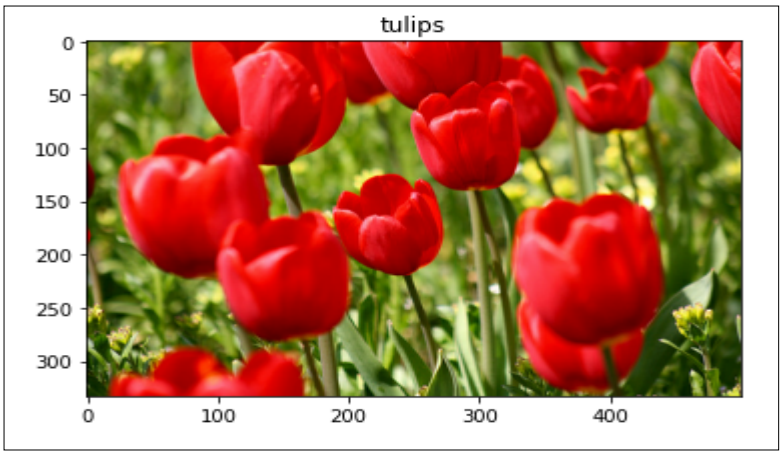
```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
import tensorflow_datasets as tfds

from tensorflow.keras import layers
(train_ds, val_ds, test_ds), metadata = tfds.load(
    'tf_flowers',
    split=['train[:80%]', 'train[80%:90%]', 'train[90%:]'],
    with_info=True,
    as_supervised=True,
)
```

```
Downloading and preparing dataset tf_flowers/3.0.1 (download: 218.21 MiB, generated: 221.83 MiB, total: 440.05 MiB) to /root/tensorflow_datasets/tf_flowers/3.0.1...
WARNING:absl:Dataset tf_flowers is hosted on GCS. It will automatically be downloaded to your
local data directory. If you'd instead prefer to read directly from our public
GCS bucket (recommended if you're running on GCP), you can instead pass
`try_gcs=True` to `tfds.load` or set `data_dir=gs://tfds-data/datasets`.

Dl Completed...: 100% 5/5 [00:02<00:00, 2.16 file/s]

Dataset tf_flowers downloaded and prepared to /root/tensorflow_datasets/tf_flowers/3.0.1. Subsequent calls will reuse this data.
```

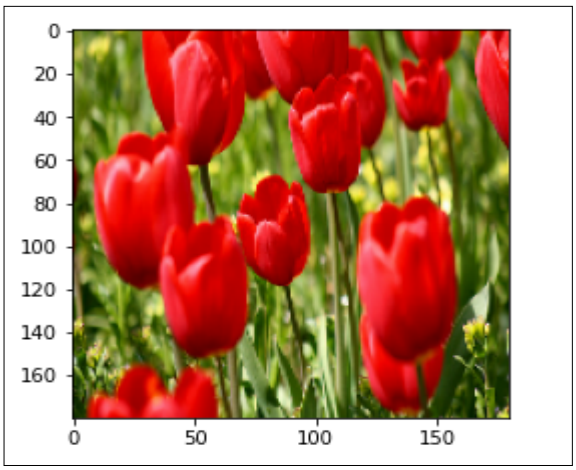
```
num_classes = metadata.features['label'].num_classes
print(num_classes)
```

```
5
```

```
get_label_name = metadata.features['label'].int2str

image, label = next(iter(train_ds))
_ = plt.imshow(image)
_ = plt.title(get_label_name(label))
```




```
IMG_SIZE = 180

resize_and_rescale = tf.keras.Sequential([
  layers.Resizing(IMG_SIZE, IMG_SIZE),
  layers.Rescaling(1./255)
])
result = resize_and_rescale(image)
_ = plt.imshow(result)
```




```
print("Min and max pixel values:", result.numpy().min(), result.numpy().max())
```

```
Min and max pixel values: 0.0 1.0
```

```
data_augmentation = tf.keras.Sequential([
  layers.RandomFlip("horizontal_and_vertical"),
  layers.RandomRotation(0.2),
])
# Add the image to a batch.
image = tf.expand_dims(image, 0)
plt.figure(figsize=(10, 10))
for i in range(9):
  augmented_image = data_augmentation(image)
```

```
 ax = plt.subplot(3, 3, i + 1)
 plt.imshow(augmented_image[0])
 plt.axis("off")
```

```
model = tf.keras.Sequential([
 # Add the preprocessing layers you created earlier.
 resize_and_rescale,
 data_augmentation,
 layers.Conv2D(16, 3, padding='same', activation='relu'),
 layers.MaxPooling2D(),
 # Rest of your model.
])
aug_ds = train_ds.map(
 lambda x, y: (resize_and_rescale(x, training=True), y))
batch_size = 32
AUTOTUNE = tf.data.AUTOTUNE

def prepare(ds, shuffle=False, augment=False):
 # Resize and rescale all datasets.
 ds = ds.map(lambda x, y: (resize_and_rescale(x), y),
             num_parallel_calls=AUTOTUNE)

 if shuffle:
   ds = ds.shuffle(1000)

 # Batch all datasets.
 ds = ds.batch(batch_size)
```

```
  # Use data augmentation only on the training set.
  if augment:
    ds = ds.map(lambda x, y: (data_augmentation(x, training=True), y),
                num_parallel_calls=AUTOTUNE)

  # Use buffered prefetching on all datasets.
  return ds.prefetch(buffer_size=AUTOTUNE)
train_ds = prepare(train_ds, shuffle=True, augment=True)
val_ds = prepare(val_ds)
test_ds = prepare(test_ds)


model = tf.keras.Sequential([
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(num_classes)
])
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])


epochs=5
history = model.fit(
  train_ds,
  validation_data=val_ds,
  epochs=epochs
)
```

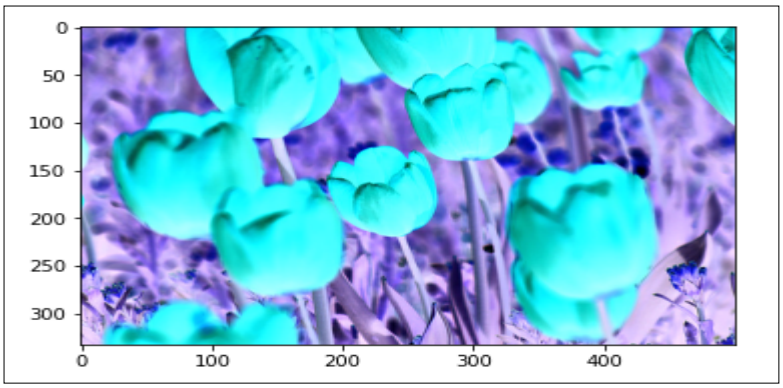
```
Epoch 1/5
92/92 [==============================] - 25s 156ms/step - loss: 1.2920 - accuracy: 0.4537 - val_loss: 1.2261 - val_accuracy: 0.4714
Epoch 2/5
92/92 [==============================] - 12s 124ms/step - loss: 1.0784 - accuracy: 0.5576 - val_loss: 0.9947 - val_accuracy: 0.6294
Epoch 3/5
92/92 [==============================] - 12s 127ms/step - loss: 0.9746 - accuracy: 0.6114 - val_loss: 0.9631 - val_accuracy: 0.6403
Epoch 4/5
92/92 [==============================] - 12s 125ms/step - loss: 0.9334 - accuracy: 0.6277 - val_loss: 0.8666 - val_accuracy: 0.6785
Epoch 5/5
92/92 [==============================] - 12s 126ms/step - loss: 0.8594 - accuracy: 0.6553 - val_loss: 0.9118 - val_accuracy: 0.6158
```

```
loss, acc = model.evaluate(test_ds)
print("Accuracy", acc)
```

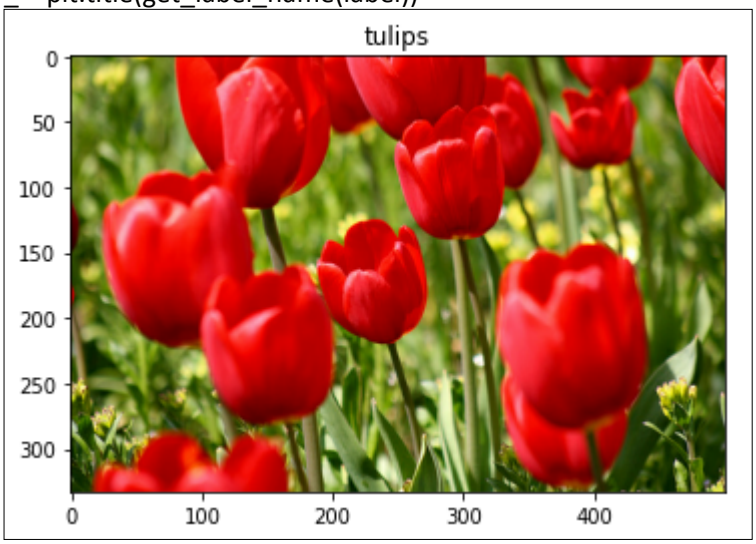
```
12/12 [==============================] - 1s 58ms/step - loss: 0.8202 - accuracy: 0.6540
Accuracy 0.6539509296417236
```

```
def random_invert_img(x, p=0.5):
  if  tf.random.uniform([]) < p:
    x = (255-x)
  else:
    x
  return x
def random_invert(factor=0.5):
  return layers.Lambda(lambda x: random_invert_img(x, factor))

random_invert = random_invert()
plt.figure(figsize=(10, 10))
for i in range(9):
  augmented_image = random_invert(image)
  ax = plt.subplot(3, 3, i + 1)
  plt.imshow(augmented_image[0].numpy().astype("uint8"))
  plt.axis("off")
```

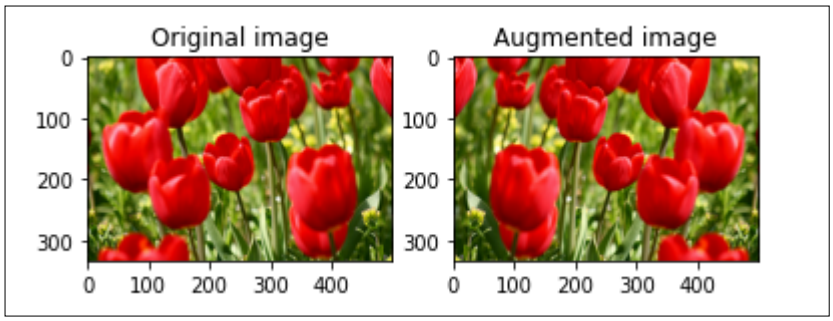
```
class RandomInvert(layers.Layer):
  def __init__(self, factor=0.5, **kwargs):
    super().__init__(**kwargs)
    self.factor = factor

  def call(self, x):
    return random_invert_img(x)
_ = plt.imshow(RandomInvert()(image)[0])
```



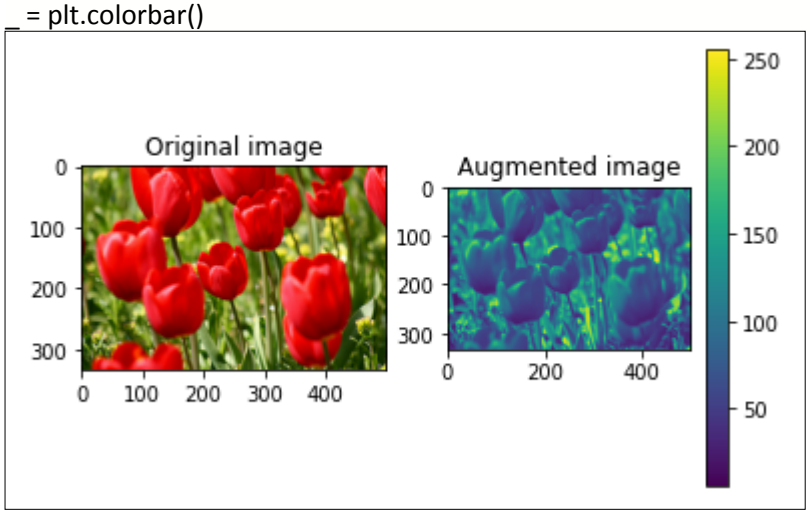

```
(train_ds, val_ds, test_ds), metadata = tfds.load(
    'tf_flowers',
    split=['train[:80%]', 'train[80%:90%]', 'train[90%:]'],
    with_info=True,
    as_supervised=True,
)
image, label = next(iter(train_ds))
_ = plt.imshow(image)
_ = plt.title(get_label_name(label))
```



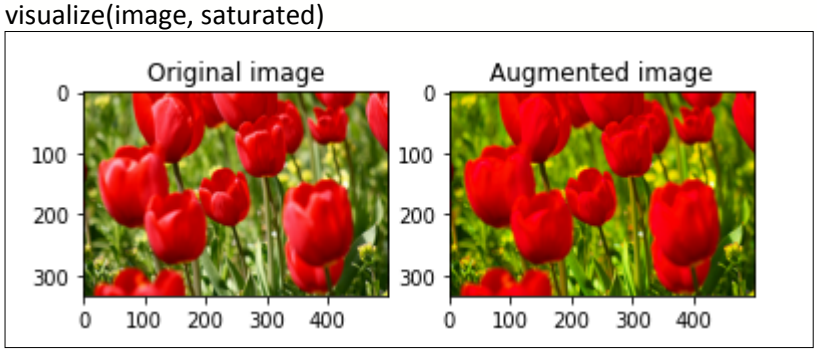

```
def visualize(original, augmented):
  fig = plt.figure()
  plt.subplot(1,2,1)
  plt.title('Original image')
  plt.imshow(original)

  plt.subplot(1,2,2)
  plt.title('Augmented image')
  plt.imshow(augmented)
flipped = tf.image.flip_left_right(image)
visualize(image, flipped)
```



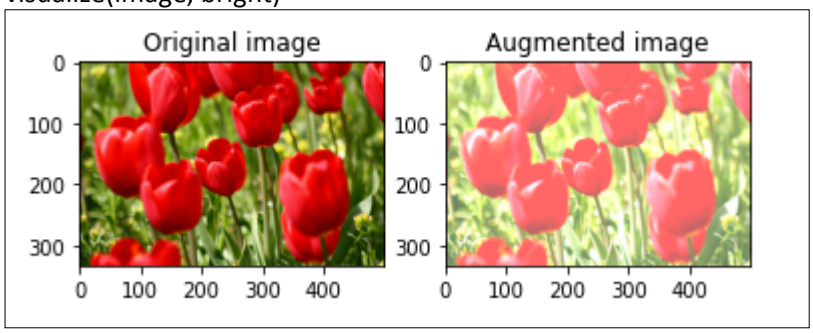

```
grayscaled = tf.image.rgb_to_grayscale(image)
visualize(image, tf.squeeze(grayscaled))
_ = plt.colorbar()
```




```
saturated = tf.image.adjust_saturation(image, 3)
visualize(image, saturated)
```

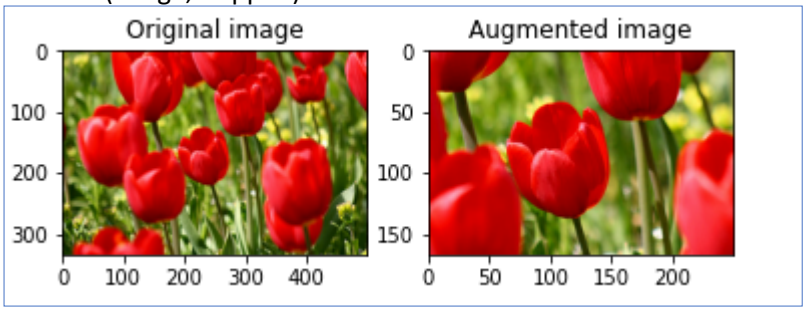



```
bright = tf.image.adjust_brightness(image, 0.4)
visualize(image, bright)
```

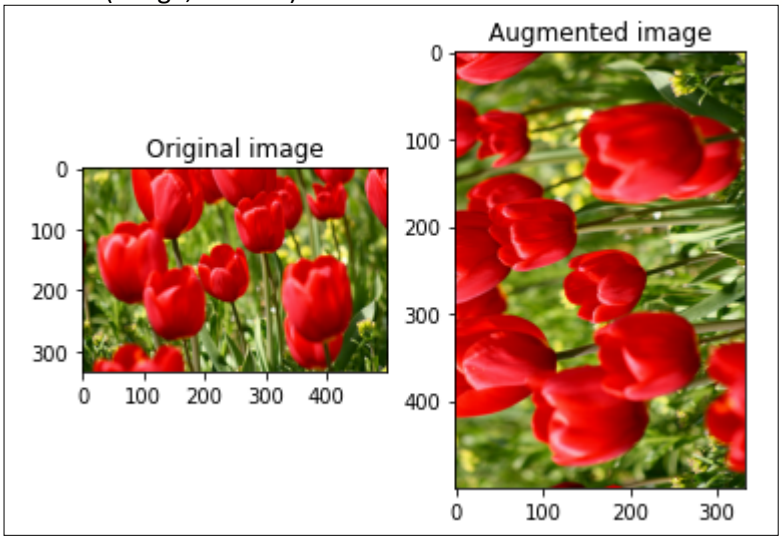



```
cropped = tf.image.central_crop(image, central_fraction=0.5)
```

visualize(image,cropped)




rotated = tf.image.rot90(image)
visualize(image, rotated)



**PRACTICAL 6**
**BUILDING RNN USING SINGLE NEURON**

Recurrent neural networks (RNN) are a class of neural networks that is powerful for modeling sequence data such as time series or natural language.

Schematically, a RNN layer uses a for loop to iterate over the timesteps of a sequence, while maintaining an internal state that encodes information about the timesteps it has seen so far.

The Keras RNN API is designed with a focus on:

- **Ease of use**: the built-in keras.layers.RNN, keras. layers.LSTM, keras.layers.GRU layers enable you to quickly build recurrent models without having to make difficult configuration choices.

**CODE**:

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
model = keras.Sequential()
# Add an Embedding layer expecting input vocab of size 1000, and
# output embedding dimension of size 64.
model.add(layers.Embedding(input_dim=1000, output_dim=64))

# Add a LSTM layer with 128 internal units.
model.add(layers.LSTM(128))

# Add a Dense layer with 10 units.
model.add(layers.Dense(10))

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, None, 64)          64000

 lstm (LSTM)                 (None, 128)               98816

 dense (Dense)               (None, 10)                1290

=================================================================
Total params: 164,106
Trainable params: 164,106
Non-trainable params: 0
_________________________________________________________________
```

```
model = keras.Sequential()
model.add(layers.Embedding(input_dim=1000, output_dim=64))

# The output of GRU will be a 3D tensor of shape (batch_size, timesteps, 256)
```

```
model.add(layers.GRU(256, return_sequences=True))

# The output of SimpleRNN will be a 2D tensor of shape (batch_size, 128)
model.add(layers.SimpleRNN(128))

model.add(layers.Dense(10))

model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 embedding_1 (Embedding)      (None, None, 64)          64000

 gru (GRU)                    (None, None, 256)         247296

 simple_rnn (SimpleRNN)       (None, 128)               49280

 dense_1 (Dense)              (None, 10)                1290

=================================================================
Total params: 361,866
Trainable params: 361,866
Non-trainable params: 0
_________________________________________________________________
```

```
encoder_vocab = 1000
decoder_vocab = 2000

encoder_input = layers.Input(shape=(None,))
encoder_embedded = layers.Embedding(input_dim=encoder_vocab, output_dim=64)(
    encoder_input
)

# Return states in addition to output
output, state_h, state_c = layers.LSTM(64, return_state=True, name="encoder")(
    encoder_embedded
)
encoder_state = [state_h, state_c]

decoder_input = layers.Input(shape=(None,))
decoder_embedded = layers.Embedding(input_dim=decoder_vocab, output_dim=64)(
    decoder_input
)

# Pass the 2 states to a new LSTM layer, as initial state
decoder_output = layers.LSTM(64, name="decoder")(
    decoder_embedded, initial_state=encoder_state
)
output = layers.Dense(10)(decoder_output)
```

model = keras.Model([encoder_input, decoder_input], output)
model.summary()

```
Model: "model"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_1 (InputLayer)           [(None, None)]       0           []

 input_2 (InputLayer)           [(None, None)]       0           []

 embedding_2 (Embedding)        (None, None, 64)     64000       ['input_1[0][0]']

 embedding_3 (Embedding)        (None, None, 64)     128000      ['input_2[0][0]']

 encoder (LSTM)                 [(None, 64),         33024       ['embedding_2[0][0]']
                                 (None, 64),
                                 (None, 64)]

 decoder (LSTM)                 (None, 64)           33024       ['embedding_3[0][0]',
                                                                  'encoder[0][1]',
                                                                  'encoder[0][2]']

 dense_2 (Dense)                (None, 10)           650         ['decoder[0][0]']

==================================================================================================
Total params: 258,698
Trainable params: 258,698
Non-trainable params: 0
__________________________________________________________________________________________________
```

**PRACTICAL 7**

**USING COVNET TO BUILD DEEP LEARNING MODEL**

This demo trains a Convolutional Neural Network on the MNIST digits dataset in your browser, with nothing but Javascript. The dataset is fairly easy and one should expect to get somewhere around 99% accuracy within few minutes. I used this python script to parse the original files into batches of images that can be easily loaded into page DOM with img tags.

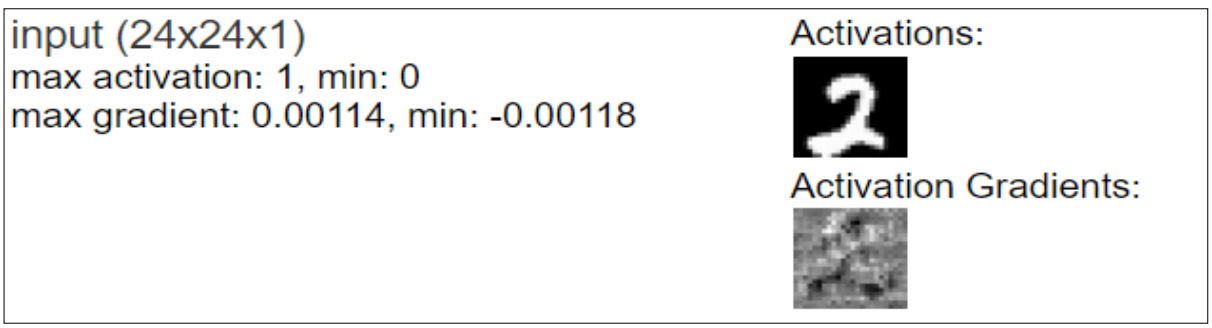
This network takes a 28x28 MNIST image and crops a random 24x24 window before training on it (this technique is called data augmentation and improves generalization). Similarly, to do prediction, 4 random crops are sampled and the probabilities across all crops are averaged to produce final predictions. The network runs at about 5ms for both forward and backward pass on my reasonably decent Ubuntu+Chrome machine.

## Instantiate a Network and Trainer

**CODE:**

```
layer_defs = [];

layer_defs.push({type:'input', out_sx:24, out_sy:24, out_depth:1});

layer_defs.push({type:'conv', sx:5, filters:8, stride:1, pad:2, activation:'relu'});

layer_defs.push({type:'pool', sx:2, stride:2});

layer_defs.push({type:'conv', sx:5, filters:16, stride:1, pad:2, activation:'relu'});

layer_defs.push({type:'pool', sx:3, stride:3});

layer_defs.push({type:'softmax', num_classes:10});

net = new convnetjs.Net();

net.makeLayers(layer_defs);

trainer = new convnetjs.SGDTrainer(net, {method:'adadelta', batch_size:20, l2_decay:0.001});
```

## Network Visualization

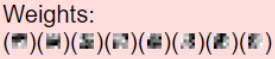


conv (24x24x8)
filter size 5x5x1, stride 1
max activation: 3.52966, min: -3.75137
max gradient: 0.01445, min: -0.02289
parameters: 8x5x5x1+8 = 208

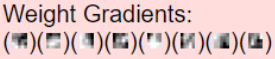
Activations:

Activation Gradients:

Weights:



Weight Gradients:



**PRACTICAL 8**

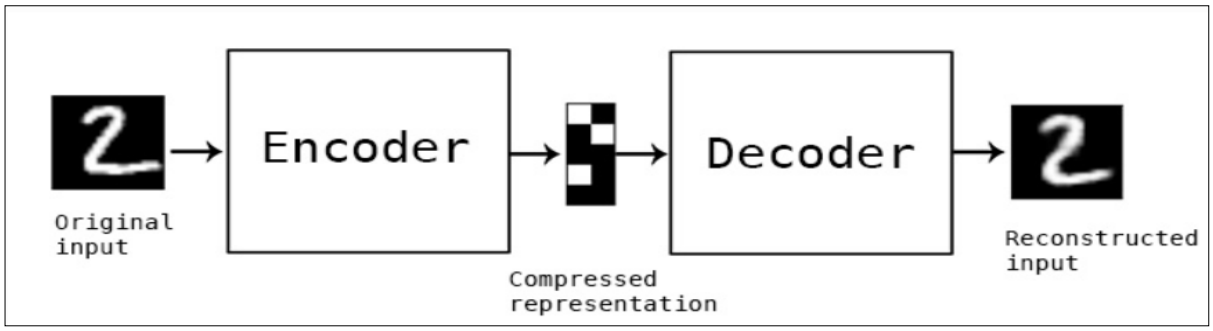
**IMPLEMENTING A SINGLE AUTOENCODER BASED ON FULLY CONNECTED LAYER**

"Autoencoding" is a data compression algorithm where the compression and decompression functions are 1) data-specific, 2) lossy, and 3) learned automatically from examples rather than engineered by a human. Additionally, in almost all contexts where the term "autoencoder" is used, the compression and decompression functions are implemented with neural networks.

1) Autoencoders are data-specific, which means that they will only be able to compress data similar to what they have been trained on. This is different from, say, the MPEG-2 Audio Layer III (MP3) compression algorithm, which only holds assumptions about "sound" in general, but not about specific types of sounds. An autoencoder trained on pictures of faces would do a rather poor job of compressing pictures of trees, because the features it would learn would be face-specific.

2) Autoencoders are lossy, which means that the decompressed outputs will be degraded compared to the original inputs (similar to MP3 or JPEG compression). This differs from lossless arithmetic compression.

3) Autoencoders are learned automatically from data examples, which is a useful property: it means that it is easy to train specialized instances of the algorithm that will perform well on a specific type of input. It doesn't require any new engineering, just appropriate training data.




**CODE:**

```
import keras
from keras import layers


# This is the size of our encoded representations
encoding_dim = 32  # 32 floats -> compression of factor 24.5, assuming the input is 784 floats


# This is our input image
input_img = keras.Input(shape=(784,))
# "encoded" is the encoded representation of the input
encoded = layers.Dense(encoding_dim, activation='relu')(input_img)
# "decoded" is the lossy reconstruction of the input
decoded = layers.Dense(784, activation='sigmoid')(encoded)

# This model maps an input to its reconstruction
autoencoder = keras.Model(input_img, decoded)
```

```
#Let's also create a separate encoder model:
# This model maps an input to its encoded representation
encoder = keras.Model(input_img, encoded)

#As well as the decoder model:
# This is our encoded (32-dimensional) input
encoded_input = keras.Input(shape=(encoding_dim,))
# Retrieve the last layer of the autoencoder model
decoder_layer = autoencoder.layers[-1]
# Create the decoder model
decoder = keras.Model(encoded_input, decoder_layer(encoded_input))

#Now let's train our autoencoder to reconstruct MNIST digits.
#First, we'll configure our model to use a per-pixel binary crossentropy loss, and the Adam
optimizer:
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

#Let's prepare our input data. We're using MNIST digits, and we're discarding the labels #(since
we're only interested in encoding/decoding the input images).
from keras.datasets import mnist
import numpy as np
(x_train, _), (x_test, _) = mnist.load_data()

#We will normalize all values between 0 and 1 and we will flatten the 28x28 images into #vectors of
size 784.
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))
print(x_train.shape)
print(x_test.shape)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
11501568/11490434 [==============================] - 0s 0us/step
(60000, 784)
(10000, 784)
```

```
#Now let's train our autoencoder for 50 epochs:
autoencoder.fit(x_train, x_train,
                epochs=50,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))
```

```
Epoch 43/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0922 - val_loss: 0.0912
Epoch 44/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0922 - val_loss: 0.0911
Epoch 45/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0922 - val_loss: 0.0911
Epoch 46/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0922 - val_loss: 0.0912
Epoch 47/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0922 - val_loss: 0.0911
Epoch 48/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0922 - val_loss: 0.0911
Epoch 49/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0922 - val_loss: 0.0912
Epoch 50/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0922 - val_loss: 0.0911
<keras.callbacks.History at 0x7f5eed3fbd50>
```

#After 50 epochs, the autoencoder seems to reach a stable train/validation loss value of #about 0.09. We can try to visualize the reconstructed inputs and the encoded #representations. We will use Matplotlib.

```
# Encode and decode some digits
# Note that we take them from the *test* set
encoded_imgs = encoder.predict(x_test)
decoded_imgs = decoder.predict(encoded_imgs)

# Use Matplotlib (don't ask)
import matplotlib.pyplot as plt
n = 10  # How many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
    # Display original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```