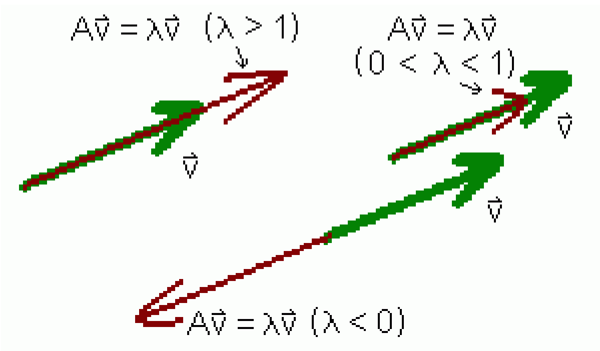
Practical 1

MATRIX MULTIPLICATION, EIGEN VECTORS, EIGENVALUE COMPUTATION USING TENSORFLOW

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=λv.Av=λv.

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

```
import tensorflow as tf
print("Matrix Multiplication Demo")
```

```
Matrix Multiplication Demo
```

```
x=tf.constant([1,2,3,4,5,6],shape=[2,3])
print(x)
```

```
tf.Tensor(
[[1 2 3]
 [4 5 6]], shape=(2, 3), dtype=int32)
```

```
y=tf.constant([7,8,9,10,11,12],shape=[3,2])
print(y)
```

```
tf.Tensor(
[[ 7  8]
 [ 9 10]
 [11 12]], shape=(3, 2), dtype=int32)
```

```
z=tf.matmul(x,y)
```

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

```
Product: tf.Tensor(
[[ 58  64]
 [139 154]], shape=(2, 2), dtype=int32)
```

```
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))
```

```
Matrix A:
[[8.7307205 9.0639515]
 [5.393942  9.614674 ]]
```

```
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))
```

```
Eigen Vectors:
[[-0.73541343 -0.6776187 ]
 [ 0.6776187  -0.73541343]]

Eigen values:
[ 3.760678 14.584717]
```

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

```
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())
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_2 (Dense)             (None, 2)                 6

 dense_3 (Dense)             (None, 1)                 3

=================================================================
Total params: 9
Trainable params: 9
Non-trainable params: 0
_________________________________________________________________
None
```

```
print(model.get_weights())
```

```
[array([[-0.8435025 , -0.37239122],
       [ 0.34912777, -1.1265315 ]], dtype=float32), array([0., 0.], dtype=float32),
       [-1.3532746 ]], dtype=float32), array([0.], dtype=float32)]
```

```
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)
```

```
Epoch 1/1000
1/1 [==============================] - 1s 1s/step - loss: 0.7042 - accuracy: 0.50
Epoch 2/1000
1/1 [==============================] - 0s 10ms/step - loss: 0.7041 - accuracy: 0.
Epoch 3/1000
1/1 [==============================] - 0s 8ms/step - loss: 0.7040 - accuracy: 0.2
Epoch 4/1000
1/1 [==============================] - 0s 8ms/step - loss: 0.7039 - accuracy: 0.2
Epoch 5/1000
1/1 [==============================] - 0s 8ms/step - loss: 0.7038 - accuracy: 0.2
Epoch 6/1000
1/1 [==============================] - 0s 9ms/step - loss: 0.7036 - accuracy: 0.2
Epoch 7/1000
1/1 [==============================] - 0s 8ms/step - loss: 0.7035 - accuracy: 0.2
Epoch 8/1000
1/1 [==============================] - 0s 7ms/step - loss: 0.7034 - accuracy: 0.2
Epoch 9/1000
1/1 [==============================] - 0s 11ms/step - loss: 0.7033 - accuracy: 0.
Epoch 10/1000
1/1 [==============================] - 0s 8ms/step - loss: 0.7032 - accuracy: 0.2
Epoch 11/1000
1/1 [==============================] - 0s 7ms/step - loss: 0.7031 - accuracy: 0.2
Epoch 12/1000
1/1 [==============================] - 0s 8ms/step - loss: 0.7030 - accuracy: 0.2
Epoch 13/1000
1/1 [==============================] - 0s 8ms/step - loss: 0.7029 - accuracy: 0.2
Epoch 14/1000
1/1 [==============================] - 0s 9ms/step - loss: 0.7028 - accuracy: 0.2
Epoch 15/1000
1/1 [==============================] - 0s 9ms/step - loss: 0.7027 - accuracy: 0.2
Epoch 16/1000
1/1 [==============================] - 0s 9ms/step - loss: 0.7026 - accuracy: 0.2
Epoch 17/1000
1/1 [==============================] - 0s 10ms/step - loss: 0.7025 - accuracy: 0.
Epoch 18/1000
1/1 [==============================] - 0s 9ms/step - loss: 0.7024 - accuracy: 0.2
Epoch 19/1000
1/1 [==============================] - 0s 8ms/step - loss: 0.7023 - accuracy: 0.2
Epoch 20/1000
1/1 [==============================] - 0s 8ms/step - loss: 0.7022 - accuracy: 0.2
Epoch 21/1000
1/1 [==============================] - 0s 14ms/step - loss: 0.7021 - accuracy: 0.
Epoch 22/1000
1/1 [==============================] - 0s 9ms/step - loss: 0.7020 - accuracy: 0.2
Epoch 23/1000
1/1 [==============================] - 0s 8ms/step - loss: 0.7019 - accuracy: 0.2
Epoch 24/1000
1/1 [==============================] - 0s 12ms/step - loss: 0.7018 - accuracy: 0.
Epoch 25/1000
1/1 [==============================] - 0s 9ms/step - loss: 0.7017 - accuracy: 0.2
Epoch 26/1000
1/1 [==============================] - 0s 10ms/step - loss: 0.7016 - accuracy: 0.
Epoch 27/1000
1/1 [==============================] - 0s 10ms/step - loss: 0.7015 - accuracy: 0.
Epoch 28/1000
1/1 [==============================] - 0s 8ms/step - loss: 0.7014 - accuracy: 0.2
Epoch 29/1000
```

```
print(model.get_weights())
```

```
[array([[-0.8435025 , -0.37239122],
       [ 0.16758746, -1.1265315 ]], dtype=float32), array([-0.18154037,  0.        ]
       [-1.3532746 ]], dtype=float32), array([2.1596843e-08], dtype=float32)]
```

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

```
[[0.5]
 [0.5]
 [0.5]
 [0.5]]
```

PRACTICAL 3A

CLASSIFICATION USING DNN

[ ] ↳ *10 cells hidden*

PRACTICAL 3B

BINARY CLASSIFICATION USING MLP

[ ] ↳ *7 cells hidden*

PRACTICAL 4

PREDICTING THE PROBABILITY OF THE CLASS

[ ] ↳ *5 cells hidden*

PRACTICAL 5A

CNN FOR CIFAR10 IMAGES

[ ] ↳ *9 cells hidden*

PRACTICAL 5B

IMAGE CLASSIFICATION

[ ] ↳ 27 cells hidden

PRACTICAL 5C

DATA AUGMENTATION

[ ] ↳ 22 cells hidden

PRACTICAL 6

BUILDING RNN USING SINGLE NEURON

[ ] ↳ 9 cells hidden

PRACTICAL 7

NLP CORPUS

[ ] ↳ 91 cells hidden

PRACTICAL 8

Lemmatization, Stemming, Tokenization, Stopwords

[ ] ↳ 46 cells hidden

PRACTICAL 9

One-Hot Encoding, Bag of Words, N-grams, TF-IDF

[ ] ↳ 11 cells hidden

PRACTICAL 10

Word Embedding

[ ] ↳ 11 cells hidden

other ipynb files

cfg - https://colab.research.google.com/drive/1y9yIwn6X8ZyN52y8tA6M2v8mDeoDFwOz?usp=sharing
text to speech -
https://colab.research.google.com/drive/1mR6gv2Yr5IYJpb6T_MdQfFlwJcjkPZ-l?usp=sharing