In [1]:
```
import tensorflow as tf
from tensorflow import keras
```

In [37]:
```
from keras.models import Sequential
from keras.layers import Dense, Activation
model = Sequential([
    Dense (units = 5, input_shape = (3,), activation ='relu'),
    Dense (units = 2, activation ='softmax')
    ])
```

OR we can write following way

In [38]:
```
from keras.models import Sequential
from keras.layers import Dense, Activation
layers = [
    Dense (units = 5, input_shape = (3,), activation ='relu'),
    Dense (units = 2, activation ='softmax')
    ]
model = Sequential(layers)
```

OR we can write following way

In [42]:
```
from keras.models import Sequential
from keras.layers import Dense, Activation
model = Sequential()
model.add(Dense (units = 5, input_shape = (3,), activation ='relu')),
model.add(Dense (units = 2, activation ='softmax'))
```

If one is unsure about the activation function to utilise, just select "RELU", which is a broad activation function that is used in most circumstances these days. ReLU activation function is currently the most commonly used function for the hidden layers (and never for the output layer) for any type of neural network. If the ReLU function does not seem to provide the best results, changing the activation to leaky ReLU might in some cases yield better results and overall performance.

Although the swish activation function does seem to outperform the ReLU function on complex applications, it should be mostly used for only larger neural networks having depths of greater than 50 layers.

If our output layer is meant to be used for binary identification/detection, the sigmoid function is an obvious choice. For binary classification applications, the output (top-most) layer should be activated by the sigmoid function — also for multi-label classification.

The softmax function is always used for multi-class classification applications since it results in the output probabilities of each of the 'k' number of classes in the application which all adds upto 1.

The linear activation function should only be used in the output layer of a simple regression neural network.

For recurrent neural networks (RNNs) the tanh activation function is preferred for the hidden layer(s). It is set by default in TensorFlow.

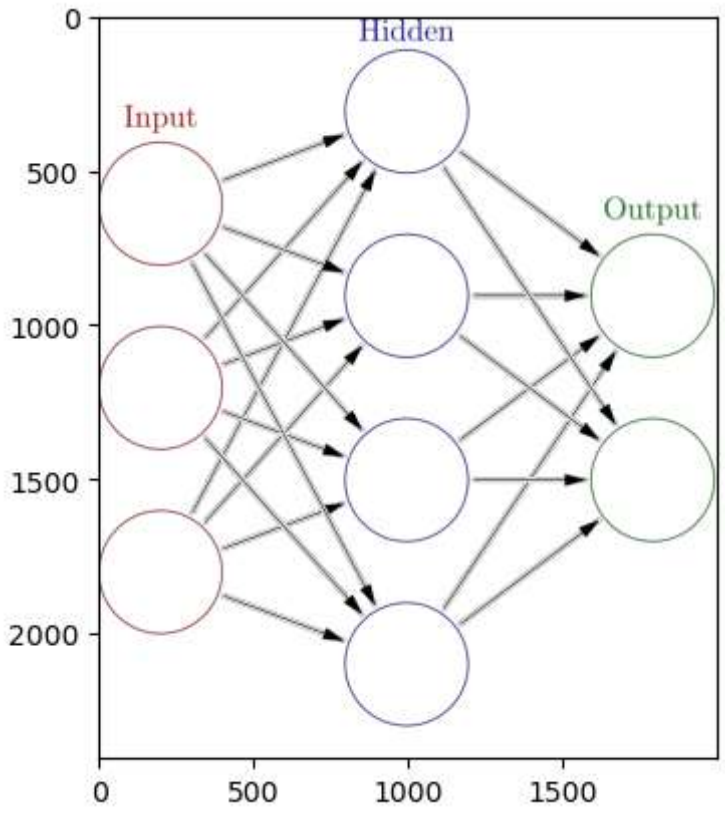
In [33]:
```
a=np.array([0,0,1,1,1,1,1,1,1,1])
print(a.shape)
b=np.array([2.3,1.1])
print(b.shape)
```

```
(10,)
(2,)
```

In [18]:
```
import numpy as  np
from scipy import *
import matplotlib.pyplot as plt
%matplotlib inline
import imageio
img = np.expand_dims(imageio.imread("neural_network.png"),0)
plt.imshow(img[0])
```

```
C:\Users\shalini193091\.conda\envs\tf\lib\site-packages\ipykernel_launcher.py:6: Depr
ecationWarning: Starting with ImageIO v3 the behavior of this function will switch to
that of iio.v3.imread. To keep the current behavior (and make this warning dissapear)
use `import imageio.v2 as imageio` or call `imageio.v2.imread` directly.
```

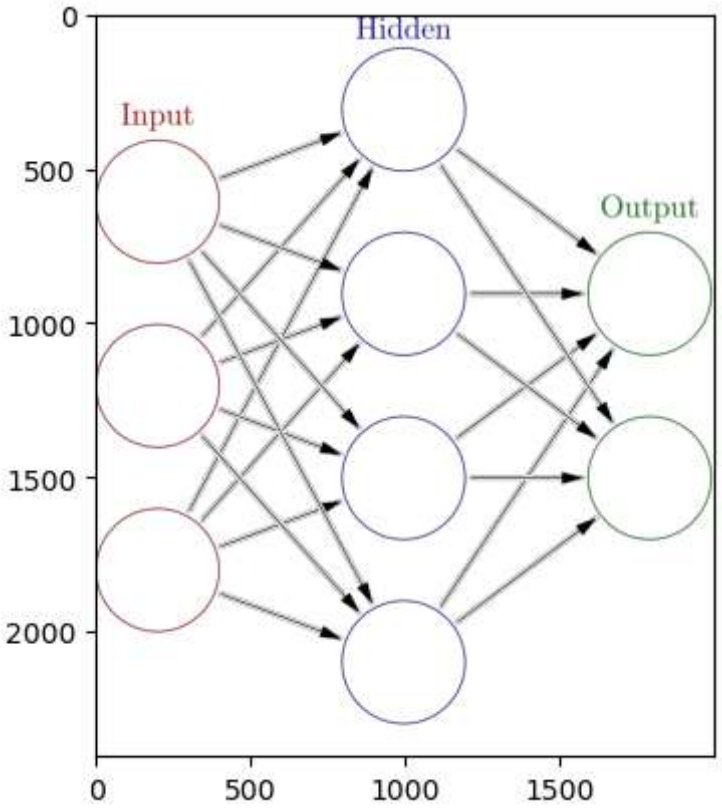
Out[18]:
```
<matplotlib.image.AxesImage at 0x1ec23a231c8>
```



In [28]:
```
import matplotlib.image as mpimg
#fig=plt.figure(figsize=(8,6),dpi=500)
```

```
img = mpimg.imread("neural_network.png")
plt.imshow(img)
```

Out[28]:
```
<matplotlib.image.AxesImage at 0x1ec27e5b348>
```



In [43]:
```
# from PIL import Image
# Open the image form working directory
image = Image.open("neural_network.png")
# summarize some details about the image
print(image.format)
print(image.size)
print(image.mode)
# show the image
image.show()
```

```
PNG
(2000, 2405)
RGBA
```

Why Loss Function is important?

Famous author Peter Druker says You can't improve what you can't measure. That's why the loss function comes into the picture to evaluate how well your algorithm is modeling your dataset.

If the value of the loss function is lower then it's a good model otherwise, we have to change the parameter of the model and minimize the loss.

Loss function vs Cost function

Most people confuse loss function and cost function. let's understand what is loss function and cost function. Cost function and Loss function are synonymous and used interchangeably but they are different.

Loss Function:

A loss function/error function is for a single training example/input.

Cost Function:

A cost function, on the other hand, is the average loss over the entire training dataset.

Loss function in Deep Learning

1. Regression

a. MSE(Mean Squared Error)

b. MAE(Mean Absolute Error)

c. Hubber loss

1. Classification

a. Binary cross-entropy (binary classification)

b. Categorical cross-entropy (multi class classification)

c. Sparse Categorical cross-entropy (If target column has One hot encoding to classes like 0 0 1, 0 1 0, 1 0 0 then use categorical cross-entropy. And if the target column has Numerical encoding to classes like 1,2,3,4….n then use sparse categorical cross-entropy). Sparse categorical cross-entropy faster than Categorical cross-entropy.

1. AutoEncoder

a. KL Divergence

1. GAN

a. Discriminator loss

b. Minmax GAN loss

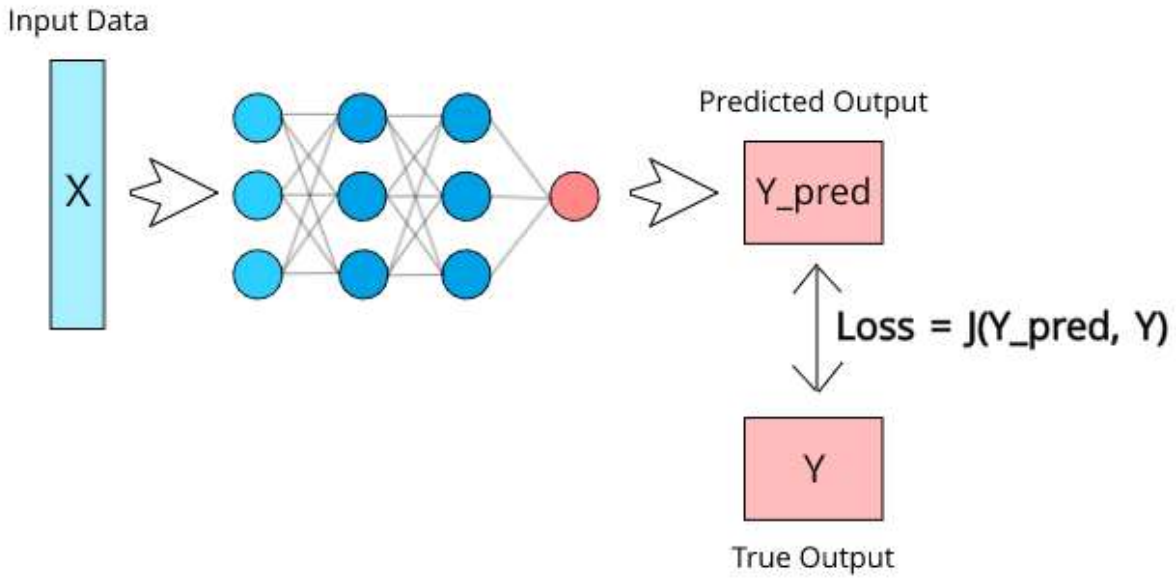
1. Object detection

a. Focal loss

6.Word embeddings

a. Triplet loss

Loss Function



In [1]:
```
import keras
from keras.models import Sequential
from keras.layers import Activation
from keras.layers.core import Dense
from keras.optimizers import Adam
from keras.metrics import categorical_crossentropy
```

In [2]:
```
model = Sequential([
    Dense(units=16, input_shape=(1,), activation='relu'),
    Dense(units=32, activation='relu'),
    Dense(units=2, activation='sigmoid')
])
```

In [3]:
```
model.compile(
    optimizer=Adam(learning_rate=0.0001),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

In [13]:
```
import numpy as np
from random import randint
from sklearn.preprocessing import MinMaxScaler
train_labels = []
train_samples = []

for i in range (1000):
    random_younger = randint (13,64)
    train_samples.append(random_younger)
    train_labels.append(0)

    random_older = randint (65,100)
```

```
    train_samples.append(random_older)
    train_labels.append(1)

for i in range (50):
    random_younger = randint (13,64)
    train_samples.append(random_younger)
    train_labels.append(1)

    random_older = randint (65,100)
    train_samples.append(random_older)
    train_labels.append(0)

#print raw data
for i in train_samples:
    print (i)

train_labels = np.array (train_labels)
train_samples = np.array (train_samples)

scaler = MinMaxScaler(feature_range = (0,1))
scaled_train_samples = scaler.fit_transform ((train_samples).reshape(-1,1))
```

53
84
55
82
24
78
37
85
13
76
53
75
32
73
49
65
37
73
24
97
50
73
51
84
34
70
41
81
49
92
47
67
38
93
22
84
27
84
46
72
37
97
46
92
34
78
51
79
59
70
33
77
62
85
55
95
47
70
31
68

26
70
54
69
22
87
34
67
18
98
51
68
43
77
29
82
39
89
13
67
49
86
54
93
43
68
21
75
49
75
60
86
54
84
32
98
22
94
29
90
61
69
34
73
63
93
13
81
59
66
63
82
26
91
59
96
14
99
58
76

39
65
29
67
20
78
35
77
62
92
45
78
27
75
31
90
41
98
16
95
58
96
57
77
45
66
37
92
21
81
44
90
38
94
39
78
39
100
36
87
53
66
16
98
24
77
24
73
45
69
29
81
26
88
62
97
55
74
21
93

26
95
24
99
57
69
63
75
20
96
17
70
50
99
45
65
24
96
40
82
47
66
39
91
27
75
24
84
40
74
44
68
16
95
14
73
48
96
42
99
42
75
28
85
19
81
41
100
36
99
56
95
51
77
17
93
14
72
26
65

14
67
27
71
42
99
16
77
22
85
58
72
46
68
45
74
31
86
38
84
54
88
22
68
38
72
55
83
58
73
37
96
34
76
63
96
28
84
40
76
26
75
50
89
47
89
30
69
55
67
50
70
44
69
64
67
13
97
32
99

27
75
57
66
32
98
36
90
36
99
53
79
40
94
16
89
14
94
33
67
54
95
40
94
16
67
60
77
17
73
27
81
23
82
58
66
57
99
14
91
30
94
40
75
58
93
14
71
63
71
52
66
32
75
46
70
47
77
30
83

37
82
57
88
27
77
23
66
29
69
25
79
24
69
16
69
59
92
40
91
22
97
23
99
23
84
44
87
60
88
63
81
46
99
44
97
42
93
63
74
43
72
29
70
20
75
53
77
48
66
47
91
33
68
22
72
34
81
48
82

55
70
62
92
42
91
52
78
43
71
22
75
47
73
36
68
44
99
13
68
59
86
46
66
31
90
56
93
23
73
55
65
31
100
30
82
33
81
47
70
35
90
48
93
63
72
64
84
43
100
62
83
45
72
44
86
59
71
25
80

28
85
35
89
18
82
22
96
48
74
52
86
42
87
31
93
29
91
54
90
63
94
22
77
29
93
59
81
42
75
13
72
47
83
21
70
31
67
44
87
49
86
34
86
29
94
25
69
16
82
44
78
27
72
56
72
26
87
49
98

21
68
17
65
44
80
35
77
45
88
29
84
26
79
60
99
49
80
29
73
44
89
28
68
14
92
27
98
36
85
31
91
21
76
33
81
22
100
30
97
54
84
42
65
59
93
30
81
26
84
34
66
35
79
25
90
53
91
19
86

46
68
32
72
38
79
26
66
53
93
15
78
49
88
40
65
62
78
46
97
35
77
55
99
63
68
52
71
19
94
32
74
63
69
49
94
23
98
52
78
14
80
52
97
53
77
55
66
50
72
42
84
51
91
16
78
19
98
27
66

38
93
25
71
46
68
57
79
61
75
16
97
40
72
27
73
63
99
38
98
51
74
39
92
49
77
60
96
14
72
13
94
55
86
56
69
19
89
55
82
35
70
48
75
42
71
41
89
62
96
45
76
19
71
19
90
27
98
24
80

50
66
61
85
40
85
31
100
55
93
57
99
26
88
55
65
51
97
45
80
26
65
41
98
55
95
26
74
24
97
18
70
53
71
25
68
31
79
20
94
62
82
57
81
16
100
55
66
51
88
29
93
37
77
24
93
61
100
21
100

36
88
59
72
13
85
17
73
41
89
25
84
38
79
30
92
13
74
38
76
49
93
24
74
37
81
62
70
25
92
21
98
58
92
54
98
23
73
44
79
34
69
46
80
42
90
42
94
50
94
47
97
42
87
45
89
47
72
55
99

32
66
36
65
53
74
23
76
31
70
46
84
49
86
61
93
57
71
58
88
33
86
58
71
58
93
45
90
23
73
21
73
58
88
32
66
42
99
41
96
62
98
45
78
51
74
47
92
50
80
19
82
45
82
25
97
35
74
61
96

41
91
26
83
23
89
47
75
16
90
32
69
54
74
51
84
41
83
18
92
62
66
45
83
59
82
40
100
57
70
40
92
63
90
62
79
63
74
64
76
36
75
56
89
21
92
37
92
21
75
60
95
16
92
32
71
29
74
18
98

62
87
24
79
29
69
62
71
58
68
56
88
53
75
43
89
22
98
64
96
15
78
41
90
34
86
24
77
50
88
45
78
47
87
32
94
22
90
40
92
39
97
15
69
63
96
47
96
63
84
24
86
25
70
64
79
55
90
24
88

25
83
46
90
32
97
61
97
16
74
18
84
40
91
45
66
18
86
40
75
15
97
25
85
19
92
35
86
44
99
42
100
26
93
41
84
60
82
58
97
16
98
36
92
17
94
43
96
47
66
41
89
51
80
41
76
27
84
20
73

61
80
32
92
63
70
43
86
42
88
25
75
60
75
25
93
56
92
35
95
54
89
62
69
16
81
29
98
58
77
40
78
17
93
42
100
25
83
15
79
25
90
17
100
35
90
44
94
57
83
56
94
35
85
42
96
35
85
23
65

41
67
58
68
21
66
20
84
31
86
19
79
52
90
25
85
32
69
30
81
58
94
18
70
20
65
47
84
57
78
47
95
59
69
24
78
57
69
53
82
40
94
30
69
13
83
57
89
17
99
54
69
55
73
50
66
42
96
18
90

26
80
40
94
24
91
61
93
45
88
57
71
17
98
30
90
38
85
53
74
21
90
16
65
61
67
48
79
41
77
24
93
37
92
29
92
41
76
54
67
63
71
22
95
35
90
18
71
13
86
38
73
52
79
23
87
58
81
45
65

52
72
46
96
48
78
34
67
18
89
41
67
15
85
29
89
23
73
21
76
19
83
35
77
22
69
53
86
56
90
23
97
51
85
51
75
34
83
41
97
63
74
27
79
53
67
14
87
32
65
49
88
43
70
13
83
40
69
14
90

38
83
17
95
52
77
56
75
38
94
43
67
27
88
16
94
16
65
60
94
48
80
54
98
44
81
37
82
51
87
15
72
63
95
14
86
16
99
34
79
60
94
58
88
18
78
15
94
41
84
30
97
57
76
52
84
44
77
34
86

45
95
41
95
58
98
31
100
38
84
31
91
40
82
29
100
44
77
22
87
33
77
52
76
27
98
17
83
55
84
21
100
17
65
50
96
64
85
51
85
34
91
50
96
19
100
14
85
60
99
32
95
59
96
33
97
41
82
26
74

64
85
64
82
38
74
23
73
28
73
55
66
46
86
30
95
37
91
34
98
23
90
36
73
21
77
17
86
45
84
57
69
61
94
57
86
48
96
63
69
51
72
41
87
27
78
16
96
28
65
35
89
61
88
40
84
58
68
27
81

34
71
17
80
32
99
14
97
28
98
63
84
42
75
14
67
41
79
14
77
61
83
43
89
54
74
42
100
16
88
40
96
56
89
64
82
58
87
29
100
59
99
54
83
55
78
41
92
31
73
45
91
56
89
17
72
38
68
63
83

27
84
57
100
16
87
36
65
14
86
30
79
35
83
32
89
55
71
25
77
18
79
54
95
30
83
39
93
42
81
39
79
26
89
36
88
55
95
39
67
24
92
42
84
60
89
34
99
25
94
30
94
37
84
13
89
27
65
33
67

62
80
39
83
30
83
54
69
51
72
28
68
64
72
42
99
29
69
50
70
57
94
50
85
56
92
64
95
39
81
37
100
52
77
40
85
47
90
31
72
13
99
24
74
60
97
60
69
32
74
45
77
46
100
29
84
18
99
43
98

34
70
57
95
29
100
53
87
33
88
58
67
31
82
58
100
23
71
55
66
16
86
40
85
30
99
54
74
59
80
36
71
48
67
25
73
58
78
61
66
45
88
64
100
47
75
63
90
28
71
43
75
18
74
30
84
36
94
28
65

38
98
34
72
48
96
58
100
25
74
51
83
39
92
42
99
56
82
27
100
15
82
40
77
64
81
29
69
62
91
19
67
23
96
45
81
41
91
56
74
42
90
44
80
41
80
17
93
36
65
63
92
24
93
35
95
51
87
26
100

48
67
52
76
60
87
15
79
58
73
63
73
57
83
64
94
34
76
60
79
14
83
61
69
58
67
31
82
43
100
53
95
40
87
19
68
51
72
16
93
61
67
30
87
30
92
32
94
63
92
39
89
48
79
16
82
26
76
18
93

47
69
18
87
57
98
46
66
64
66
58
89
18
71
29
81
19
79
53
66
54
100
47
81
18
66
26
98
31
93
20
97
36
87
18
67
42
68
16
100
30
86
39
89
27
68
60
96
32
96
33
81
48
88
18
65
49
91
64
76

28
97
21
85
36
86
49
88
51
71
41
68
41
69
21
88
38
72
37
79
40
65
34
70
40
90
25
89
55
74
28
89
17
72
62
68
30
90
41
70
25
67
52
91
17
94
25
79
28
69
42
74
13
68
32
88
56
78
58
77

46
92
41
93
50
86
30
87
48
92
21
93
17
95
23
75
24
97
38
86
39
81
15
66
41
87
62
76
41
77
44
96
38
90
42
65
38
81
35
85
14
70
23
99
23
65
57
82
54
100
57
83
53
70
19
99
16
88
38
83

44
69
22
69
13
71
42
75
32
99
52
93
13
86
14
91
26
73
50
93
63
72
48
86
26
66
64
93
51
88
51
95
60
85
19
65
54
83
39
96
27
68
47
92
48
75
49
83
59
78
42
91
26
84
49
95
26
91
41
73

In [14]:
```
model.fit(
    x=scaled_train_samples,
    y=train_labels,
    batch_size=10,
    epochs=20,
    shuffle=True,
    verbose=2
)
```

```
Epoch 1/20
210/210 - 1s - loss: 0.6937 - accuracy: 0.4124 - 797ms/epoch - 4ms/step
Epoch 2/20
210/210 - 0s - loss: 0.6729 - accuracy: 0.6138 - 215ms/epoch - 1ms/step
Epoch 3/20
210/210 - 0s - loss: 0.6436 - accuracy: 0.7462 - 199ms/epoch - 948us/step
Epoch 4/20
210/210 - 0s - loss: 0.6164 - accuracy: 0.7695 - 183ms/epoch - 870us/step
Epoch 5/20
210/210 - 0s - loss: 0.5893 - accuracy: 0.8000 - 219ms/epoch - 1ms/step
Epoch 6/20
210/210 - 0s - loss: 0.5606 - accuracy: 0.8138 - 189ms/epoch - 900us/step
Epoch 7/20
210/210 - 0s - loss: 0.5309 - accuracy: 0.8386 - 185ms/epoch - 883us/step
Epoch 8/20
210/210 - 0s - loss: 0.5007 - accuracy: 0.8452 - 188ms/epoch - 897us/step
Epoch 9/20
210/210 - 0s - loss: 0.4712 - accuracy: 0.8705 - 179ms/epoch - 854us/step
Epoch 10/20
210/210 - 0s - loss: 0.4432 - accuracy: 0.8714 - 201ms/epoch - 956us/step
Epoch 11/20
210/210 - 0s - loss: 0.4170 - accuracy: 0.8857 - 234ms/epoch - 1ms/step
Epoch 12/20
210/210 - 0s - loss: 0.3929 - accuracy: 0.9043 - 245ms/epoch - 1ms/step
Epoch 13/20
210/210 - 0s - loss: 0.3717 - accuracy: 0.9076 - 268ms/epoch - 1ms/step
Epoch 14/20
210/210 - 0s - loss: 0.3537 - accuracy: 0.9090 - 198ms/epoch - 941us/step
Epoch 15/20
210/210 - 0s - loss: 0.3386 - accuracy: 0.9148 - 197ms/epoch - 937us/step
Epoch 16/20
210/210 - 0s - loss: 0.3259 - accuracy: 0.9190 - 185ms/epoch - 881us/step
Epoch 17/20
210/210 - 0s - loss: 0.3154 - accuracy: 0.9214 - 196ms/epoch - 931us/step
Epoch 18/20
210/210 - 0s - loss: 0.3067 - accuracy: 0.9190 - 201ms/epoch - 958us/step
Epoch 19/20
210/210 - 0s - loss: 0.2994 - accuracy: 0.9281 - 231ms/epoch - 1ms/step
Epoch 20/20
210/210 - 0s - loss: 0.2934 - accuracy: 0.9238 - 234ms/epoch - 1ms/step
```

Out[14]:
```
<keras.callbacks.History at 0x1d0f9136cc8>
```

In [ ]: