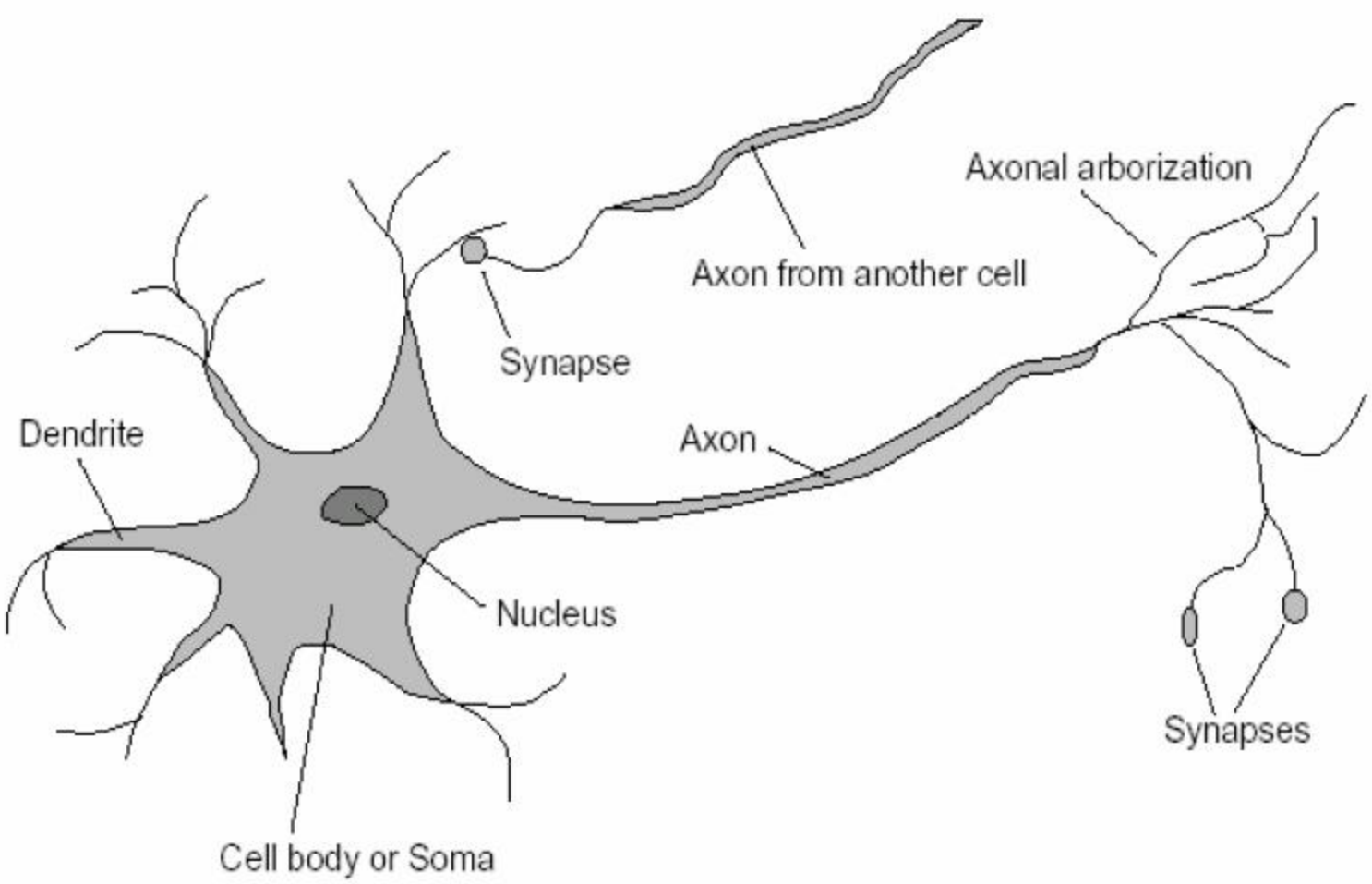
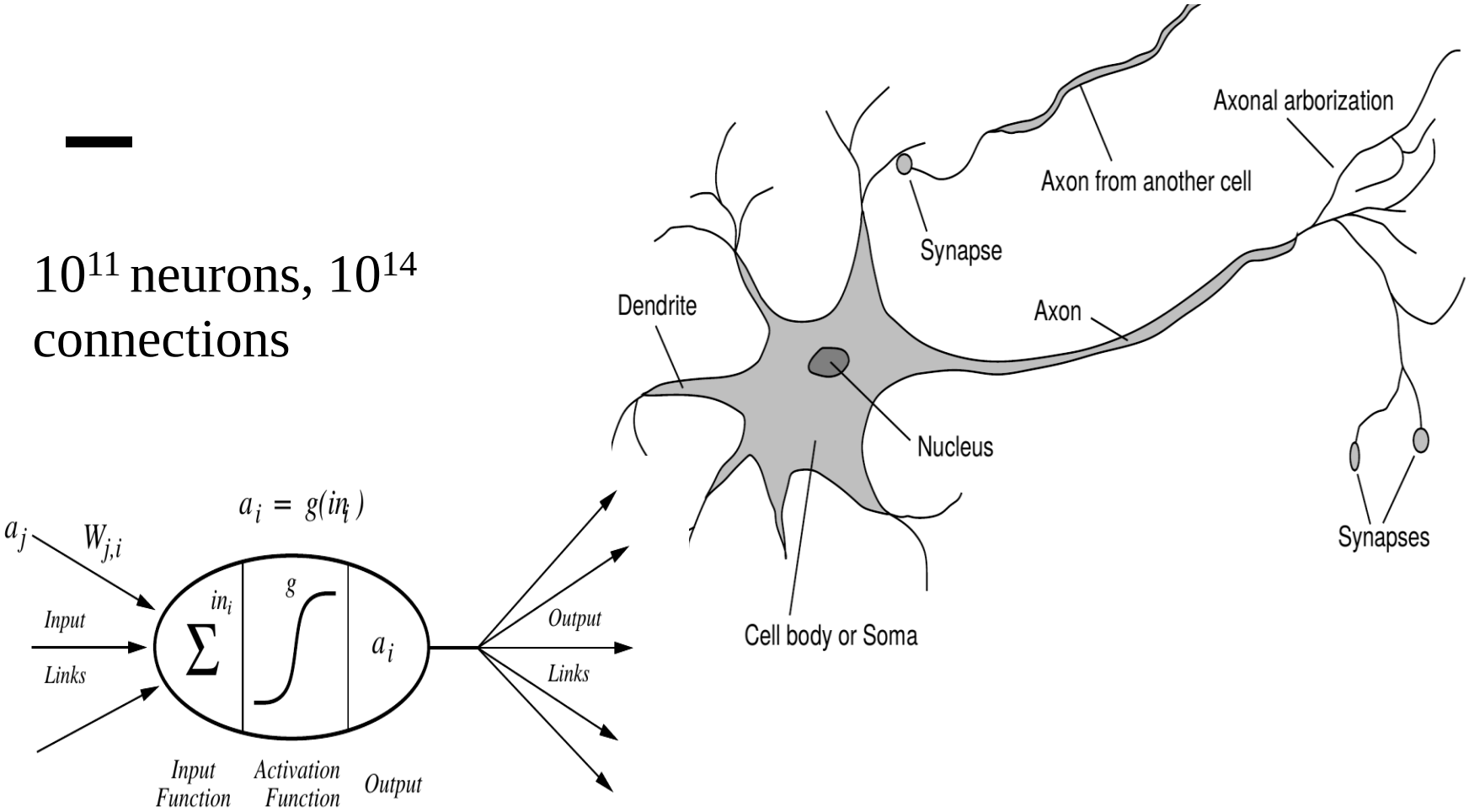
Artificial Neural Network

Neurons in the Brain



—

10^{11} neurons, 10^{14} connections



$$a_i = g(\sum_j W_{j,i} a_j)$$

Neural Networks

- Artificial neural network (ANN) is a machine learning approach that models human brain and consists of a number of artificial neurons.

- Each neuron in ANN receives a number of inputs.

Neural Networks

- An Artificial Neural Network is specified by:
  - **neuron model**: the information processing unit of the NN
  - **an architecture**: a set of neurons and links connecting neurons. Each link has a weight
  - **a learning algorithm**: used for training the NN by modifying the weights in order to model a particular learning task correctly on the training examples.

Perceptron

Introduced in the late 50s – Minsky and Papert.

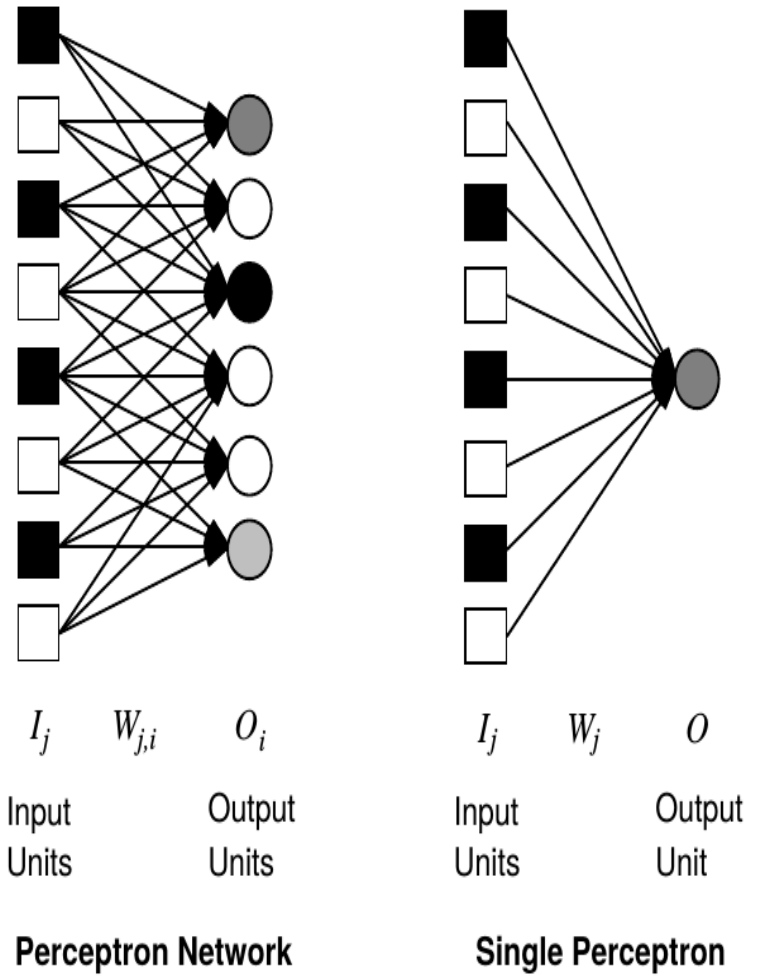
Classifies a linearly separable set of inputs.

Multi-layer perceptrons – found as a "solution" to represent nonlinearly separable functions – 1950s.

Many local minima – Perceptron convergence theorem does not apply.

1950s - Intuitive Conjecture was: There is no learning algorithm for multi-layer perceptrons.

Perceptron convergence theorem Rosenblatt 1962: Perceptron will learn to classify any linearly separable set of inputs.



Neuron

- The neuron is the basic information processing unit of a NN. It consists of:

1 A set of links, describing the neuron inputs, with weights W_1, W_2, …, W_m

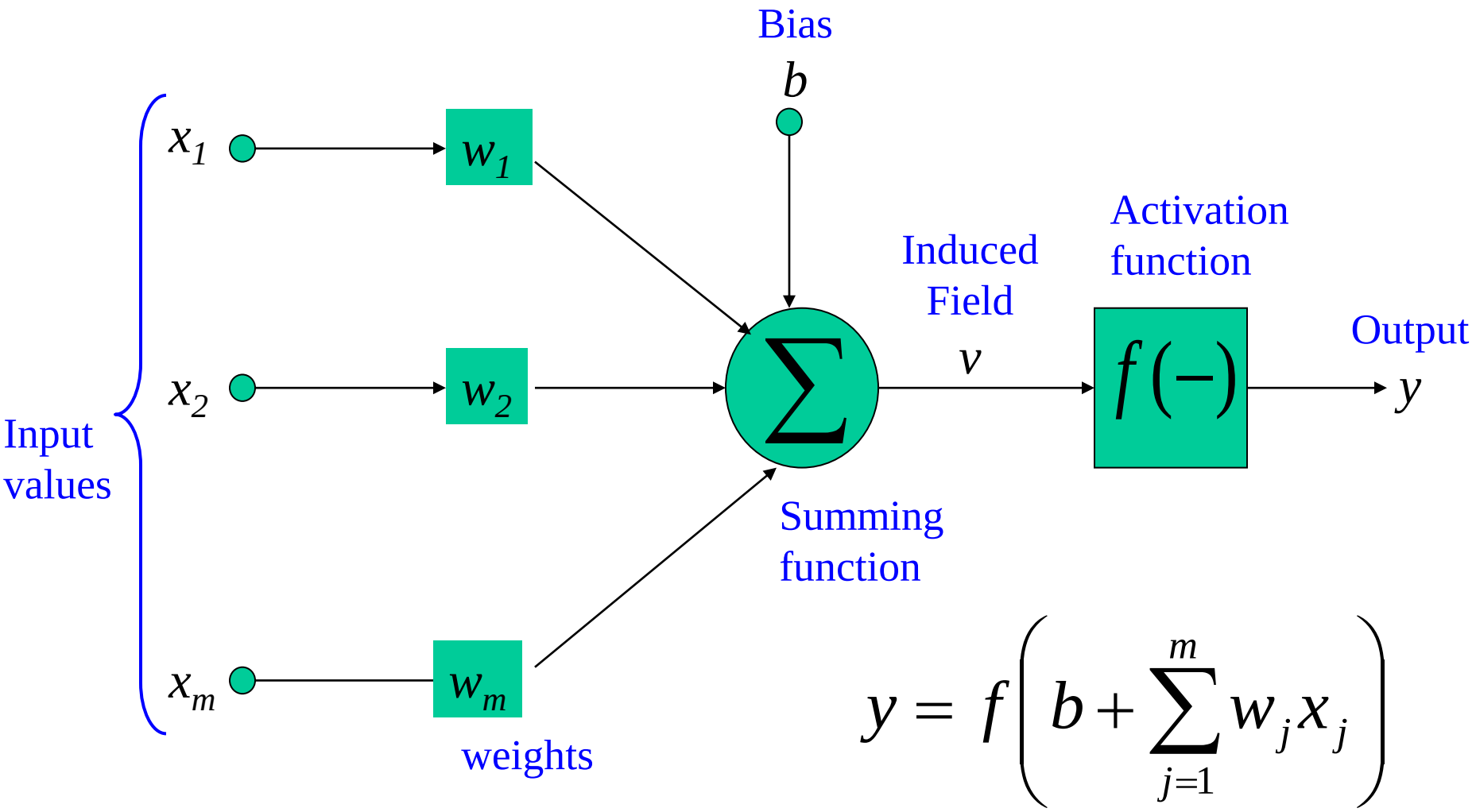
2 An adder function (linear combiner) for computing the weighted sum of the inputs:
(real numbers)

$$u = \sum_{j=1}^{m} w_j x_j$$

3 Activation function f for limiting the amplitude of the neuron output. Here 'b' denotes bias.

$$y = f(u + b)$$

The Neuron Diagram



$$y = f\left(b + \sum_{j=1}^{m} w_j x_j \right)$$

One Neuron as a Network

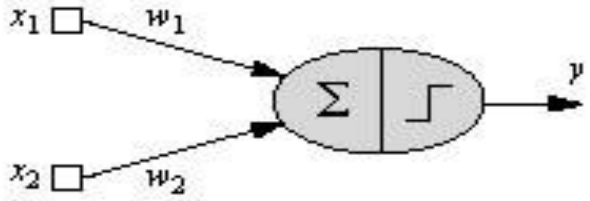


Fig1: an artificial neuron

- **x1 and x2 are normalized attribute value of data.**
- **y is the output of the neuron , i.e the class label.**
- **x1 and x2 values multiplied by weight values w1 and w2 are input to the neuron**
- **Given that**
  - **w1 = 0.5 and w2 = 0.5**
  - **Say value of x1 is 0.3 and value of x2 is 0.8,**
  - **So, weighted sum is :**
  - **sum= w1 * x1 + w2 * x2 = 0.5 x 0.3 + 0.5 x 0.8 = 0.55**

One Neuron as a Network

- **The neuron receives the weighted sum as input and calculates the output as a function of input as follows :**

- **y = f(x) , where f(x) is defined as**

- **f(x) = 0 { when x< 0.5 }**
- **f(x) = 1 { when x >= 0.5 }**

- **For our example, weighted sum is 0.55, so y = 1 ,**

- That means corresponding input attribute values are classified in class 1.

- **If for another input values , x = 0.45 , then f(x) = 0,**
- **so we could conclude that input values are classified to class 0.**

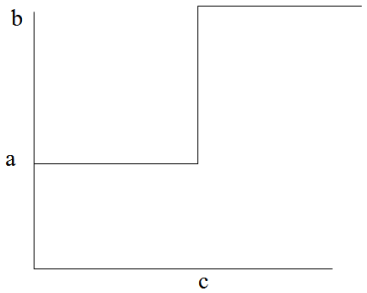
Neuron Models

- The choice of activation function φ determines the neuron model.

Examples:

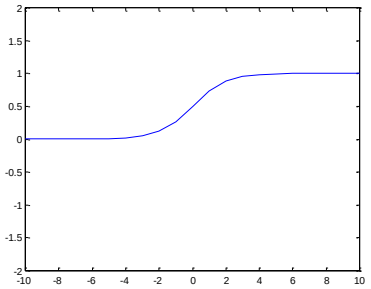
- step function:

$$f(v) = \begin{cases} a & \text{if } v < c \\ b & \text{if } v > c \end{cases}$$



- Logistic (Sigmoid function)

$$f(v) = \frac{1}{1 + \exp(-v)}$$



Bias of a Neuron

- The bias ***b*** has the effect of applying a transformation to the weighted sum ***u***

  $$v = u + b$$

- The bias is an external parameter of the neuron. It can be modeled by adding an extra input.
- ***v*** is called **induced field** of the neuron

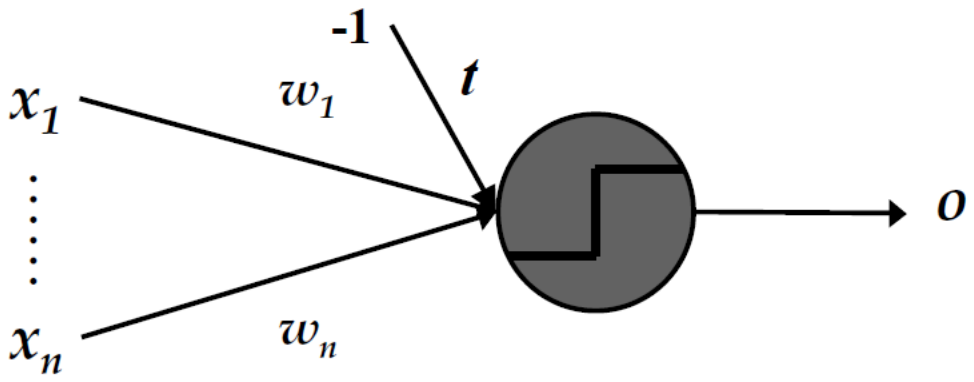
$$v = \sum_{j=0}^{m} w_j x_j$$

$$w_0 = b$$

Bias of a Neuron

- Really, the threshold t is just another weight (called the bias):

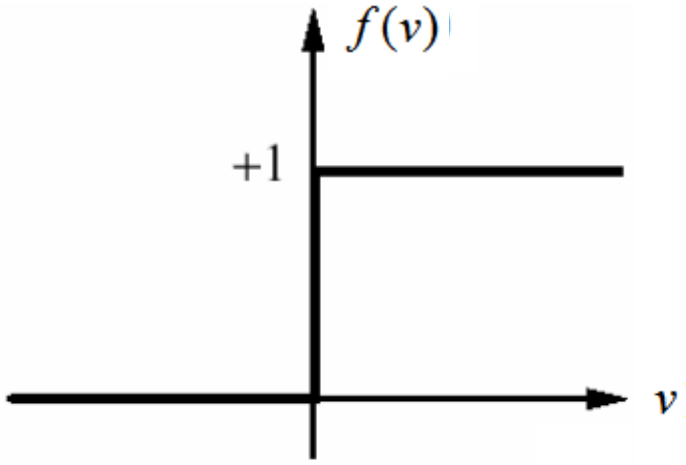
$$(w_1 \times x_1) + (w_2 \times x_2) + \ldots + (w_n \times x_n) \geq t$$
$$= (w_1 \times x_1) + (w_2 \times x_2) + \ldots + (w_n \times x_n) - t \geq 0$$
$$= (w_1 \times x_1) + (w_2 \times x_2) + \ldots + (w_n \times x_n) + (t \times -1) \geq 0$$



Linear Threshold Unit
Simple Perceptron Unit
Threshold Logic Unit

Use "hard-limiting" squashing function

$$f(v)=\begin{cases}1 & \text{if } v>0\\ 0 & \text{if } v\le 0\end{cases}$$



Boolean interpretation: 0 ⇔ false, 1 ⇔ true

Perceptron for Classification

- The perceptron is used for binary classification.
- First train a perceptron for a classification task.
  - Find suitable weights in such a way that the training examples are correctly classified.

- The perceptron can only model linearly separable classes.
- Given training examples of classes C_1, C_2 train the perceptron in such a way that :
  - *If the output of the perceptron is +1 then the input is assigned to class* C_1
  - *If the output is -1 then the input is assigned to* C_2

Perceptron Training

Learning Procedure:

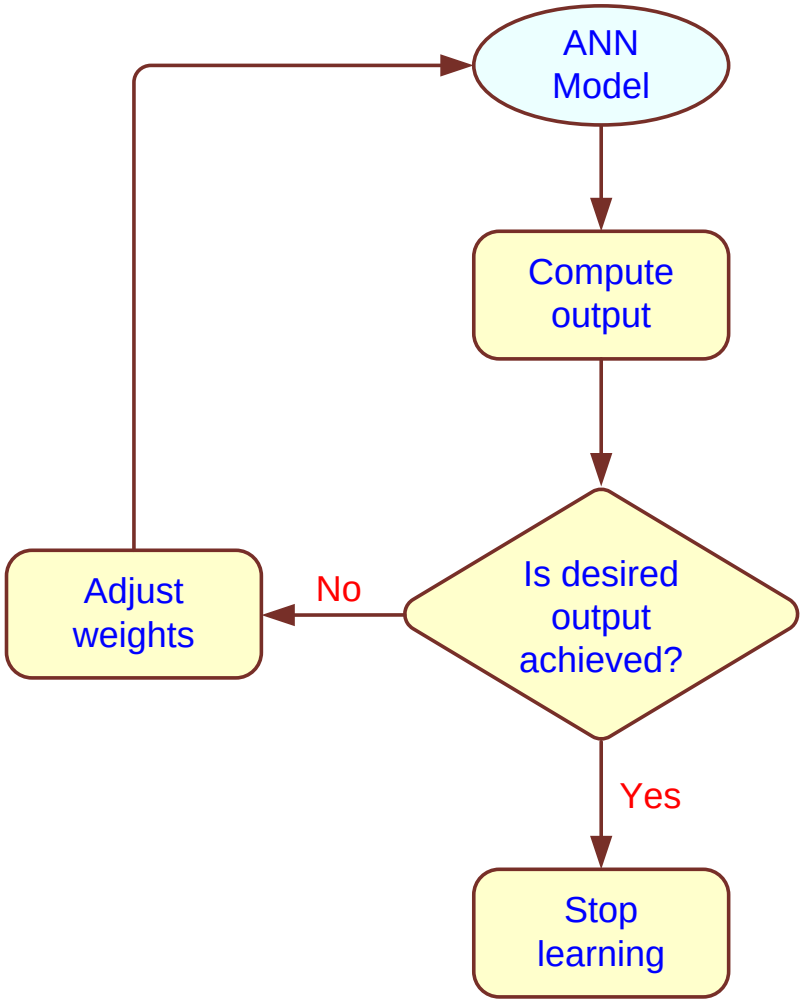
Randomly assign weights (between 0-1)

Present inputs from training data

Get output O, modify weights to gives results toward our desired output T

Repeat; stop when no errors, or enough epochs completed

A Supervised Learning Process



Three-step process:

1. Compute temporary outputs
2. Compare outputs with desired targets
3. Adjust the weights and repeat the process

Perceptron algorithm

$\mathbf{w} \leftarrow \mathbf{0}$ (any initial values ok)

repeat

  for r=1 to R

    $\mathbf{w} \leftarrow \mathbf{w} + \eta(d^r - y^r)\mathbf{x}^r$

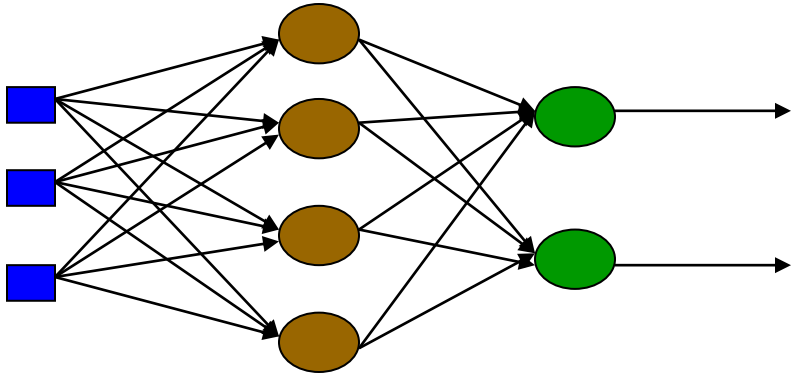
until no errors

$\eta > 0$ is the learning rate

It can be taken to be 1 when inputs are 0 and 1

Multi layer feed-forward NN (FFNN)

- FFNN is a more general network architecture, where there are hidden layers between input and output layers.
- Hidden nodes do not directly receive inputs nor send outputs to the external environment.
- FFNNs overcome the limitation of single-layer NN.
- They can handle non-linearly separable learning tasks.



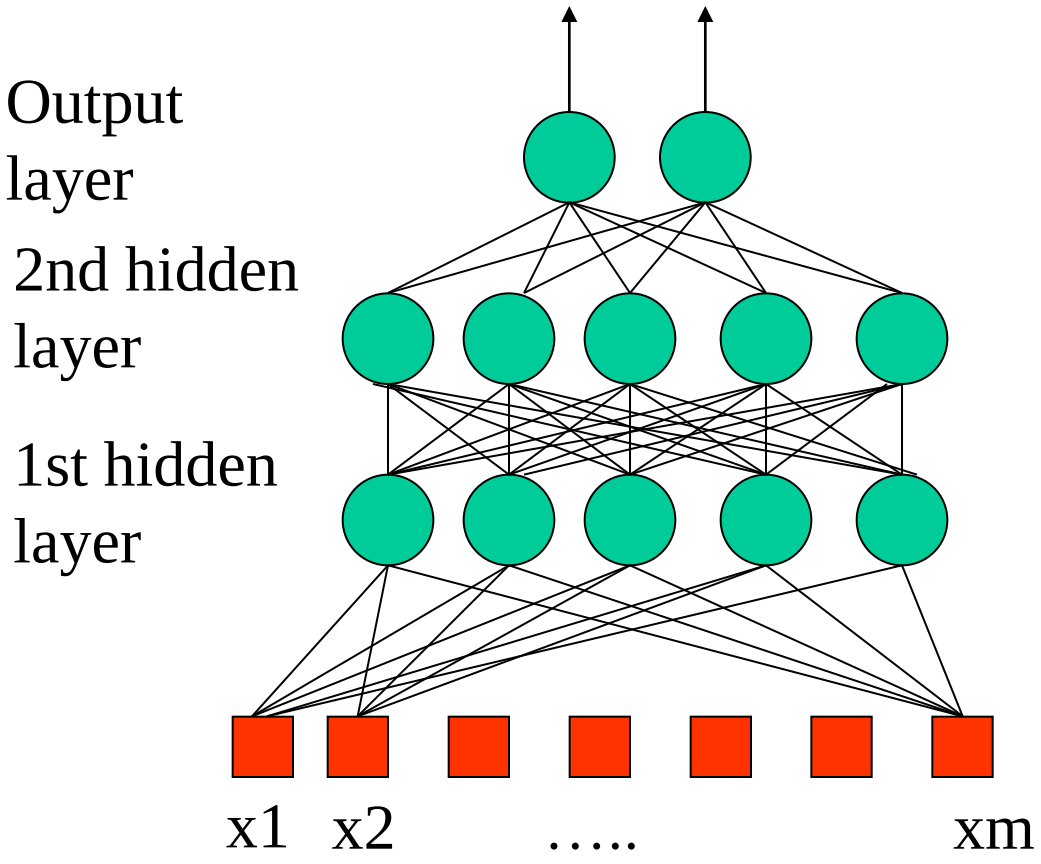
Input layer

Output layer

Hidden Layer

3-4-2 Network

Feed Forward Neural Networks



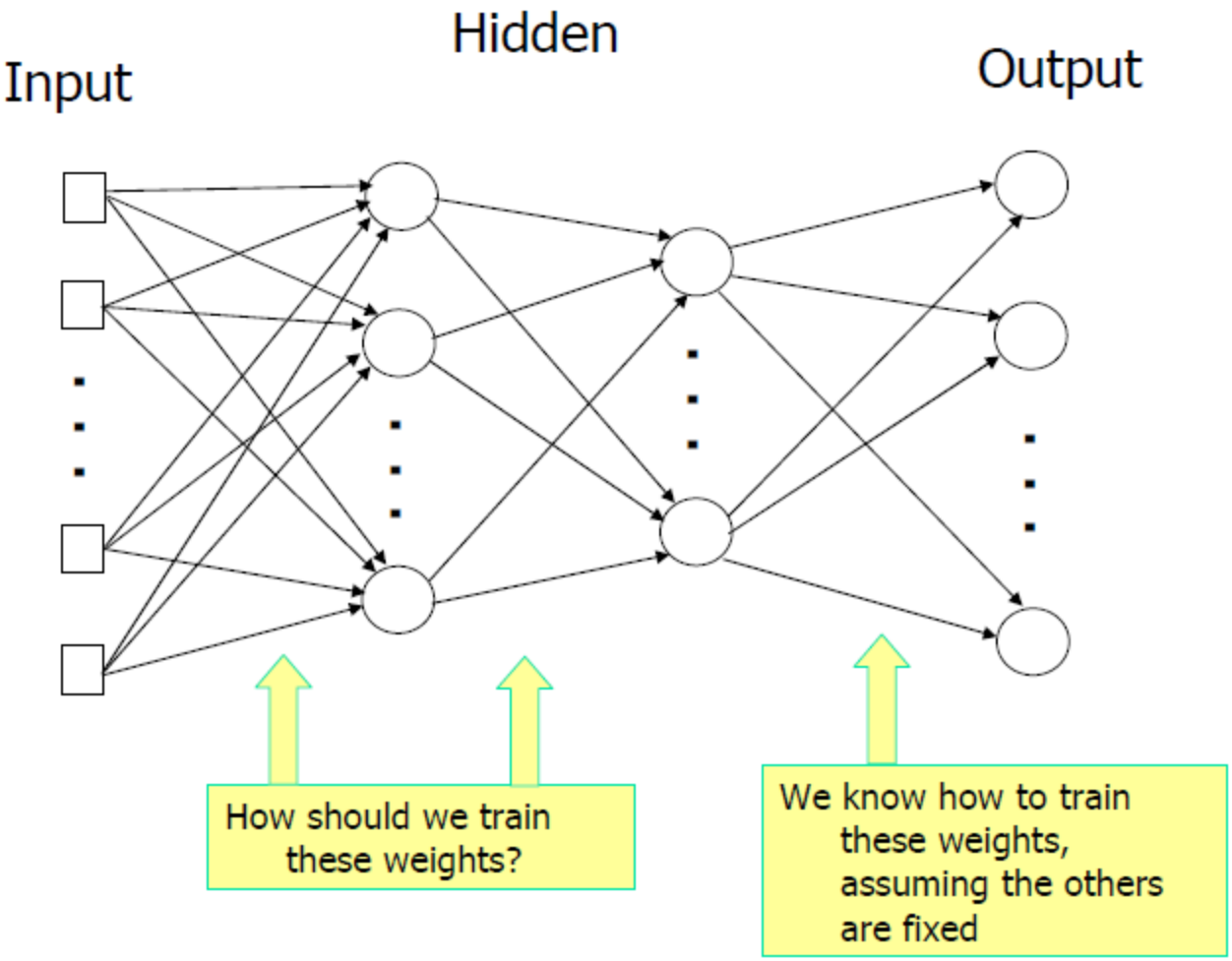
- The information is propagated from the inputs to the outputs
- Time has no role (NO cycle between outputs and inputs)

Hidden Layers

- In some cases, there may be many independencies among the input variables and adding an extra hidden layer can be helpful

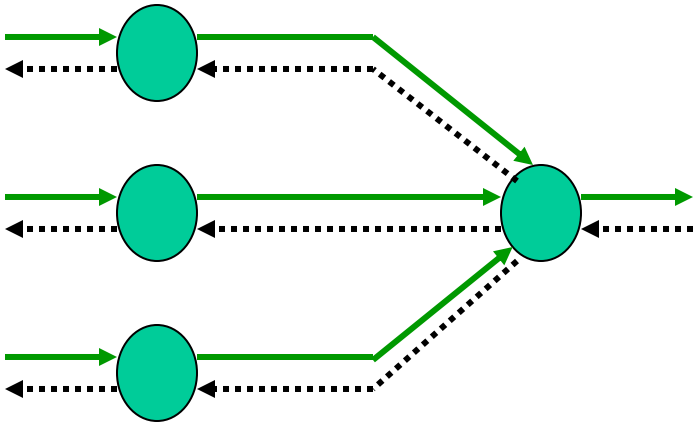
- MLP with two hidden layers can approximate any non-continuous functions

Multilayer Networks



Backpropagation

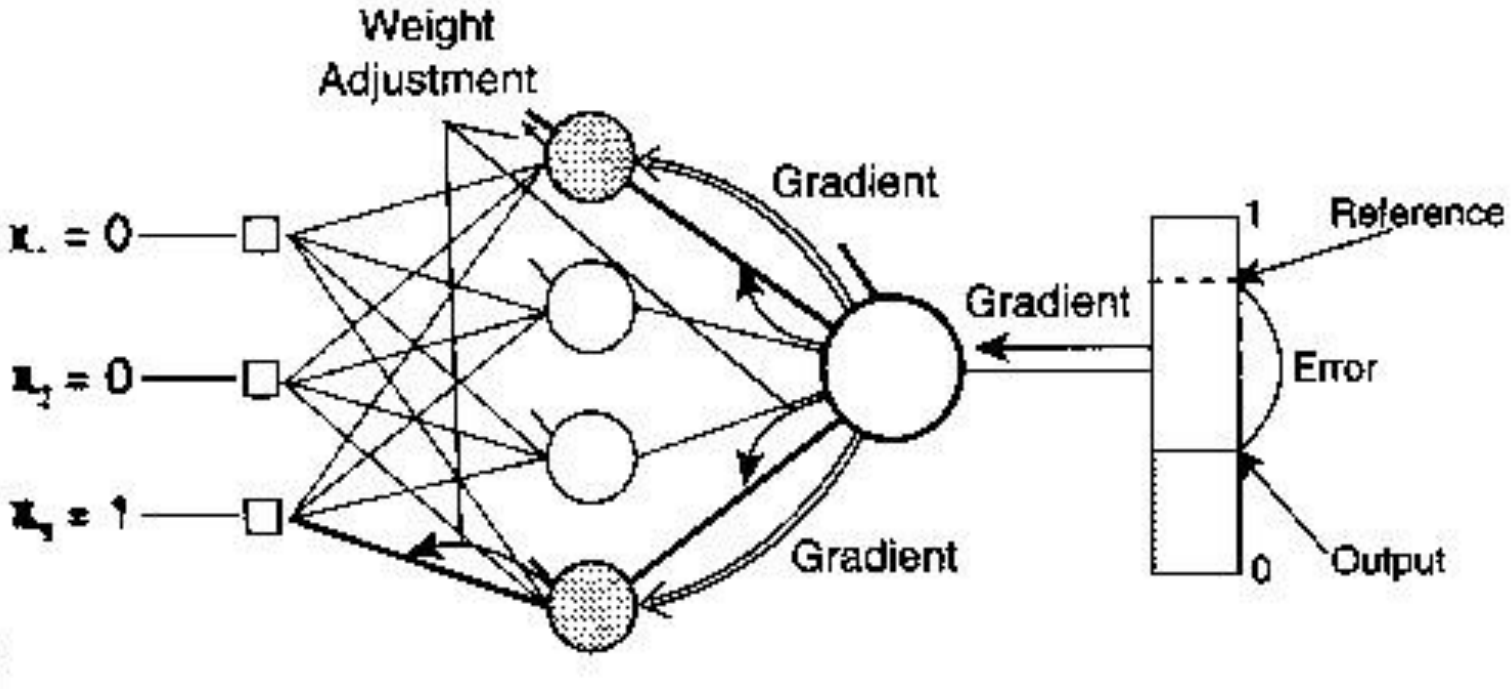
- Back-propagation training algorithm



Network activation
Forward Step

Error propagation
Backward Step

- Backpropagation adjusts the weights of the NN in order to minimize the network total mean squared error.


Weight
Adjustment
Gradient
Gradient
Gradient
1
Reference
Error
0
Output

Backpropagation Algorithm

- The Backpropagation algorithm learns in the same way as single perceptron.
- It searches for weight values that minimize the total error of the network over the set of training examples (training set).

Given: set of input-output pairs

Task: compute weights for n-layer network to minimize the total error of the network

Backpropagation Algorithm

1.Determine the number of neurons required

2.Initialize weights to random values

3.Set activation values for threshold units

Backpropagation Algorithm

4. Choose an input-output pair and assign activation levels to input neurons
5. Propagate activations from input neurons to hidden layer neurons for each neuron

$$h_j = 1/(1 + e^{-\Sigma\, w1_{ij} x_i})$$

6. Propagate activations from hidden layer neurons to output neurons for each neuron

$$o_j = 1/(1 + e^{-\Sigma\, w2_{ij} h_i})$$

Backpropagation Algorithm

7. Compute error for output neurons by comparing pattern to actual
8. Compute error for neurons in hidden layer
9. Adjust weights in between hidden layer and output layer
10. Adjust weights between input layer and hidden layer
11. Go to step 4

Backprop learning algorithm (incremental-mode)

n=1;

initialize **weights** randomly;

while (stopping criterion not satisfied or n <max_iterations)

for each example (**x**,*d*)

- run the network with input x and compute the output y

- update the weights in backward order starting from those of the output layer:

$$w_{ji} = w_{ji} + \Delta w_{ji}$$

with Δw_{ji} computed using the (generalized) Delta rule

end-for

n = n+1;

end-while;

Total Mean Squared Error

- $E[w] \equiv \frac{1}{2} \sum_d (true_d - o_d)^2$
- Where $E[w]$ is the sum of squared errors for the weight vector w, and d ranges over examples in the training set

Derivation of Back-propagation

Backpropagation

It work for Multi-layer perceptron.

It changes Weights proportional to output errors.

It is used in Gradient descent and chaining.

After some training no more improvement in model.

When to stop training?

Advantages

- Good mathematical foundation
- If solution exists it can be found
- Deals well with noise

Hidden Neurons

- Choosing the number of neurons in the hidden layer often depends on the AI designer's intuition and experience
- As the number of dimensions grows the complexity of the decision surface (path through hidden layer) increases

1 Fully Connected Networks

Keras is a powerful and easy-to-use Python library for developing and evaluating deep learning models. It wraps the efficient numerical computation libraries Theano and TensorFlow and allows you to define and train neural network models in a few short lines of code. In this tutorial you will discover how to create your first neural network model in Python using Keras. After completing this lesson you will know:

- How to load a dataset for use with Keras.
- How to define and compile a Multilayer Perceptron model in Keras.
- How to evaluate a Keras model on a validation dataset.

1.1 Load Libraries

First step is to load the required libraries.

```
import numpy

import matplotlib.pyplot as plt

from keras.models import Sequential

from keras.layers import Dense

from keras.utils import np_utils

#
```

1.2 Load Dataset in Numpy Format

MNIST is a simple computer vision dataset. It consists of images of handwritten digits as shown in figure 1. It also includes labels for each image, telling us which digit it is. For example, the labels for the above images

Figure 1: Sample images of MNIST dataset

are 5, 0, 4, and 1.

The MNIST data is split into three parts: 55,000 data points of training data (mnist.train), 10,000 points of test data (mnist.test), and 5,000 points of validation data (mnist.validation). Each image is 28 pixels by 28 pixels.

```
X_train = numpy.load('./Data/mnist/x_train.npy')

X_test  =  numpy.load('./Data/mnist/x_test.npy')

y_train = numpy.load('./Data/mnist/y_train.npy')

y_test  = numpy.load('./Data/mnist/y_test.npy')

#
```

1.3 Formatting Data and Labels for Keras

We can flatten this array into a vector of $28 \times 28 = 784$ numbers. It doesn't matter how we flatten the array, as long as we're consistent between images. From this perspective, the MNIST images are just a bunch of points in a 784-dimensional vector space. The data should always be of the format (Number of data points, data point dimension). In this case the training data will be of format $55,000 \times 784$.

```
num_pixels = X_train.shape[1] * X_train.shape[2]

X_train = X_train.reshape(X_train.shape[0], num_pixels).astype('float32')

X_test = X_test.reshape(X_test.shape[0], num_pixels).astype('float32')

X_train = X_train / 255

X_test = X_test / 255

y_train = np_utils.to_categorical(y_train)

y_test = np_utils.to_categorical(y_test)

num_classes = y_test.shape[1]

#
```

1.4 Defining a single layer neural network model

Here we will define a single layer neural network. It will have a input layer of 784 neurons, i.e. the input dimension and output layer of 10 neurons, i.e. number of classes. The activation function used will be softmax activation.

```
# create model

model = Sequential()
```

```
model.add(Dense(num_classes, input_dim=num_pixels, activation='softmax'))

#
```

1.5 Compiling the model

Once the model is defined, we have to compile it. While compiling we provide the loss function to be used, the optimizer and any metric. Here we will use crossentropy loss with Adam optimizer and accuracy as a metric.

```
# Compile model

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

#
```

1.6 Training/Fitting the model

Now the model is ready to be trained. We will provide training data to the network. Also we will specify the validation data, over which the model will only be validated.

```
# Training model

model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10, batch_size=200,
↪  verbose=2)

#
```

1.7 Evaluating the model

Finally we will evaluate the model on the testing dataset.

```
# Final evaluation of the model

scores = model.evaluate(X_test, y_test, verbose=0)

print("Baseline Error: %.2f%%" % (100-scores[1]*100))

#
```

1.8 Defining a multi-layer model

Now we will define a multi layer neural network in which we will add 2 hidden layers having 500 and 100 neurons.

```
model = Sequential()

model.add(Dense(500, input_dim=num_pixels, activation='relu'))

model.add(Dense(100, activation='relu'))

model.add(Dense(num_classes, activation='softmax'))

#
```

1.9 Programming task

Now to put it all together to form a 3 layer neural network for 10 class digit classification.

```
import numpy
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense
from keras.utils import np_utils

# load data from the path specified by the user
X_train = numpy.load('./Data/mnist/x_train.npy')
X_test  =  numpy.load('./Data/mnist/x_test.npy')
y_train = numpy.load('./Data/mnist/y_train.npy')
y_test  = numpy.load('./Data/mnist/y_test.npy')

# flatten 28*28 images to a 784 vector for each image
num_pixels = X_train.shape[1] * X_train.shape[2]
# forcing the precision of the pixel values to be 32 bit
X_train = X_train.reshape(X_train.shape[0], num_pixels).astype('float32')
X_test = X_test.reshape(X_test.shape[0], num_pixels).astype('float32')
# normalize inputs from 0-255 to 0-1
X_train = X_train / 255
X_test = X_test / 255
# one hot encode outputs using np_utils.to_categorical inbuilt function
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
# define baseline model
```

```
#The model is a simple neural network with one hidden layer with the same number of
↪  neurons as there are inputs (784)
def baseline_model():
        # create model
        model = Sequential()
        model.add(Dense(500,kernel_initializer='normal', input_dim=num_pixels,
        ↪  activation='relu'))
        model.add(Dense(100,kernel_initializer='normal', activation='relu'))
        #A softmax activation function is used on the output
        #to turn the outputs into probability-like values and
        #allow one class of the 10 to be selected as the model's output #prediction.
        model.add(Dense(num_classes, kernel_initializer='normal', activation='softmax'))
        # Compile model
        model.compile(loss='categorical_crossentropy', optimizer='adam',
        ↪  metrics=['accuracy'])
        return model
# build the model
model = baseline_model()
# Fit the model
#The model is fit over 10 epochs with updates every 200 images. The test data is used as
↪  the validation dataset
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10, batch_size=200,
↪  verbose=2)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Baseline Error: %.2f%%" % (100-scores[1]*100))
```