Member-only story

A Visual Understanding of Neural Networks

The math behind neural networks visually explained

Reza Bagheri · Follow
Published in Towards Data Science · 27 min read · 5 days ago

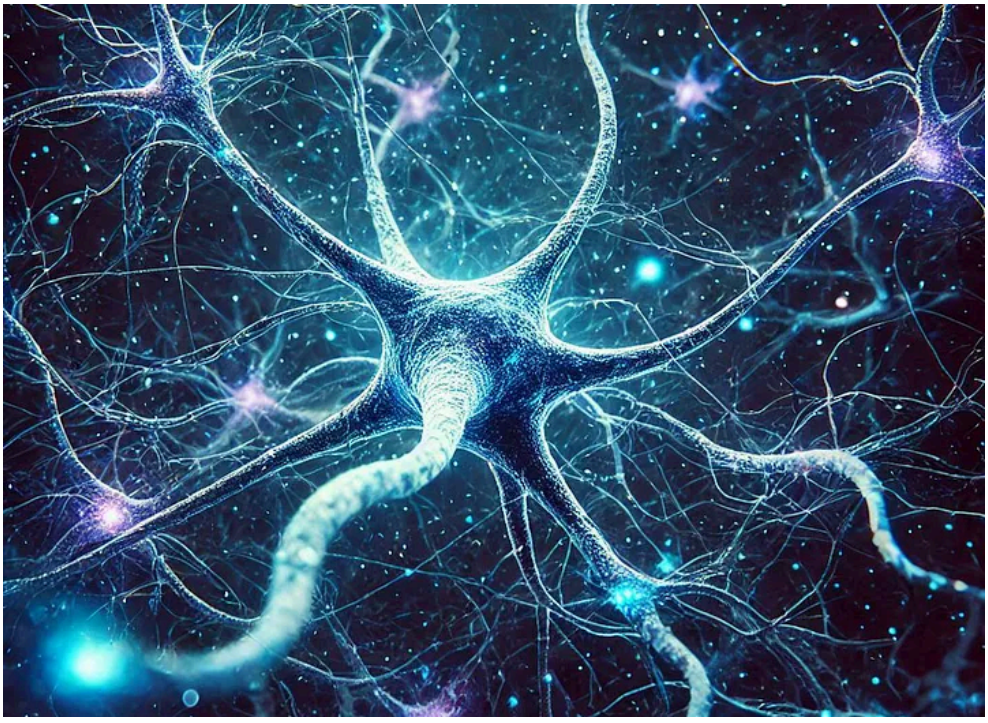

Image generated using DALL-E

Artificial neural networks are the most powerful and at the same time the most complicated machine learning models. They are particularly useful for complex tasks where traditional machine learning algorithms fail. The main advantage of neural networks is their ability to learn intricate patterns and

relationships in data, even when the data is highly dimensional or unstructured.

Many articles discuss the math behind neural networks. Topics like different activation functions, forward and backpropagation algorithms, gradient descent, and optimization methods are discussed in detail. In this article, we take a different approach and present a visual understanding of a neural network layer by layer. We will first focus on the visual explanation of single-layer neural networks in both classification and regression problems and their similarities to other machine learning models. Then we will discuss the importance of hidden layers and non-linear activation functions. All the visualizations are created using Python.

All the images in this article were created by the author.

Neural networks for classification

We start with classification problems. The simplest type of classification problem is a *binary classification* in which the target has only two categories or labels. If the target has more than two labels, then we have a *multi-class classification* problem.

Single-layer networks: perceptron

A single-layer neural network is the simplest form of an artificial neural network. Here we only have an input layer which receives the input data and an output layer that produces the output of the network. The input layer isn't considered a true layer in this network since it merely passes the input data. That's why this architecture is called a single-layer network. Perceptron, the first neural network ever created, is the simplest example of a single-layer neural network.

The perceptron was created in 1957 by Frank Rosenblatt. He believed that perceptron can simulate brain principles, with the ability to learn and make decisions. The original perceptron was designed to solve a binary classification problem.

Figure 1 shows the architecture of a perceptron. The input data has n features denoted by x_1 to x_n. The target y has only two labels (y=0 and y=1).

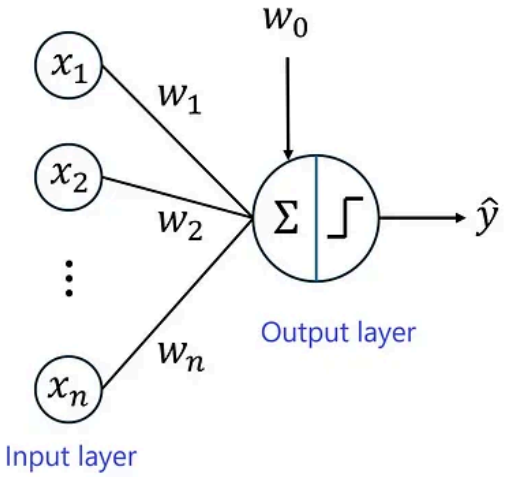

Figure 1

The input layer receives the features and passes them to the output layer. The neuron in the output layer calculates the weighted sum of the input features. Each input feature, x_i, is associated with the weight w_i. The neuron multiplies each input by its corresponding weight and sums up the results. A bias term, w_0, is also added to this sum. If we denote the sum by z, we have:

$$z = w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n$$

The activation function is a step function defined as:

$$g(z) = \begin{cases} 1 & if\ z \geq 0 \\ 0 & Otherwise \end{cases} \qquad (1)$$

This activation function is plotted in Figure 2.

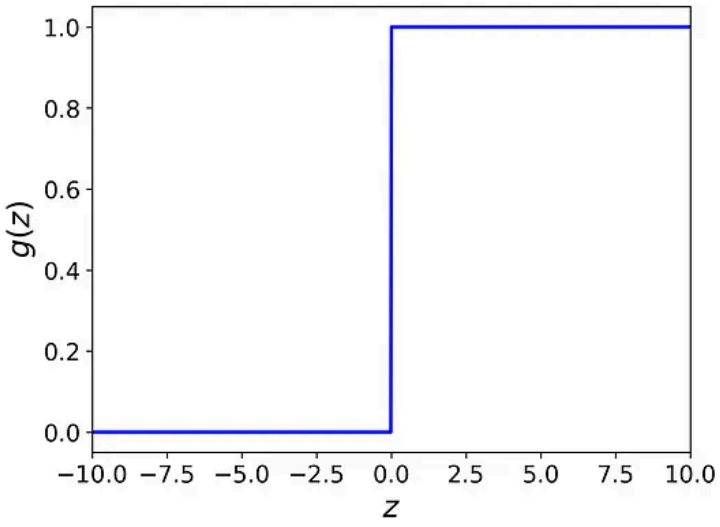

Figure 2

The output of the perceptron denoted by $y^$ is calculated as follows:

$$\hat{y} = g(w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n)$$

To visualize how a perceptron works, we use a simple training dataset with only two features x_1 and x_2. This dataset is created in Listing 1. It is defined randomly, and the target y has only two labels (y=0 and y=1). We also import all the Python libraries needed in this article at the beginning of this listing. The dataset is plotted in Figure 3.

```
# Listing 1

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
import random
import tensorflow as tf
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import Dense, Input
from tensorflow.keras.utils import to_categorical
from tensorflow.keras import backend

np.random.seed(3)
n = 30
X1 = np.random.randn(n,2)

y1 = np.random.choice((0, 1),size=n)
X1[y1>0,0] -= 4
X1[y1>0,1] += 4
scaler = StandardScaler()
X1 = scaler.fit_transform(X1)

plt.figure(figsize=(5, 5))
marker_colors = ['red', 'blue']
target_labels = np.unique(y1)
n = len(target_labels)
for i, label in enumerate(target_labels):
    plt.scatter(X1[y1==label, 0], X1[y1==label,1], label="y="+str(label),
                edgecolor="white", color=marker_colors[i])
plt.xlabel('$x_1$', fontsize=16)
plt.ylabel('$x_2$', fontsize=16)
plt.legend(loc='best', fontsize=11)
ax = plt.gca()
ax.set_aspect('equal')
plt.xlim([-2.3, 1.8])
plt.ylim([-1.9, 2.2])
plt.show()
```

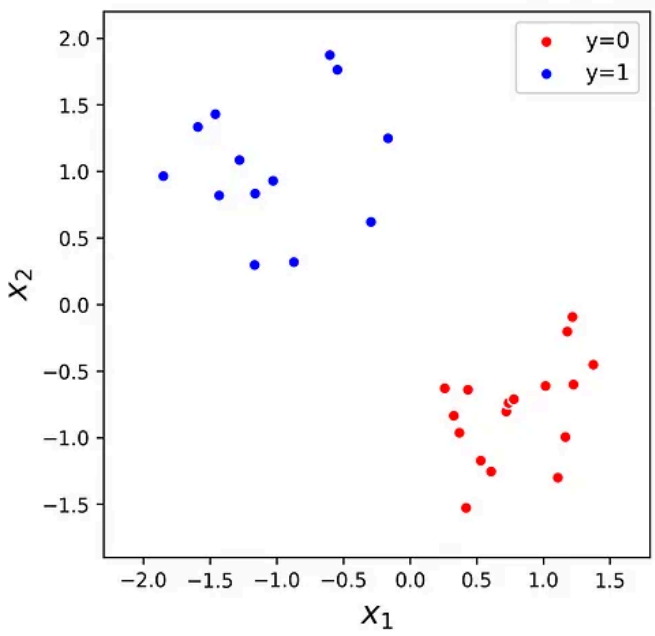

Figure 3

This article does not go into detail about the neural network training process. Instead, we focus on the behaviour of an already trained neural network. In Listing 2, we define and train a perceptron using the previous dataset.

```
# Listing 2

class Perceptron(object):
    def __init__(self, eta=0.01, epochs=50):
        self.eta = eta
        self.epochs = epochs

    def fit(self, X, y):

        self.w = np.zeros(1 + X.shape[1])

        for epoch in range(self.epochs):
            for xi, target in zip(X, y):
                error = target - self.predict(xi)
                self.w[1:] +=  self.eta * error * xi
                self.w[0] +=  self.eta * error
        return self

    def net_input(self, X):
        return np.dot(X, self.w[1:]) + self.w[0]

    def predict(self, X):
        return np.where(self.net_input(X) >= 0.0, 1, 0)

perc = Perceptron(epochs=150, eta=0.05)
perc.fit(X1, y1)
```

Now we want to see how this model classifies our training dataset. Hence, we define a function that plots the decision boundary of the trained neural network. This function defined in Listing 3, creates a mesh grid on the 2D space and then uses a trained model to predict the target of all the points on that grid. The points with different labels are colored differently. Therefore, the decision boundary of the model can be visualized using this function.

```
# Listing 3

def plot_boundary(X, y, clf, lims, alpha=1):
    gx1, gx2 = np.meshgrid(np.arange(lims[0], lims[1],
                                     (lims[1]-lims[0])/500.0),
                           np.arange(lims[2], lims[3],
                                     (lims[3]-lims[2])/500.0))
    backgd_colors = ['lightsalmon', 'aqua', 'lightgreen', 'yellow']
    marker_colors = ['red', 'blue', 'green', 'orange']
    gx1l = gx1.flatten()
    gx2l = gx2.flatten()
    gx = np.vstack((gx1l,gx2l)).T
    gyhat = clf.predict(gx)
    if len(gyhat.shape)==1:
        gyhat = gyhat.reshape(len(gyhat), 1)
    if gyhat.shape[1] > 1:
        gyhat = gyhat.argmax(axis=1)
    gyhat = gyhat.reshape(gx1.shape)
    target_labels = np.unique(y)
    n = len(target_labels)
    plt.pcolormesh(gx1, gx2, gyhat, cmap=ListedColormap(backgd_colors[:n]))
    for i, label in enumerate(target_labels):
        plt.scatter(X[y==label, 0], X[y==label,1],
                    label="y="+str(label),
                    alpha=alpha, edgecolor="white",
                    color=marker_colors[i])
```

Now, we use this function to plot the decision boundary of the perceptron for the training dataset. The result is shown in Figure 4.

```
# Listing 4

plt.figure(figsize=(5, 5))
# Plot the vector w
plt.quiver([0], [0], perc.w[1], perc.w[2], color=['black'],
           width=0.008, angles='xy', scale_units='xy',
           scale=0.4, zorder=5)
# Plot the boundary
plot_boundary(X1, y1, perc, lims=[-2.3, 1.8, -1.9, 2.2])
ax = plt.gca()
ax.set_aspect('equal')
plt.xlabel('$x_1$', fontsize=16)
plt.ylabel('$x_2$', fontsize=16)
plt.legend(loc='best', fontsize=11)
plt.xlim([-2.3, 1.8])
plt.ylim([-1.9, 2.2])
plt.show()
```

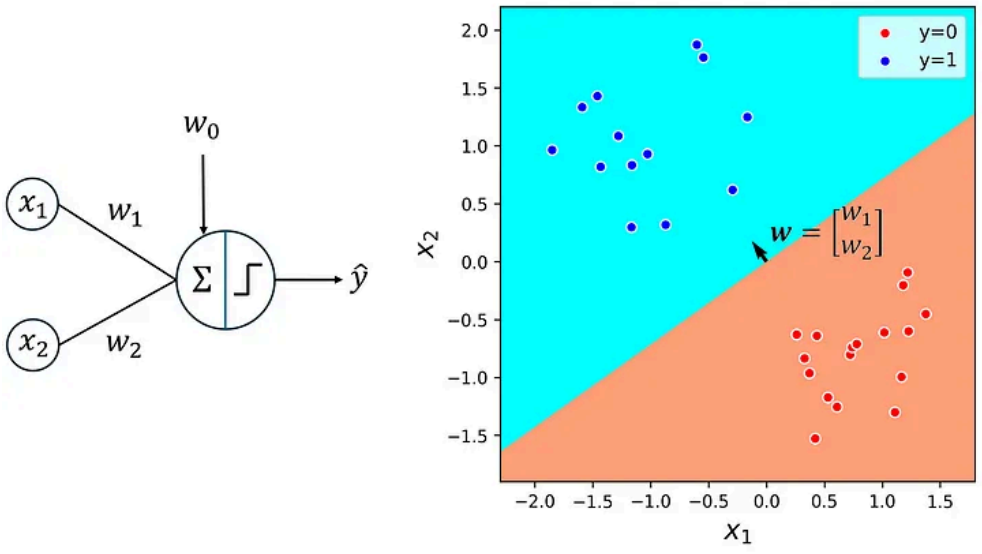

Figure 4

The figure clearly shows that the decision boundary is a straight line. We define the vector $\boldsymbol{w}$ using the weights of the perceptron:

$$\boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$$

This vector is also plotted in Figure 4, which shows that it is perpendicular to the decision boundary of the perceptron (the vector was small, so we scaled it in the plot). We can now explain the mathematical reasons behind these results.

For a dataset with two features, we have:

$$z = w_0 + w_1 x_1 + w_2 x_2$$

Based on Equation 1, we know that the predicted label of all the data points for which z=0 is 1. On the other hand, the predicted label for any data point with z<0 will be 0. Hence, the decision boundary is the location of the data points for which z=0, and it is defined by the following equation:

$$w_0 + w_1 x_1 + w_2 x_2 = 0$$

This is the equation of a straight line, and the normal vector of this line (the vector which is perpendicular to this line) is:

$$\boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$$

This explains why the decision boundary is perpendicular to the vector $\boldsymbol{w}$.

Single-layer networks: sigmoid neuron

The perceptron can predict the label of a data point, but it cannot provide the prediction probability. In fact, this network cannot tell you how confident it is in its prediction. We need a different activation function called *sigmoid* to get the prediction probability. The sigmoid activation function is defined as follows:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

A plot of this function is given in Figure 5.

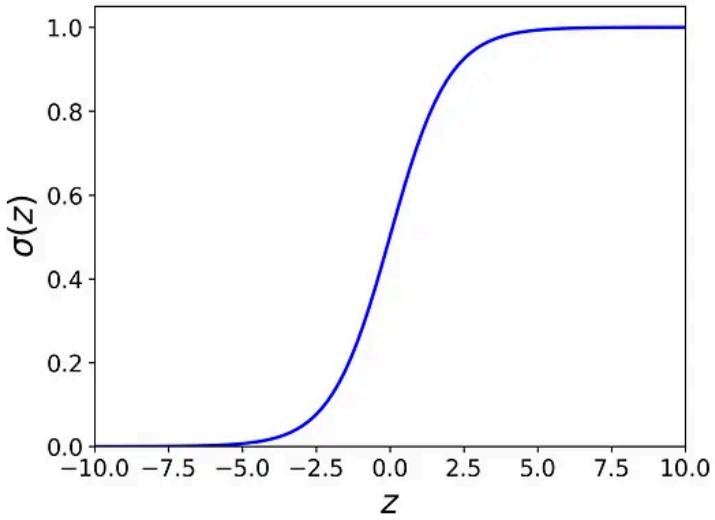

Figure 5

We know that the probability of an event is a number between 0 and 1. As this plot shows the range of the sigmoid function is (0, 1), so it can be used to represent the probability of an outcome. Now, we replace the activation function of the perceptron with a sigmoid function to get the network shown in Figure 6.

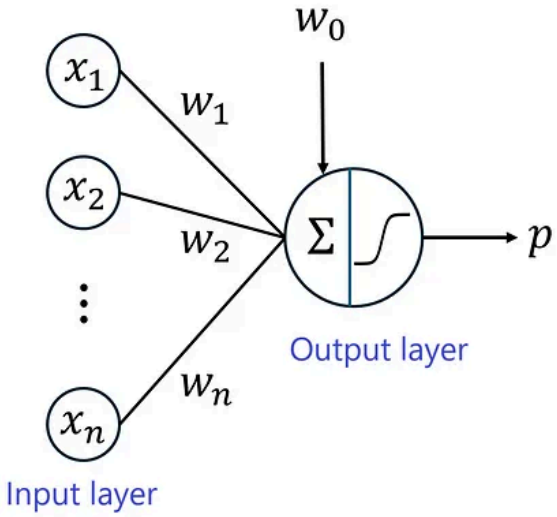

Figure 6

In this network, we denote the output of the network with p, so we can write:

$$z = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$

$$p = \sigma(z) = \frac{1}{1 + e^{-z}}$$

Here p is the probability that the predicted label is 1 (y^=1). To obtain the predicted target, we must compare this probability with a threshold which is 0.5 by default:

$$\hat{y} = \begin{cases} 1 & if\ p \geq 0.5 \\ 0 & Otherwise \end{cases} \qquad (2)$$

To visualize this network, we use the dataset defined in Listing 1 to train it. Listing 5 creates this network using the `keras` library.

```
# Listing 5

np.random.seed(0)
random.seed(0)
tf.random.set_seed(0)

model1 = Sequential()
model1.add(Dense(1, activation='sigmoid', input_shape=(2,)))

model1.compile(loss = 'binary_crossentropy',
               optimizer='adam', metrics=['accuracy'])
model1.summary()
```

The cost function of this neural network is called *cross-entropy*. Next, we use the dataset defined in Listing 1 to train this model.

```
# Listing 6

history1 = model1.fit(X1, y1, epochs=1500, verbose=0, batch_size=X1.shape[0])
plt.plot(history1.history['accuracy'])
plt.title('Accuracy vs Epochs')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.show()
```

Figure 7 shows the plot of accuracy versus epochs for this model.

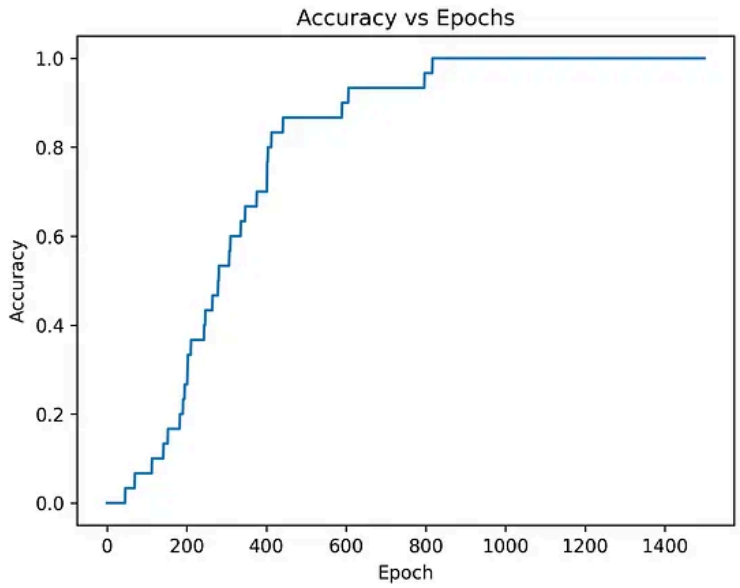

Figure 7

After training the model, we can retrieve the weights of the output layer (w_1 and w_2).

```
# Listing 7

output_layer_weights = model1.layers[0].get_weights()[0]
model1_w1, model1_w2 = output_layer_weights[0, 0], output_layer_weights[1, 0]
```

Finally, we plot the decision boundary of this network. The result is shown in Figure 8.

```
# Listing 8

plt.figure(figsize=(5, 5))
# Plot the vector w
output_layer_weights = model1.layers[0].get_weights()[0]
plt.quiver([0], [0], model1_w1,
           model1_w2, color=['black'],
           width=0.008, angles='xy', scale_units='xy',
           scale=1, zorder=5)
# Plot the boundary
plot_boundary(X1, y1, model1, lims=[-2.3, 1.8, -1.9, 2.2])
ax = plt.gca()
ax.set_aspect('equal')
plt.xlabel('$x_1$', fontsize=16)
plt.ylabel('$x_2$', fontsize=16)
plt.legend(loc='best', fontsize=11)
plt.xlim([-2.3, 1.8])
plt.ylim([-1.9, 2.2])
plt.show()
```

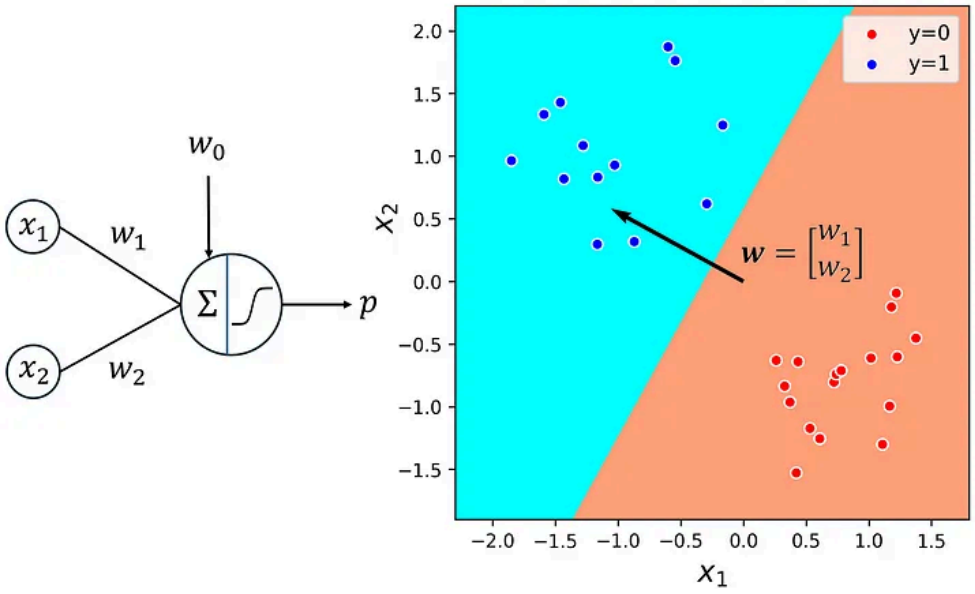

Figure 8

Again we see that the decision boundary is a straight line. We define the vector $\boldsymbol{w}$ using the weights of the output layer:

$$\boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$$

The vector $\boldsymbol{w}$ is perpendicular to the decision boundary, just as we saw with the perceptron. Let's explain the mathematical reasons behind these results. According to Equation 2, the predicted label of all the data points for which p=0.5 is 1. On the other hand, the predicted label for any data point with

p<0.5 will be 0. As a result, the decision boundary is the location of all the data points for which p=0.5:

$$p = \sigma(z) = \frac{1}{1+e^{-z}} = 0.5 \Rightarrow z = w_0 + w_1 x_1 + w_2 x_2 = 0$$

Hence, the decision boundary is the location of all the data points defined by the following equation:

$$w_0 + w_1 x_1 + w_2 x_2 = 0$$

As mentioned before, this is the equation of a straight line, and the normal vector of this line (the vector which is perpendicular to this line) is:

$$\boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$$

Adding more features

So far, we only considered a toy dataset with only two features. Let's see what happens when we have three features. Listing 9 defines another dataset with 3 features. This dataset is plotted in Figure 9.

```
# Listing 9

fig = plt.figure(figsize=(7, 7))
ax = fig.add_subplot(111, projection='3d')

ax.scatter(X2[y2==0, 0], X2[y2==0,1], X2[y2==0,2],
           label="y=0", alpha=0.8, color="red")
ax.scatter(X2[y2==1, 0], X2[y2==1,1], X2[y2==1,2],
           label="y=1", alpha=0.8, color="blue")
ax.legend(loc="upper left", fontsize=12)
ax.set_xlabel("$x_1$", fontsize=18)
ax.set_ylabel("$x_2$", fontsize=18)
ax.set_zlabel("$x_3$", fontsize=15, labelpad=-0.5)
ax.view_init(5, -50)
plt.show()
```

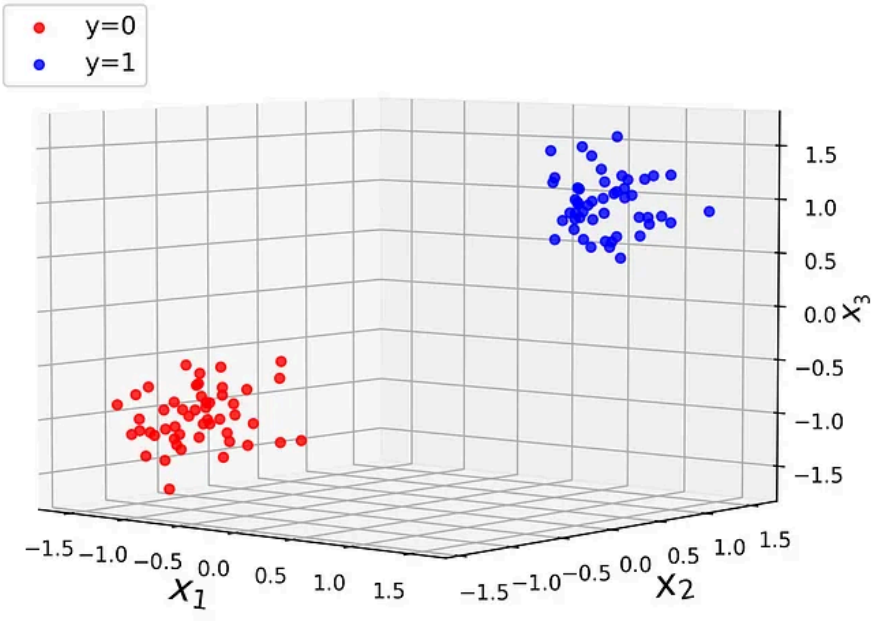

Figure 9

Now we create a new network with a sigmoid neuron and train it using this dataset.

```
# Listing 10

backend.clear_session()
np.random.seed(0)
random.seed(0)
tf.random.set_seed(0)

model2 = Sequential()
model2.add(Dense(1, activation='sigmoid', input_shape=(3,)))

model2.compile(loss = 'binary_crossentropy',
               optimizer='adam', metrics=['accuracy'])
history2 = model2.fit(X2, y2, epochs=1500, verbose=0,
                      batch_size=X2.shape[0])
```

Next, we retrieve the weights of the output layer in the trained model and plot the data points and the decision boundary of the model in Figure 10.

```
# Listing 11

model2_w0 = output_layer_biases[0]
model2_w1,  model2_w2,  model2_w3 = output_layer_weights[0, 0], \
      output_layer_weights[1, 0], output_layer_weights[2, 0]

fig = plt.figure(figsize=(7, 7))
ax = fig.add_subplot(111, projection='3d')

lims=[-2, 2, -2, 2]
```

```
ga1, ga2 = np.meshgrid(np.arange(lims[0], lims[1], (lims[1]-lims[0])/500.0),
                       np.arange(lims[2], lims[3], (lims[3]-lims[2])/500.0))

ga1l = ga1.flatten()
ga2l = ga2.flatten()
ga3 = -(model2_w0 + model2_w1*ga1l + model2_w2*ga2l) / model2_w3
ga3 = ga3.reshape(500, 500)
ax.plot_surface(ga1, ga2, ga3, alpha=0.5)
ax.quiver([0], [0], [0], model2_w1, model2_w2, model2_w3,
          color=['black'], length=0.5, zorder=5)
ax.scatter(X2[y2==0, 0], X2[y2==0,1], X2[y2==0,2],
           label="y=0", alpha=0.8, color="red")
ax.scatter(X2[y2==1, 0], X2[y2==1,1], X2[y2==1,2],
           label="y=1", alpha=0.8, color="blue")
ax.legend(loc="upper left", fontsize=12)
ax.set_xlabel("$x_1$", fontsize=16)
ax.set_ylabel("$x_2$", fontsize=16)
ax.set_zlabel("$x_3$", fontsize=15, labelpad=-0.5)
ax.view_init(5, -50)
plt.show()
```

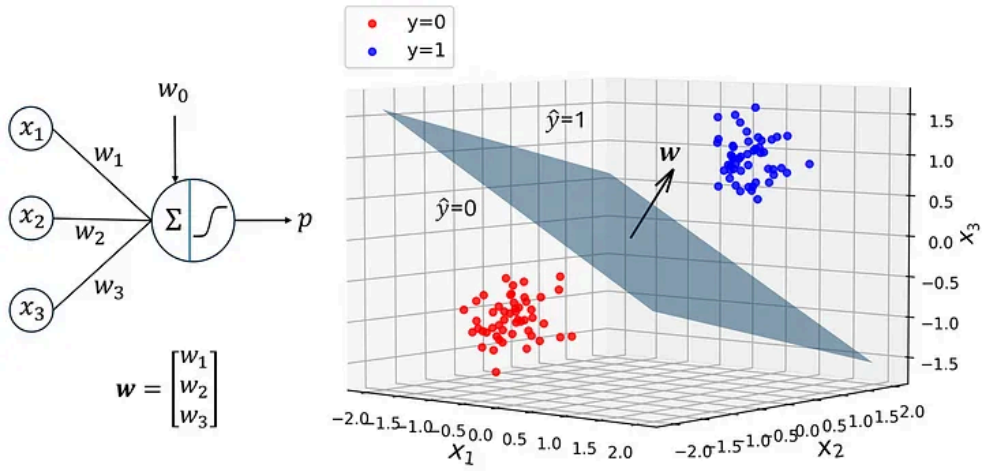

Figure 10

As this figure shows the decision boundary is a plane perpendicular to the vector

$$\boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} \qquad (3)$$

which is formed using the weights of the output layer. Here the decision boundary was calculated as follows:

$$p = \sigma(z) = \frac{1}{1+e^{-z}} = 0.5 \Rightarrow z = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3 = 0$$

So, the decision boundary is the solution of this equation

$$w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3 = 0$$

This is the equation of a plane, and the vector $\boldsymbol{w}$ (defined in Equation 3) is the normal vector of this plane.

Linear classifiers

What happens if we have more than 3 features in the input data? We can easily extend the same idea to find the decision boundary of a network with n features for a perceptron or a sigmoid neuron. In both cases, the decision boundary is the solution to this equation:

$$w_0 + w_1 x_1 + \cdots w_n x_n = 0$$

This equation describes a hyperplane in an n-dimensional space which is perpendicular to the vector

$$\boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$$

In 2D space, the hyperplane becomes a 1-dimensional line while in 3D space, it becomes a 2D plane. A line or a plane has no curvature, and though we cannot visualize hyperplanes in higher dimensions, the concept remains the same. In n-dimensional space, a hyperplane is an n-1-dimensional subspace which is flat and has no curvature.

In machine learning, a *linear classifier* is a classification model that makes its decisions based on a linear combination of the input features. As a result, the decision boundary of a linear classifier is a hyperplane. Perceptron and sigmoid neurons are two examples of a linear classifier.

It is worth mentioning that a sigmoid neuron with a cross-entropy cost function is equivalent to a logistic regression model. The next Listing trains a logistic regression model (from the `scikit-learn` library) on the 2D dataset defined in Listing 1. The decision boundary of this model is plotted in Figure 11. Though it is a straight line, it is not exactly the same line obtained with a sigmoid neuron in Figure 8.

Though the logistic regression and sigmoid neuron (with the cross-entropy cost function) are equivalent models, different approaches are used to find their paramters during the training process. In an aneural network, the gradient descent algorithm with random initialization is used for training, however, the logistic regression model uses a deterministic solver called lbfgs (Limited-memory Broyden–Fletcher–Goldfarb–Shanno) for that purpose. As a result, the final values of the parameters in these two models may differ, changing the position of the decision boundary line.

```
# Listing 12

# Comparing with a logistic regression model
lr_model = LogisticRegression().fit(X1, y1)

plt.figure(figsize=(5, 5))
# Plot the boundary
plot_boundary(X1, y1, lr_model, lims=[-2.3, 1.8, -1.9, 2.2])
ax = plt.gca()
ax.set_aspect('equal')
plt.xlabel('$x_1$', fontsize=16)
plt.ylabel('$x_2$', fontsize=16)
plt.legend(loc='best', fontsize=11)
plt.xlim([-2.3, 1.8])
plt.ylim([-1.9, 2.2])
plt.show()
```

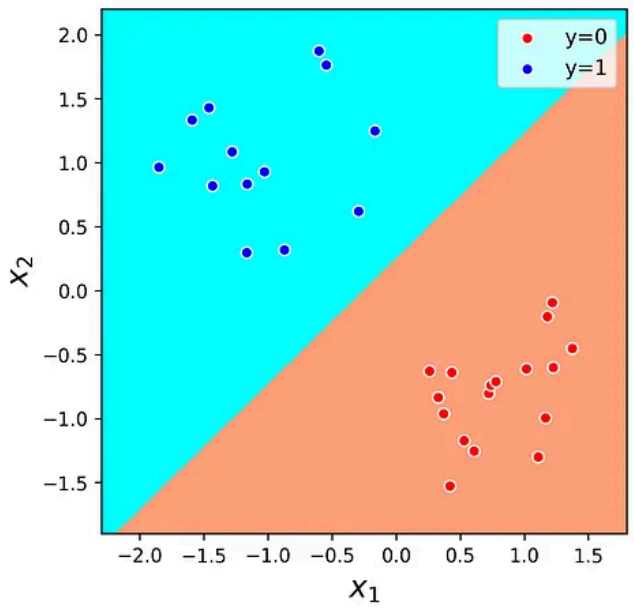

Figure 11

Multi-class classification and the softmax layer

Our attention has been on a binary classification problem so far. If the target of the dataset has more than two labels, then we have a multi-class classification problem, and a *softmax layer* is needed for such a problem.

Suppose that a dataset has n features and its target has C labels. This dataset can be used for training a single-layer neural network with a softmax layer shown in Figure 12.

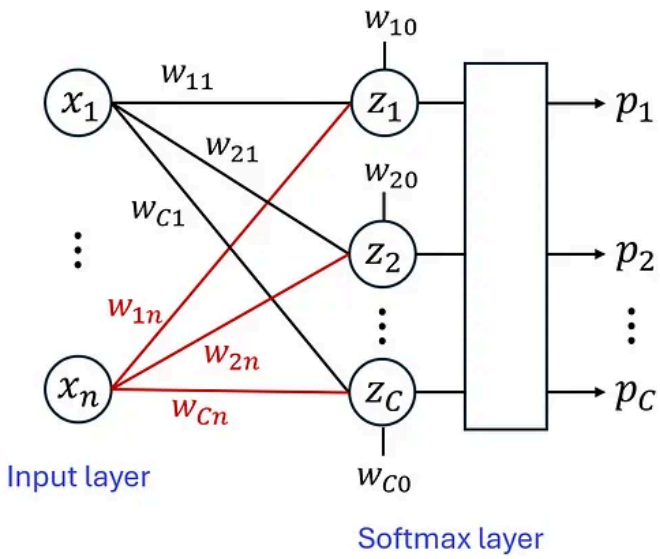

Figure 12

The softmax function is a generalization of the sigmoid function to a multi-class classification problem in which the target has more than 2 labels. The neurons in the output layer give a linear combination of the input features:

$$z_i = w_{i0} + w_{i1}x_1 + w_{i2}x_2 + \cdots w_{in}x_n$$

Each output of the softmax layer is calculated as follows:

$$p_i = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}$$

In this equation, p_i represents the probability of the predicted target being equal to ith label. In the end, the predicted label is the one with the highest probability:

$$\hat{y} = \underset{i}{\text{argmax}}\ p_i$$

Now we create another toy dataset to visualize the softmax layer. In this dataset, we have two features and the target has 3 labels. It is plotted in

Figure 13.

```
# Listing 13

np.random.seed(0)
xt1 = np.random.randn(50, 2) * 0.4 + np.array([2, 1])
xt2 = np.random.randn(50, 2) * 0.7 + np.array([6, 4])
xt3 = np.random.randn(50, 2) * 0.5 + np.array([2, 6])

y3 = np.array(50*[1]+50*[2]+50*[3])
X3 = np.vstack((xt1, xt2, xt3))
scaler = StandardScaler()
X3 = scaler.fit_transform(X3)

plt.figure(figsize=(6, 6))
plt.scatter(X3[y3==1, 0], X3[y3==1,1], label="y=1", alpha=0.7, color="red")
plt.scatter(X3[y3==2, 0], X3[y3==2,1], label="y=2", alpha=0.7, color="blue")
plt.scatter(X3[y3==3, 0], X3[y3==3,1], label="y=3", alpha=0.7, color="green")
plt.legend(loc="best", fontsize=11)
plt.xlabel("$x_1$", fontsize=16)
plt.ylabel("$x_2$", fontsize=16)
ax = plt.gca()
ax.set_aspect('equal')
plt.show()
```

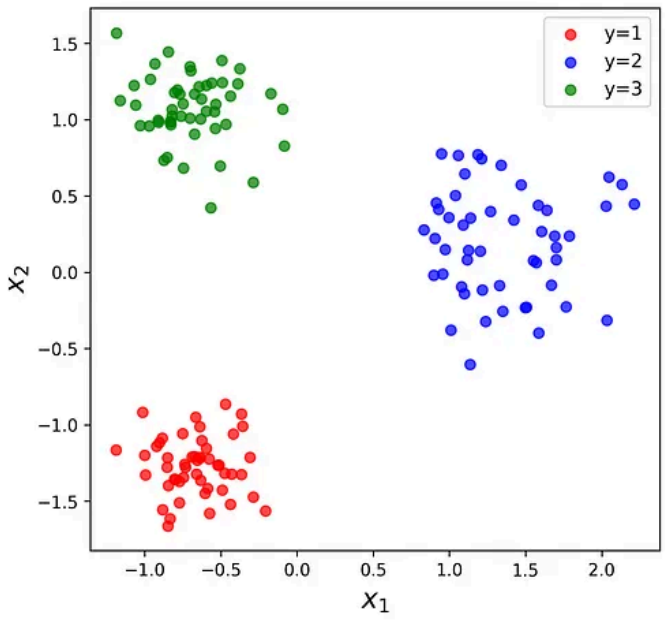

Figure 13

Next, we create a single-layer neural network and train it using this dataset. This network has a softmax layer.

```
# Listing 14

backend.clear_session()
```

```
np.random.seed(0)
random.seed(0)
tf.random.set_seed(0)
y3_categorical = to_categorical(y3-1, num_classes=3)
model3 = Sequential()
model3.add(Dense(3, activation='softmax', input_shape=(2,)))
model3.compile(loss = 'categorical_crossentropy',
               optimizer='adam', metrics=['accuracy'])
history3 = model3.fit(X3, y3_categorical, epochs=2200,
                      verbose=0, batch_size=X3.shape[0])
```

Next, we retrieve the weight and biases of this network:

```
# Listing 15

output_layer_weights = model3.layers[-1].get_weights()[0]
output_layer_biases = model3.layers[-1].get_weights()[1]

model3_w10, model3_w20, model3_w30 = output_layer_biases[0], \
output_layer_biases[1], output_layer_biases[2]

model3_w1 = output_layer_weights[:, 0]
model3_w2 = output_layer_weights[:, 1]
model3_w3 = output_layer_weights[:, 2]
```

Finally, we can plot the decision boundary of this model using Listing 16.

```
# Listing 16

plt.figure(figsize=(5, 5))
plt.quiver([1.7], [0.7], model3_w3[0]-model3_w2[0],
           model3_w3[1]-model3_w2[1], color=['black'],
           width=0.008, angles='xy', scale_units='xy',
           scale=1, zorder=5)
plt.quiver([-0.5], [-2.2], model3_w2[0]-model3_w1[0],
           model3_w2[1]-model3_w1[1], color=['black'],
           width=0.008, angles='xy', scale_units='xy',
           scale=1, zorder=5)
plt.quiver([-1.8], [-1.7], model3_w3[0]-model3_w1[0],
           model3_w3[1]-model3_w1[1], color=['black'],
           width=0.008, angles='xy', scale_units='xy',
           scale=1, zorder=5)
plt.text(0.25, 1.85, "$\mathregular{w_3-w_2}$", color="black",
         fontsize=12, weight="bold", style="italic")
plt.text(1.2, -1.1, "$\mathregular{w_2-w_1}$", color="black",
         fontsize=12, weight="bold", style="italic")
plt.text(-1.5, -0.5, "$\mathregular{w_3-w_1}$", color="black",
         fontsize=12, weight="bold", style="italic")
plot_boundary(X3, y3, model3,lims=[-2.2, 2.4, -2.5, 2.1],
              alpha= 0.7)
ax = plt.gca()
ax.set_aspect('equal')
plt.xlabel('$x_1$', fontsize=16)
plt.ylabel('$x_2$', fontsize=16)
plt.legend(loc='best', fontsize=11)
plt.xlim([-2.2, 2.4])
```

```
plt.ylim([-2.5, 2.1])
plt.show()
```

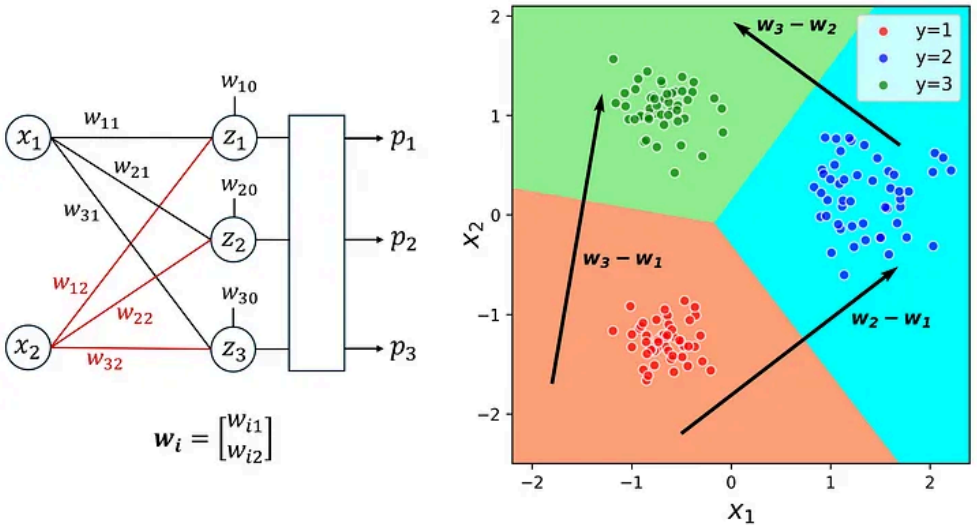

Figure 14

As Figure 14 shows, softmax creates 3 decision boundaries each being a straight line. For example, the decision boundary between labels 1 and 2 is the location of the points with an equal prediction probability for labels 1 and 2. Hence we can write:

$$p_i = p_2 \Rightarrow \frac{e^{z_1}}{\sum_{j=1}^{C} e^{z_j}} = \frac{e^{z_2}}{\sum_{j=1}^{C} e^{z_j}} \Rightarrow z_1 = z_2 \Rightarrow$$

$$w_{10} + w_{11}x_1 + w_{12}x_2 = w_{20} + w_{21}x_1 + w_{22}x_2$$

By simplifying the last equation we get:

$$(w_{20} - w_{10}) + (w_{21} - w_{11})x_1 + (w_{22} - w_{12})x_2 = 0$$

This is again, the equation of a straight line. If we define the vector w_i as:

$$\boldsymbol{w_i} = \begin{bmatrix} w_{i1} \\ w_{i2} \end{bmatrix}$$

The normal vector of this line can be written as:

$$\boldsymbol{w} = \begin{bmatrix} w_{21} - w_{11} \\ w_{22} - w_{12} \end{bmatrix} = \boldsymbol{w}_2 - \boldsymbol{w}_1$$

Hence the decision boundary is perpendicular to $\boldsymbol{w}_2$ -$\boldsymbol{w}_1$. Similarly, it can be shown that the other decision boundaries are all straight lines and the line between the labels i and j is perpendicular to the vector $\boldsymbol{w}_i$-$\boldsymbol{w}\_j$.

More generally, if we have n features in the training dataset, the decision boundaries will be hyperplanes in an n-dimensional space. Here, the hyperplane for the labels i and j is perpendicular to the vector $\boldsymbol{w}_i$-$\boldsymbol{w}\_j$ where

$$\boldsymbol{w}_i = \begin{bmatrix} w_{i1} \\ w_{i2} \\ \vdots \\ w_{in} \end{bmatrix}$$

A single-layer neural network with a softmax activation is a generalization of a linear classifeir to higher dimensions. It continues to use hyperplanes to predict the target's label, but more than one hyperplane is required for predicting all labels.

All the datasets shown so far were *linearly separable* which means that we can separate the data points with different labels using hyperplanes. In reality, a dataset is rarely linearly separable. In the following section, we will look at the difficulties of classifying non-linearly separable datasets.

Multiple-layer networks

Listing 17 creates a toy dataset which is not linearly separable. This dataset is plotted in Figure 15.

```
# Listing 17

np.random.seed(0)
n = 1550
Xt1 = np.random.uniform(low=[0, 0], high=[4, 4], size=(n,2))
drop = (Xt1[:, 0] < 3) & (Xt1[:, 1] < 3)
Xt1 = Xt1[~drop]
yt1= np.ones(len(Xt1))

Xt2 = np.random.uniform(low=[0, 0], high=[4, 4], size=(n,2))
drop = (Xt2[:, 0] > 2.3) | (Xt2[:, 1] > 2.3)

Xt2 = Xt2[~drop]
yt2= np.zeros(len(Xt2))

X4 = np.concatenate([Xt1, Xt2])
y4 = np.concatenate([yt1, yt2])
```

```
scaler = StandardScaler()
X4 = scaler.fit_transform(X4)

colors = ['red', 'blue']
plt.figure(figsize=(6, 6))
for i in np.unique(y4):
    plt.scatter(X4[y4==i, 0], X4[y4==i, 1], label = "y="+str(i),
                color=colors[int(i)], edgecolor="white", s=50)

plt.xlim([-1.9, 1.9])
plt.ylim([-1.9, 1.9])
ax = plt.gca()
ax.set_aspect('equal')
plt.xlabel('$x_1$', fontsize=16)
plt.ylabel('$x_2$', fontsize=16)
plt.legend(loc='upper right', fontsize=11, framealpha=1)
plt.show()
```

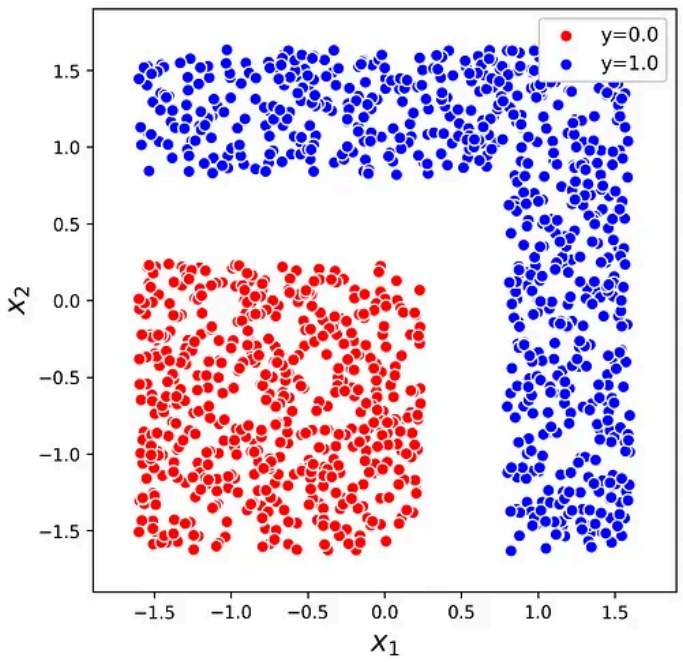

Figure 15

This dataset has two features and a binary target. First, we try to use it to train a sigmoid neuron.

```
# Listing 18

backend.clear_session()
np.random.seed(2)
random.seed(2)
tf.random.set_seed(2)

model4 = Sequential()
model4.add(Dense(1, activation='sigmoid', input_shape=(2,)))
model4.compile(loss = 'binary_crossentropy',
               optimizer='adam', metrics=['accuracy'])
```

```
history4 = model4.fit(X4, y4, epochs=4000, verbose=0,
                      batch_size=X4.shape[0])
```

After training the network, we can plot the decision boundary using Listing 19. Figure 16 shows this plot.

```
# Listing 19

plt.figure(figsize=(5,5))
plot_boundary(X4, y4, model5, lims=[-2, 2, -2, 2])
ax = plt.gca()
ax.set_aspect('equal')
plt.xlabel('$x_1$', fontsize=16)
plt.ylabel('$x_2$', fontsize=16)
plt.legend(loc='upper right', fontsize=11, framealpha=1)
plt.xlim([-1.9, 1.9])
plt.ylim([-1.9, 1.9])
plt.show()
```

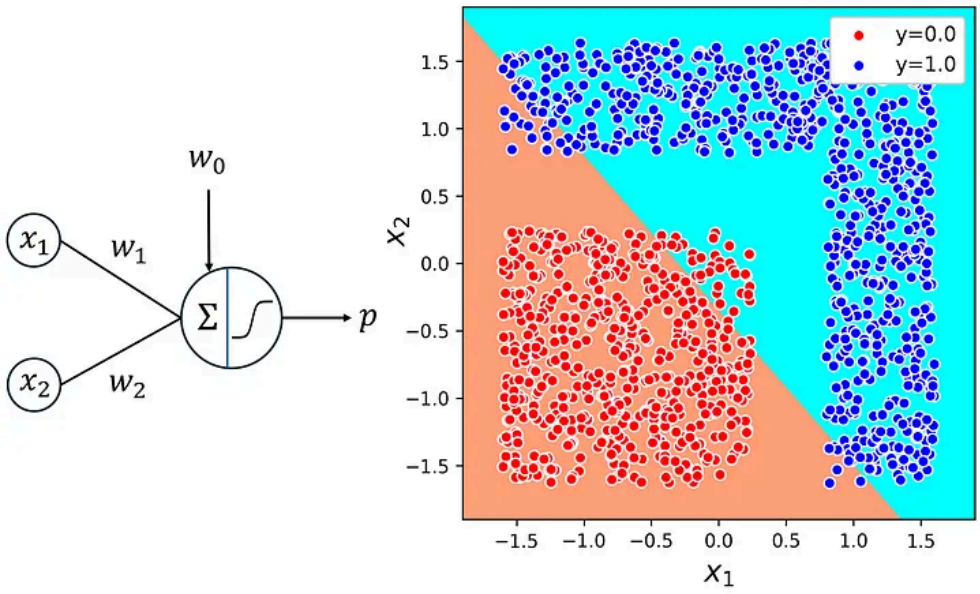

Figure 16

The decision boundary is a straight line, as expected. However, in this dataset, a straight line cannot separate data points with different labels since the dataset is not linearly separable. We can only separate a fraction of the data points using this model resulting in a low prediction accuracy.

Hidden layers

We learned that a single-layer neural network acts as a linear classifier. So, before proceeding to the output layer, we must first convert the original dataset into a linearly separable one. That is precisely what the *hidden layers*

in a multi-layer network do. The input layer receives the features from the original dataset. The features are then transferred to one or more hidden layers, which attempt to turn them into linearly separable features. Finally, the new features are transmitted to the output layer, which acts as a linear classifier.

The performance of a multiple-layer network is determined by the hidden layers' capacity to linearize the input dataset. If the hidden layer is unable to turn the original dataset into a linearly separable one (or at least something close to it), the output layer will fail to provide an accurate classification.

Let's create a multiple-layer network that can be trained using the previous dataset. Listing 20 defines a neural network with one hidden layer, depicted in Figure 17.

```
# Listing 20

backend.clear_session()
np.random.seed(2)
random.seed(2)
tf.random.set_seed(2)

input_layer = Input(shape=(2,))
hidden_layer = Dense(3, activation='relu')(input_layer)
output_layer = Dense(1, activation='sigmoid')(hidden_layer)
model5 = Model(inputs=input_layer, outputs=output_layer)

model5.compile(loss = 'binary_crossentropy', optimizer='adam',
               metrics=['accuracy'])
```

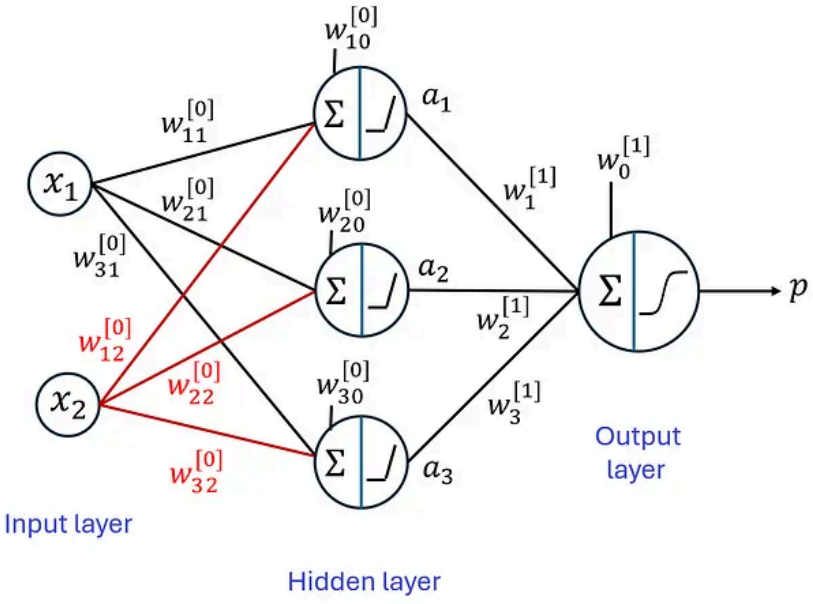

Figure 17

The input layer has 2 neurons since the dataset has only two features. The hidden layer has 3 neurons, and each neuron has a ReLU (Rectified Linear Unit) activation function. This nonlinear activation function is defined as follows:

$$ReLU(z) = \max(0, z)$$

Figure 18 shows a plot of ReLU.

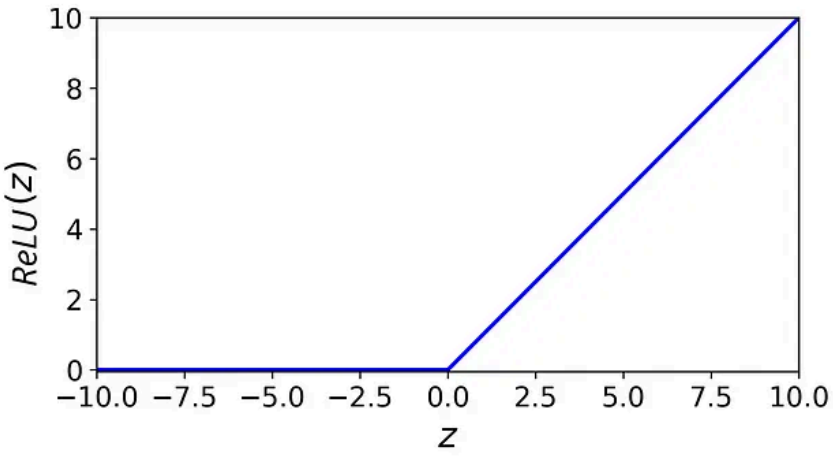

Figure 18

Finally, we have a sigmoid neuron in the output layer. Now, we train this model using our dataset and plot the decision boundary.

```
# Listing 21

history5 = model5.fit(X4, y4, epochs=2200, verbose=0,
                      batch_size=X4.shape[0])

plt.figure(figsize=(5,5))
plot_boundary(X4, y4, model5, lims=[-2, 2, -2, 2])
ax = plt.gca()
ax.set_aspect('equal')
plt.xlabel('$x_1$', fontsize=16)
plt.ylabel('$x_2$', fontsize=16)
plt.legend(loc='upper right', fontsize=11, framealpha=1)
plt.xlim([-1.9, 1.9])
plt.ylim([-1.9, 1.9])
plt.show()
```

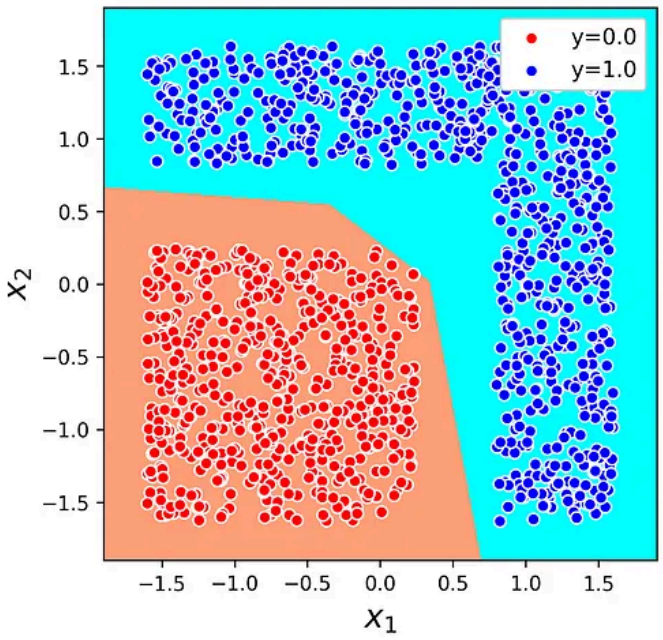

Figure 19

The model can properly separate the data points with labels 0 and 1, but the decision boundary is not a straight line anymore. How the model could achieve that? Let's take a look at the output of the hidden and output layers. Listing 22 plots the output of the hidden layer (Figure 20). Please note that we have three neurons in the hidden layer and their output is denoted by a_1, a_2 and a_3. Hence, we need to plot them in a 3D space. In this case, the decision boundary of the output layer is a plane that separates the data points of the hidden space.

```
# Listing 22

hidden_layer_model = Model(inputs=model5.input,
                           outputs=model5.layers[1].output)
hidden_layer_output = hidden_layer_model.predict(X4)
output_layer_weights = model5.layers[-1].get_weights()[0]
output_layer_biases = model5.layers[-1].get_weights()[1]

 w0 = output_layer_biases[0]
 w1, w2, w3= output_layer_weights[0, 0], \
  output_layer_weights[1, 0], output_layer_weights[2, 0]

fig = plt.figure(figsize=(7, 7))
ax = fig.add_subplot(111, projection='3d')
# Plot the bounday
lims=[0, 4, 0, 4]
ga1, ga2 = np.meshgrid(np.arange(lims[0], lims[1], (lims[1]-lims[0])/500.0),
                       np.arange(lims[2], lims[3], (lims[3]-lims[2])/500.0))

ga1l = ga1.flatten()
ga2l = ga2.flatten()
ga3 = (0.5 - (w0 + w1*ga1l + w2*ga2l)) / w3
ga3 = ga3.reshape(500, 500)
ax.plot_surface(ga1, ga2, ga3, alpha=0.5)
```

```
marker_colors = ['red', 'blue']
target_labels = np.unique(y4)
n = len(target_labels)
for i, label in enumerate(target_labels):
    ax.scatter(hidden_layer_output[y4==label, 0],
               hidden_layer_output[y4==label, 1],
               hidden_layer_output[y4==label, 2],
               label="y="+str(label),
               color=marker_colors[i])

ax.view_init(0, 25)
ax.set_xlabel('$a_1$', fontsize=14)
ax.set_ylabel('$a_2$', fontsize=14)
ax.set_zlabel('$a_3$', fontsize=14)
ax.legend(loc="best")
plt.show()
```

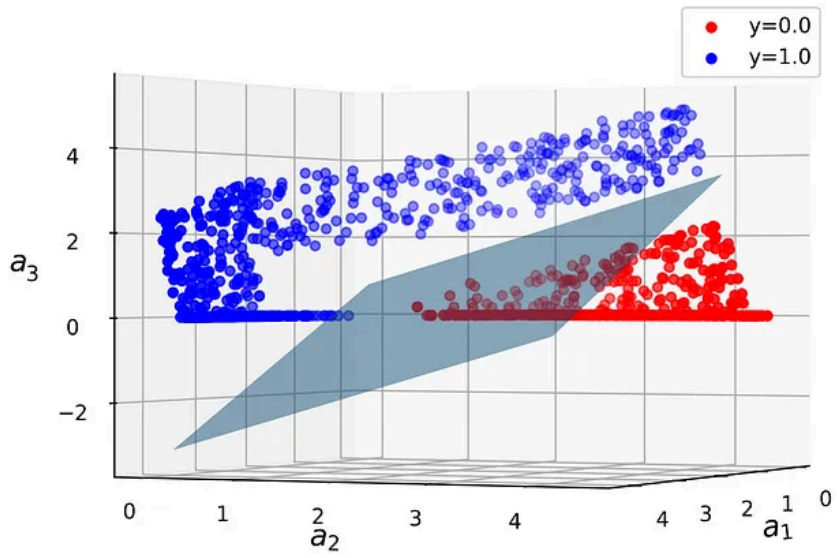

Figure 20

The original dataset was two-dimensional and non-linearly separable. Hence the hidden layer transformed it into a 3D dataset which is now linearly separable. Then the plane created by the output layer easily classifies it.

So, we conclude that the nonlinear decision boundary shown in Figure 19 is like an illusion, and we still have a linear classifier at the output layer. However, when the plane is mapped to the original 2D dataset, it appears as a nonlinear decision boundary (Figure 21).

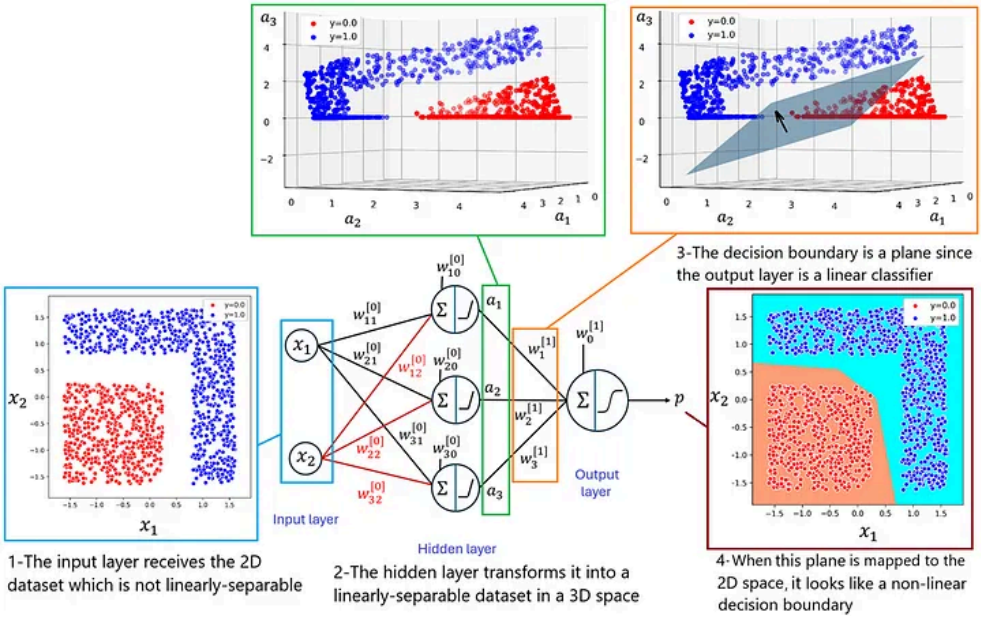

Figure 21

The game of dimensions

When a data point passes through each layer of a neural network, the number of neurons in that layer determines its dimension. Here each neuron encodes one of the dimensions. Since the original dataset is 2D, we need two neurons in the input layer. The hidden layer has three neurons, so it transforms the 2D data points into 3D data points. The additional dimension somehow unfolds the input dataset and helps with converting it into a linearly separable dataset. Finally, the output layer is simply a linear classifier in a 3D space.

The performance of a multiple-layer network is determined by the hidden layers' capacity to linearize the input dataset. The hidden layer of the neural network defined in this example could transform the original dataset into a linearly separable dataset. In reality, though, that isn't always possible. A dataset that is roughly linearly separable is sometimes the best result that the hidden layer can produce. As a result, certain data points may be mislabeled by the output layer. However, this is acceptable as long as the model's overall accuracy is sufficient for practical applications.

Additionally, it is common to have a neural network with multiple hidden layers. In that case, the hidden layers combine to create a linearly separable dataset in the end.

The need for a nonlinear action function

It is crucial to have a nonlinear activation function (such as ReLU) in the hidden layers. We can explain the importance of nonlinear activation functions using an example. Let's replace the ReLU activation functions in

the previous neural network with a linear activation function. A linear activation function is defined as follows:

$$g(z) = z$$

Figure 22 shows the plot of this activation function.

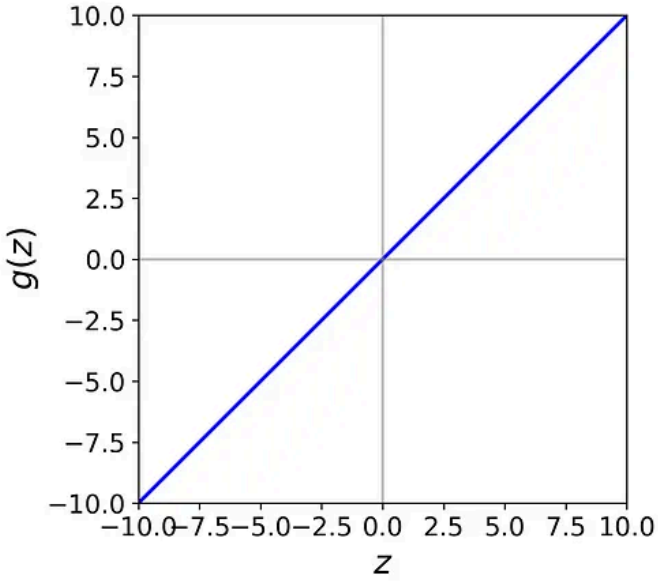

Figure 22

Let's now use a linear activation function for the hidden layer of the prior neural network in Figure 17. This redesigned neural network is illustrated in Figure 23.

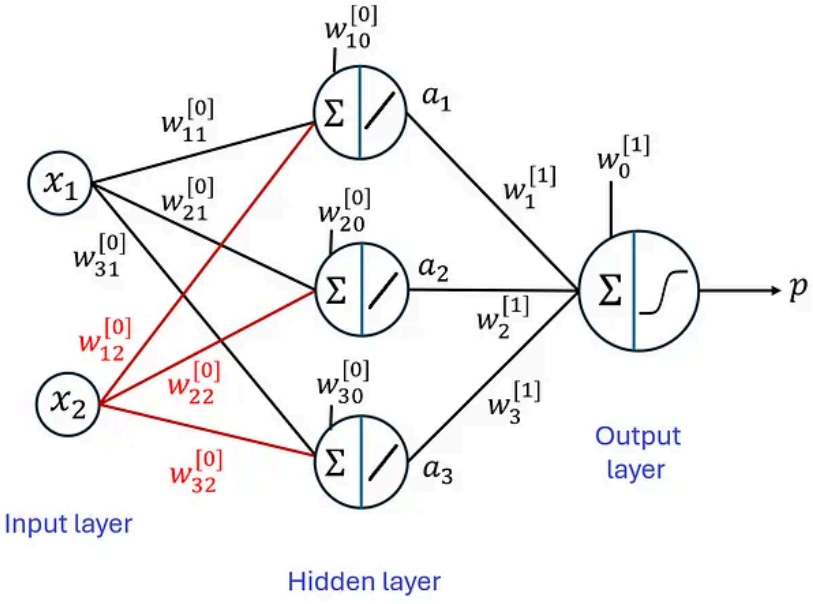

Figure 23

Listing 23 defines the neural network and trains it with the previous dataset. The decision boundary is plotted in Figure 24.

```
# Listing 23

backend.clear_session()
np.random.seed(2)
random.seed(2)
tf.random.set_seed(2)

input_layer = Input(shape=(2,))
hidden_layer_linear = Dense(3, activation='linear')(input_layer)
output_layer = Dense(1, activation='sigmoid')(hidden_layer_linear)
model6 = Model(inputs=input_layer, outputs=output_layer)

model6.compile(loss = 'binary_crossentropy',
               optimizer='adam', metrics=['accuracy'])

history6 = model6.fit(X4, y4, epochs=1000, verbose=0,
                      batch_size=X4.shape[0])

plt.figure(figsize=(5,5))
plot_boundary(X4, y4, model6, lims=[-2, 2, -2, 2])
ax = plt.gca()
ax.set_aspect('equal')
plt.xlabel('$x_1$', fontsize=16)
plt.ylabel('$x_2$', fontsize=16)
plt.legend(loc='upper right', fontsize=11, framealpha=1)
plt.xlim([-1.9, 1.9])
plt.ylim([-1.9, 1.9])
plt.show()
```

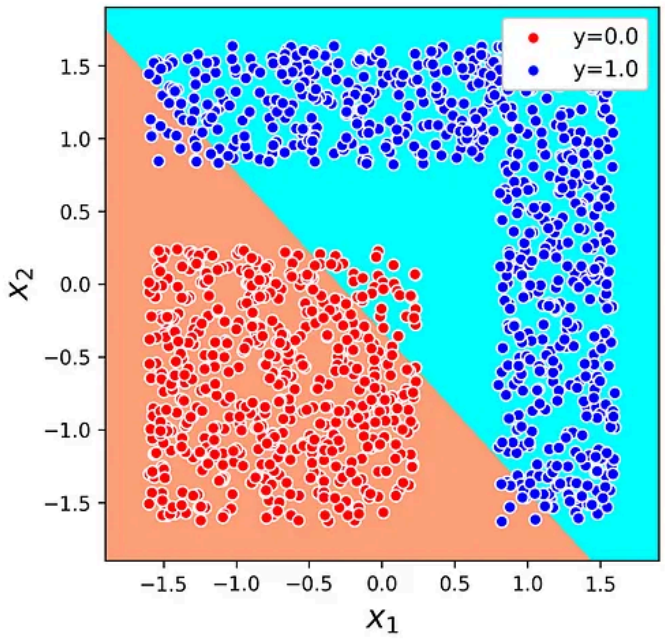

Figure 24

We see that the decision boundary is still a straight line. This means that the hidden layer fails to linearize the dataset. Let's explain the reason for that. Since we are using a linear activation function, the output of the hidden layer is as follows:

$$a_1 = w_{10}^{[0]} + w_{11}^{[0]} x_1 + w_{12}^{[0]} x_2$$

$$a_2 = w_{20}^{[0]} + w_{21}^{[0]} x_1 + w_{22}^{[0]} x_2$$

$$a_3 = w_{30}^{[0]} + w_{31}^{[0]} x_1 + w_{32}^{[0]} x_2$$

These equations can be expressed in a vector form:

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} w_{10}^{[0]} \\ w_{20}^{[0]} \\ w_{30}^{[0]} \end{bmatrix} + \begin{bmatrix} w_{11}^{[0]} \\ w_{21}^{[0]} \\ w_{31}^{[0]} \end{bmatrix} x_1 + \begin{bmatrix} w_{12}^{[0]} \\ w_{22}^{[0]} \\ w_{32}^{[0]} \end{bmatrix} x_2$$

This means that each data point in the $a_1 a_2 a_3$ space is on a plane parallel to the vectors:

$$v_1 = \begin{bmatrix} w_{11}^{[0]} \\ w_{21}^{[0]} \\ w_{31}^{[0]} \end{bmatrix}, \quad v_2 = \begin{bmatrix} w_{12}^{[0]} \\ w_{22}^{[0]} \\ w_{32}^{[0]} \end{bmatrix}$$

Listing 24 plots the output of the hidden layer with the vectors v_1 and v_2. This plot is shown on the right-hand side of Figure 25.

```
# Listing 24

fig = plt.figure(figsize=(7, 7))
ax = fig.add_subplot(111, projection='3d')
# Plot the bounday
lims=[-3, 4, -3, 4]
ga1, ga2 = np.meshgrid(np.arange(lims[0], lims[1], (lims[1]-lims[0])/500.0),
                       np.arange(lims[2], lims[3], (lims[3]-lims[2])/500.0))

ga1l = ga1.flatten()
ga2l = ga2.flatten()
ga3 = (0.5 - (w0 + w1*ga1l + w2*ga2l)) / w3
ga3 = ga3.reshape(500, 500)
```

```
ax.plot_surface(ga1, ga2, ga3, alpha=0)


marker_colors = ['red', 'blue']
target_labels = np.unique(y4)
n = len(target_labels)
for i, label in enumerate(target_labels):
    ax.scatter(hidden_layer_output[y4==label, 0],
               hidden_layer_output[y4==label, 1],
               hidden_layer_output[y4==label, 2],
               label="y="+str(label),
               color=marker_colors[i], alpha=0.15)

ax.quiver([0], [0], [0], hidden_layer_weights[0,0],
          hidden_layer_weights[0,1], hidden_layer_weights[0,2],
          color=['black'], length=1.1, zorder=15)
ax.quiver([0], [0], [0], hidden_layer_weights[1,0],
          hidden_layer_weights[1,1], hidden_layer_weights[1,2],
          color=['black'], length=1.1, zorder=15)

ax.view_init(30, 100)
ax.set_xlabel('$a_1$', fontsize=14)
ax.set_ylabel('$a_2$', fontsize=14)
ax.set_zlabel('$a_3$', fontsize=14)
ax.legend(loc="best")
plt.show()
```

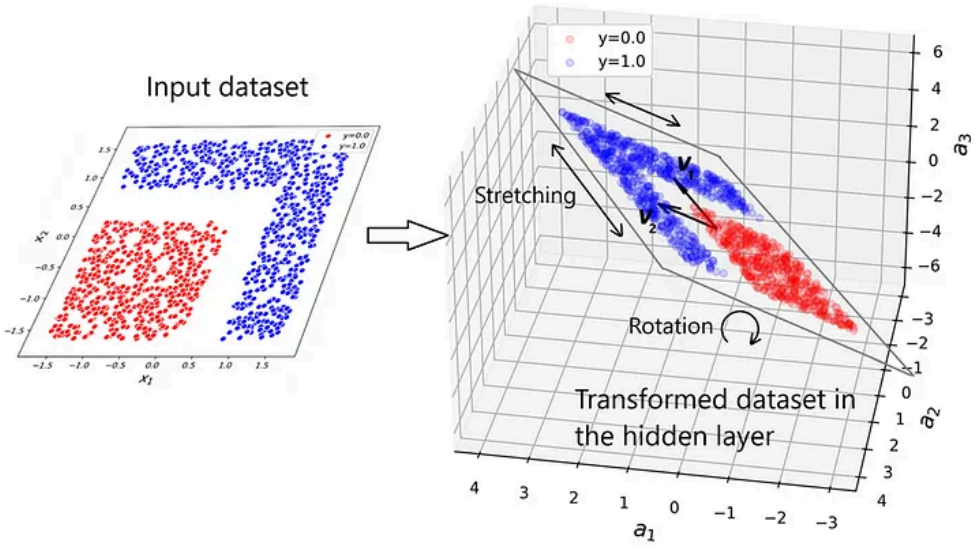

Figure 25

The data points in the $a_1a_2a_3$ space are apparently 3 dimensional, however, their mathematical dimension is 2 since they all lie on a 2D plane. While the hidden layer has 3 neurons, it cannot generate a real 3D dataset. It can only rotate the original dataset in a 3D space and stretch it along the vectors v_1 and v_2. However, these operations don't break the structure of the original dataset and the transformed dataset remains non-linearly separable. Hence, the plane created by the output layer cannot classify the data points properly. When this plane is mapped back into the 2D space, it appears as a straight line (Figure 26).

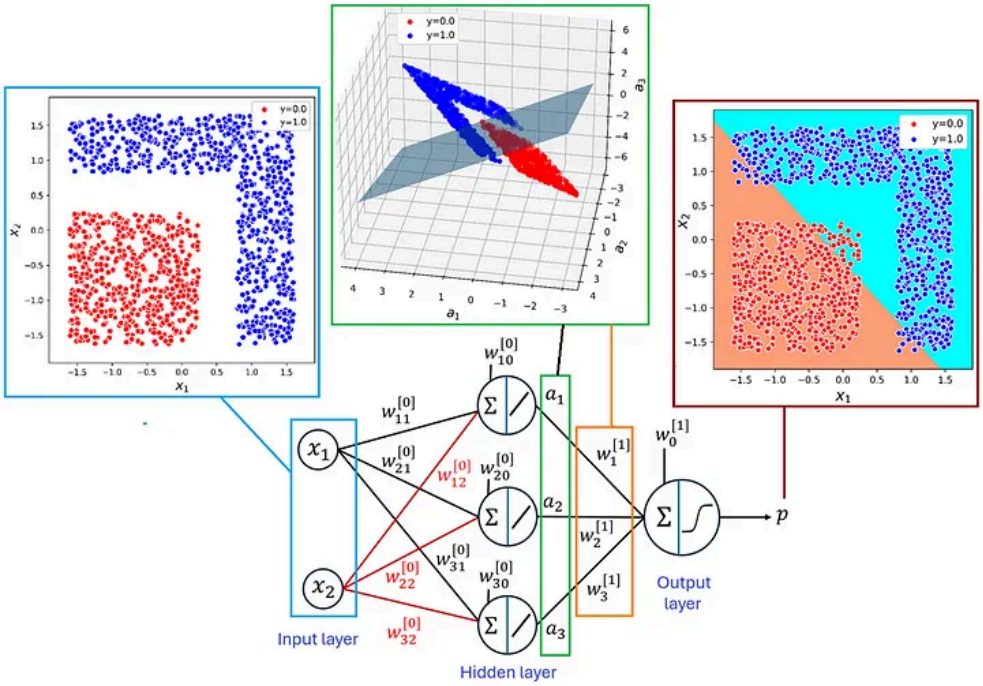

Figure 26

In conclusion, the number of neurons in the hidden layer is not the only factor that defines the mathematical dimension of the transformed dataset. Without a nonlinear activation function, the mathematical dimension of the original dataset doesn't change, and the hidden layer fails to serve its purpose.

Neural networks for regression

In this section, we will see how a neural network can solve a regression problem. In a regression problem, the target of the dataset is a continuous variable. We first create an example of such a dataset in Listing 25 and plot it in Figure 27.

```
# Listing 25

np.random.seed(0)
num_points = 100
X5 = np.linspace(0,1, num_points)
y5 = -(X5-0.5)**2 + 0.25

fig = plt.figure(figsize=(5, 5))
plt.scatter(X5, y5)
plt.xlabel('x', fontsize=14)
plt.ylabel('y', fontsize=14)
plt.show()
```

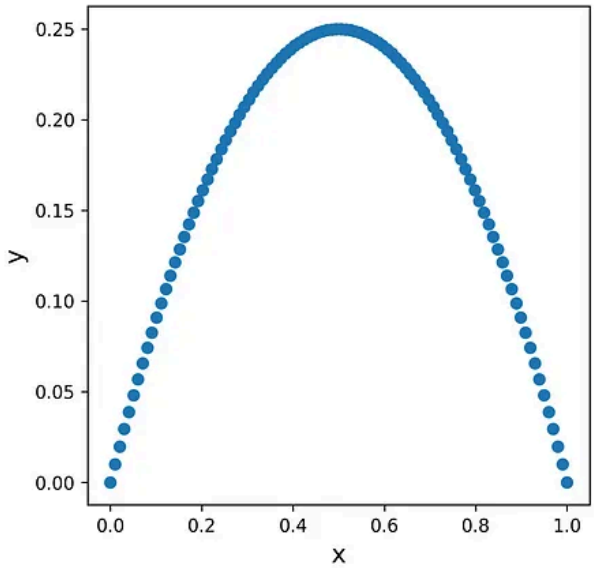

Figure 27

Single-layer networks

We first try a single-layer neural network. Here the output layer has a single neuron with a linear activation function. This neural network is shown in Figure 28.

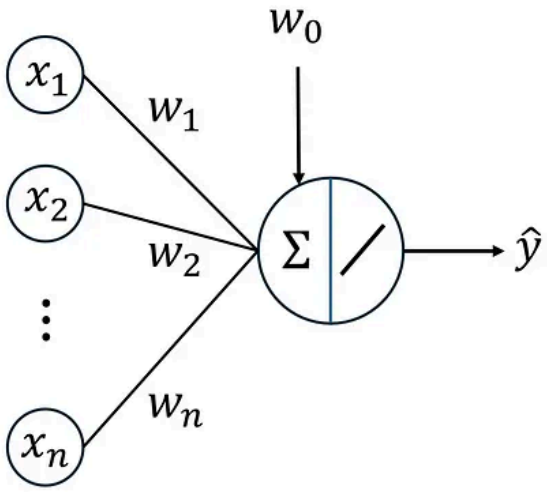

Figure 28

Its output can be written as:

$$\hat{y} = w_0 + w_1 x + w_2 x_2 + \cdots + w_n x_n$$

Now if we use a Mean Squared Error (MSE) cost function for that, it becomes like a linear regression model. Listing 26 uses the previous dataset to train such a network. Since the dataset has only one feature, the neural network ends up having only one neuron (Figure 29).

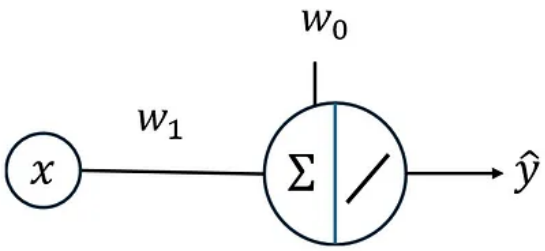

Figure 29

```
# Listing 26

backend.clear_session()
np.random.seed(0)
random.seed(0)
tf.random.set_seed(0)

model6 = Sequential()
model6.add(Dense(1, activation='linear', input_shape=(1,)))
model6.compile(optimizer='adam', loss='mse', metrics=['mse'])
history7 = model6.fit(X5, y5, epochs=500, verbose=0,
                      batch_size=X5.shape[0])
```

After training the model, we can plot its prediction versus the original data points.

```
# Listing 27

X5_test = np.linspace(0,1, 1000)
yhat1 = model6.predict(X5_test)

fig = plt.figure(figsize=(5, 5))
plt.scatter(X5, y5, label="Train data")
plt.plot(X5_test, yhat1, color="red", label="Prediction")
plt.xlabel('x', fontsize=14)
plt.ylabel('y', fontsize=14)
plt.legend(loc="best", fontsize=11)
plt.show()
```

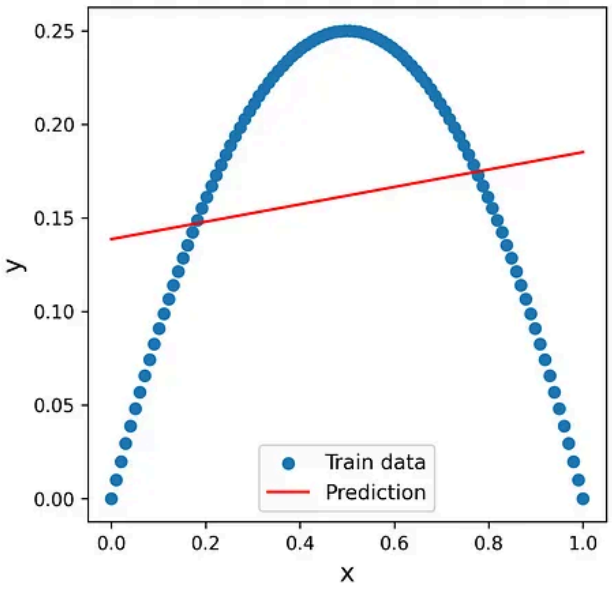

Figure 30

So, we conclude that a single-layer neural network with a linear activation function and an MSE cost function behaves similarly to a linear regression model.

Multiple-layer networks

To learn a nonlinear dataset, we need to add hidden layers. Figure 31 shows an example of such a network. Here we have one hidden layer with linear activation functions.

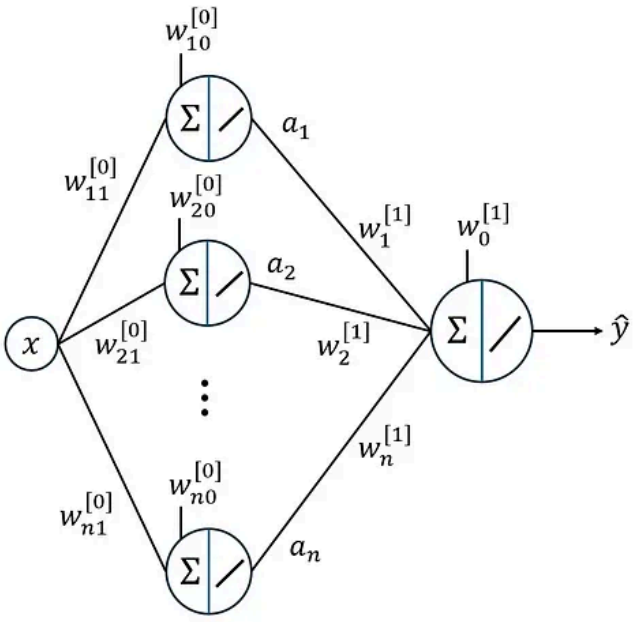

Figure 31

However, this neural network also acts like a linear regression model. To explain the reason for that, we first write the outputs of the hidden layer:

$$
\begin{aligned}
a_1 &= w_{10}^{[0]} + w_{11}^{[0]} x \\
a_2 &= w_{20}^{[0]} + w_{21}^{[0]} x \\
&\vdots \\
a_n &= w_{n0}^{[0]} + w_{n1}^{[0]} x
\end{aligned}
$$

Now, we can calculate the output of the neural network:

$$
\begin{aligned}
\hat{y} = w_0^{[1]} + w_1^{[1]} a_1 + \cdots w_n^{[1]} a_n &= w_0^{[1]} + \left(w_1^{[1]} w_{10}^{[0]} + \cdots + w_n^{[1]} w_{n0}^{[0]} \right) \\
&\quad + \left(w_1^{[1]} w_{11}^{[0]} + \cdots + w_n^{[1]} w_{n1}^{[0]} \right) x \\
&= c + dx
\end{aligned}
$$

This means that with an MSE cost function, the neural network is still behaving like a linear model. To avoid this problem, we need to use a nonlinear activation function in the hidden layer.

In the next example, we replace the activation functions of the hidden layer with ReLU as shown in Figure 32. Here, the hidden layer has 10 neurons.

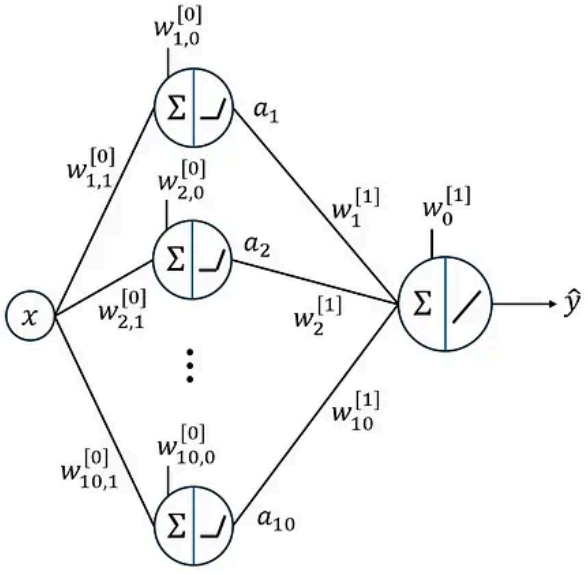

Figure 32

Listing 28 implements and trains this neural network.

```
# Listing 28

backend.clear_session()
np.random.seed(15)
random.seed(15)
tf.random.set_seed(15)

input_layer = Input(shape=(1,))
x = Dense(10, activation='relu')(input_layer)
output_layer = Dense(1, activation='linear')(x)
model7 = Model(inputs=input_layer, outputs=output_layer)

model7.compile(optimizer='adam', loss='mse', metrics=['mse'])

history8 = model7.fit(X5, y5, epochs=1500, verbose=0,
                      batch_size=X5.shape[0])

hidden_layer_model = Model(inputs=model7.input,
                           outputs=model7.layers[1].output)
hidden_layer_output = hidden_layer_model.predict(X5_test)
output_layer_weights = model7.layers[-1].get_weights()[0]
output_layer_biases = model7.layers[-1].get_weights()[1]
```

After training, we can finally plot the prediction of this neural network.

```
# Listing 29

X5_test = np.linspace(0,1, 1000)
yhat2 = model7.predict(X5_test)

fig = plt.figure(figsize=(5, 5))
plt.scatter(X5, y5, label="Train data", alpha=0.7)
plt.plot(X5_test, yhat2, color="red", label="Prediction")
plt.xlabel('x', fontsize=14)
plt.ylabel('y', fontsize=14)
plt.legend(loc="best", fontsize=11)
plt.show()
```

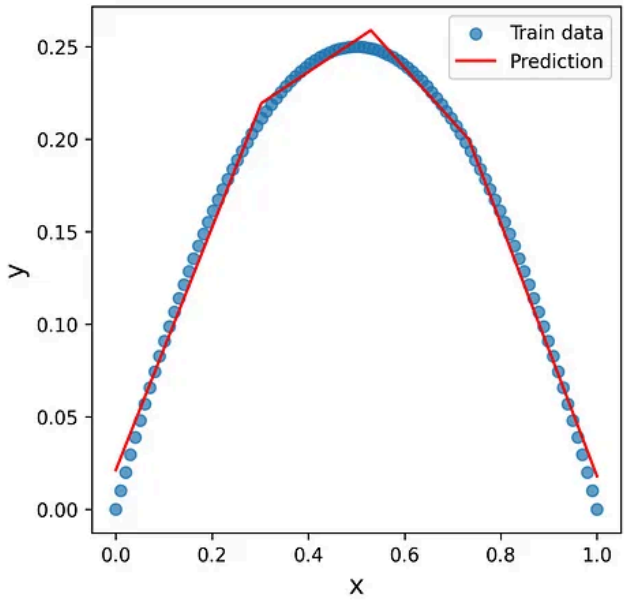

Figure 33

We see that the network can now generate a nonlinear prediction. Let's take a look at the hidden layer. The next Listing plots the outputs of the hidden layer $a_1 \ldots a_{10}$ in Figure 34. The output neuron first multiplies each a_i by its corresponding weight (w_i^[1]a_i). Finally, it computes the following sum

$$\hat{y} = w_0^{[1]} + w_1^{[1]} a_1 + \cdots w_{10}^{[1]} a_{10}$$

which is the prediction of the neural network. All these terms are plotted in Figure 34.

```
# Listing 30

fig, axs = plt.subplots(10, 4, figsize=(18, 24))
plt.subplots_adjust(wspace=0.55, hspace=0.2)

for i in range(10):
    axs[i, 0].plot(X5_test, hidden_layer_output[:, i], color="black")
    axs[i, 1].plot(X5_test,
                   hidden_layer_output[:, i]*output_layer_weights[i],
                   color="black")
    axs[i, 0].set_ylabel(r'$a_{%d}$' % (i+1), fontsize=21)
    axs[i, 1].set_ylabel(r'$w^{[1]}_{%d}a_{%d}$' % (i+1, i+1), fontsize=21)
    axs[i, 2].axis('off')
    axs[i, 3].axis('off')
axs[i, 0].set_xlabel("x", fontsize=21)
axs[i, 1].set_xlabel("x", fontsize=21)

axs[4, 2].axis('on')
axs[6, 2].axis('on')
axs[4, 2].plot(X5_test, [output_layer_biases]*len(X5_test))
axs[6, 2].plot(X5_test,
               (hidden_layer_output*output_layer_weights.T).sum(axis=1))
axs[6, 2].set_xlabel("x", fontsize=21)
```

```
axs[4, 2].set_ylabel("$w^{[1]}_0$", fontsize=21)
axs[4, 2].set_xlabel("x", fontsize=21)
axs[6, 2].set_ylabel("Sum", fontsize=21)
axs[5, 3].axis('on')
axs[5, 3].scatter(X5, y5, alpha=0.3)
axs[5, 3].plot(X5_test, yhat2, color="red")
axs[5, 3].set_xlabel("x", fontsize=21)
axs[5, 3].set_ylabel("$\hat{y}$", fontsize=21)
plt.show()
```

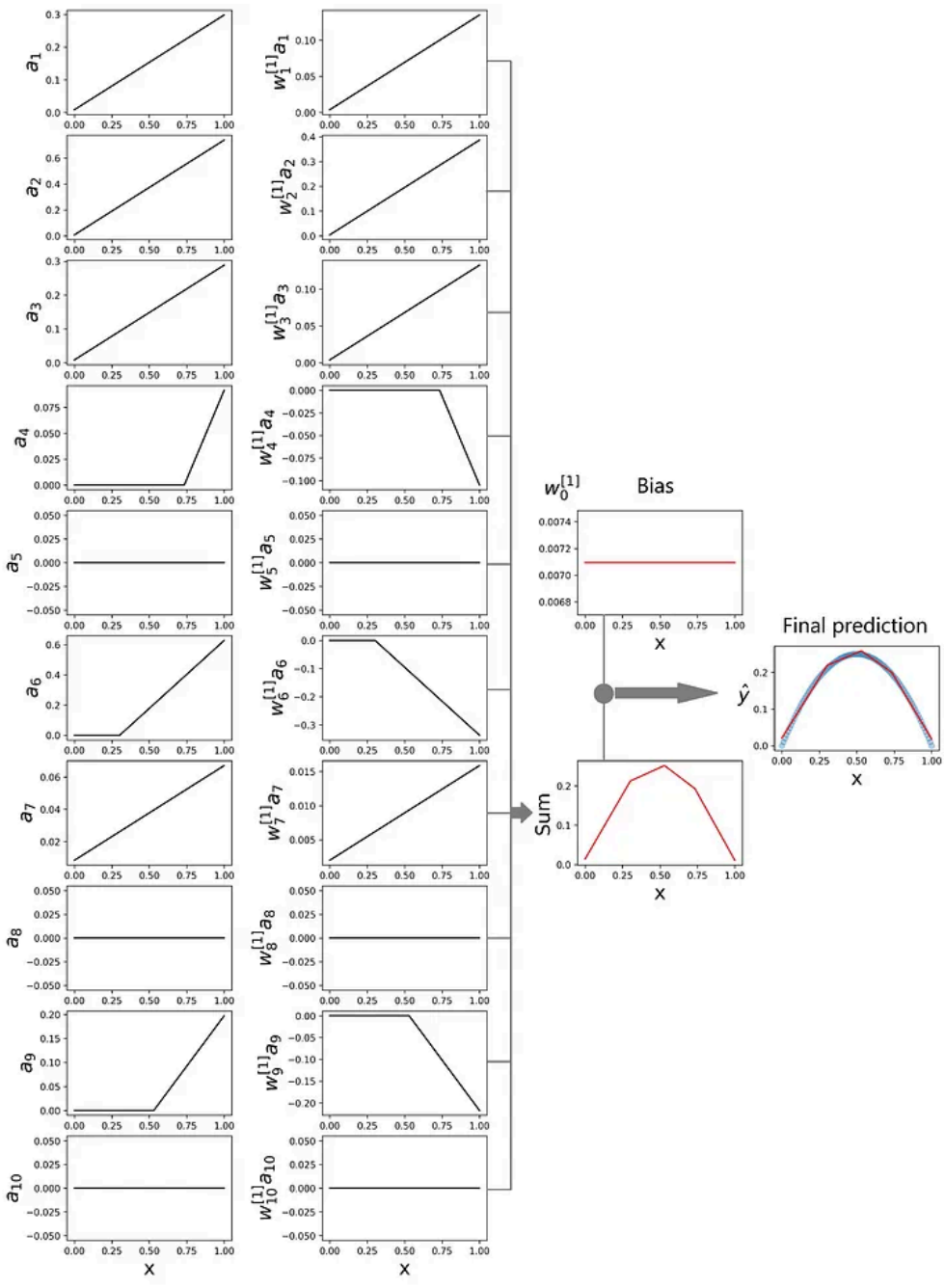

Figure 34

In our neural network, each neuron in the hidden layer has a ReLU activation function. We showed the plot of the ReLU activation function in Figure 18. It consists of two lines that intersect at the origin. The one on the

left is horizontal, while the other has a slope of one. The weight and bias of each neuron in the hidden layer modifies the shape of ReLU. It can change the location of the intersection point, the order of these lines, and the slope of the non-horizontal line. After that, the weight of the output layer can also change the slope of the non-horizontal line. An example of such changes is shown in Figure 35.

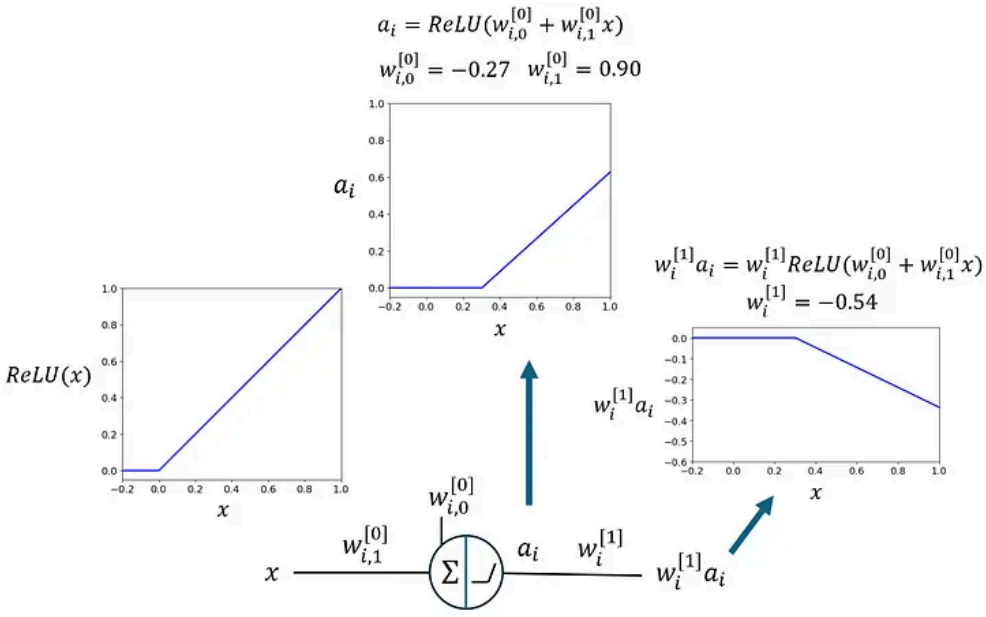

Figure 35

The modified ReLU functions are then combined to approximate the shape of the target of the dataset, as shown in Figure 36. Each modified ReLU function has a simple structure, but a large number of them when combined can approximate any continuous function. At the end, the bias of the output layer is added to the sum of the ReLU functions to adjust them vertically.

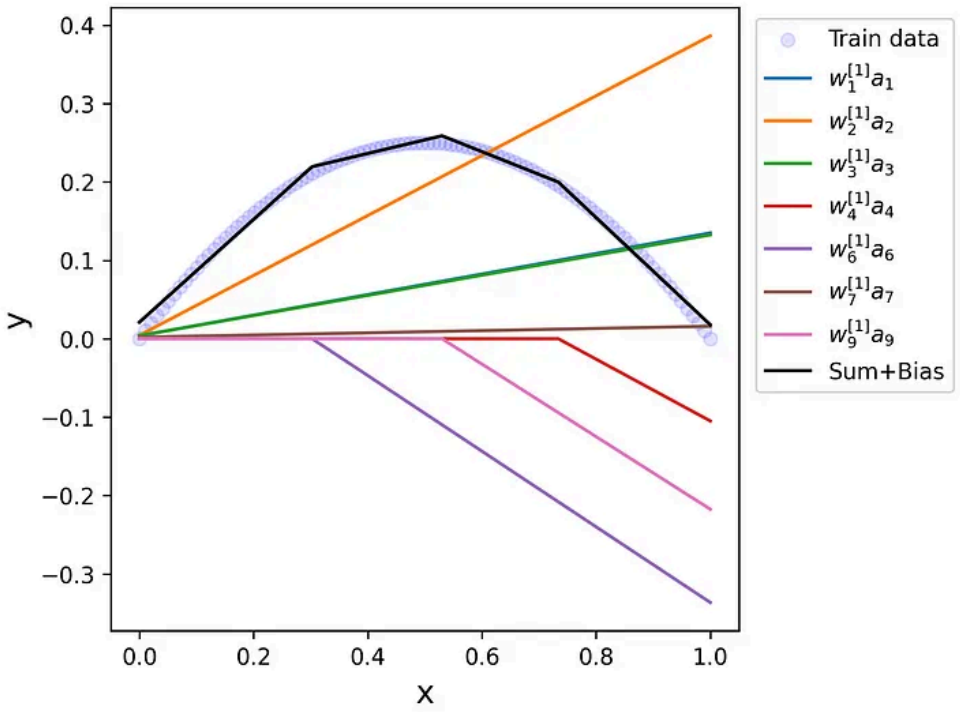

Figure 36

The *Universal Approximation Theorem* states that a feedforward neural network with one hidden layer containing a sufficiently large number of neurons can approximate any continuous function on a subset of inputs with any desired accuracy, provided the activation function is non-constant, bounded, and continuous. To demonstrate this in practice, we used the same neural network from Listing 28, but with 400 neurons in the hidden layer this time. Figure 37 shows the prediction of this neural network. You can see that adding more neurons to the hidden layer significantly improves the neural network's ability to approximate the target.

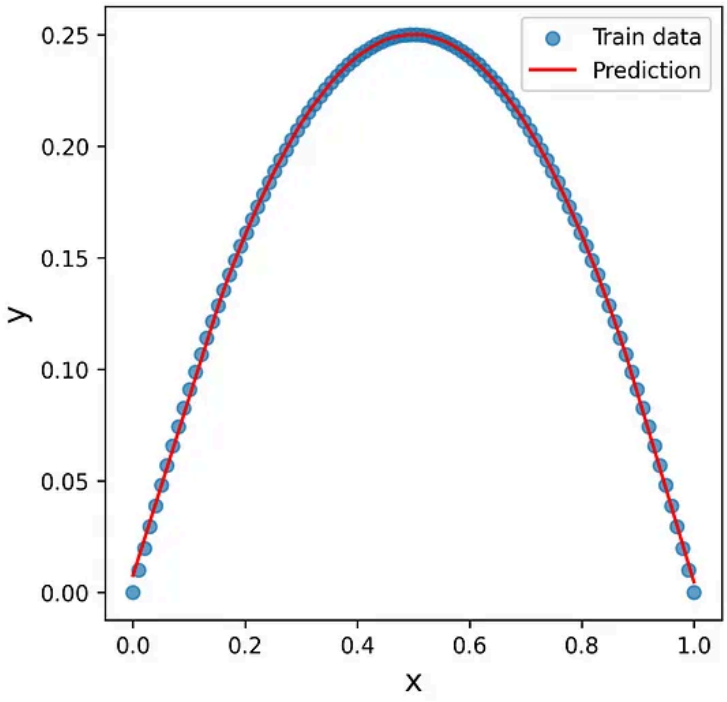

Figure 37

In this article, we presented a visual understanding of neural networks and the role that each layer plays in making the final prediction. We started with perceptrons and showed the limitations of a single-layer network. We saw that in a classification problem, a single-layer neural network is equal to a linear classifier and in a regression problem the behaviour is like a linear regression mode. The role of hidden layers and nonlinear activation function was explained. In a classification problem, the hidden layer tries to linearize a non-linearly separable dataset. In regression problems, the output of the neurons in the hidden layer is like nonlinear building blocks that are added together to make the final prediction.

I hope that you enjoyed reading this article. If you feel my articles are helpful, please follow me on Medium. All the Code Listings in this article are available for download as a Jupyter Notebook from GitHub at:

https://github.com/reza-bagheri/neural_nets_visualization/blob/main/neural_nets_visulization.ipynb

Neural Networks | Perceptron | Activation Functions | Deep Learning | Deep Dives

394 6

Published in Towards Data Science

791K Followers · Last published 3 hours ago

Follow

Your home for data science and AI. The world's leading publication for data science, data analytics, data engineering, machine learning, and artificial intelligence professionals.

Written by Reza Bagheri

1.6K Followers · 2 Following

Follow

Data Scientist and Researcher. LinkedIn: https://www.linkedin.com/in/reza-bagheri-71882a76/

Responses (6)

What are your thoughts?

Respond

poorvamudgal
1 day ago

Thanks for this tutorial, Reza. It is very useful to understand the NN through plots. I have experience with linear SVM and decision boundary (hyperplane) in a 3d plot. I am currently struggling with the non linear SVM plot. Do you have any experience to show non linear (radial) hyperplane in a 3dplot?

1 Reply

Pgeorges
2 days ago

Thanks for this tutorial, Reza. It is very useful. But could you please re-check Listing 9. As it is one cannot plot Figure 9. Thanks in advance if you have time to do/update this!

1 Reply

Paulo Frade
12 hours ago

Thank you for this great and detailed explanation of the mathematics behind artificial neural networks. I only believe that the founders of these AI algorithms should have chosen different terminology, as using the same terms as in the field of... more

Reply

See all responses

More from Reza Bagheri and Towards Data Science

In Towards Data Science by Reza Bagheri

A Visual Understanding of Decision Trees and Gradient...

A visual explanation of the math behind decision trees and gradient boosting

Jul 26, 2024 818 4

In Towards Data Scien... by Maria Mouschoutzi, P...

Water Cooler Small Talk: Benford's Law

A look into the strange first digit distribution of naturally occurring datasets

1d ago 278 3

In Towards Data Scien... by Alejandro Alvarez Pér...

Mastering the Poisson Distribution: Intuition and Foundations

Take a dive into the math and exemplifying use cases of the Poisson distribution

Jan 5 93 1

In Towards Data Science by Reza Bagheri

A Visual Understanding of the Softmax Function

The math and intuition behind the softmax function and its application in neural...

Nov 3, 2024 341 2

See all from Reza Bagheri

See all from Towards Data Science

Recommended from Medium

In Towards Data Science by Farzad Nobar

Going Beyond Bias-Variance Tradeoff Into Double Descent...

It's not how many times you get knocked down that count, it's how many times you ge...

3d ago 119 3

In Level Up Coding by Jacob Bennett

The 5 paid subscriptions I actually use in 2025 as a Staff Software...

Tools I use that are cheaper than Netflix

Jan 7 4.1K 96

Lists

Natural Language Processing
1888 stories · 1542 saves

Practical Guides to Machine Learning
10 stories · 2154 saves

data science and AI
40 stories · 316 saves

Stories to Help You Grow as a Software Developer
19 stories · 1560 saves

Austin Starks

You are an absolute moron for believing in the hype of "AI...

All of my articles are 100% free to read. Non-members can read for free by clicking my...

5d ago 1.3K 72

Andrew Zuo

Nvidia Is About To Collapse The Price Of AI Models

At CES Nvidia showed off a few interesting new things. The biggest of which is Jensen...

Jan 9 733 32

In Stackademic by Hassan Trabelsi

Linux Creator Reveals the Future of Programming with AI

Linus Torvalds, the creator of Linux and Git, two of the most influential software tools in...

Aug 31, 2024 1.3K 15

In Towards AI by Shenggang Li

Decoding Ponzi Schemes with Math and AI

Experiments in Action: Unveiling Hidden Trends in Stocks and Markets Using Mamba...

3d ago 376 2

See more recommendations