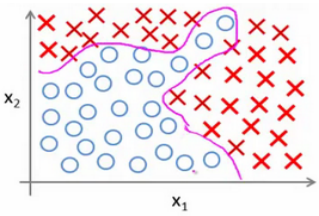
第 4 周

6、 神经网络 Neural Networks

6.1 非线性假设 Non-linear Hypotheses

我们之前学的，无论是线性回归还是逻辑回归都有这样一个缺点，即：当特征太多时，计算的负荷会非常大。

下面是一个例子：



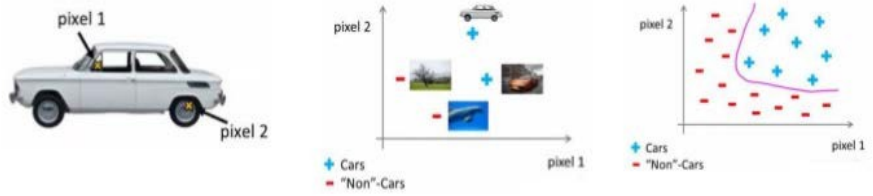
$$g(\theta_0+\theta_1x_1+\theta_2x_2+\theta_3x_1x_2+\theta_4x_1^2x_2+\theta_5x_1^3x_2+\theta_6x_1x_2^2+\dots)$$

当我们使用x_1，x_2 的多次项式进行预测时，我们可以应用的很好。

之前我们已经看到过，使用非线性的多项式项，能够帮助我们建立更好的分类模型。假设我们有非常多的特征，例如大于 100 个变量，我们希望用这 100 个特征来构建一个非线性的多项式模型，结果将是数量非常惊人的特征组合，即便我们只采用两两特征的组合$(x_1x_2+x_1x_3+x_1x_4+\dots+x_2x_3+x_2x_4+\dots+x_{99}x_{100})$，我们也会有接近 5000 个组合而成的特征。这对于一般的逻辑回归来说需要计算的特征太多了。

假设我们希望训练一个模型来识别视觉对象（例如识别一张图片上是否是一辆汽车），我们怎样才能这么做呢？一种方法是我们利用很多汽车的图片和很多非汽车的图片，然后利用这些图片上一个个像素的值（饱和度或亮度）来作为特征。

假如我们只选用灰度图片，每个像素则只有一个值（而非 RGB 值），我们可以选取图片上的两个不同位置上的两个像素，然后训练一个逻辑回归算法利用这两个像素的值来判断图片上是否是汽车：



假使我们采用的都是 50x50 像素的小图片，并且我们将所有的像素视为特征，则会有 2500 个特征，如果我们要进一步将两两特征组合构成一个多项式模型，则会有约$2500^2/2$个（接近 3 百万个）特征。普通的逻辑回归模型，不能有效地处理这么多的特征，这时候我们需要神经网络。

6.2 神经元和大脑 Neurons and the Brain

神经网络是一种很古老的算法，它最初产生的目的是制造能模拟大脑的机器。

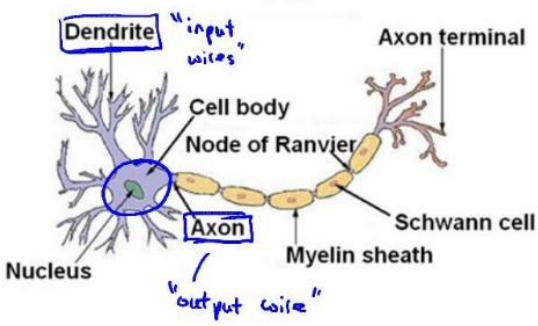
Neural networks is to mimic the computation of the brain.

Speech → images (computer vision) → text (Natural language processing)

神经网络逐渐兴起于二十世纪八九十年代，应用得非常广泛。但由于各种原因，在 90 年代的后期应用减少了。但是最近，神经网络又东山再起了。其中一个原因是：神经网络是计算量有些偏大的算法。然而大概由于近些年计算机的运行速度变快，才足以真正运行起大规模的神经网络。正是由于这个原因和其他一些我们后面会讨论到的技术因素，如今的神经网络对于许多应用来说是最先进的技术。当你想模拟大脑时，是指想制造出与人类大脑作用效果相同的机器。大脑可以学会去以看而不是听的方式处理图像，学会处理我们的触觉。

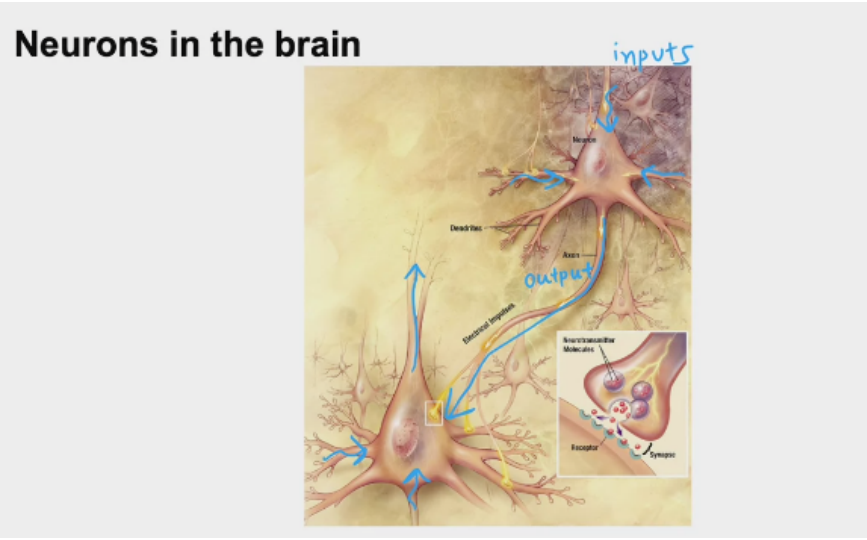
6.3 模型表示 Model Representation

为了构建神经网络模型，我们需要首先思考大脑中的神经网络是怎样的？每一个神经元都可以被认为是一个处理单元/神经核（**processing unit/Nucleus**），它含有许多输入/树突（**input/Dendrite**），并且有一个输出/轴突（**output/Axon**）。神经网络是大量神经元相互链接并通过电脉冲来交流的一个网络。



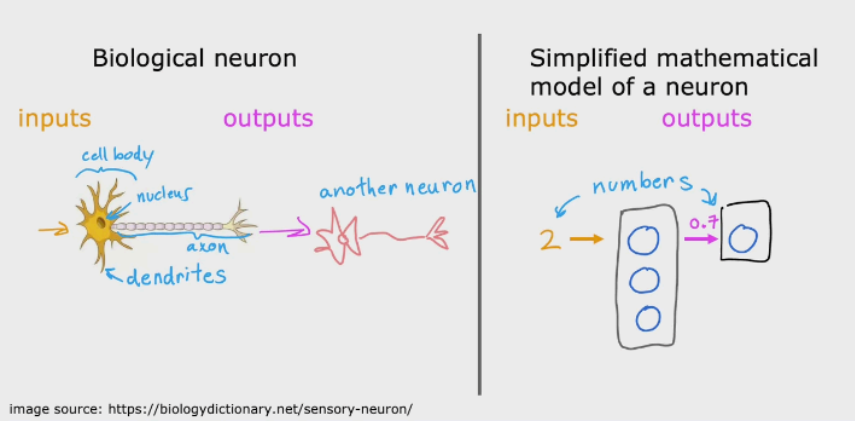
下面是一组神经元的示意图，神经元利用微弱的电流进行沟通。这些弱电流也称作动作电位，其实就是一些微弱的电流。所以如果神经元想要传递一个消息，它就会就通过它的轴突，发送一段微弱电流给其他神经元，这就是轴突。

这里是一条连接到输入神经，或者连接另一个神经元树突的神经，接下来这个神经元接收这条消息，做一些计算，它有可能会反过来将在轴突上的自己的消息传给其他神经元。这就是所有人类思考的模型：我们的神经元把自己的收到的消息进行计算，并向其他神经元传递消息。这也是我们的感觉和肌肉运转的原理。如果你想活动一块肌肉，就会触发一个神经元给你的肌肉发送脉冲，并引起你的肌肉收缩。如果一些感官：比如说眼睛想要给大脑传递一个消息，那么它就像这样发送电脉冲给大脑的。



To mimic

A circle denotes a neuron, any many circles (neuron) are favored for parallel computation.



Applications:

Demand Prediction

Activation: how much a neuron issending a high output to another neuron

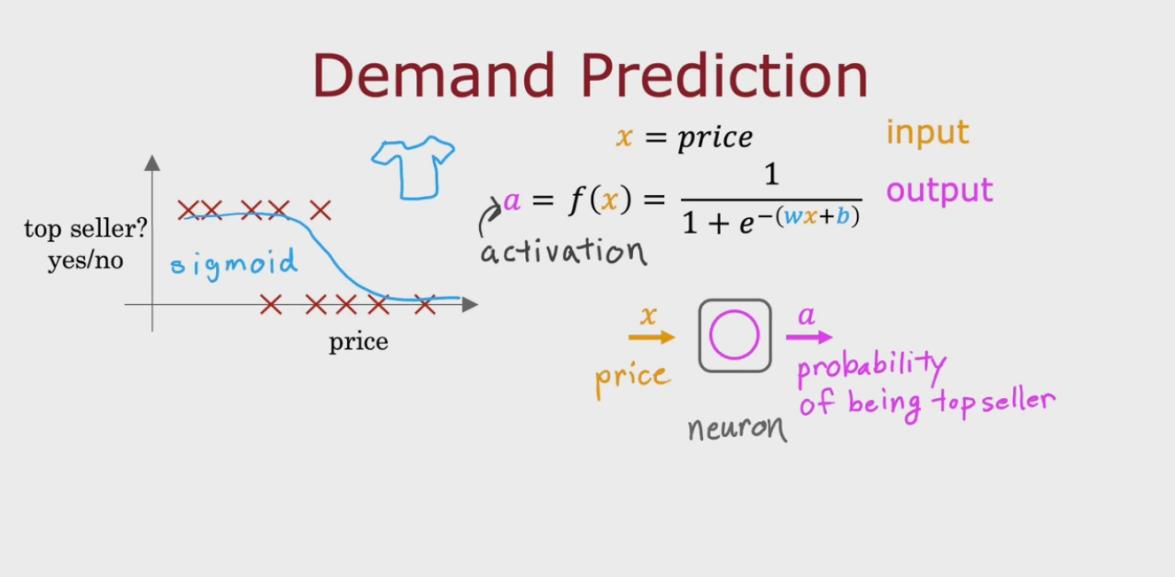
The probability of activation, however, depends on several factors/features that comprise of varied properties/activations and hence forming layered structure of influence chain connected by wires.

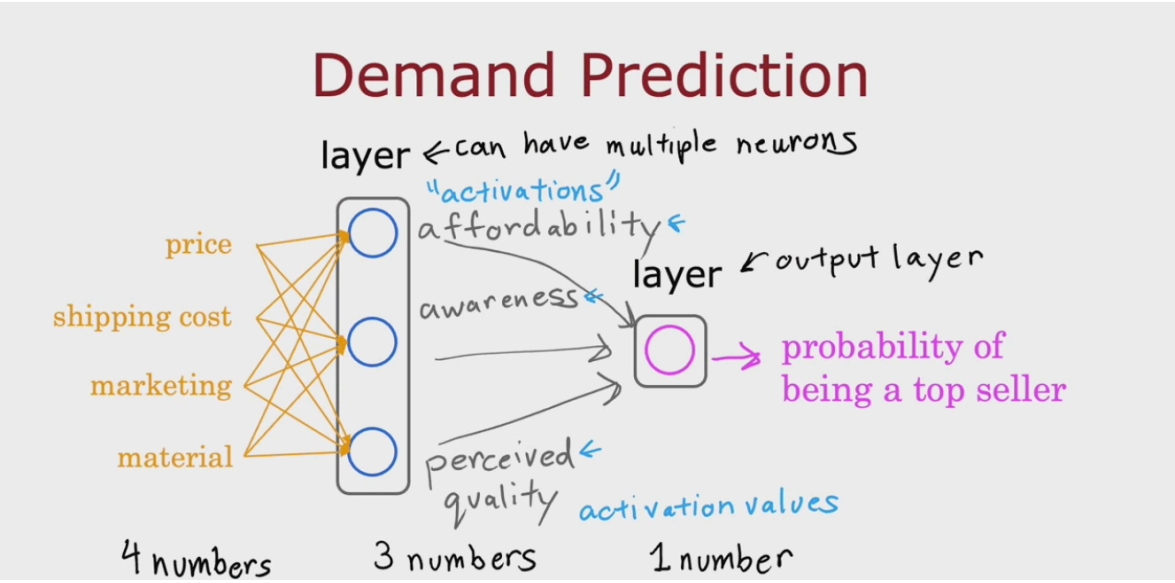
A layer is hence a analogy to grouping of neuron.

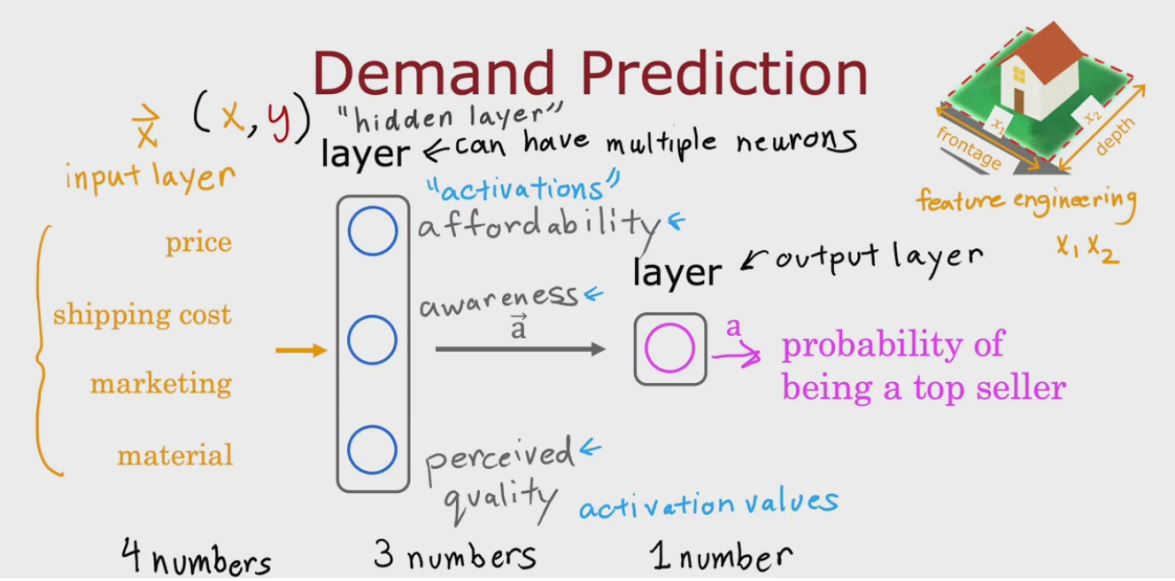
However notabily, each activation is a function of feature whose relationship between activation and feature is conventionally identified by human to decide which features should be the input parameters for the function (feature engineering), and then formulate the funtions. To reduce such manual work, we allow each neuron to access the feature of previous layer. To enable such access and relationship formulation, we represent each layer into a vector. For futhre logistic regression.

Nerual network architecture

The only work you will need to do, is to decide how many layers you want to have in your model


Demand Prediction
top seller?
yes/no
sigmoid
price
$x = price$
input
$a = f(x) = \frac{1}{1+e^{-(wx+b)}}$
output
activation
x
price
a
probability
of being top seller
neuron


Demand Prediction
layer ← can have multiple neurons
"activations"
affordability ←
awareness ←
perceived
quality
activation values
price
shipping cost
marketing
material
layer ← output layer
probability of
being a top seller
4 numbers
3 numbers
1 number

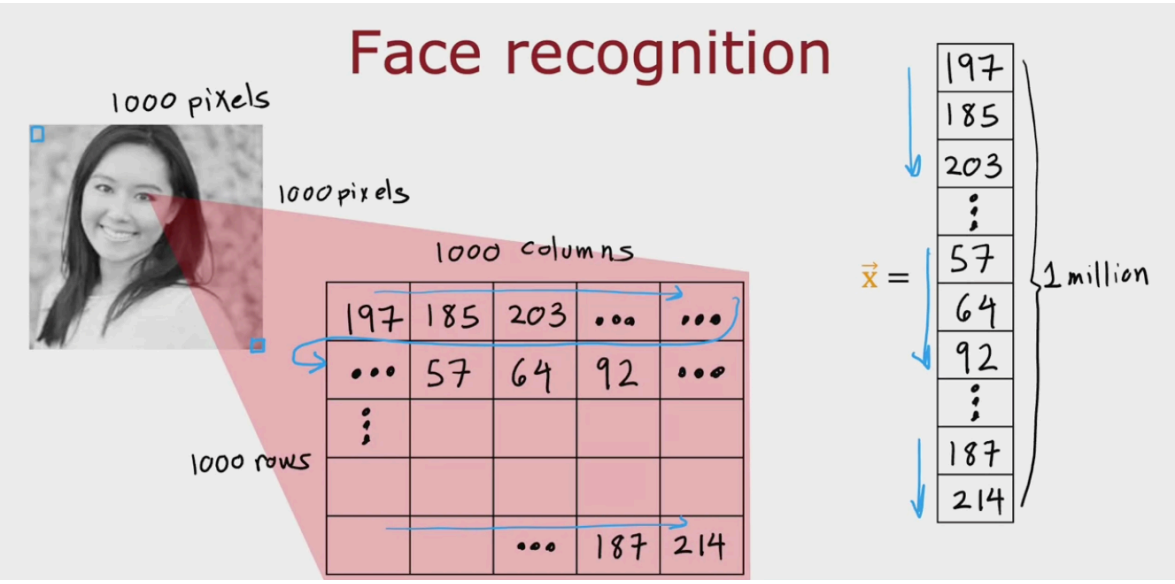


Recognizing example

Example one: face recognition

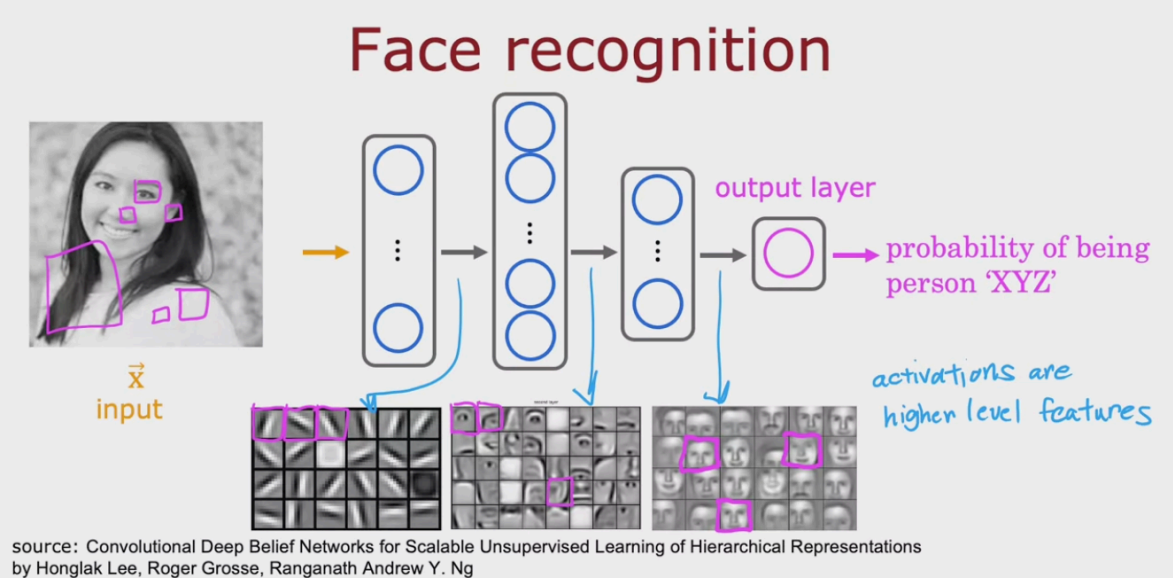
Access the image as a matrix composed of vectors of pixels intensity values.

Unroll this matrix of size NxN into a vector N.N → then you will find each neuron is looking for different pattern for relationship/feature identification

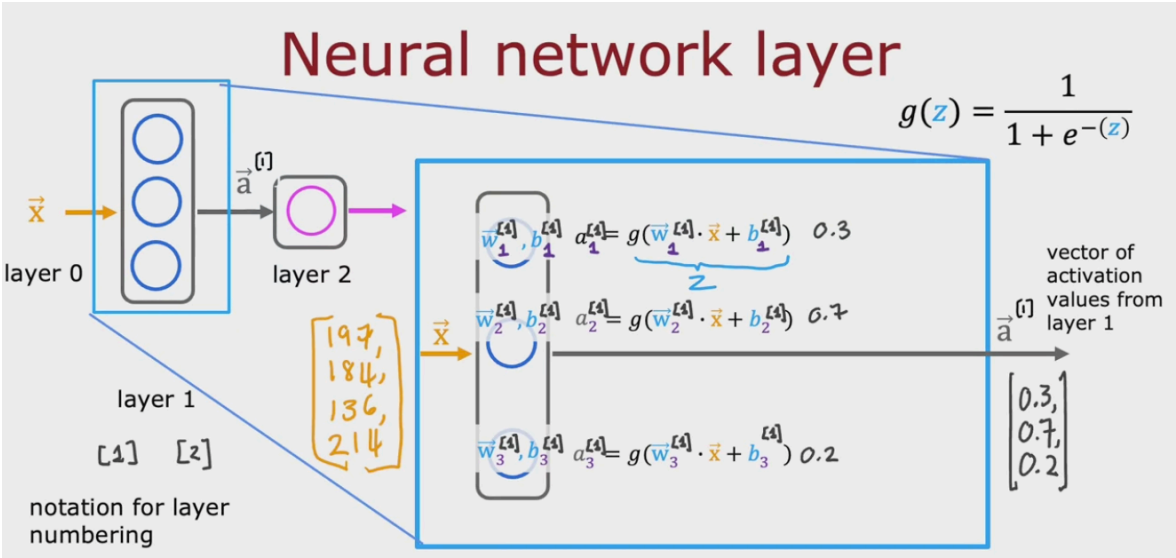


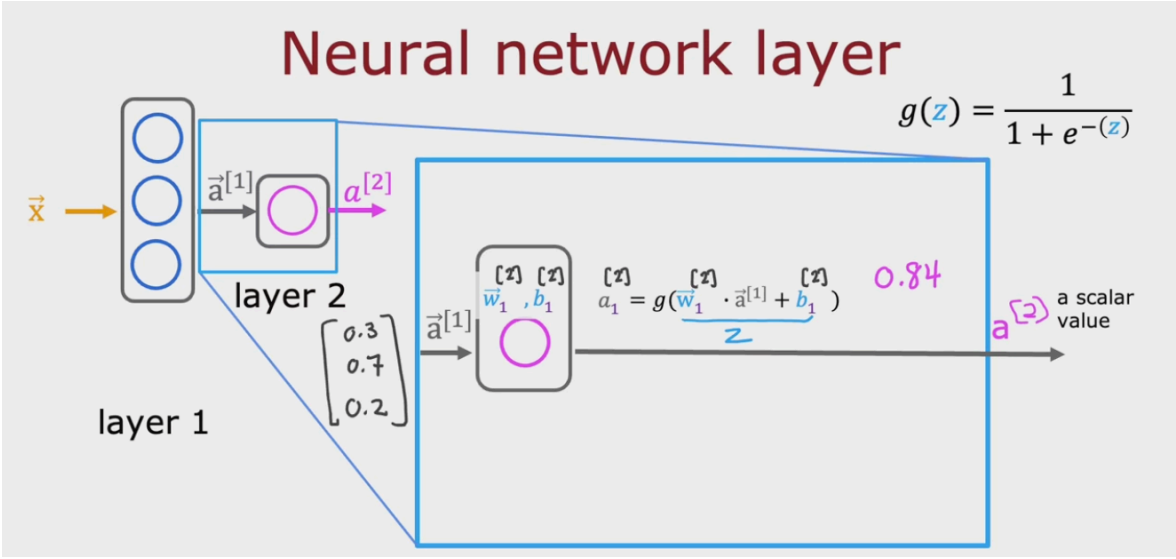
In this example, in the first layer, a neuron is looking for a very short line/edge per orientation, the next layer, this neuron learns to group together to learn and detect a segment or parts of surface, then the mext layer, may be learning to aggregation different parts of faces and to determine the

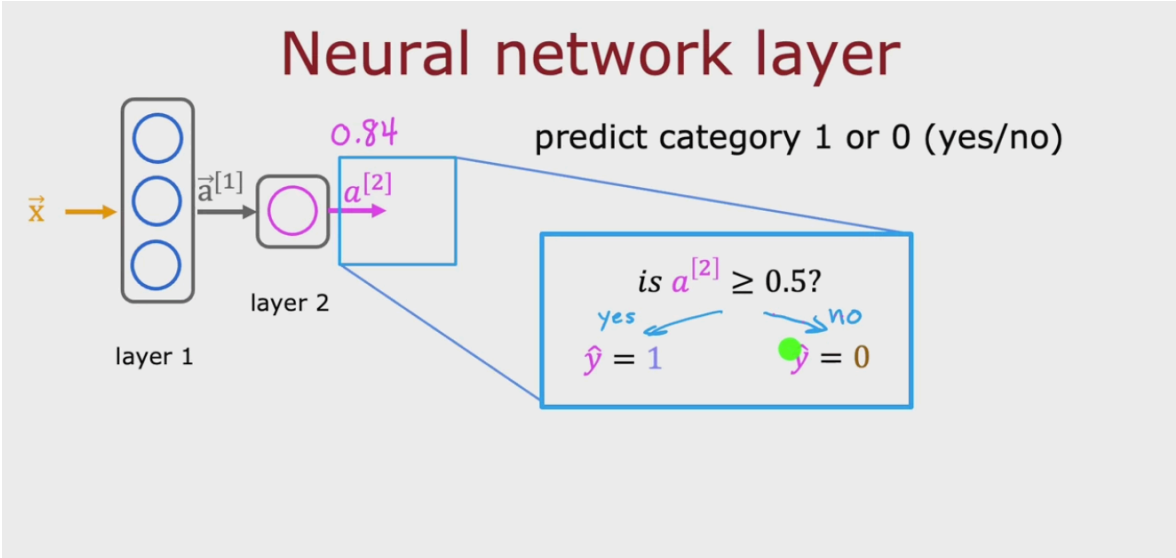
identity of the object. We can see that each layer operates on a lager group that is analogy to larger window.

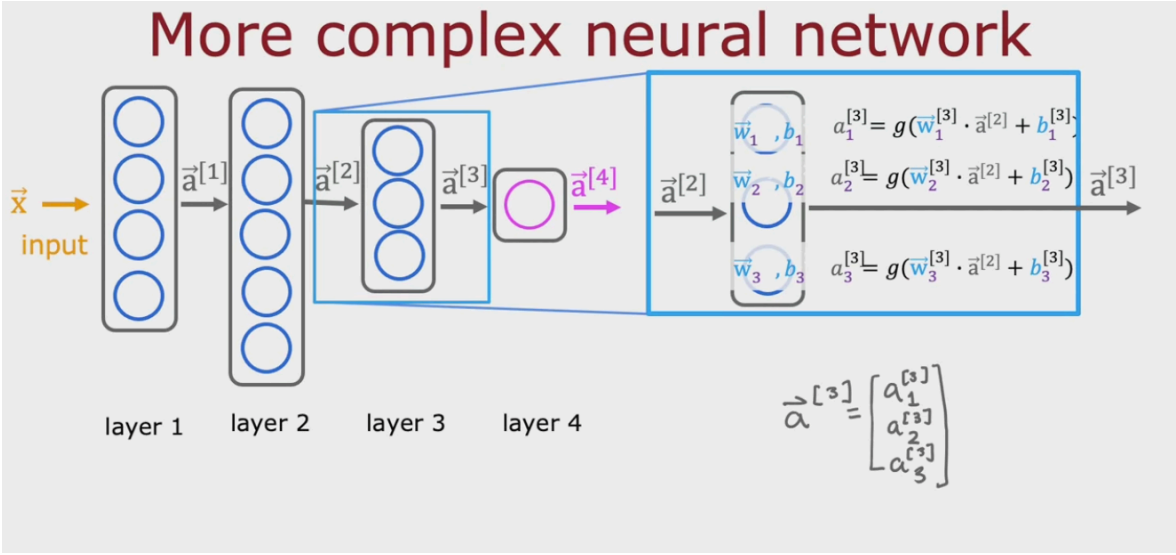


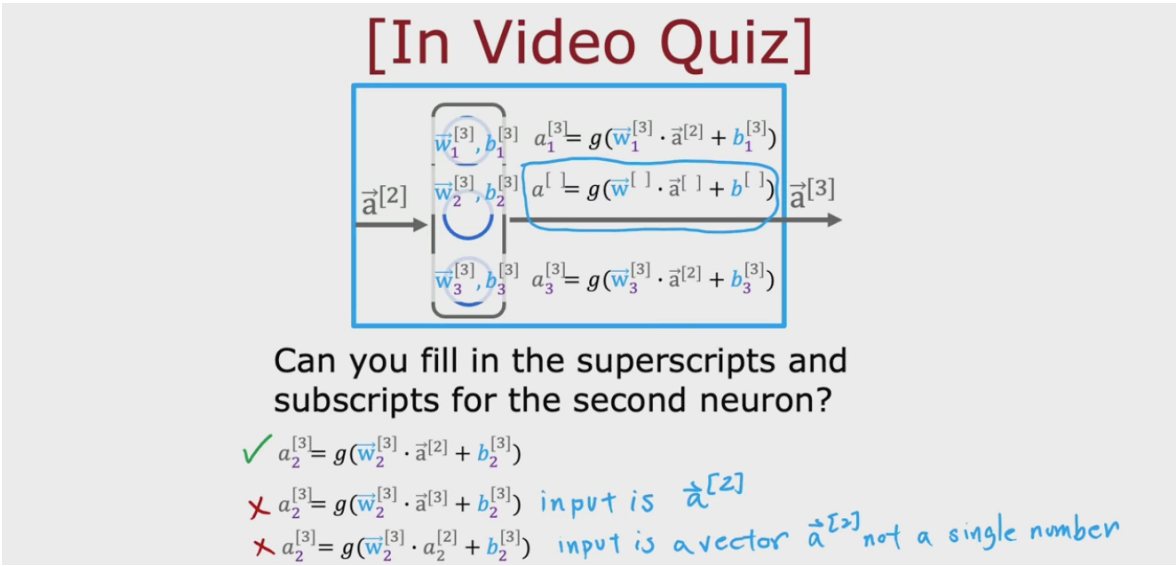
神经网络模型建立在很多神经元之上，每一个神经元又是一个个学习模型。这些神经元（也叫激活单元，**activation unit**）采纳一些特征作为输出，并且根据本身的模型提供一个输出。下图是一个以逻辑回归模型作为自身学习模型的神经元示例，在神经网络中，参数又可被成为权重（**weight**）。

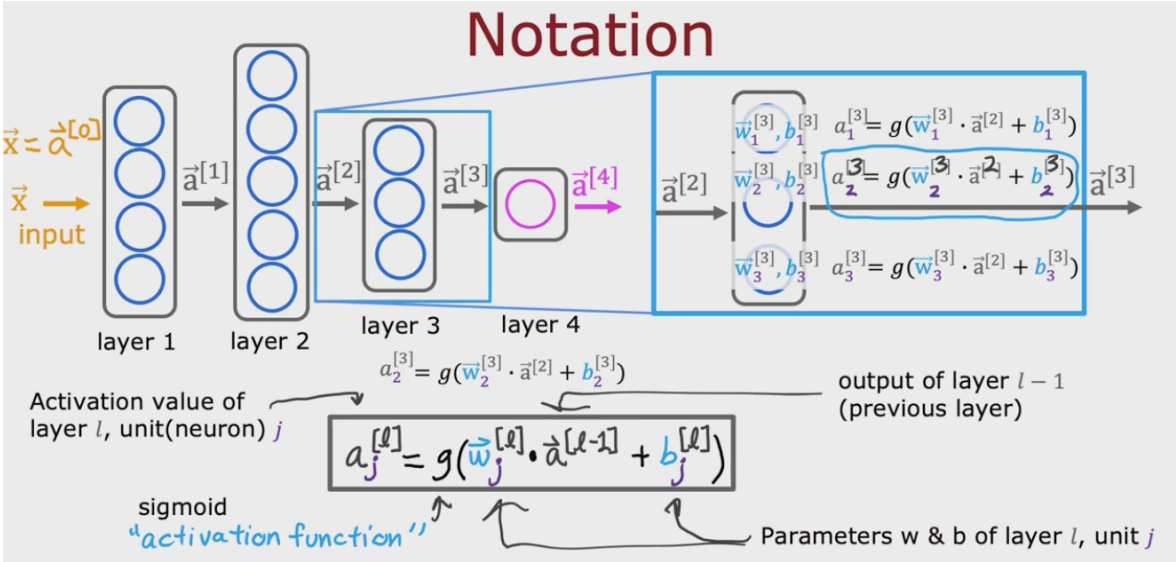


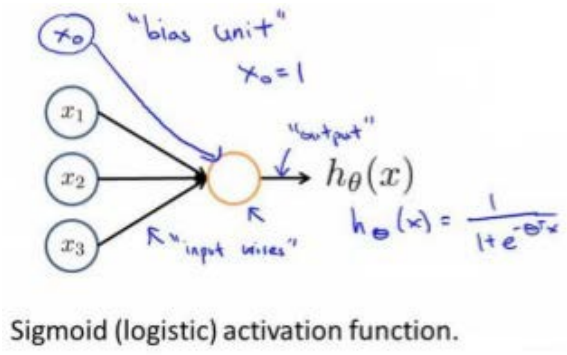

Neural network layer
$g(z) = \frac{1}{1+e^{-(z)}}$
$\vec{x}$
$\vec{a}^{[1]}$
$a^{[2]}$
layer 2
layer 1
0.3
0.7
0.2
$\vec{w}_1^{[1]}, b_1^{[1]}$
$a_1^{[1]} = g(\vec{w}_1^{[1]} \cdot \vec{a}^{[1]} + b_1^{[1]})$
z
0.84
$a^{[2]}$
a scalar value


Neural network layer
$\vec{x}$
$\vec{a}^{[1]}$
0.84
$a^{[2]}$
layer 2
layer 1
predict category 1 or 0 (yes/no)
is $a^{[2]} \geq 0.5$?
yes
no
$\hat{y} = 1$
$\hat{y} = 0$

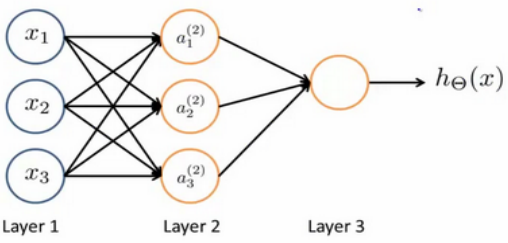

More complex neural network
x
input
a[1]
a[2]
a[3]
a[4]
layer 1
layer 2
layer 3
layer 4
a[2]
a[3]
$a_1^{[3]} = g(\vec{w}_1^{[3]} \cdot \vec{a}^{[2]} + b_1^{[3]})$
$a_2^{[3]} = g(\vec{w}_2^{[3]} \cdot \vec{a}^{[2]} + b_2^{[3]})$
$a_3^{[3]} = g(\vec{w}_3^{[3]} \cdot \vec{a}^{[2]} + b_3^{[3]})$
$\vec{a}^{[3]} = \begin{bmatrix} a_1^{[3]} \\ a_2^{[3]} \\ a_3^{[3]} \end{bmatrix}$


[In Video Quiz]
a[2]
a[3]
$a_1^{[3]} = g(\vec{w}_1^{[3]} \cdot \vec{a}^{[2]} + b_1^{[3]})$
$a^{[\ ]} = g(\vec{w}^{[\ ]} \cdot \vec{a}^{[\ ]} + b^{[\ ]})$
$a_3^{[3]} = g(\vec{w}_3^{[3]} \cdot \vec{a}^{[2]} + b_3^{[3]})$
Can you fill in the superscripts and subscripts for the second neuron?
✓ $a_2^{[3]} = g(\vec{w}_2^{[3]} \cdot \vec{a}^{[2]} + b_2^{[3]})$
✗ $a_2^{[3]} = g(\vec{w}_2^{[3]} \cdot \vec{a}^{[3]} + b_2^{[3]})$ input is $\vec{a}^{[2]}$
✗ $a_2^{[3]} = g(\vec{w}_2^{[3]} \cdot a_2^{[2]} + b_2^{[3]})$ input is a vector $\vec{a}^{[2]}$ not a single number





我们设计出了类似于神经元的神经网络，效果如下：

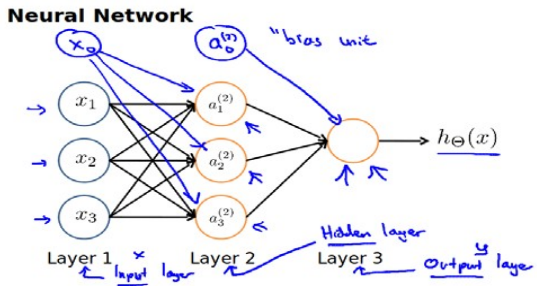


其中x_1，x_2，x_3是输入单元（**input units**），我们将原始数据输入给它们。

a_1，a_2，a_3是中间单元，它们负责将数据进行处理，然后呈递到下一层。

最后是输出单元，它负责计算$h_\theta(x)$。

神经网络模型是许多逻辑单元按照不同层级组织起来的网络，每一层的输出变量都是下一层的输入变量。下图为一个 3 层的神经网络，第一层成为输入层（**Input Layer**），最后一层称为输出层（**Output Layer**），中间一层成为隐藏层（**Hidden Layers**）。我们为每一层都增加一个偏差单位（**bias unit**）：



下面引入一些标记法来帮助描述模型：

$a_i^{(j)}$ 代表第j 层的第 i 个激活单元。$\theta^{(j)}$代表从第 j 层映射到第$j+1$ 层时的权重的矩阵，例如$\theta^{(1)}$代表从第一层映射到第二层的权重的矩阵。其尺寸为：以第 $j+1$层的激活单元数量为行数，以第 j 层的激活单元数加一为列数的矩阵。例如：上图所示的神经网络中$\theta^{(1)}$的尺寸为 3*4。

对于上图所示的模型，激活单元和输出分别表达为：

$$a_1^{(2)} = g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3)$$

$$a_2^{(2)} = g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3)$$

$$a_3^{(2)} = g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3)$$

$$h_\theta(x) = g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)})$$

上面进行的讨论中只是将特征矩阵中的一行（一个训练实例）喂给了神经网络，我们需要将整个训练集都喂给我们的神经网络算法来学习模型。

我们可以知道：每一个a都是由上一层所有的x和每一个x所对应的决定的。

（我们把这样从左到右的算法称为前向传播算法(**FORWARD PROPAGATION**)）

把x，θ，a 分别用矩阵表示，我们可以得到$\theta \cdot X = a$ ：

$$X = \begin{matrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{matrix}, \quad \theta = \begin{matrix} \theta_{10} & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \theta_{33} \end{matrix}, \quad a = \begin{matrix} a_1 \\ a_2 \\ a_3 \end{matrix}$$

6.4 模型表示 Inference: making predictions (forward propogation)

（**FORWARD PROPAGATION**）相对于使用循环来编码，利用向量化的方法会使得计算更为简便。以上面的神经网络为例，试着计算第二层的值：

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad z^{(2)} = \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{bmatrix}$$

$$z^{(2)} = \Theta^{(1)} x$$
$$a^{(2)} = g(z^{(2)})$$

$$g\left(\begin{bmatrix} \theta_{10}^{(1)} & \theta_{11}^{(1)} & \theta_{12}^{(1)} & \theta_{13}^{(1)} \\ \theta_{20}^{(1)} & \theta_{21}^{(1)} & \theta_{22}^{(1)} & \theta_{23}^{(1)} \\ \theta_{30}^{(1)} & \theta_{31}^{(1)} & \theta_{32}^{(1)} & \theta_{33}^{(1)} \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = g\left(\begin{bmatrix} \theta_{10}^{(1)} x_0 + \theta_{11}^{(1)} x_1 + \theta_{12}^{(1)} x_2 + \theta_{13}^{(1)} x_3 \\ \theta_{20}^{(1)} x_0 + \theta_{21}^{(1)} x_1 + \theta_{22}^{(1)} x_2 + \theta_{23}^{(1)} x_3 \\ \theta_{30}^{(1)} x_0 + \theta_{31}^{(1)} x_1 + \theta_{32}^{(1)} x_2 + \theta_{33}^{(1)} x_3 \end{bmatrix}\right) = \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \end{bmatrix}$$

我们令 $z^{(2)} = \theta^{(1)} x$，则 $a^{(2)} = g(z^{(2)})$ ，计算后添加 $a_0^{(2)} = 1$。 计算输出的值为：

$$g\left(\begin{bmatrix} \theta_{10}^{(2)} & \theta_{11}^{(2)} & \theta_{12}^{(2)} & \theta_{13}^{(2)} \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \end{bmatrix}\right) = g\left(\theta_{10}^{(2)} a_0^{(2)} + \theta_{11}^{(2)} a_1^{(2)} + \theta_{12}^{(2)} a_2^{(2)} + \theta_{13}^{(2)} a_3^{(2)}\right) = h_\theta(x)$$

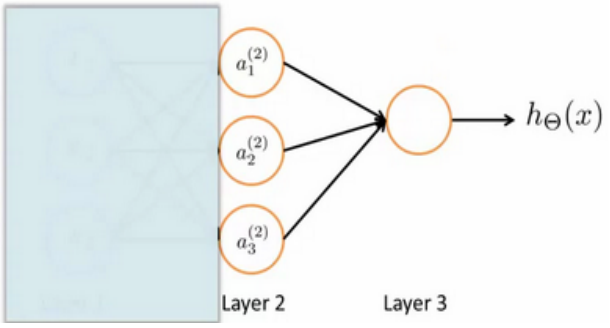
我们令 $z^{(3)} = \theta^{(2)} a^{(2)}$，则 $h_\theta(x) = a^{(3)} = g(z^{(3)})$。

这只是针对训练集中一个训练实例所进行的计算。如果我们要对整个训练集进行计算，我们需要将训练集特征矩阵进行转置，使得同一个实例的特征都在同一列里。即：

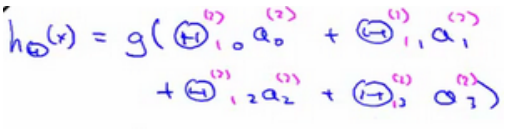
$$z^{(2)} = \Theta^{(1)} \times X^T$$

$$a^{(2)} = g(z^{(2)})$$

为了更好了了解 **Neuron Networks** 的工作原理，我们先把左半部分遮住：



右半部分其实就是以a_0, a_1, a_2, a_3，按照 **Logistic Regression** 的方式输出$h_\theta(x)$：



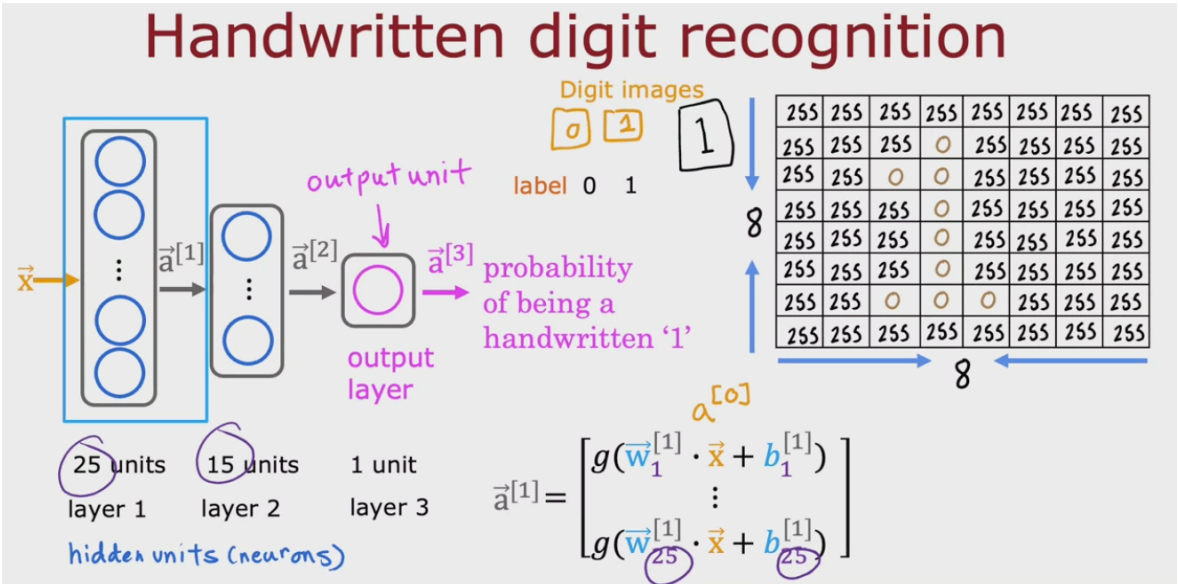
其实神经网络就像是 **logistic regression**，只不过我们把 **logistic regression** 中的输入向量$[x_1 \sim x_3]$ 变成了中间层的$\left[a_1^{(2)} \sim a_3^{(2)}\right]$，即：

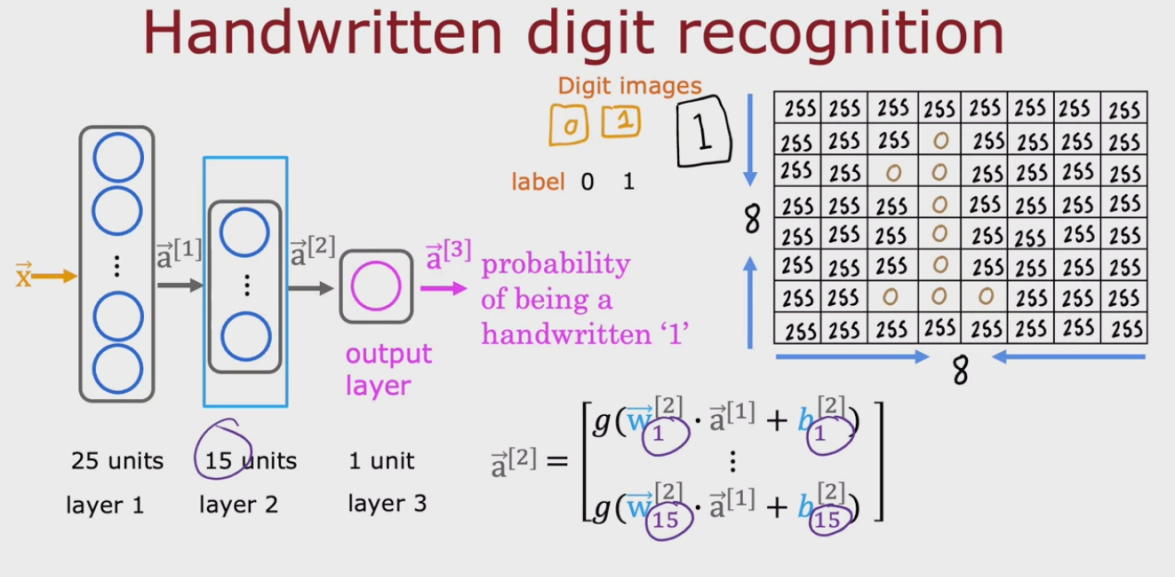
$$h_\theta(x) = g\left(\theta_0^{(2)}a_0^{(2)} + \theta_1^{(2)}a_1^{(2)} + \theta_2^{(2)}a_2^{(2)} + \theta_3^{(2)}a_3^{(2)}\right)$$

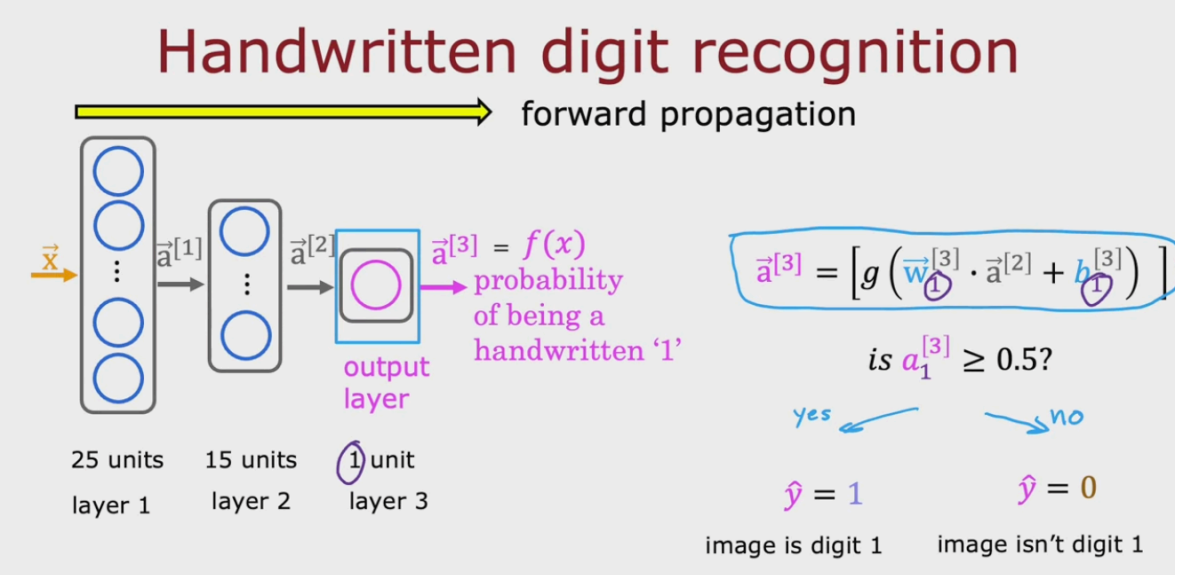
我们可以把a_0, a_1, a_2, a_3看成更为高级的特征值，也就是x_0, x_1, x_2, x_3的进化体，并且它们是由 x与决定的，因为是梯度下降的，所以a是变化的，并且变得越来越厉害，所以这些更高级的特征值远比仅仅将 x次方厉害，也能更好的预测新数据。

这就是神经网络相比于逻辑回归和线性回归的优势。

Example: 0 – 255 to represent the gray levels of an image/matrix. 0 denotes black pixel and 255 denotes bright pixel

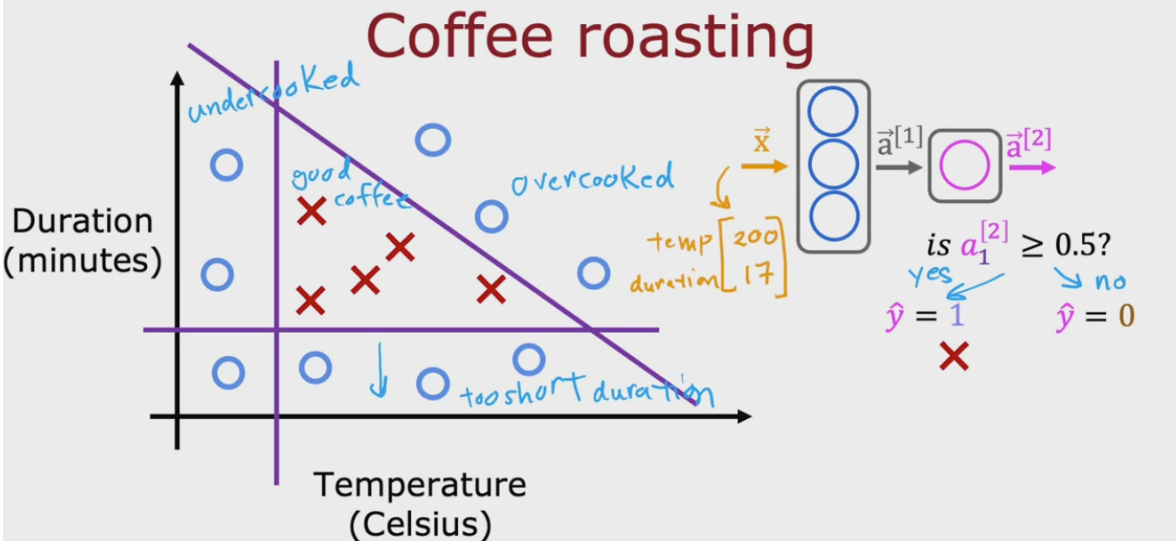






Implementation of inferencing in tensorflow

Example: coffee roasting

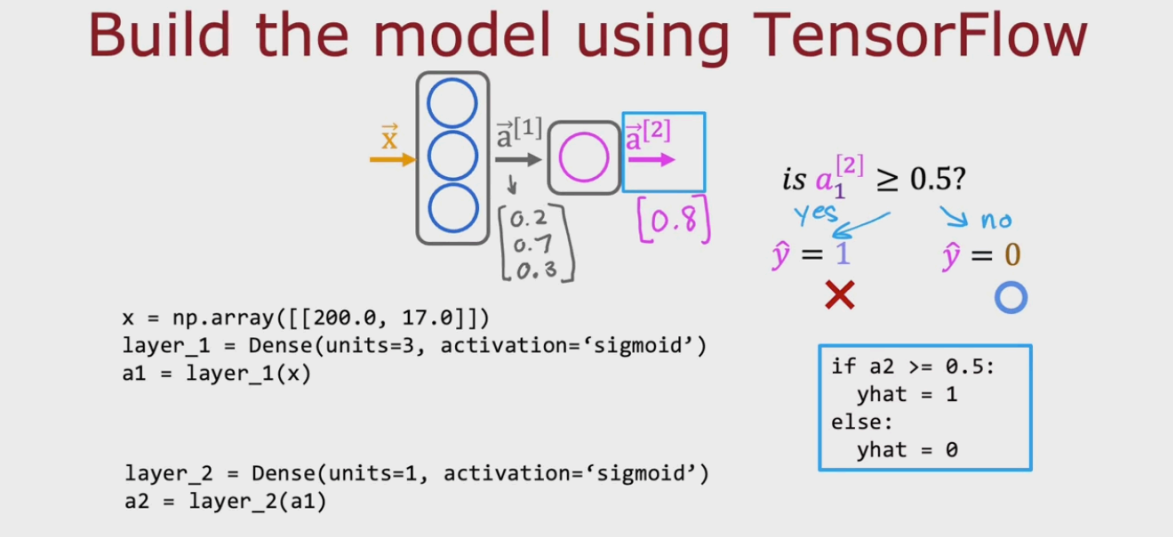


Layer one as the first hidden layer

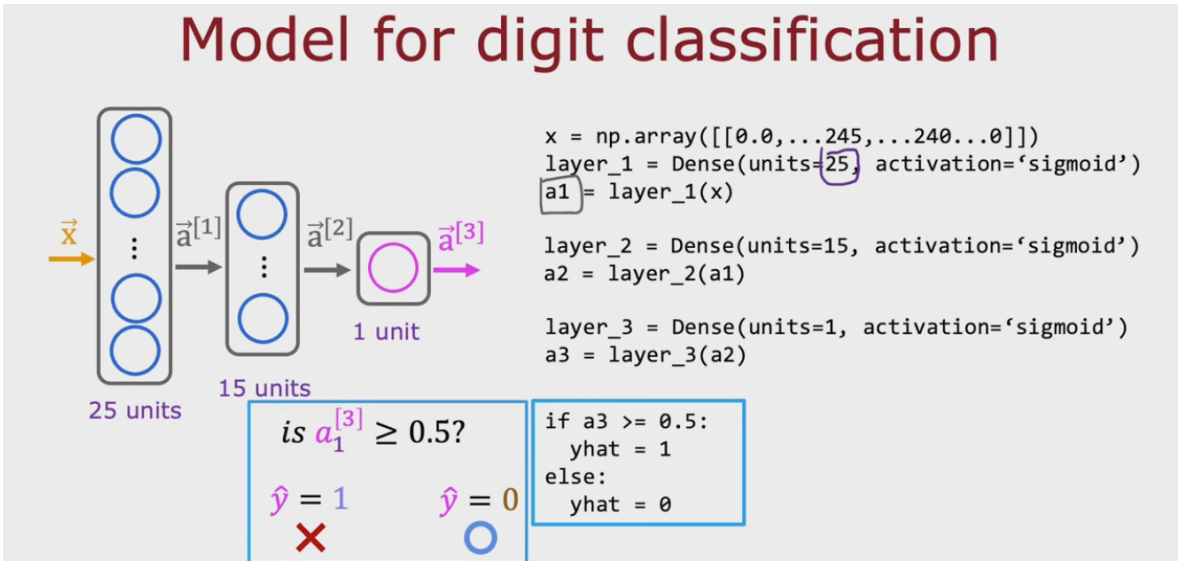
Dense is another name for the layers in neural networks:

Layer_1 = Dense(units = 3, activation='sigmoid') means that this layer has three neurons and uses sigmoid function to predict.

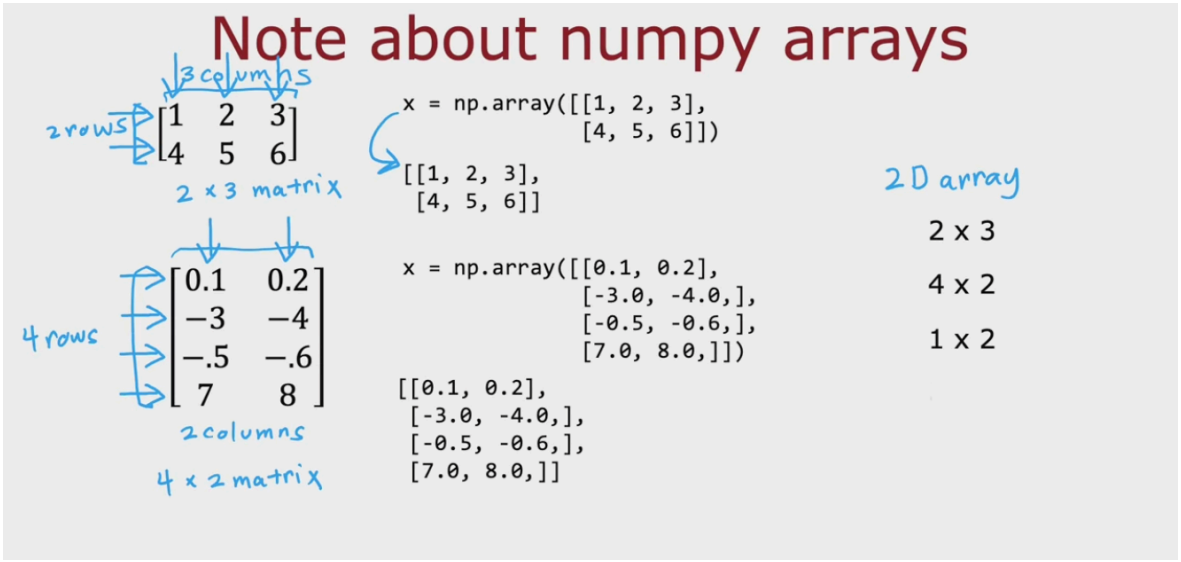
A1 = Layer_1(x) means that we apply this layer one on x input signal

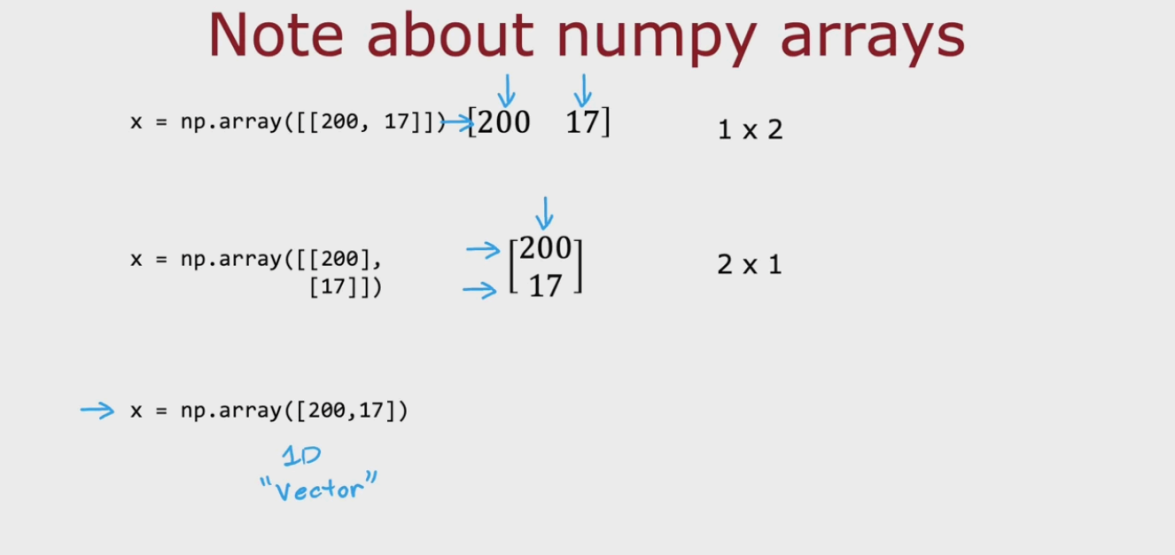


Example: image classification (model for digit classification)

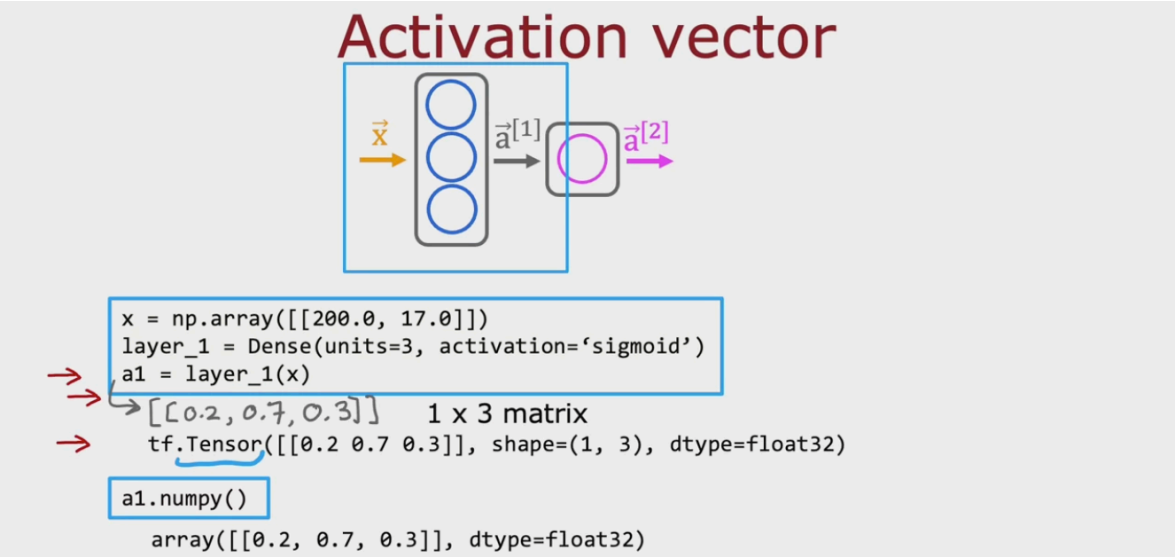


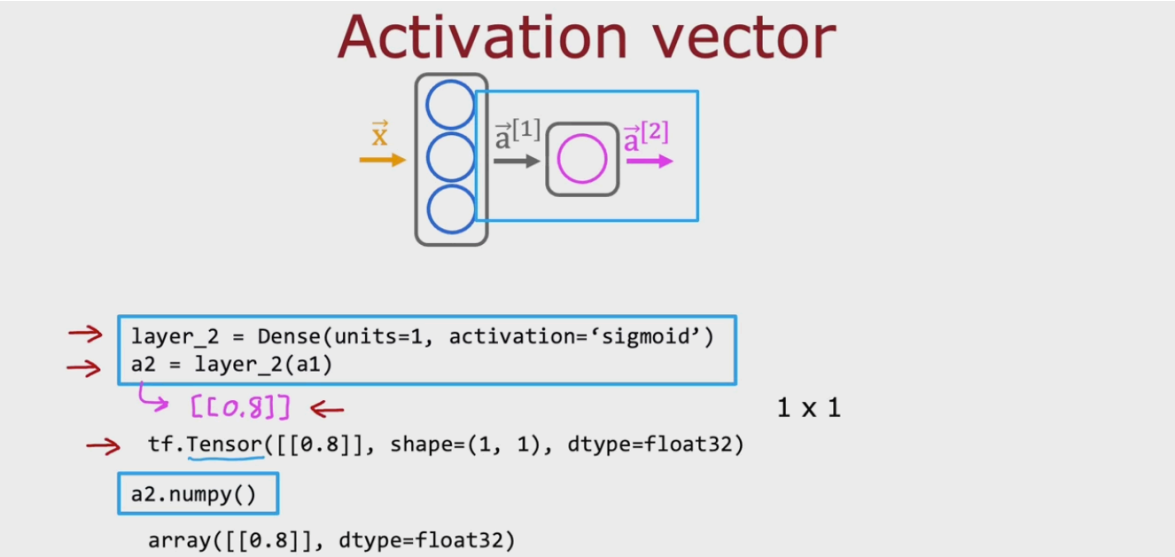
Data in Tensorflow and Conventions





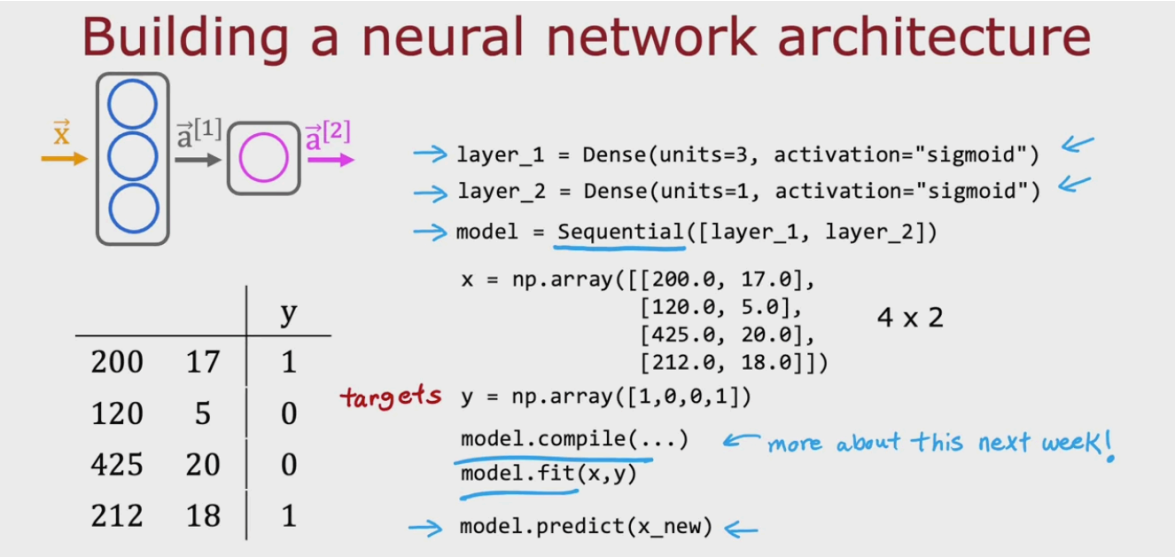
Tensor in tensorflow is to represent a matrix same as np.array and can be convert back to the form of numpy matrix/array by specifying .numpy



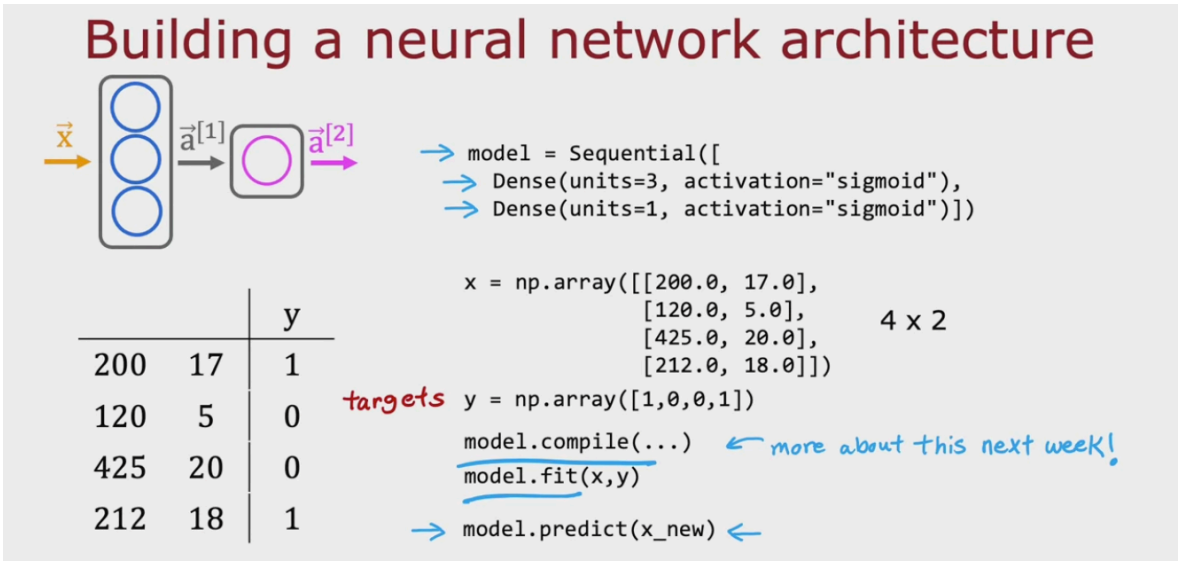


Building a neural network architecture

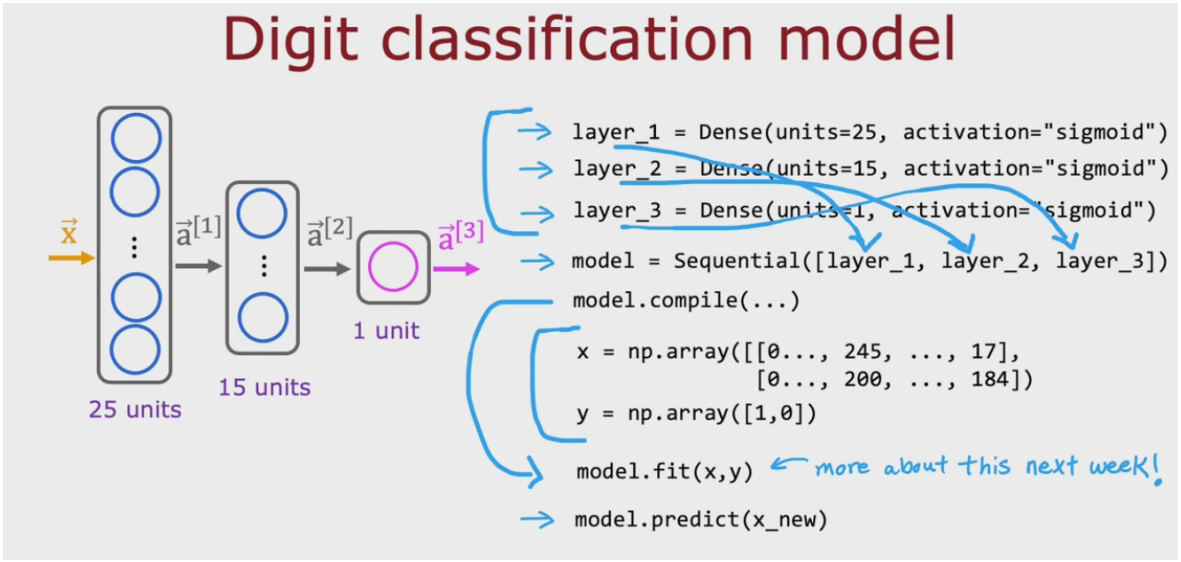
Streaming by using sequential framework



Is equivalent to



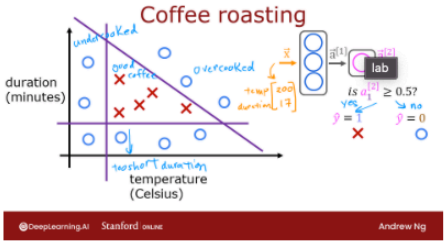
Example: digit calssificatoin model



6.4.1 Forward propagation from scratch

Optional Lab - Simple Neural Network

In this lab we will build a small neural network using Tensorflow.



In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('./deeplearning.mplstyle')
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from lab_utils_common import dlc
from lab_coffee_utils import load_coffee_data, plt_roast, plt_prob, plt_layer, plt_network, plt_output_unit
import logging
logging.getLogger("tensorflow").setLevel(logging.ERROR)
tf.autograph.set_verbosity(0)
```

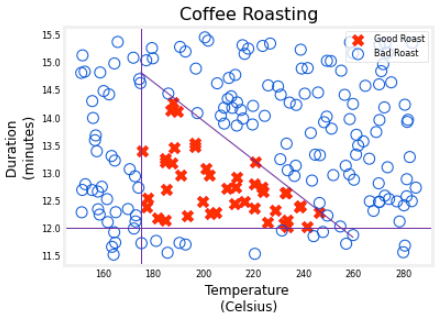
DataSet

In [2]:
```
X,Y = load_coffee_data();
print(X.shape, Y.shape)
```

```
(200, 2) (200, 1)
```

Let's plot the coffee roasting data below. The two features are Temperature in Celsius and Duration in minutes. Coffee Roasting at Home suggests that the duration is best kept between 12 and 15 minutes while the temp should be between 175 and 260 degrees Celsius. Of course, as temperature rises, the duration should shrink.

In [3]:
```
plt_roast(X,Y)
```



Normalize Data

Fitting the weights to the data (back-propagation, covered in next week's lectures) will proceed more quickly if the data is normalized. This is the same procedure you used in Course 1 where features in the data are each normalized to have a similar range. The procedure below uses a Keras normalization layer. It has the following steps:

- create a "Normalization Layer". Note, as applied here, this is not a layer in your model.
- 'adapt' the data. This learns the mean and variance of the data set and saves the values internally.
- normalize the data.
  It is important to apply normalization to any future data that utilizes the learned model.

In [ ]:
```
print(f"Temperature Max, Min pre normalization: {np.max(X[:,0]):0.2f}, {np.min(X[:,0]):0.2f}")
print(f"Duration    Max, Min pre normalization: {np.max(X[:,1]):0.2f}, {np.min(X[:,1]):0.2f}")
norm_l = tf.keras.layers.Normalization(axis=-1)
norm_l.adapt(X)  # learns mean, variance
Xn = norm_l(X)
print(f"Temperature Max, Min post normalization: {np.max(Xn[:,0]):0.2f}, {np.min(Xn[:,0]):0.2f}")
print(f"Duration    Max, Min post normalization: {np.max(Xn[:,1]):0.2f}, {np.min(Xn[:,1]):0.2f}")
```

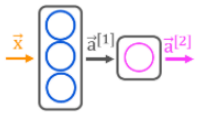
Tile/copy our data to increase the training set size and reduce the number of training epochs.

In [ ]:
```
Xt = np.tile(Xn,(1000,1))
Yt= np.tile(Y,(1000,1))
print(Xt.shape, Yt.shape)
```

lab

Tensorflow Model

Model



Let's build the "Coffee Roasting Network" described in lecture. There are two layers with sigmoid activations as shown below:

In [ ]:
```
tf.random.set_seed(1234)  # applied to achieve consistent results
model = Sequential(
    [
        tf.keras.Input(shape=(2,)),
        Dense(3, activation='sigmoid', name = 'layer1'),
        Dense(1, activation='sigmoid', name = 'layer2')
     ]
)
```

> **Note 1:** The `tf.keras.Input(shape=(2,))` specifies the expected shape of the input. This allows Tensorflow to size the weights and bias parameters at this point. This is useful when exploring Tensorflow models. This statement can be omitted in practice and Tensorflow will size the network parameters when the input data is specified in the `model.fit` statement.
> **Note 2:** Including the sigmoid activation in the final layer is not considered best practice. It would instead be accounted for in the loss which improves numerical stability. This will be described in more detail in a later lab.

The `model.summary()` provides a description of the network:

In [7]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 layer1 (Dense)              (None, 3)                 9

 layer2 (Dense)              (None, 1)                 4

=================================================================
Total params: 13
Trainable params: 13
Non-trainable params: 0
_________________________________________________________________
```

The parameter counts shown in the summary correspond to the number of elements in the weight and bias arrays as shown below.

In [8]:
```
L1_num_params = 2 * 3 + 3   # W1 parameters  + b1 parameters
L2_num_params = 3 * 1 + 1   # W2 parameters  + b2 parameters
print("L1 params = ", L1_num_params, ", L2 params = ", L2_num_params  )
```

```
L1 params =  9 , L2 params =  4
```

Let's examine the weights and biases Tensorflow has instantiated. The weights W should be of size (number of features in input, number of units in the layer) while the bias b size should match the number of units in the layer:

- In the first layer with 3 units, we expect W to have a size of (2,3) and b should have 3 elements.
- In the second layer with 1 unit, we expect W to have a size of (3,1) and b should have 1 element.

In [9]:
```
W1, b1 = model.get_layer("layer1").get_weights()
W2, b2 = model.get_layer("layer2").get_weights()
print(f"W1{W1.shape}:\n", W1, f"\nb1{b1.shape}:", b1)
print(f"W2{W2.shape}:\n", W2, f"\nb2{b2.shape}:", b2)
```

```
W1(2, 3):
 [[ 0.08 -0.3   0.18]
 [-0.56 -0.15  0.89]]
b1(3,): [0. 0. 0.]
W2(3, 1):
 [[-0.43]
 [-0.88]
 [ 0.36]]
b2(1,): [0.]
```

The following statements will be described in detail in Week2. For now:

- The `model.compile` statement defines a loss function and specifies a compile optimization.
- The `model.fit` statement runs gradient descent and fits the weights to the data.

In [10]:
```
model.compile(
    loss = tf.keras.losses.BinaryCrossentropy(),
    optimizer = tf.keras.optimizers.Adam(learning_rate=0.01),
)

model.fit(
    Xt,Yt,
    epochs=10,
)
```

```
Epoch 1/10
6250/6250 [==============================] - 5s 783us/step - loss: 0.1782
Epoch 2/10
6250/6250 [==============================] - 5s 818us/step - loss: 0.1165
Epoch 3/10
6250/6250 [==============================] - 5s 794us/step - loss: 0.0426
Epoch 4/10
6250/6250 [==============================] - 5s 786us/step - loss: 0.0160
Epoch 5/10
6250/6250 [==============================] - 5s 797us/step - loss: 0.0104
Epoch 6/10
6250/6250 [==============================] - 5s 781us/step - loss: 0.0073
Epoch 7/10
6250/6250 [==============================] - 5s 788us/step - loss: 0.0052
Epoch 8/10
6250/6250 [==============================] - 5s 799us/step - loss: 0.0037
Epoch 9/10
6250/6250 [==============================] - 5s 786us/step - loss: 0.0027
Epoch 10/10
```

Epochs and batches

In the `compile` statement above, the number of `epochs` was set to 10. This specifies that the entire data set should be applied during training 10 times. During training, you see output describing the progress of training that looks like this:

```
Epoch 1/10
6250/6250 [==============================] - 6s 910us/step - loss: 0.1782
```

lab

The first line, `Epoch 1/10` , describes which epoch the model is currently running. For efficiency, the training data set is broken into 'batches'. The default size of a batch in Tensorflow is 32. There are 200000 examples in our expanded data set or 6250 batches. The notation on the 2nd line `6250/6250 [====` is describing which batch has been executed.

Updated Weights

After fitting, the weights have been updated:

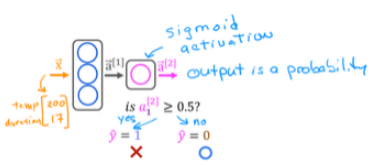
In [11]:
```
W1, b1 = model.get_layer("layer1").get_weights()
W2, b2 = model.get_layer("layer2").get_weights()
print("W1:\n", W1, "\nb1:", b1)
print("W2:\n", W2, "\nb2:", b2)
```

```
W1:
 [[ -0.13   14.3   -11.1 ]
 [ -8.92   11.85   -0.25]]
b1: [-11.16    1.76 -12.1 ]
W2:
 [[-45.71]
 [-42.95]
 [-50.19]]
b2: [26.14]
```

Next, we will load some saved weights from a previous training run. This is so that this notebook remains robust to changes in Tensorflow over time. Different training runs can produce somewhat different results and the discussion below applies to a particular solution. Feel free to re-run the notebook with this cell commented out to see the difference.

In [12]:
```
W1 = np.array([
    [-8.94,  0.29, 12.89],
    [-0.17, -7.34, 10.79]] )
b1 = np.array([-9.87, -9.28,  1.01])
W2 = np.array([
    [-31.38],
    [-27.86],
    [-32.79]])
b2 = np.array([15.54])
model.get_layer("layer1").set_weights([W1,b1])
model.get_layer("layer2").set_weights([W2,b2])
```

Predictions



Once you have a trained model, you can then use it to make predictions. Recall that the output of our model is a probability. In this case, the probability of a good roast. To make a decision, one must apply the probability to a threshold. In this case, we will use 0.5

Let's start by creating input data. The model is expecting one or more examples where examples are in the rows of matrix. In this case, we have two features so the matrix will be (m,2) where m is the number of examples. Recall, we have normalized the input features so we must normalize our test data as well.
To make a prediction, you apply the `predict` method.

In [13]:
```
X_test = np.array([
    [200,13.9],  # postive example
    [200,17]])   # negative example
X_testn = norm_l(X_test)
predictions = model.predict(X_testn)
print("predictions = \n", predictions)
```

```
predictions =
 [[9.63e-01]
 [3.03e-08]]
```

To convert the probabilities to a decision, we apply a threshold:

In [14]:
```
yhat = np.zeros_like(predictions)
for i in range(len(predictions)):
    if predictions[i] >= 0.5:
        yhat[i] = 1
    else:
        yhat[i] = 0
print(f"decisions = \n{yhat}")
```

```
decisions =
[[1.]
 [0.]]
```

This can be accomplished more succinctly:

In [15]:
```
yhat = (predictions >= 0.5).astype(int)
print(f"decisions = \n{yhat}")
```

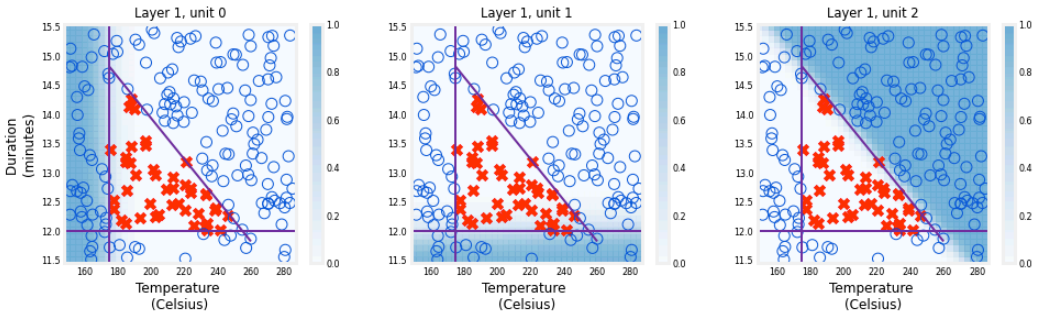
```
decisions =
[[1]
 [0]]
```

Layer Functions

Let's examine the functions of the units to determine their role in the coffee roasting decision. We will plot the output of each node for all values of the inputs (duration,temp). Each unit is a logistic function whose output can range from zero to one. The shading in the graph represents the output value.

> Note: In labs we typically number things starting at zero while the lectures may start with 1.

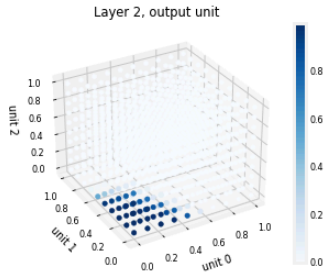
In [16]:
```
plt_layer(X,Y.reshape(-1,),W1,b1,norm_l)
```



The shading shows that each unit is responsible for a different "bad roast" region. unit 0 has larger values when the temperature is too low. unit 1 has larger values when the duration is too short and unit 2 has larger values for bad combinations of time/temp. It is worth noting that the network learned these functions on its own through the process of gradient descent. They are very much the same sort of functions a person might choose to make the same decisions.

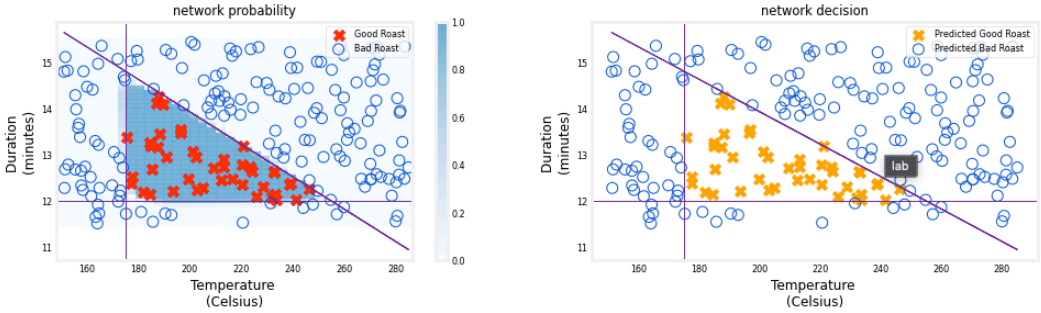
The function plot of the final layer is a bit more difficult to visualize. It's inputs are the output of the first layer. We know that the first layer uses sigmoids so their output range is between zero and one. We can create a 3-D plot that calculates the output for all possible combinations of the three inputs. This is shown below. Above, high output values correspond to 'bad roast' area's. Below, the maximum output is in area's where the three inputs are small values corresponding to 'good roast' area's.

In [17]:
```
plt_output_unit(W2,b2)
```

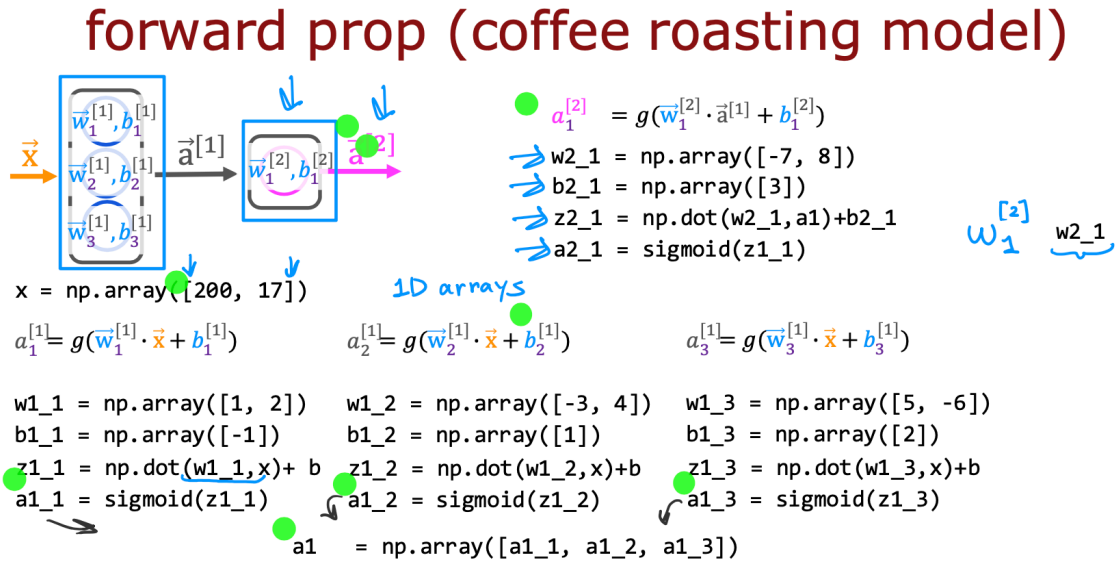


The final graph shows the whole network in action.
The left graph is the raw output of the final layer represented by the blue shading. This is overlaid on the training data represented by the X's and O's.
The right graph is the output of the network after a decision threshold. The X's and O's here correspond to decisions made by the network.
The following takes a moment to run

In [18]:
```
netf= lambda x : model.predict(norm_l(x))
plt_network(X,Y,netf)
```



Forward Propagation in a single layer



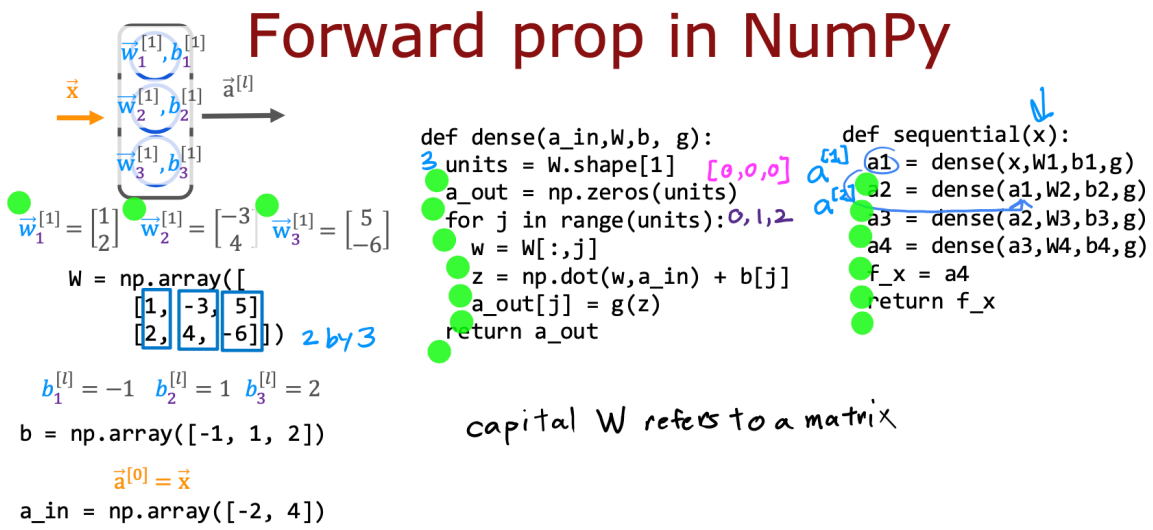
6.4.2 Forward proprogation in Numpy

Always remember to normalize data before inputing into dense layer functions
Define Desne layer

- Stack weight vectors into a matrix (number of units means number of neurons)
- Stack bias into 1D array
- Pull out jth column in the weight matrix.

Note that signmoid function g() is deined outside of dense().

As described in lecture, one can utilize a for loop to visit each unit (j) in the layer and perform the dot product of the weights for that unit (W[:,j]) and sum the bias for the unit (b[j]) to form z. An activation function g(z) can then be applied to that result. Let's try that below to build a "dense layer" subroutine.
First, you will define the activation function g(). You will use the sigmoid()function which is already implemented for you in the lab_utils_common.py file outside this notebook.



The following cell builds a two-layer neural network utilizing the my_dense subroutine above.

The following cell builds a two-layer neural network utilizing the my_dense subroutine above.

In [10]:
```
def my_sequential(x, W1, b1, W2, b2):
    a1 = my_dense(x,  W1, b1)
    a2 = my_dense(a1, W2, b2)
    return(a2)
```

We can copy trained weights and biases from the previous lab in Tensorflow.

In [11]:
```
W1_tmp = np.array( [[-8.93,  0.29, 12.9 ], [-0.1,  -7.32, 10.81]] )
b1_tmp = np.array( [-9.82, -9.28,  0.96] )
W2_tmp = np.array( [[-31.18], [-27.59], [-32.56]] )
b2_tmp = np.array( [15.41] )
```

6.4.3 Predictions

Once you have a trained model, you can then use it to make predictions. Recall that the output of our model is a probability. In this case, the probability of a good roast. To make a decision, one must apply the probability to a threshold. In this case, we will use 0.5.

Let's start by writing a routine similar to Tensorflow's model.predict(). This will take a matrix X with all m examples in the rows and make a prediction by running the model.

In [15]:
```
def my_predict(X, W1, b1, W2, b2):
    m = X.shape[0]
    p = np.zeros((m,1))
    for i in range(m):
        p[i,0] = my_sequential(X[i], W1, b1, W2, b2)
    return(p)
```

We can try this routine on two examples:

In [14]:
```
X_tst = np.array([
    [200,13.9],  # postive example
    [200,17]])   # negative example
X_tstn = norm_l(X_tst)  # remember to normalize
predictions = my_predict(X_tstn, W1_tmp, b1_tmp, W2_tmp, b2_tmp)
```

To convert the probabilities to a decision, we apply a threshold:

In [ ]:
```
yhat = np.zeros_like(predictions)
for i in range(len(predictions)):
    if predictions[i] >= 0.5:
        yhat[i] = 1
    else:
        yhat[i] = 0
print(f"decisions = \n{yhat}")
```

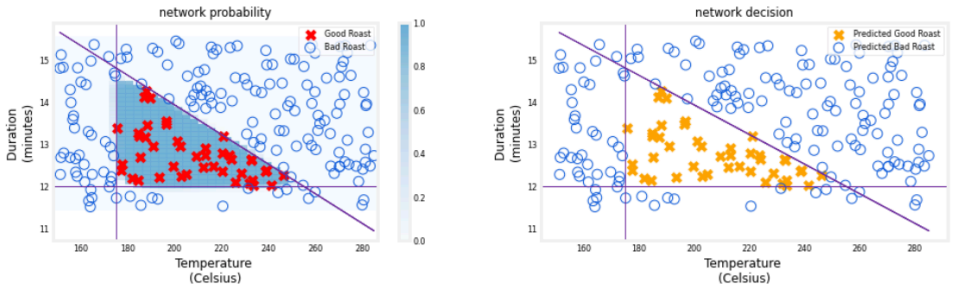
This can be accomplished more succinctly:

In [16]:
```
yhat = (predictions >= 0.5).astype(int)
print(f"decisions = \n{yhat}")
```

6.5 Network function

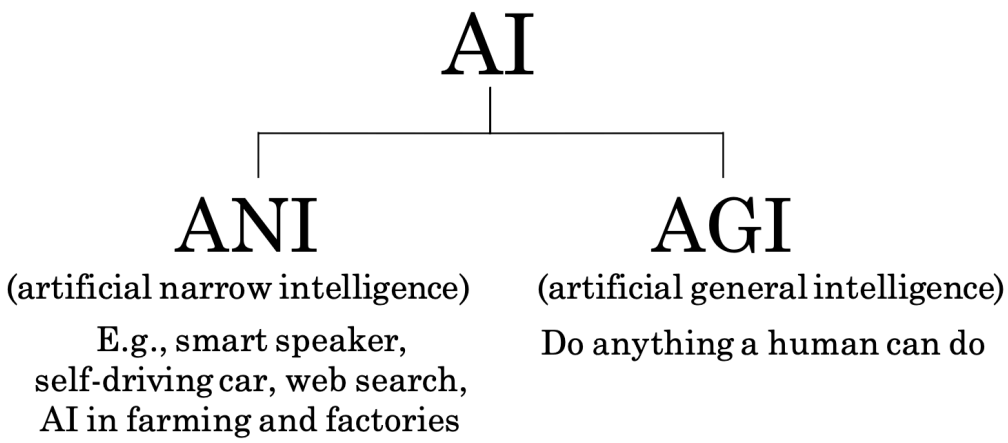
This graph shows the operation of the whole network and is identical to the Tensorflow result from the previous lab. The left graph is the raw output of the final layer represented by the blue shading. This is overlaid on the training data represented by the X's and O's.
The right graph is the output of the network after a decision threshold. The X's and O's here correspond to decisions made by the network.

```
In [17]: netf= lambda x : my_predict(norm_l(x),W1_tmp, b1_tmp,
                                     W2_tmp, b2_tmp)
         plt_network(X,Y,netf)
```



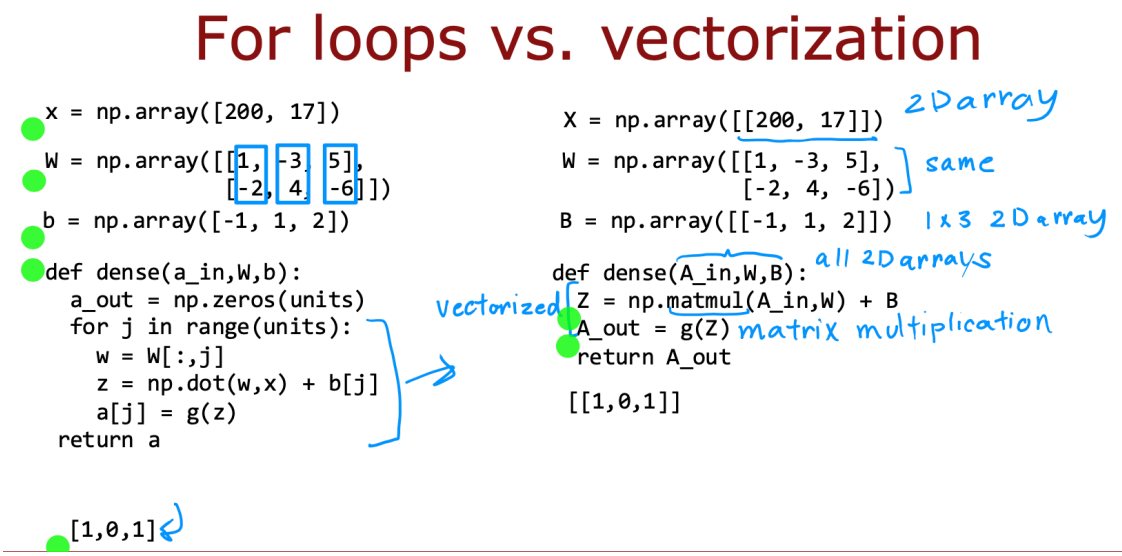
6.5.1 Path to Artifical general intelligence (AGI)

Definition of AGI

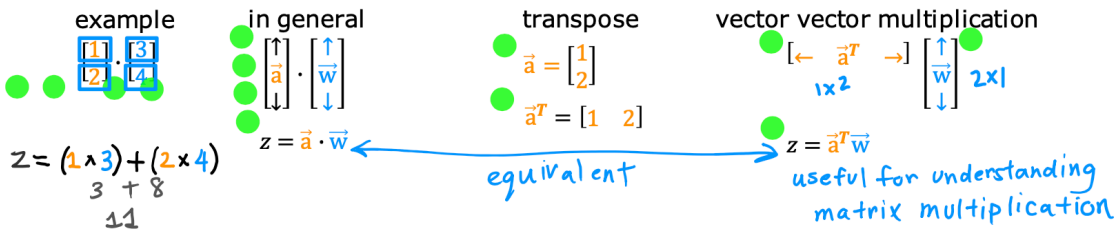


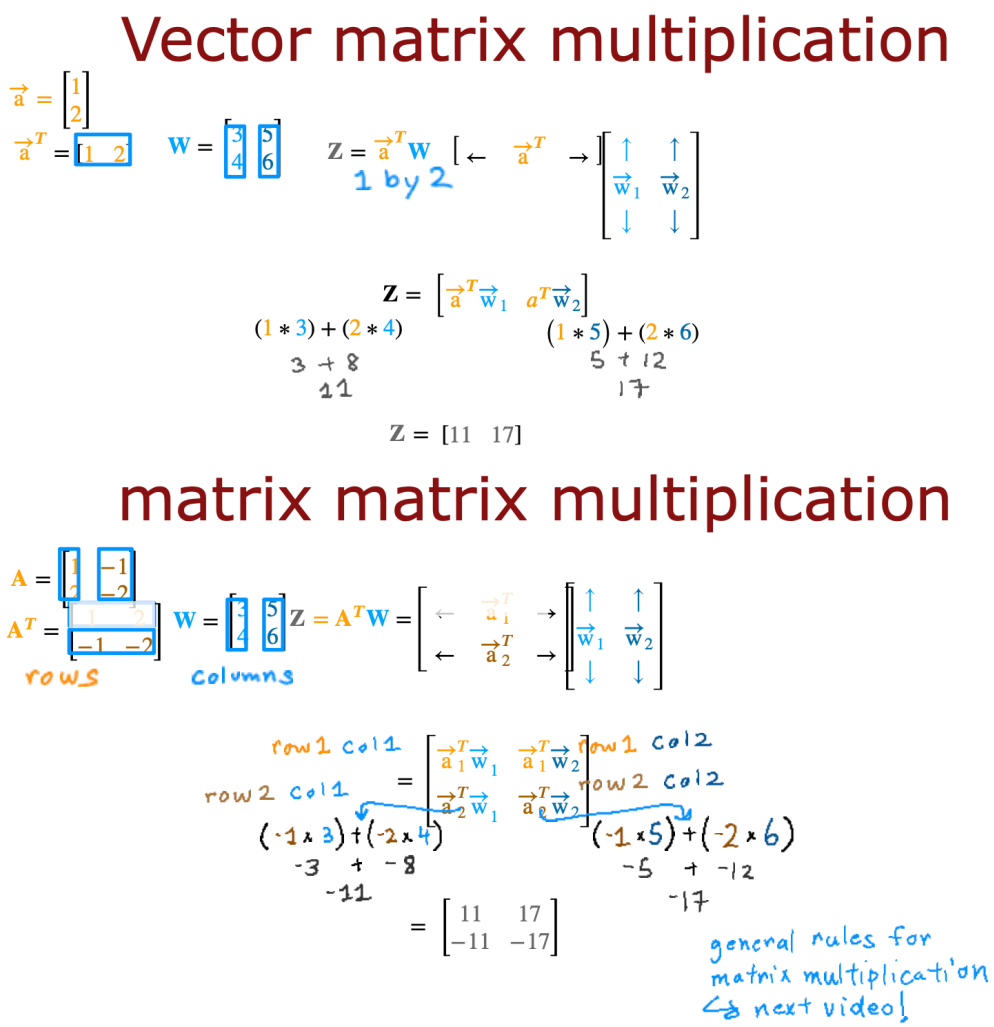
6.5.2 How Neural Networks Implemented Efficiently

By vectorizeing and exploiting parallel processing
To understand the vectorization's advantage over for-loop:
For loop can be replaced by vectorized implementation: numpy.matmul (matrix multiplication)



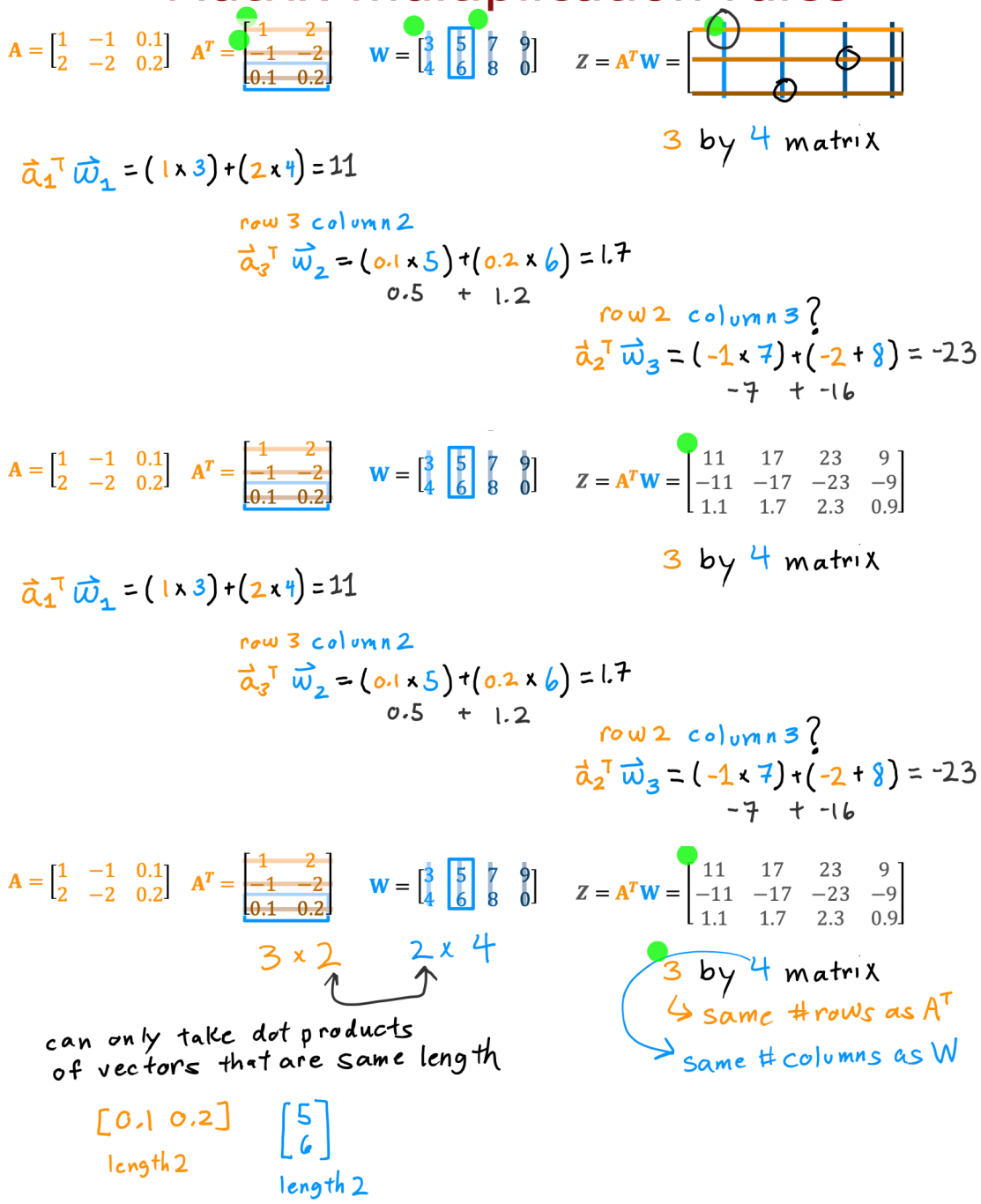
6.5.3 Vectorization: Matrix multiplication





Matrix multiplication rules

Matrix multiplication rules

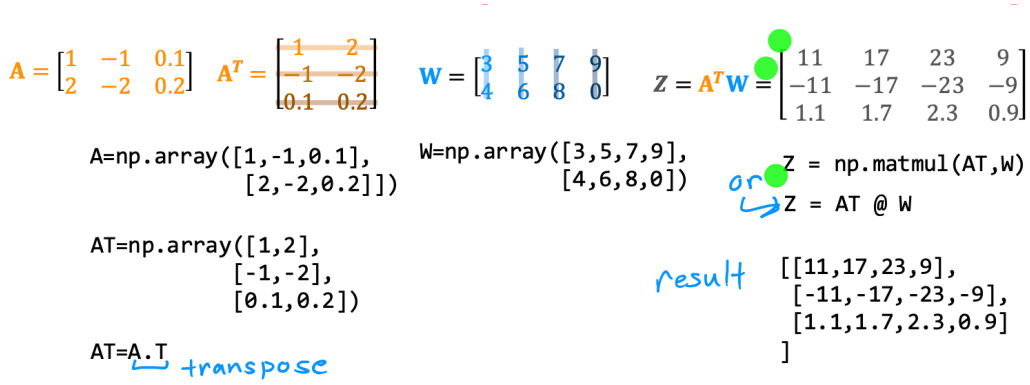


Matrix multiplication code

Matrix multiplication in NumPy

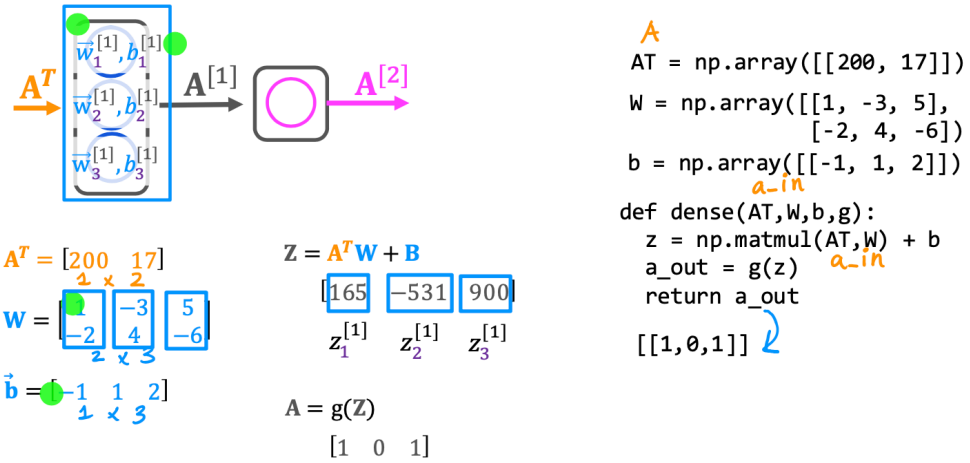
(numpy.matmul)

Transpose → A, a matrix. → A_transpose = A.T
Matrix multiplication → np.matmul



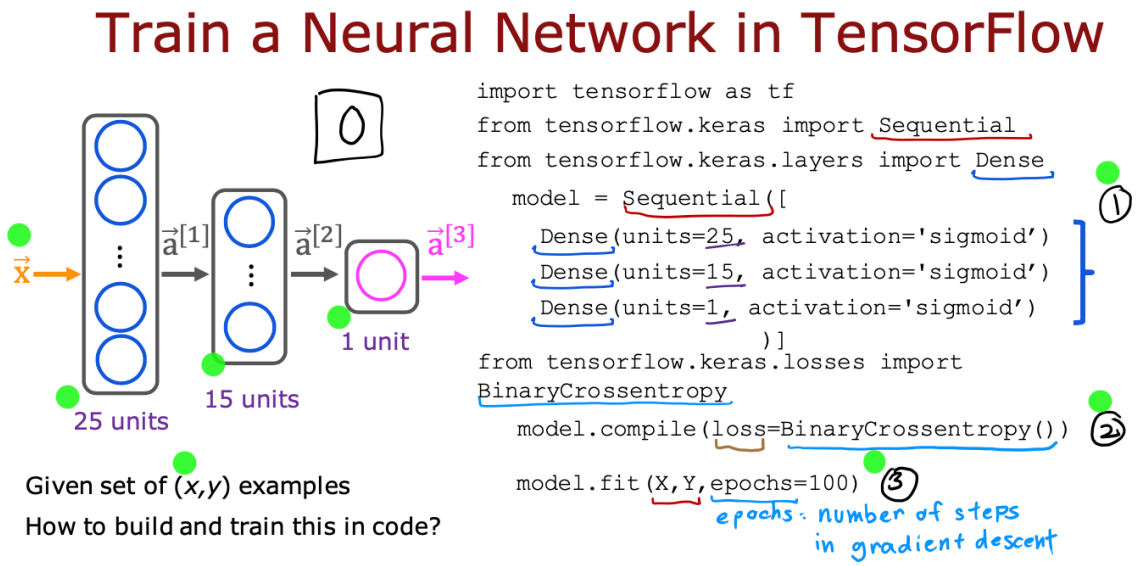
Dense layer vectorized

Transpose → A, an input matrix. → A_transpose = A.T
Matrix multiplication → np.matmul



6.5.4 TensorFlow Implementations

Key function to compile is the loss - **BinraryCrossentropy**
Then fit the model using model.fit(X,Y, **ephochs**) with ephochs denoting number of steps in gradient descent



6.5.5 TensorFlow Neural Netowrks Training Details

Conceptually, the model traning steps are 1st specify how to comput output given input x and parameters w, b. Namely, step one is to define the model. Step two is to specify the loss and cost of the model when applied to training data set. The loss computation $Loss\left(f_{\vec{w},b}(\vec{x}),y\right)$ is for one data only. The cost function $J(\vec{w},b)=\frac{1}{m}\sum Loss\left(f_{\vec{w},b}(\vec{x}),y\right)$ is the average loss, or cost, of the entire datasets. The third step is to train on data to minimize the cost function output.

Correspondingly in mathematical representation, the first step of model definition is identical to logistic regression where the z is dirst defined as the dot product between weight and input and then plus bias, then the sigmoid function is computed as $g(z)=\frac{1}{1+e^{-z}}$. The logistic loss is computed as $loss=-y\cdot\ln\left(g(z)\right)-(1-y)\cdot\ln\left(1-g(z)\right)$. The minimizaing cost function training is by updating weights and bias simutaneouly and iteratively.

In Tensor flow, the first model definition step is by building the model with $model=Sequential([layers])$ where layers is defined as eg. $Dense(\dots)$. Then, to comput the loss, we use the function $model.compile\left(loss=BinaryCrossentropy(\ )\right)$ then use $model.fit(X,y,epochs)$ to define the number of steps in gradient descent that is relating to the weight and bias updating speed (updates the network parameters in order to reduce the cost) and perfrom gradient descent using a “backprogation” technique.

Model Training Steps TensorFlow

1. specify how to compute output given input x and parameters w,b (define model)

$f_{\vec{w},b}(\vec{x})$ =?

2. specify loss and cost

$L(f_{\vec{w},b}(\vec{x}), y)$ 1 example

$J(\vec{w}, b) = \frac{1}{m}\sum_{i=1}^{m} L(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)})$

3. Train on data to minimize $J(\vec{w}, b)$

logistic regression

```
z = np.dot(w,x)+ b

f_x = 1/(1+np.exp(-z))
```

logistic loss

```
loss = -y * np.log(f_x)
   -(1-y) * np.log(1-f_x)

w = w - alpha * dj_dw
b = b - alpha * dj_db
```

neural network

```
model = Sequential([
          Dense(...)
          Dense(...)
          Dense(...)
                    ])
```

binary cross entropy

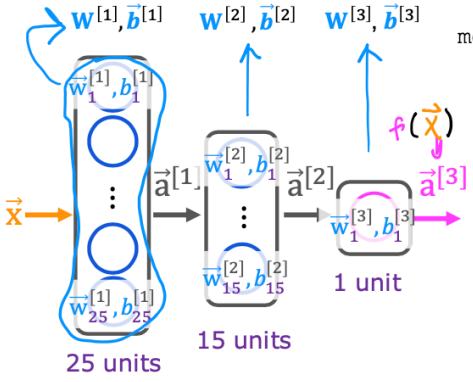
```
model.compile(
loss=BinaryCrossentropy())

model.fit(X,y,epochs=100)
```

1. Create the model

define the model

$f(\vec{x})$ =?




```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

model = Sequential([
  Dense(units=25, activation='sigmoid')
  Dense(units=15, activation='sigmoid')
  Dense(units=1, activation='sigmoid')
                    ])
```

2. Loss and cost functions

Mnist digit classification problem — binary classification

$L(f(\vec{x}), y) = -y\log(f(\vec{x})) - (1-y)\log(1-f(\vec{x}))$

Compare prediction vs. target

logistic loss
also known as binary cross entropy

$J(\mathbf{W}, \mathbf{B}) = \frac{1}{m}\sum_{i=1}^{m} L(f(\vec{x}^{(i)}), y^{(i)})$

$\mathbf{W}^{[1]}, \mathbf{W}^{[2]}, \mathbf{W}^{[3]}$ $\vec{b}^{[1]}, \vec{b}^{[2]}, \vec{b}^{[3]}$

$f_{W,B}(\vec{x})$

```
model.compile(loss= BinaryCrossentropy())
```

```
from tensorflow.keras.losses import
  BinaryCrossentropy
```

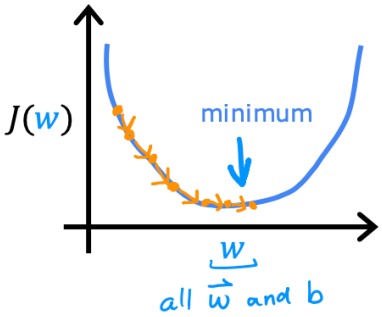
Keras

regression (predicting numbers and not categories) — mean squared error

```
model.compile(loss= MeanSquaredError())
```

```
from tensorflow.keras.losses import
  MeanSquaredError
```

3. Gradient descent



repeat {

$$w_j^{[l]} = w_j^{[l]} - \alpha \frac{\partial}{\partial w_j} J(\vec{w}, b)$$

$$b_j^{[l]} = b_j^{[l]} - \alpha \frac{\partial}{\partial b} J(\vec{w}, b)$$

}

Compute derivatives for gradient descent using "back propagation"

```
model.fit(X,y,epochs=100)
```

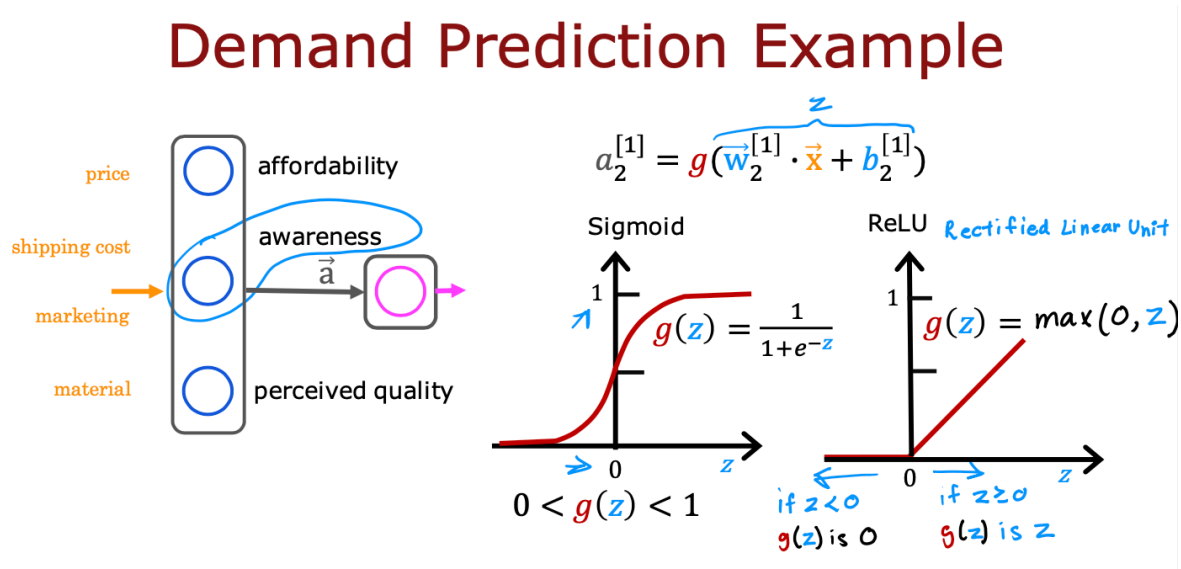
6.5.6 Activation Functions

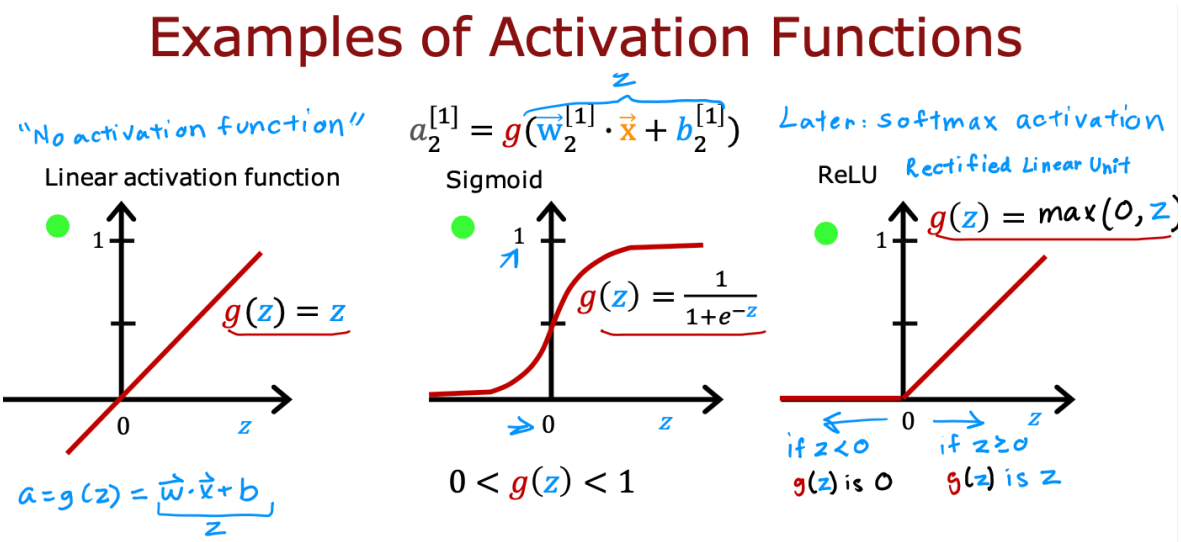
Sigmoid

$$\boldsymbol{a = g(\vec{w} \cdot x = b)}$$

Rectified Linear Unit

$$\boldsymbol{g(z) = max(0, z)}$$

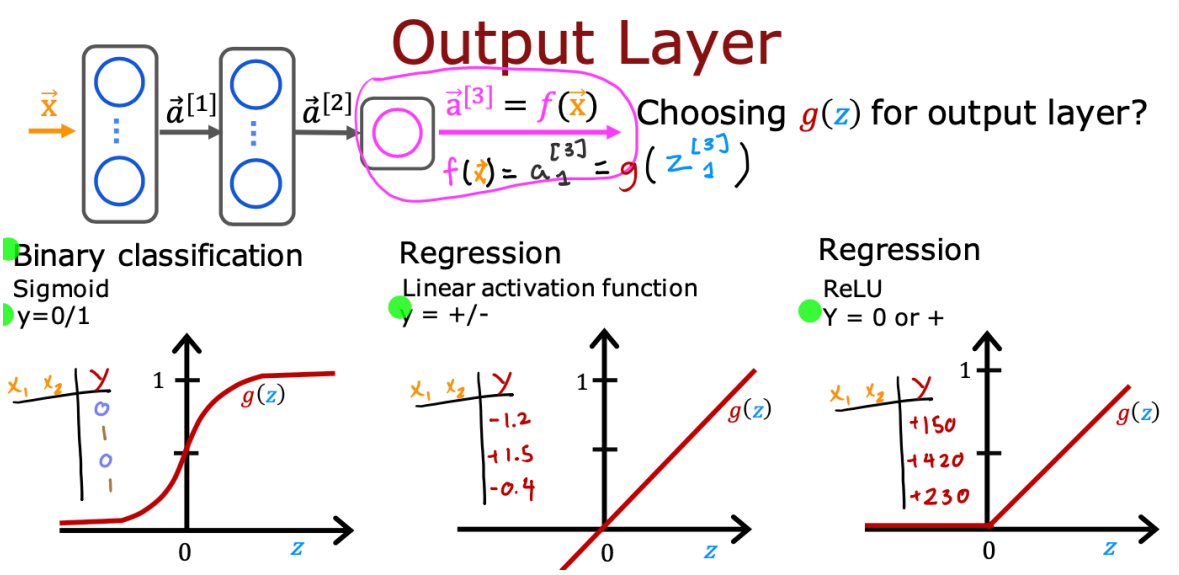




6.5.7 Choosing activation functions

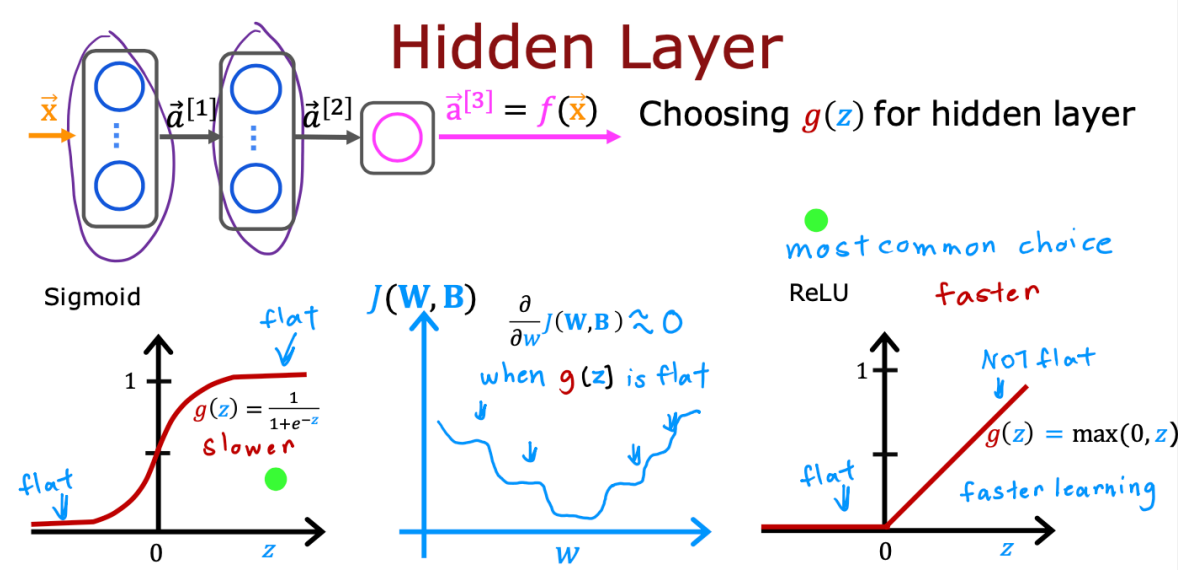
For the output layer: Depending on the ground true lable x
Binary classification problem: sigmoid
Regression classification:

- **linear activation for possible positive and negative values**
- **ReLU activation for only positive values**

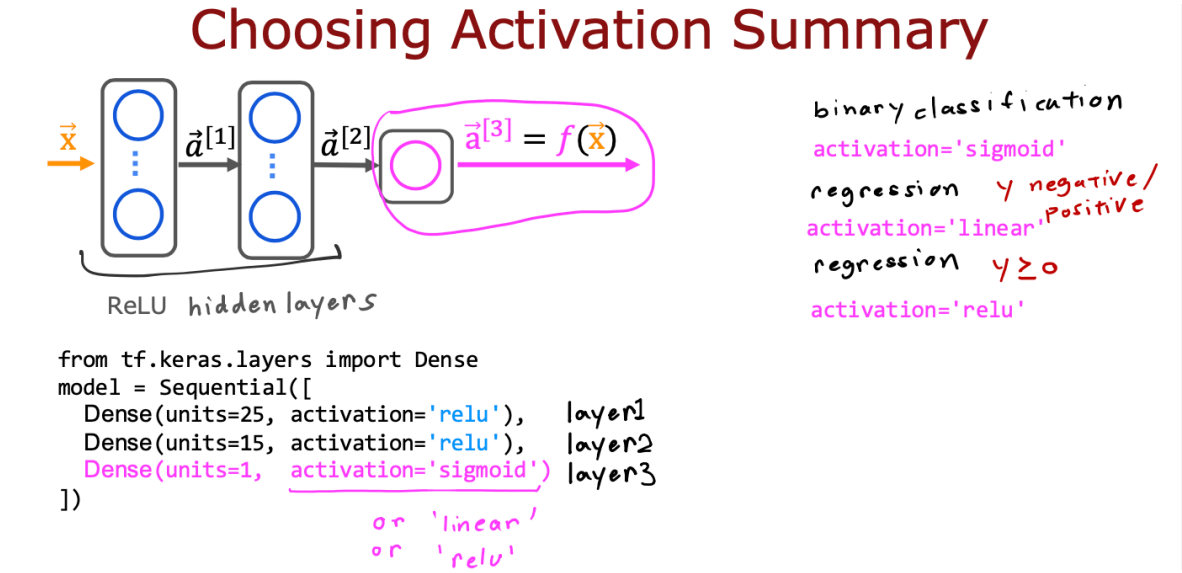


For hiden layers:

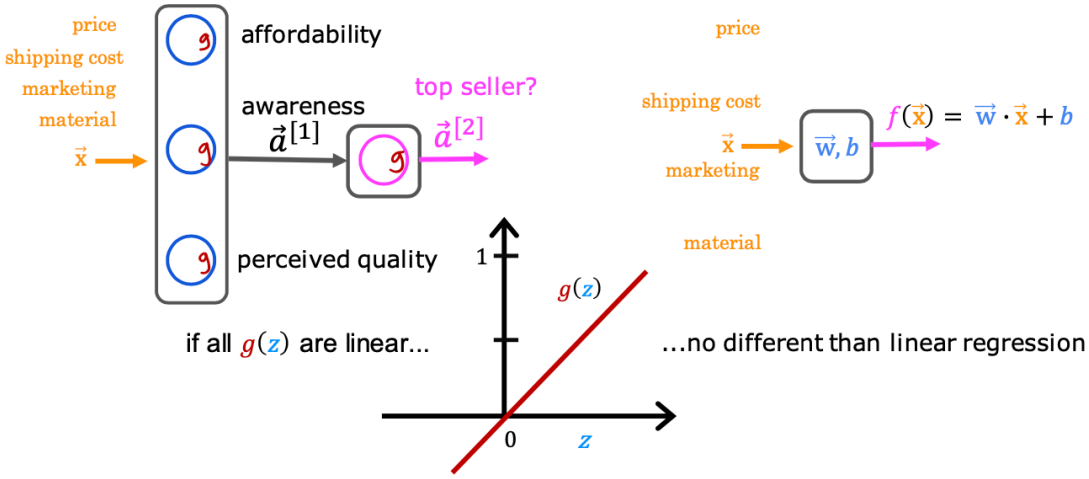
ReLU is by far the most common choice. Since that flat output will result in slower gradient descent given smaller gradients. ReLU has nly one flat region while sigmoid has two flat regions. ReLU is hence faster in gradient descent.

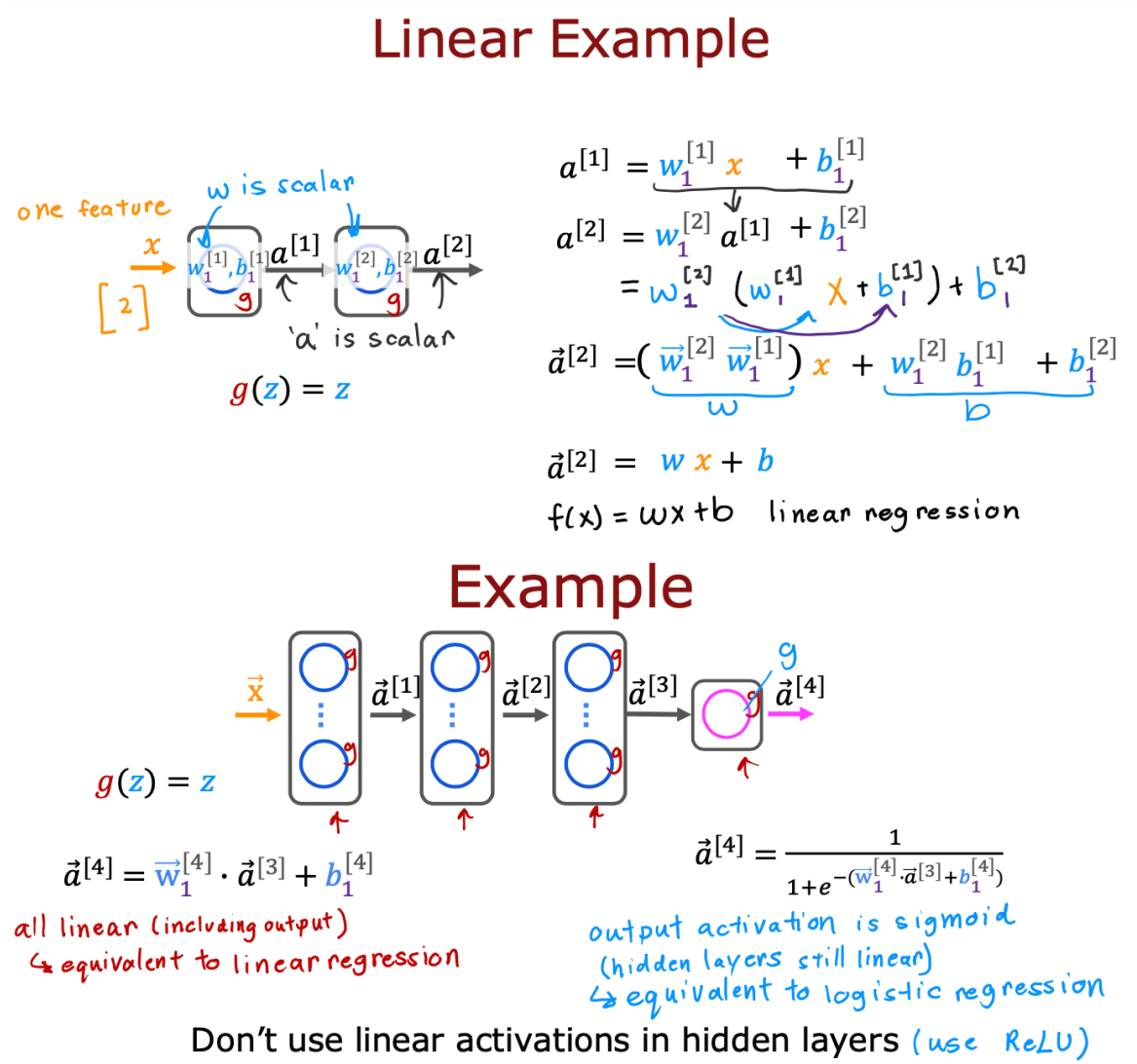


Choosing activation summary



6.5.8 Why do we need activation functions?





6.6 特征和直观理解 Examples and Intuitions

从本质上讲，神经网络能够通过学习得出其自身的一系列特征。在普通的逻辑回归中，我们被限制为使用数据中的原始特征$x_1, x_2, \ldots, x_n$，我们虽然可以使用一些二项式项来组合这些特征，但是我们仍然受到这些原始特征的限制。在神经网络中，原始特征只是输入层，在我们上面三层的神经网络例子中，第三层也就是输出层做出的预测利用的是第二层的特征，而非输入层中的原始特征，我们可以认为第二层中的特征是神经网络通过学习后自己得出的一系列用于预测输出变量的新特征。

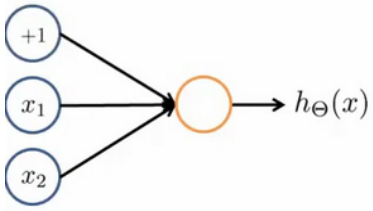
神经网络中，单层神经元（无中间层）的计算可用来表示逻辑运算，比如逻辑与(AND)、逻辑或(OR)。

举例说明：逻辑与(AND)；下图中左半部分是神经网络的设计与 output 层表达式，右边上部分是 sigmod

函数，下半部分是真值表。

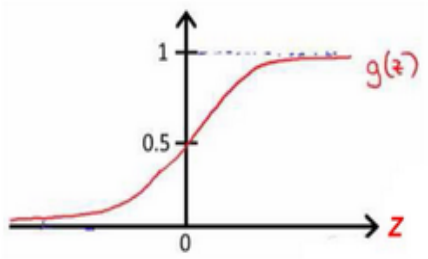
我们可以用这样的一个神经网络表示 **AND** 函数：

$x_1, x_2 \in \{0, 1\}$
$y = x_1 \text{ AND } x_2$



其中$\theta_0 = -30, \theta_1 = 20, \theta_2 = 20$ 我们的输出函数$h_\theta(x)$即为：$h_\theta(x) = g(-30 + 20x_1 + 20x_2)$

我们知道$g(x)$的图像是：



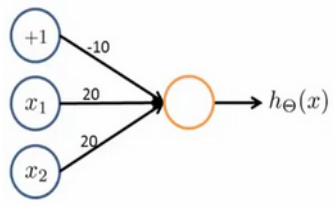
x_1	x_2	$h_\Theta(x)$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

所以我们有：$h_\theta(x) \approx \mathrm{x}_1 \mathrm{AND}\, \mathrm{x}_2$

所以我们的：

这就是 **AND** 函数。

接下来再介绍一个 **OR** 函数：

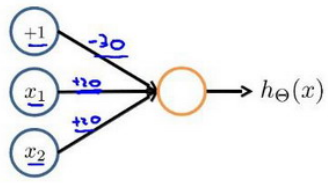


x_1	x_2	$h_\Theta(x)$
0	0	0
0	1	1
1	0	1
1	1	1

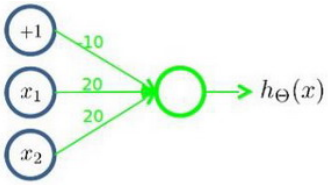
OR 与 **AND** 整体一样，区别只在于的取值不同。

二元逻辑运算符（**BINARY LOGICAL OPERATORS**）当输入特征为布尔值（0 或 1）时，我们可以用一个单一的激活层可以作为二元逻辑运算符，为了表示不同的运算符，我们只需要选择不同的权重即可。

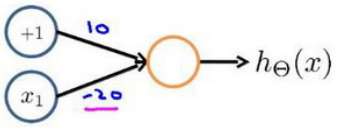
下图的神经元（三个权重分别为-30，20，20）可以被视为作用同于逻辑与（**AND**）：



下图的神经元（三个权重分别为-10，20，20）可以被视为作用等同于逻辑或（**OR**）：



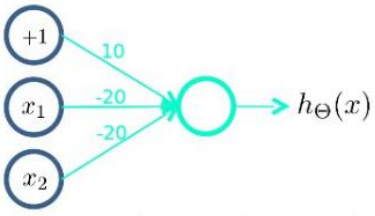
下图的神经元（两个权重分别为 10，-20）可以被视为作用等同于逻辑非（**NOT**）：



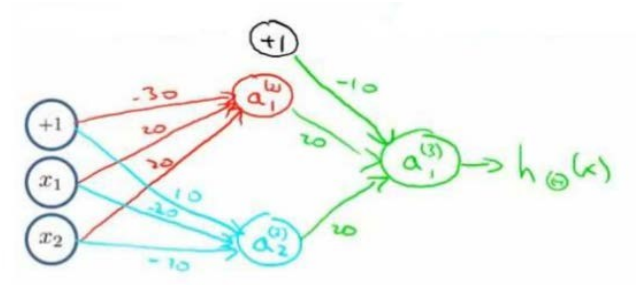
我们可以利用神经元来组合成更为复杂的神经网络以实现更复杂的运算。例如我们要实现 **XNOR** 功能（输入的两个值必须一样，均为 1 或均为 0），即：

$\mathrm{XNOR} = (\mathrm{x}_1 \ \mathrm{AND}\ \mathrm{x}_2)\ \mathrm{OR}\big((\mathrm{NOT}\ \mathrm{x}_1)\mathrm{AND}(\mathrm{NOT}\ \mathrm{x}_2)\big)$

首先构造一个能表达$(\mathrm{NOT}\ \mathrm{x}_1)\mathrm{AND}(\mathrm{NOT}\ \mathrm{x}_2)$部分的神经元：



然后将表示 **AND** 的神经元和表示$(\mathrm{NOT}\ \mathrm{x}_1)\mathrm{AND}(\mathrm{NOT}\ \mathrm{x}_2)$的神经元以及表示 **OR** 的神经元进行组合：



我们就得到了一个能实现 XNOR 运算符功能的神经网络。

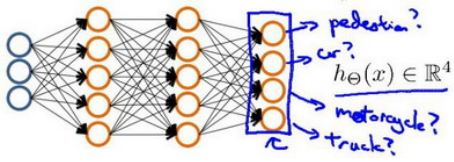
按这种方法我们可以逐渐构造出越来越复杂的函数，也能得到更加厉害的特征值。

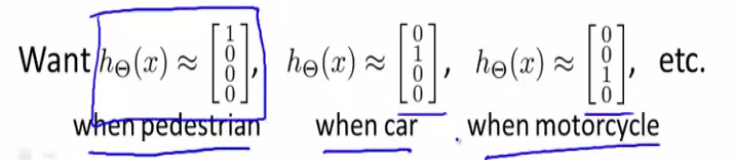
这就是神经网络的厉害之处。

7、 多类分类 Multiclass Classification

当我们有不止两种分类时（也就是$y = 1,2,3 \ldots.$），比如以下这种情况，该怎么办？如果我们要训练一个神经网络算法来识别路人、汽车、摩托车和卡车，在输出层我们应该有 4 个值。例如，第一个值为 1 或 0 用于预测是否是行人，第二个值用于判断是否为汽车。

输入向量x有三个维度，两个中间层，输出层 4 个神经元分别用来表示 4 类，也就是每一个数据在输出层都会出现$[a\ b\ c\ d]^T$，且a, b, c, d中仅有一个为 1，表示当前类。下面是该神经网络的可能结构示例：





神经网络算法的输出结果为四种可能情形之一：

$$\begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$$

Softmax regression for N possible outputs (multiclassification)

To estimate the chance of particular class being the output among the probability of all classes in the pool.

a_n represents the probability, the model estimate

K inidicates the index of summation

$$\boldsymbol{a_j = \frac{e_j^z}{\sum_k^N e_k^z} = P(y = j \mid \vec{x})}$$

Softmax is a function of z1, z2, …, to zN that depends on all values of z simultaneuously. The other activation functions such as sigmoid, ReLU, and linear, only depends on specific z_{index} value.

7.1 Cost functions using softmax regression.

The higher the possibility, the smaller the loss per crossentropy loss of soft regression

7.1.1 Cost Function

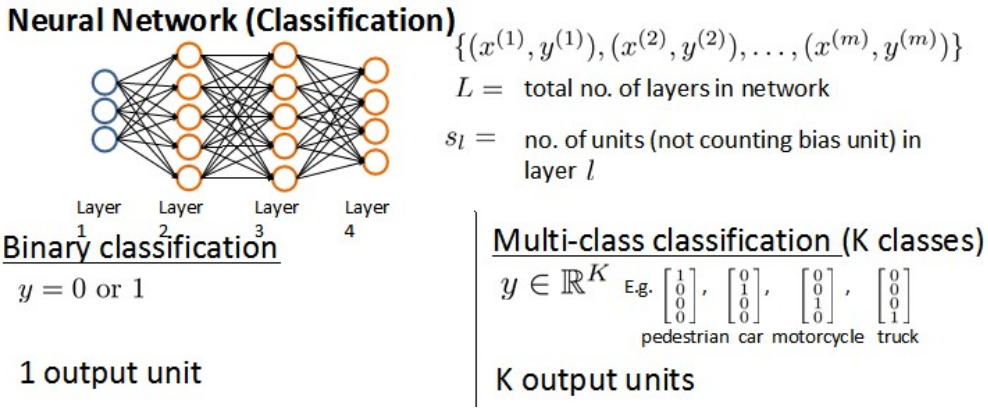
首先引入一些便于稍后讨论的新标记方法：

假设神经网络的训练样本有m个，每个包含一组输入x和一组输出信号y，L表示神经网络层数，S_I表示每层的 **neuron** 个数(S_l表示输出层神经元个数)，S_L代表最后一层中处理单元的个数。

将神经网络的分类定义为两种情况：二类分类和多类分类，

二类分类：$S_L = 0, y = 0\ or\ 1$表示哪一类；

K类分类：$S_L = k, y_i = 1$表示分到第 i 类；$(k > 2)$



我们回顾逻辑回归问题中我们的代价函数为：

$$J(\theta) = -\frac{1}{m}\left[\sum_{j=1}^{n} y^{(i)} \log h_\theta(x^{(i)}) + \left(1 - y^{(i)}\right) log\left(1 - h_\theta\left(x^{(i)}\right)\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^{\,2}$$

在逻辑回归中，我们只有一个输出变量，又称标量（**scalar**），也只有一个因变量y，但是在神经网络中，我们可以有很多输出变量，我们的$h_\theta(x)$是一个维度为K的向量，并且我们训练集中的因变量也是同样维度的一个向量，因此我们的代价函数会比逻辑回归更加复杂一些，为：$h_\theta(x) \in \mathbb{R}^K$ ，$\left(h_\theta(x)\right)_i = i^{th}\text{output}$

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{k} y_k^{\,(i)} \log\left(h_\Theta\left(x^{(i)}\right)\right)_k + \left(1 - y_k^{(i)}\right)\log\left(1 - \left(h_\Theta\left(x^{(i)}\right)\right)_k\right)\right]$$

$$+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}\left(\Theta_{ji}^{(l)}\right)^2$$

这个看起来复杂很多的代价函数背后的思想还是一样的，我们希望通过代价函数来观察算法预测的结果与真实情况的误差有多大，唯一不同的是，对于每一行特征，我们都会给出K个预测，基本上我们可以利用循环，对每一行特征都预测K个不同结果，然后在利用循环在K个预测中选择可能性最高的一个，将其与y中的实际数据进行比较。

正则化的那一项只是排除了每一层θ_0后，每一层的θ 矩阵的和。最里层的循环j循环所有的行（由s_l +1层的激活单元数决定），循环i则循环所有的列，由该层（s_l层）的激活单元数所决定。即：$h_\theta(x)$与真实值之间的距离为每个样本-每个类输出的加和，对参数进行 **regularization** 的 **bias** 项处理所有参数的平方和。

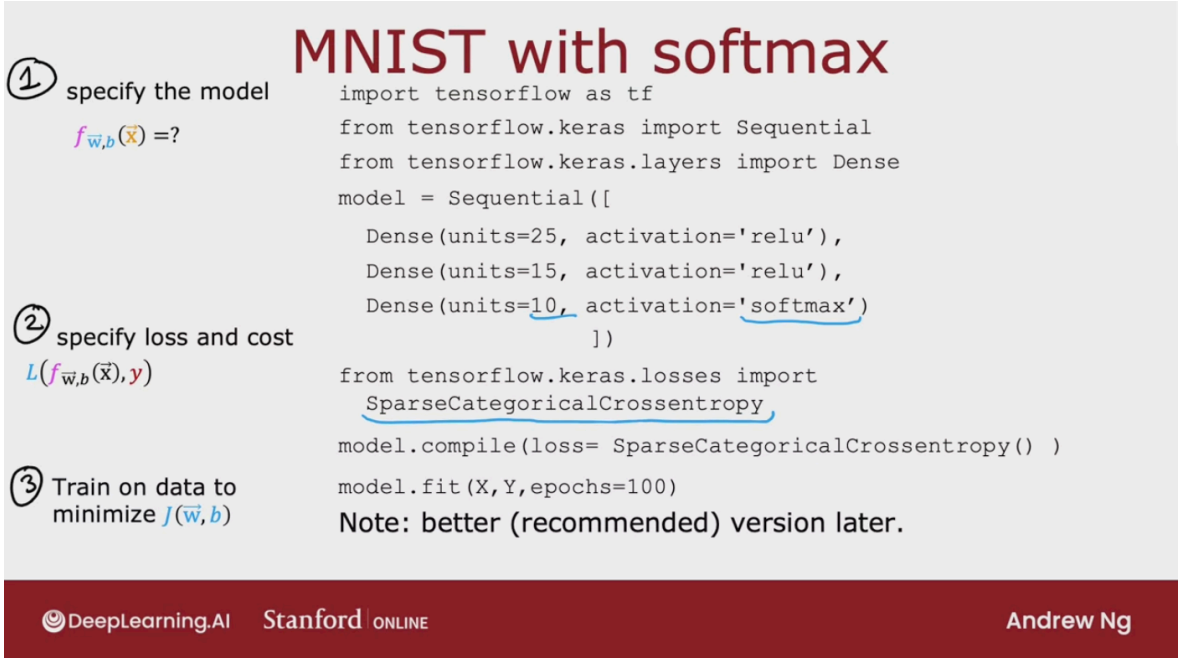
7.2 Neural network with softmax output

The output layer of NN should have the number of classes you desire to classified. The output layer should be softmax layer.

The hidden layers will be using ReLU.

The following softmax NN is a demo and yet should not be used. TensorFlow has a better implementation.

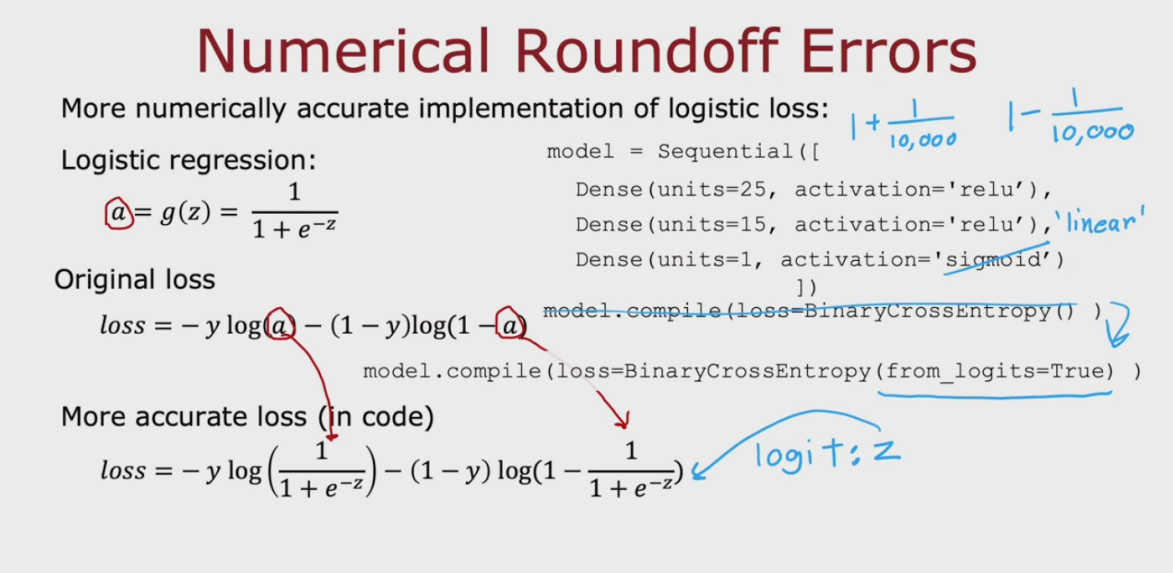
Note: MNIST stands for MNIST is a database. The acronym stands for “Modified National Institute of Standards and Technology.” The MNIST database contains handwritten digits (0 through 9), and can provide a baseline for testing image processing systems.



7.2.1 Improved implementation of softmax

Finite register length: Numerial roundoff errors

Instead of having a as intermedia variable, directly using z as intermediate variable would reduce the layers of rounding such that the round off error is less.



7.2.2 Numerically acuratne implementation of softmax

Use z as intermediate variables and last layer with linear, activation function in the model phase, while integrating the softmax computation in the SparseCaegoricalCrossEntropy in the loss computation stage.

The development flow is: model → loss computation → fit (compile) → predict

When predicting, we should use softmax at the end for multiclass classificatoin

More numerically accurate implementation of softmax

Softmax regression

$$(a_1, \ldots, a_{10}) = g(z_1, \ldots, z_{10})$$

$$\text{Loss} = L(\vec{a}, y) = \begin{cases} -\log a_1 & \text{if } y = 1 \\ \vdots & \\ -\log a_{10} & \text{if } y = 10 \end{cases}$$

```
model = Sequential([
  Dense(units=25, activation='relu'),
  Dense(units=15, activation='relu'),
  Dense(units=10, activation='softmax')
  ])
```

'linear'

~~model.compile(loss=SparseCategoricalCrossEntropy())~~

More Accurate

$$L(\vec{a}, y) = \begin{cases} -\log \frac{e^{z_1}}{e^{z_1} + \cdots + e^{z_{10}}} & \text{if } y = 1 \\ \vdots & \\ -\log \frac{e^{z_{10}}}{e^{z_1} + \cdots + e^{z_{10}}} & \text{if } y = 10 \end{cases}$$

```
model.compile(loss=SparseCategoricalCrossEntropy(from_logits=True))
```

MNIST (more numerically accurate)

model

```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
model = Sequential([
  Dense(units=25, activation='relu'),
  Dense(units=15, activation='relu'),
  Dense(units=10, activation='linear') ])
```

loss

```
from tensorflow.keras.losses import
  SparseCategoricalCrossentropy
model.compile(...,loss=SparseCategoricalCrossentropy(from_logits=True) )
```

fit

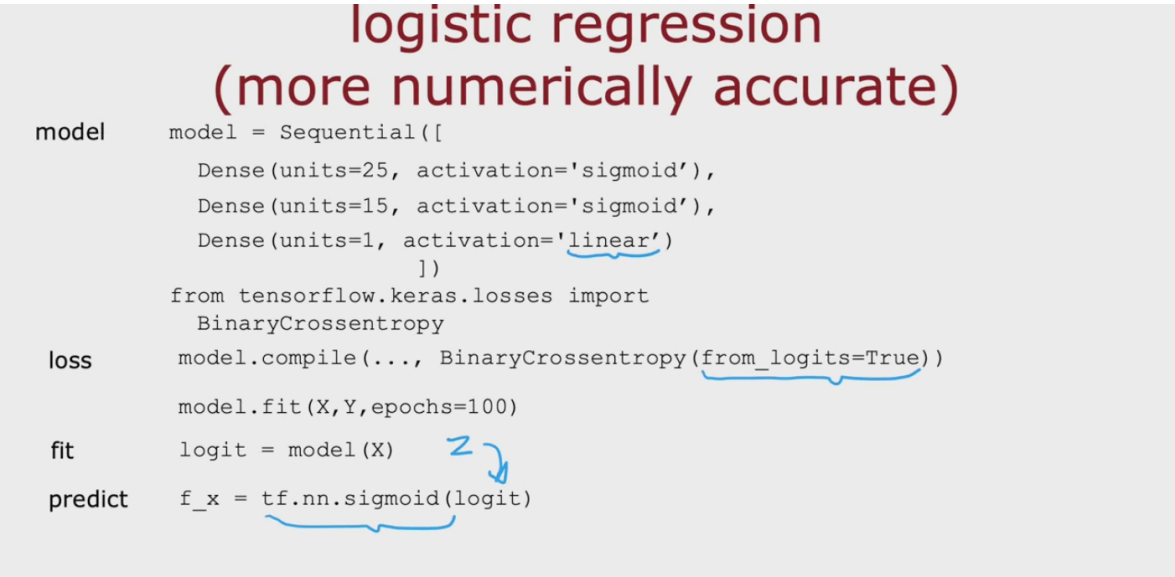
```
model.fit(X,Y,epochs=100)
```

predict

```
logits = model(X)
f_x = tf.nn.softmax(logits)
```

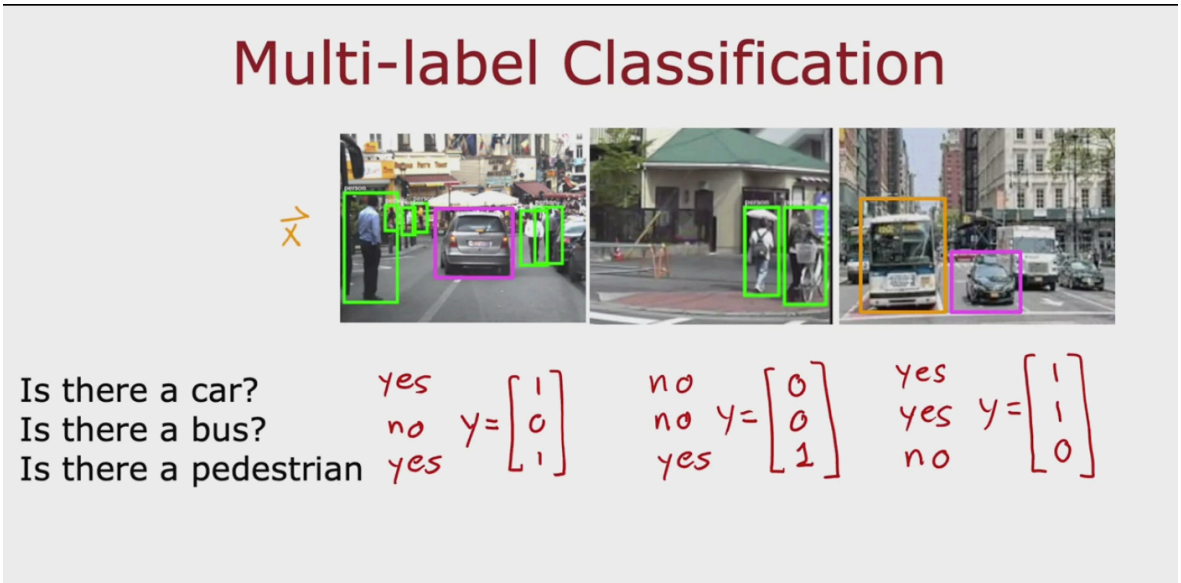
not is $a_1 \ldots a_{10}$ $z_1 \ldots z_{10}$

7.3 The same applies to logistic regression.



7.4 Classificatoin with Multiple Outputs (Multiple labels)

Multilabel classification: types including car, bus, and pedestrian, classification in an image.
First approach: one NN for each class
Second approach: one NN for simultaneuously identifying each class and at the end output a vector of N class numbers indicating if the image contains that class or no.



In this example, at the end, we wish to identify if the image **contains** a car **or no**, a bus or on, and a pedestrian or no, so the last layer we can use **sigmoid** function.

Multi-label classification and multi-class classification is different, such that the output layer in multi-label will ue sigmoid function to for each neuron determine a binary classification (cat/not_cat, dog/not_dog, others/no_others)**,** to determine if an image **CONTAINS** the object**.**

7.5 Advanced Optimization

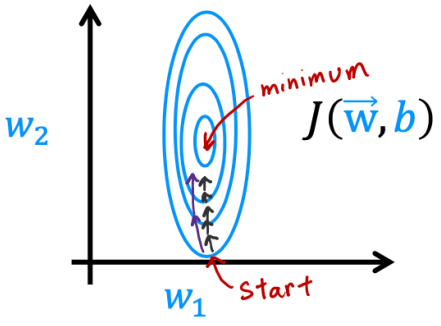
7.5.1 Adam Algorithm Intuition

Adaptive Moment estimation

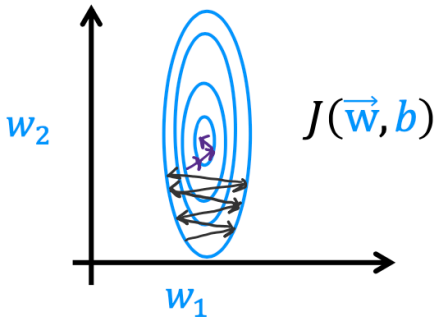
That allows faster learning

Adam: Adaptive Moment estimation not just one α

$$w_1 = w_1 - \alpha_1 \frac{\partial}{\partial w_1} J(\vec{w}, b)$$
$$\vdots$$
$$w_{10} = w_{10} - \alpha_{10} \frac{\partial}{\partial w_{10}} J(\vec{w}, b)$$
$$b = b - \alpha_{11} \frac{\partial}{\partial b} J(\vec{w}, b)$$



If w_j (or b) keeps moving in same direction, increase α_j.



If w_j (or b) keeps oscillating, reduce α_j.

7.5.2 ADAM implementation

model

```
model = Sequential([
          tf.keras.layers.Dense(units=25, activation='sigmoid')
          tf.keras.layers.Dense(units=15, activation='sigmoid')
          tf.keras.layers.Dense(units=10, activation='linear')
])
```

compile

$\alpha = 10^{-3} = 0.001$

```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True))
```

fit

```
model.fit(X,Y,epochs=100)
```

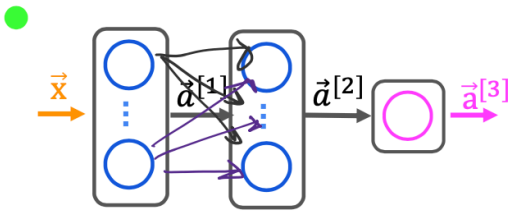
7.6 Additional Layer Types

Unlike dense layers, neurons in convolutional layers only process a specific region of the input, such as a portion of the digit image. This approach speeds up computation and reduces training data needs or overfitting risks.

The video further explores convolutional neural networks (CNNs) using electrocardiogram (ECG) signals as an example. In a CNN, neurons in each layer focus on a limited window of the input, such as segments of the ECG signal. This structure leads to specialized processing of different input parts. The video explains that CNNs provide multiple architectural choices, like the size of the input window for each neuron and the number of neurons per layer, which can enhance neural network effectiveness for certain applications.

7.6.1 Dense layer

fully connected layer with activation function that every neuron in the layer gets its inputs all the activations from the previous layer.

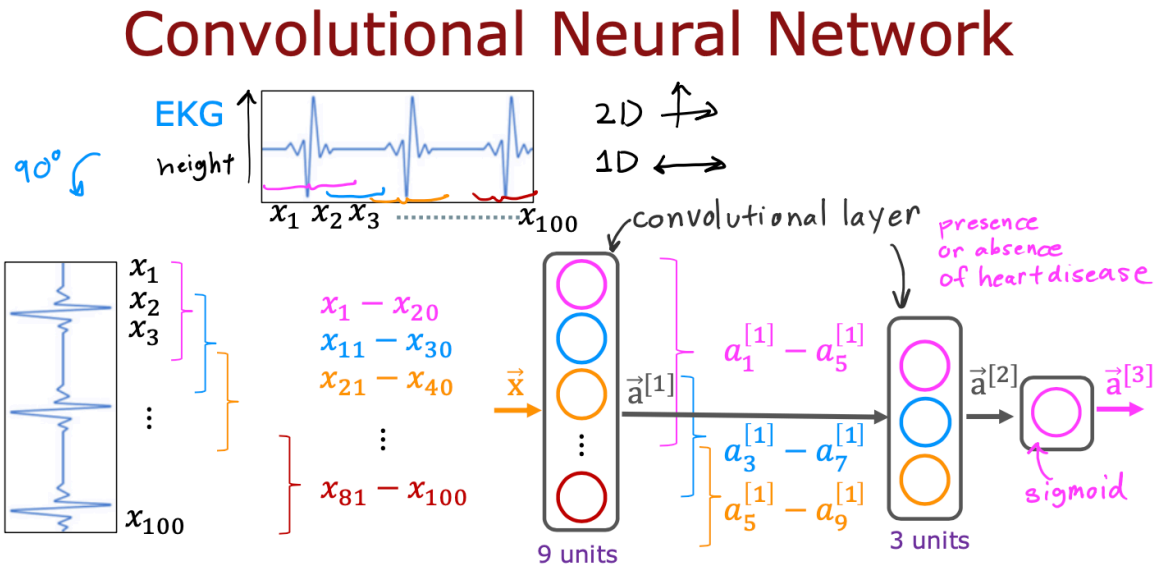


Each neuron output is a function of all the activation outputs of the previous layer.

$$\vec{a}_1^{[2]} = g\left(\vec{w}_1^{[2]} \cdot \vec{a}^{[1]} + b_1^{[2]}\right)$$

7.6.2 Convolution layer:

Each Neuron only looks at part of the previous layer's inputs.
e.g. ECG image output analysis: overlap-save linear convolution
1st layer:
Length of frame: 20 → each neuron looks at 20 rows.
Length of overlap window: frame_length / 2 = 10
signal length = number of rows = $\vec{X}[0].shape$ = 100
output neurons = (signal_legnth / overlap_window_length) - 1 = 9
2nd layer:
Length of frame: 5 → each neuron looks at 5 rows.
Length of overlap window: frame_length / 2 = 2.5 ≈ 2
signal length = number of rows = $\vec{X}[0].shape$ = 9
output neurons = (signal_legnth / overlap_window_length) - 1 = 3.5 ≈ 3



第 5 周

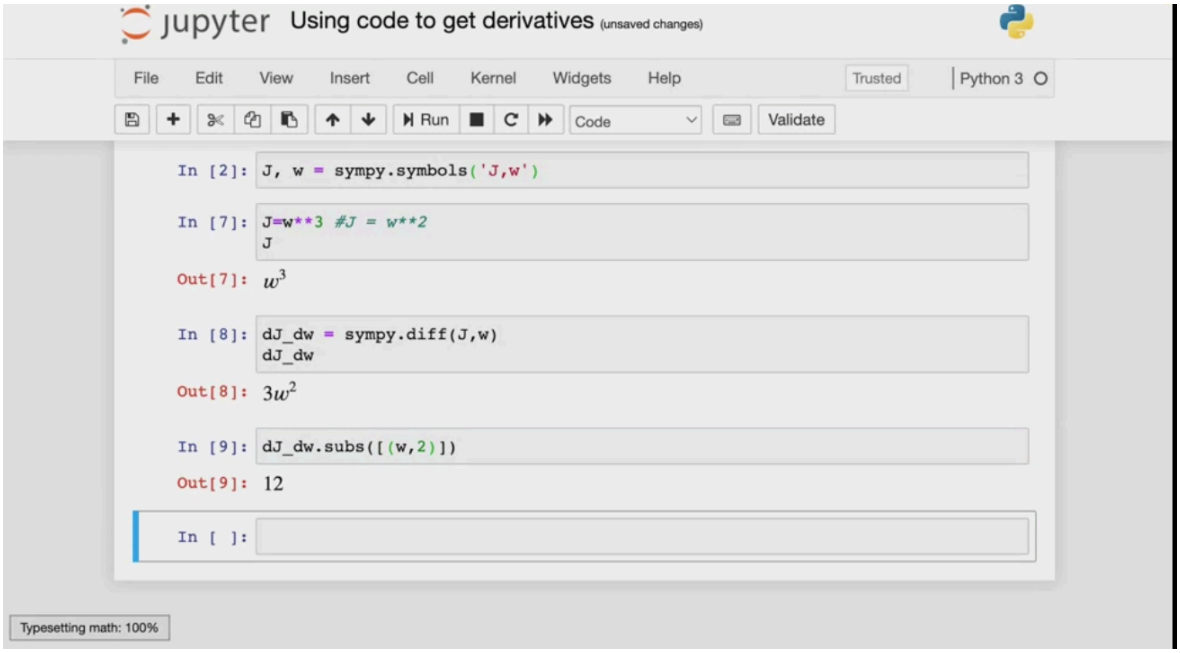
8、 反向传播算法 & 代价函数 BackPropagation and Cost Function

8.1 反向传播算法 Backpropagation Algorithm

Backpropagation Basics:

- TensorFlow automates backpropagation for neural network training.
- Backpropagation calculates derivatives of the cost function with respect to network parameters.
- Essential for gradient descent or Adam optimization in training neural networks.
- **Backpropagation is the chain rule of derivative**

Understanding Derivatives:

- Derivative: How much the function's output changes for a tiny change in input.
- Example: $J(w) = w^2$, the derivative of J with respect to w illustrates how J changes as w is modified.
- Gradient descent uses these derivatives for parameter updates.
- 

Calculus and Derivatives:

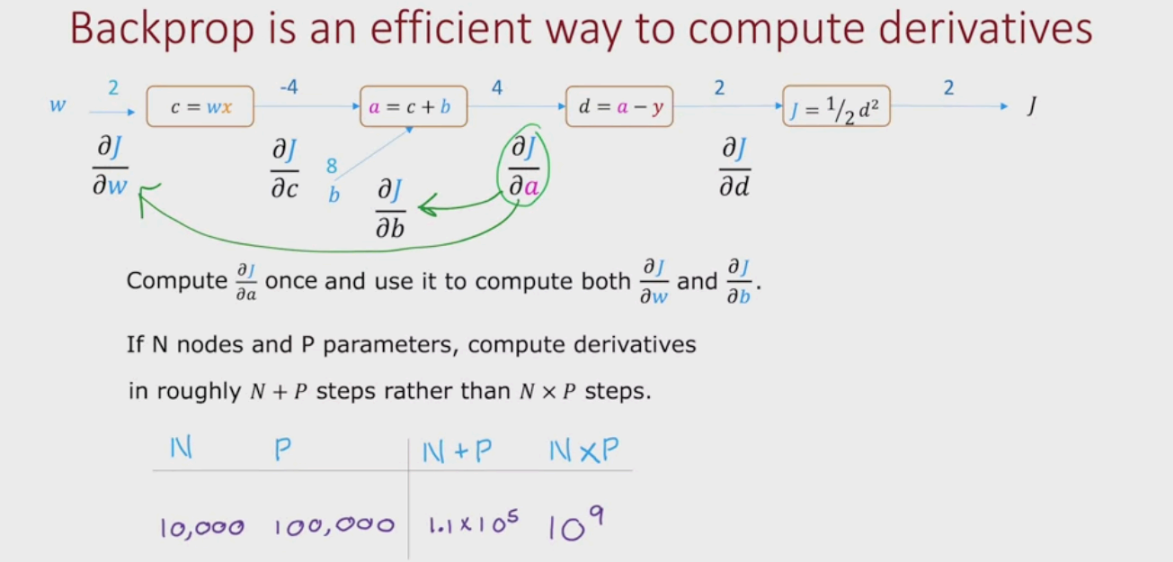
- Derivative calculation involves infinitesimally small changes.
- The derivative of $J(w)$ with respect to w indicates the rate of change of J as w varies.
- Calculus is used for deriving these derivatives but is simplified by software like TensorFlow.

Computation Graphs:

- Essential for understanding backpropagation in neural networks.
- Break down neural network operations into a series of steps.
- Allows for efficient calculation of derivatives during backpropagation.

Backpropagation Mechanics:

- Operates from output to input (right-to-left) in computation graphs.
- Calculates derivatives with respect to each parameter efficiently.
- Involves the chain rule from calculus for computing derivatives.



-

Practical Implications in Neural Networks:

- Enables efficient training of networks by computing gradients for parameter updates.
- Automatic differentiation (autodiff) in frameworks like TensorFlow simplifies the derivative calculation process.
-

Historical Context:

- Earlier, neural network training required manual derivative calculations.
- Advancements in autodiff have lowered the calculus requirements for neural network implementation.

Conclusion:

- Backpropagation and autodiff have revolutionized neural network training.
- The understanding of computation graphs and derivatives is crucial for grasping how neural networks learn.

之前我们在计算神经网络预测结果的时候我们采用了一种正向传播方法，我们从第一层开始正向一层一层进行计算，直到最后一层的$h_\theta(x)$。

现在，为了计算代价函数的偏导数$\frac{\partial}{\partial\Theta_{ij}^{(l)}}J(\Theta)$，我们需要采用一种反向传播算法，也就是首先计算最后一层的误差，然后再一层一层反向求出各层的误差，直到倒数第二层。 以一个例子来说明反向传播算法。

假设我们的训练集只有一个实例$\left(x^{(1)},y^{(1)}\right)$，我们的神经网络是一个四层的神经网络，其中$K=4$，$S_L=4$，$L=4$：

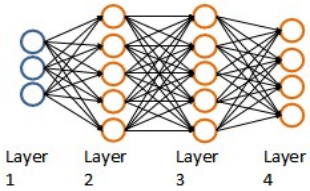
8.2 Forward Propagation：

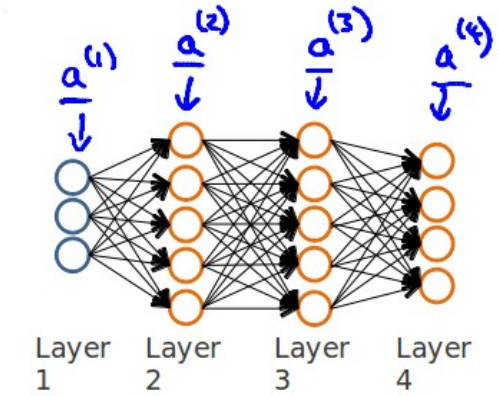
Gradient computation

Given one training example (x, y):

Forward propagation:

$$a^{(1)} = x$$
$$z^{(2)} = \Theta^{(1)}a^{(1)}$$
$$a^{(2)} = g(z^{(2)}) \quad (\text{add } a_0^{(2)})$$
$$z^{(3)} = \Theta^{(2)}a^{(2)}$$
$$a^{(3)} = g(z^{(3)}) \quad (\text{add } a_0^{(3)})$$
$$z^{(4)} = \Theta^{(3)}a^{(3)}$$
$$a^{(4)} = h_\Theta(x) = g(z^{(4)})$$





我们从最后一层的误差开始计算，误差是激活单元的预测（$a_k^{(4)}$）与实际值（y^k）之间的误差，（$k=1:k$）。

我们用δ来表示误差，则：$\delta^{(4)}=a^{(4)}-y$

我们利用这个误差值来计算前一层的误差：$\delta^{(3)} = \left(\Theta^{(3)}\right)^T \delta^{(4)} * g'\left(z^{(3)}\right)$ 其中 $g'(z^{(3)})$是 S 形函数的导数，$g'(z^{(3)}) = a^{(3)} * (1 - a^{(3)})$。而$(\theta^{(3)})^T \delta^{(4)}$则是权重导致的误差的和。下一步是继续计算第二层的误差：$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} * g'(z^{(2)})$

因为第一层是输入变量，不存在误差。我们有了所有的误差的表达式后，便可以计算代价函数的偏导数了，假设$\lambda = 0$，即我们不做任何正则化处理时有： $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{l+1}$

重要的是清楚地知道上面式子中上下标的含义：

l 代表目前所计算的是第几层。

j 代表目前计算层中的激活单元的下标，也将是下一层的第j个输入变量的下标。

i 代表下一层中误差单元的下标，是受到权重矩阵中第i行影响的下一层中的误差单元的下标。

如果我们考虑正则化处理，并且我们的训练集是一个特征矩阵而非向量。在上面的特殊情况中，我们需要计算每一层的误差单元来计算代价函数的偏导数。在更为一般的情况中，我们同样需要计算每一层的误差单元，但是我们需要为整个训练集计算误差单元，此时的误差单元也是一个矩阵，我们用$\Delta_{ij}^{(l)}$来表示这个误差矩阵。第 l 层的第 i 个激活单元受到第 j 个参数影响而导致的误差。

我们的算法表示为：

for i=1:m {
set $a^{(i)}=x^{(i)}$
perform foward propagation to compute $a^{(l)}$ *for l=1,2,3...L*
Using $\delta^{(L)}=a^{(L)}-y^i$
perform back propagation to compute all previous layer error vector
$\Delta_{ij}^{(l)}:=\Delta_{ij}^{(l)}+a_j^{(l)}\delta_i^{l+1}$
}

即首先用正向传播方法计算出每一层的激活单元，利用训练集的结果与神经网络预测的结果求出最后一层的误差，然后利用该误差运用反向传播法计算出直至第二层的所有误差。

在求出了$\Delta_{ij}^{(l)}$之后，我们便可以计算代价函数的偏导数了，计算方法如下：

$D_{ij}^{(l)} := \frac{1}{m}\Delta_{ij}^{(l)} + \lambda\Theta_{ij}^{(l)} \quad if\ j \neq 0$

$D_{ij}^{(l)} := \frac{1}{m}\Delta_{ij}^{(l)} \quad if\ j = 0$

8.3 随机初始化 Random Initialization

任何优化算法都需要一些初始的参数。到目前为止我们都是初始所有参数为 0，这样的初始方法对于逻辑回归来说是可行的，但是对于神经网络来说是不可行的。如果我们令所有的初始参数都为 0，这将意味着

我们第二层的所有激活单元都会有相同的值。同理，如果我们初始所有的参数都为一个非 0 的数，结果也是一样的。

我们通常初始参数为正负ε之间的随机值，假设我们要随机初始一个尺寸为 10×11 的参数矩阵

8.4 综合起来 Putting It Together

小结一下使用神经网络时的步骤：

网络结构：第一件要做的事是选择网络结构，即决定选择多少层以及决定每层分别有多少个单元。

第一层的单元数即我们训练集的特征数量。

最后一层的单元数是我们训练集的结果的类的数量。

如果隐藏层数大于 1，确保每个隐藏层的单元个数相同，通常情况下隐藏层单元的个数越多越好。

我们真正要决定的是隐藏层的层数和每个中间层的单元数。

训练神经网络：

1. 参数的随机初始化
2. 利用正向传播方法计算所有的$h_\theta(x)$
3. 编写计算代价函数 J 的代码
4. 利用反向传播方法计算所有偏导数
5. 利用数值检验方法检验这些偏导数
6. 使用优化算法来最小化代价函数

8.5 自主驾驶 Autonomous Driving

ALVINN (Autonomous Land Vehicle In a Neural Network)是一个基于神经网络的智能系统，通过观察人类的驾驶来学习驾驶，**ALVINN** 能够控制 **NavLab**，装在一辆改装版军用悍马，这辆悍马装载了传感器、计算机和驱动器用来进行自动驾驶的导航试验。实现 **ALVINN** 功能的第一步，是对它进行训练，也就是训练一个人驾驶汽车。

然后让 **ALVINN** 观看，**ALVINN** 每两秒将前方的路况图生成一张数字化图片，并且记录驾驶者的驾驶方向，得到的训练集图片被压缩为 30x32 像素，并且作为输入提供给 **ALVINN** 的三层神经网络，通过使用反向传播学习算法，**ALVINN** 会训练得到一个与人类驾驶员操纵方向基本相近的结果。一开始，我们的网络选择

出的方向是随机的，大约经过两分钟的训练后，我们的神经网络便能够准确地模拟人类驾驶者的驾驶方向，对其他道路类型，也重复进行这个训练过程，当网络被训练完成后，操作者就可按下运行按钮，车辆便开始行驶了。

每秒钟 **ALVINN** 生成 12 次数字化图片，并且将图像传送给神经网络进行训练，多个神经网络同时工作，每一个网络都生成一个行驶方向，以及一个预测自信度的参数，预测自信度最高的那个神经网络得到的行驶方向。比如这里，在这条单行道上训练出的网络将被最终用于控制车辆方向，车辆前方突然出现了一个交叉十字路口，当车辆到达这个十字路口时，我们单行道网络对应的自信度骤减，当它穿过这个十字路口时，前方的双车道将进入其视线，双车道网络的自信度便开始上升，当它的自信度上升时，双车道的网络，将被选择来控制行驶方向，车辆将被安全地引导进入双车道路。

这就是基于神经网络的自动驾驶技术。当然，我们还有很多更加先进的试验来实现自动驾驶技术。在美国，欧洲等一些国家和地区，他们提供了一些比这个方法更加稳定的驾驶控制技术。但我认为，使用这样一个简单的基于反向传播的神经网络，训练出如此强大的自动驾驶汽车，的确是一次令人惊讶的成就。