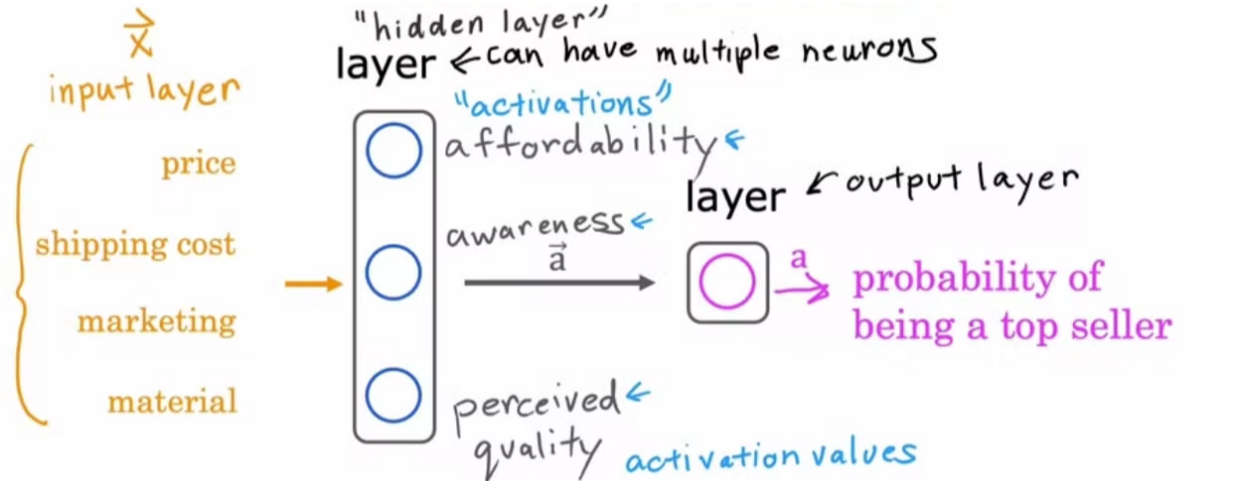
Advanced Learning Algorithms

Origins: algorithms that try to mimic the brain

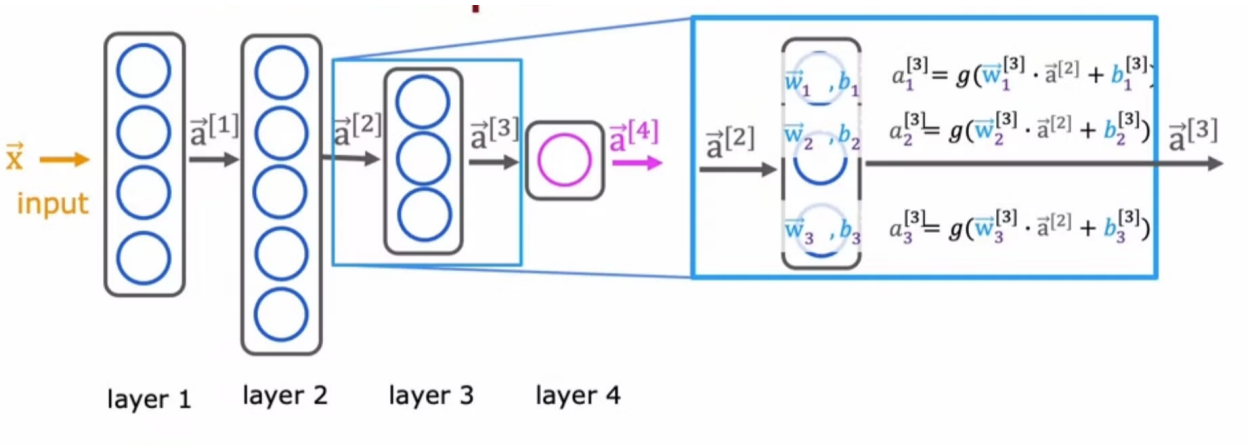
Ressurge from around 2005 under the term *deep learning*, as the amount of data rapidly increased. The rise of GPUs was also a major force.

Applications: speech recognition → computer vision → NLP …

Example: predict if t-shirt will become a top-seller



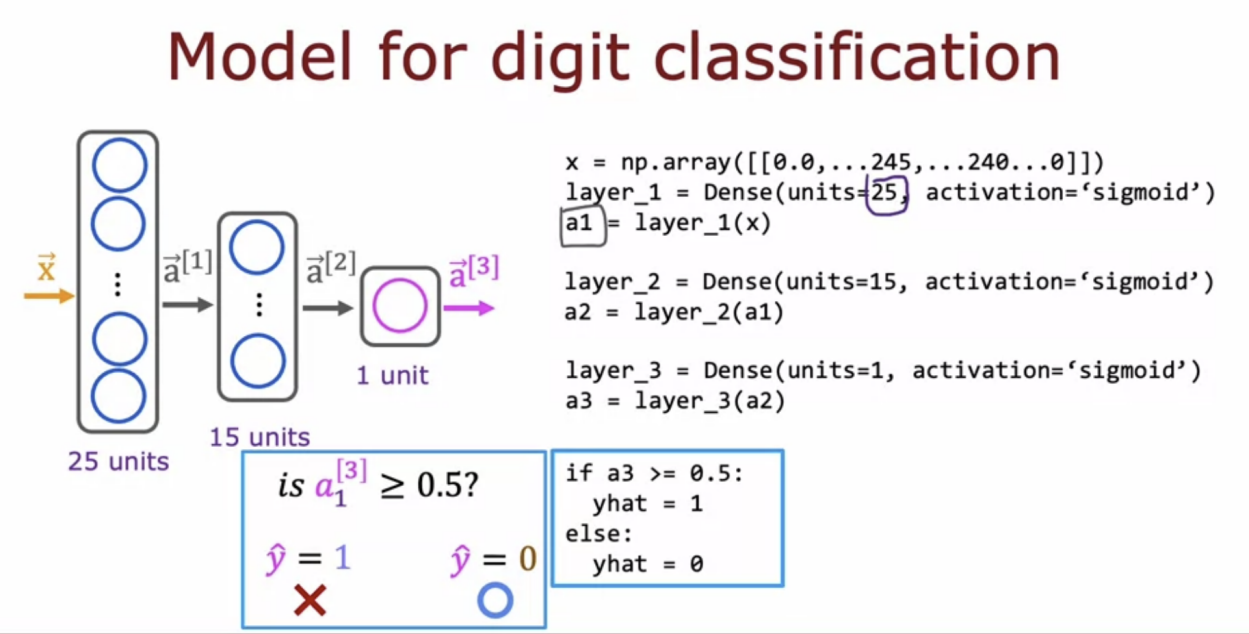
It's like a version of logistic regression that learns its own features layer by layer. That replaces manual feature engineering.

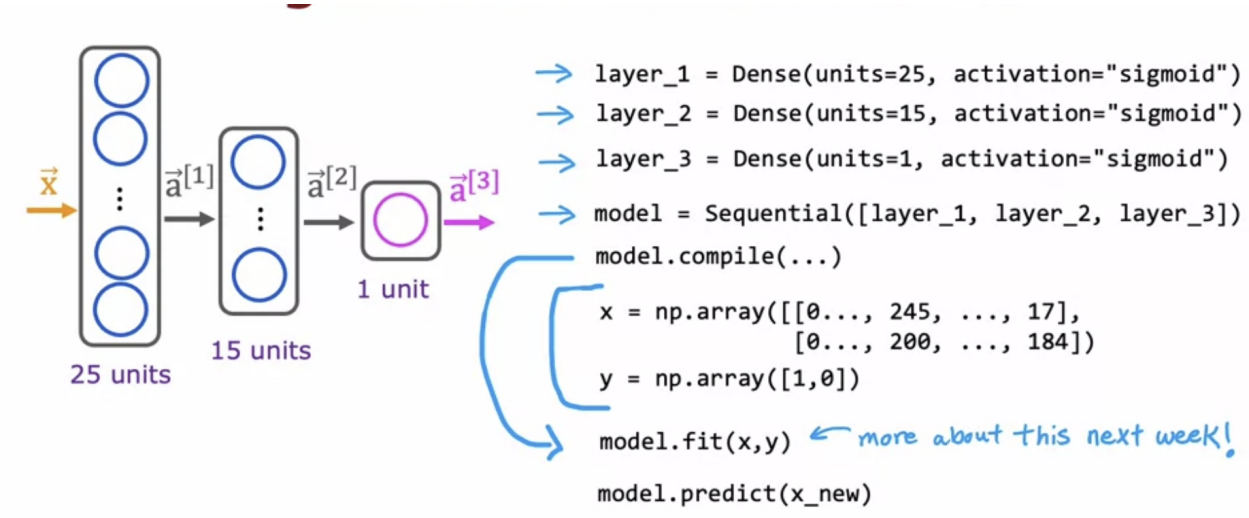


The activation function g could be the sigmoid function or other functions.

Forward propagation to make a prediction (inference).

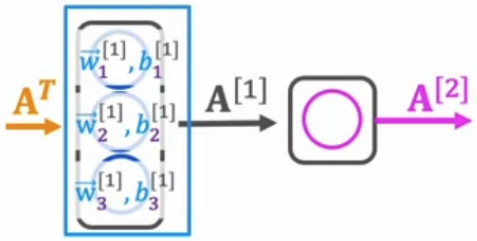
TensorFlow





Vectorized implementation makes neural networks run much faster!

Dense layer vectorized




```
A
AT = np.array([[200, 17]])
W = np.array([[1, -3, 5],
              [-2, 4, -6]])
b = np.array([[-1, 1, 2]])
          a_in
def dense(AT,W,b):
  z = np.matmul(AT,W) + b
  a_out = g(z)
  return a_out
```

[[1,0,1]]

$\mathbf{A}^T = [200 \quad 17]$ (1 × 2)

$\mathbf{W} = \begin{bmatrix} 1 & -3 & 5 \\ -2 & 4 & -6 \end{bmatrix}$ (2 × 3)

$\mathbf{B} = [-1 \quad 1 \quad 2]$ (1 × 3)

$\mathbf{Z} = \mathbf{A}^T\mathbf{W} + \mathbf{B}$

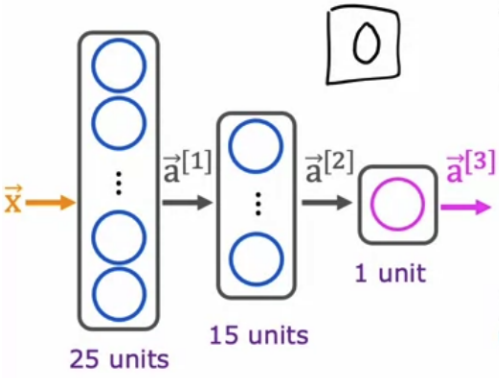
$[165 \quad -531 \quad 900]$

$z_1^{[1]} \quad z_2^{[1]} \quad z_3^{[1]}$

$\mathbf{A} = g(\mathbf{Z})$

$[1 \quad 0 \quad 1]$

Train a Neural Network in TensorFlow



0

```
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'),
                    ])
from tensorflow.keras.losses import
BinaryCrossentropy
   model.compile(loss=BinaryCrossentropy())
   model.fit(X,Y,epochs=100)
```

① ② ③

epochs: number of steps in gradient descent

Given set of (x,y) examples

How to build and train this in code?

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)
```

2. Loss and cost functions

handwritten 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

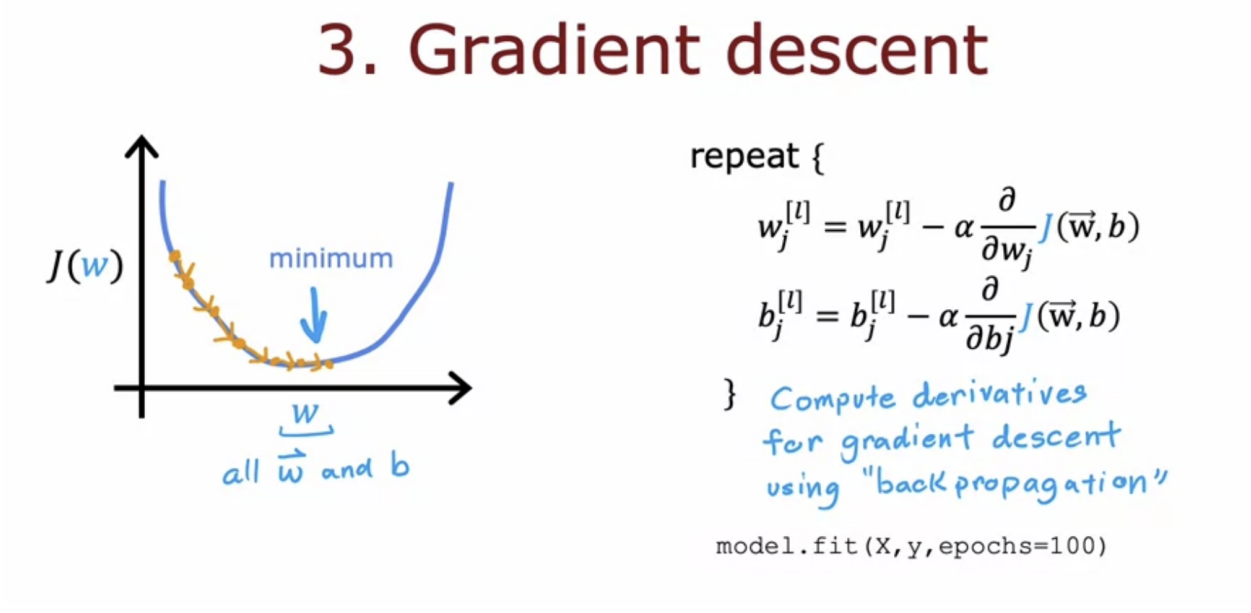
logistic loss

also known as binary cross entropy

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

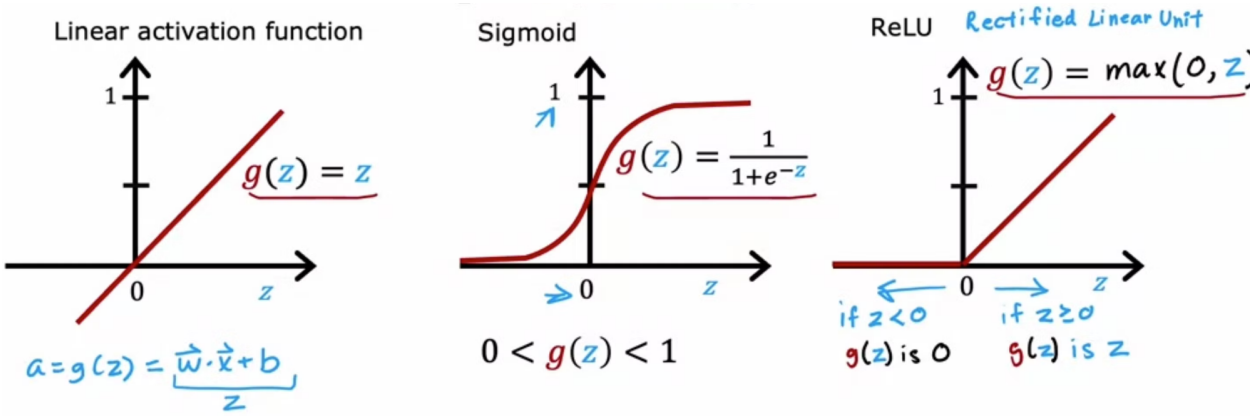
```
from tensorflow.keras.losses import
  BinaryCrossentropy
```

Keras

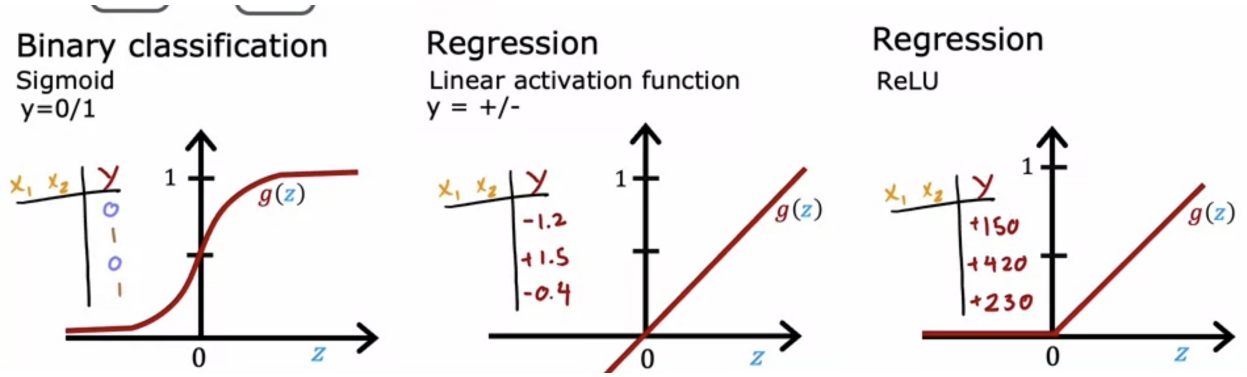


Another popular activation function is **ReLU** (rectified linear unit).

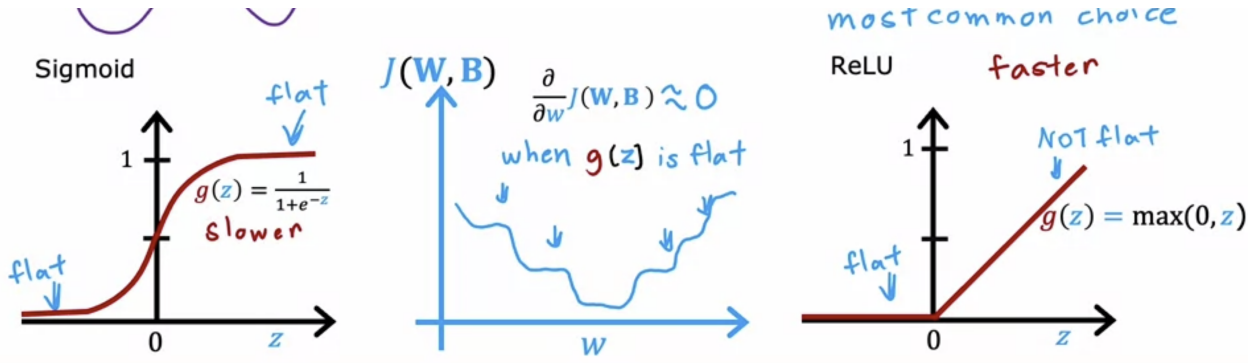
$$g(z) = max(0, z)$$



We choose the activation function for the output layer depending on the nature of the label y we're trying to predict.



For the hidden layers, ReLU is the most commonly used. It's faster to compute and makes gradient descent faster. This is because ReLU is flat to the left only, whereas sigmoid is flat to the left and right.



Why don't we use linear activation function in the entire neural network? Because that would be equivalent to a single linear regression. And if the hidden layers are all linear and the output layer is sigmoid, then the neural network is equivalent to logistic regression.

The "off" or disable feature of the ReLU activation enables models to stitch together linear segments to model complex non-linear functions.

Multiclass classification

Softmax regression (4 possible outputs) $y = 1, 2, 3, 4$

$z_1 = \vec{w}_1 \cdot \vec{x} + b_1$

$$a_1 = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}} = P(y = 1|\vec{x})$$

$z_2 = \vec{w}_2 \cdot \vec{x} + b_2$

$$a_2 = \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}} = P(y = 2|\vec{x})$$

$z_3 = \vec{w}_3 \cdot \vec{x} + b_3$

$$a_3 = \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}} = P(y = 3|\vec{x})$$

$z_4 = \vec{w}_4 \cdot \vec{x} + b_4$

$$a_4 = \frac{e^{z_4}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}} = P(y = 4|\vec{x})$$

Softmax regression (N possible outputs) $y = 1, 2, 3, \ldots, N$

$$z_j = \vec{w}_j \cdot \vec{x} + b_j \quad j = 1, \ldots, N$$

parameters $w_1, w_2, \ldots, w_N$ $b_1, b_2, \ldots, b_N$

$$a_j = \frac{e^{z_j}}{\sum_{k=1}^{N} e^{z_k}} = P(y = j|\vec{x})$$

Cost

Logistic regression

$$z = \vec{w} \cdot \vec{x} + b$$

$$a_1 = g(z) = \frac{1}{1+e^{-z}} = P(y=1|\vec{x})$$

$$a_2 = 1 - a_1 = P(y=0|\vec{x})$$

$$loss = -y \log a_1 - (1-y) \log(1-a_1)$$

if $y=1$ — if $y=0$

$$J(\vec{w}, b) = \text{average loss}$$

Softmax regression

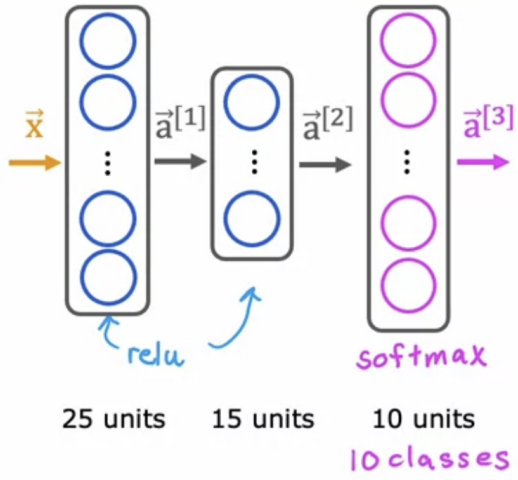
$$a_1 = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + \cdots + e^{z_N}} = P(y=1|\vec{x})$$

$$\vdots$$

$$a_N = \frac{e^{z_N}}{e^{z_1} + e^{z_2} + \cdots + e^{z_N}} = P(y=N|\vec{x})$$

$$loss(a_1, \ldots, a_N, y) = \begin{cases} -\log a_1 & \text{if } y=1 \\ -\log a_2 & \text{if } y=2 \\ \vdots & \\ -\log a_N & \text{if } y=N \end{cases}$$

Neural Network with Softmax output



$$z_1^{[3]} = \vec{w}_1^{[3]} \cdot \vec{a}^{[2]} + b_1^{[3]} \qquad a_1^{[3]} = \frac{e^{z_1^{[3]}}}{e^{z_1^{[3]}} + \cdots + e^{z_{10}^{[3]}}} = P(y=1|\vec{x})$$

$$\vdots$$

$$z_{10}^{[3]} = \vec{w}_{10}^{[3]} \cdot \vec{a}^{[2]} + b_{10}^{[3]} \qquad a_{10}^{[3]} = \frac{e^{z_{10}^{[3]}}}{e^{z_1^{[3]}} + \cdots + e^{z_{10}^{[3]}}} = P(y=10|\vec{x})$$

MNIST with softmax

1. specify the model

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

```
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='softmax')
                  ])
```

2. specify loss and cost

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

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

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

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

Numerical Roundoff Errors

More numerically accurate implementation of logistic loss:

Logistic regression:

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

Original loss

$$loss = -y\log(a) - (1-y)\log(1-a)$$

More accurate loss (in code)

$$loss = -y\log\left(\frac{1}{1+e^{-z}}\right) - (1-y)\log\left(1-\frac{1}{1+e^{-z}}\right)$$

```
model = Sequential([
  Dense(units=25, activation='relu'),
  Dense(units=15, activation='relu'),
  Dense(units=1, activation='sigmoid')
                ])
model.compile(loss=BinaryCrossEntropy() )
```

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

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

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

```
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)
```

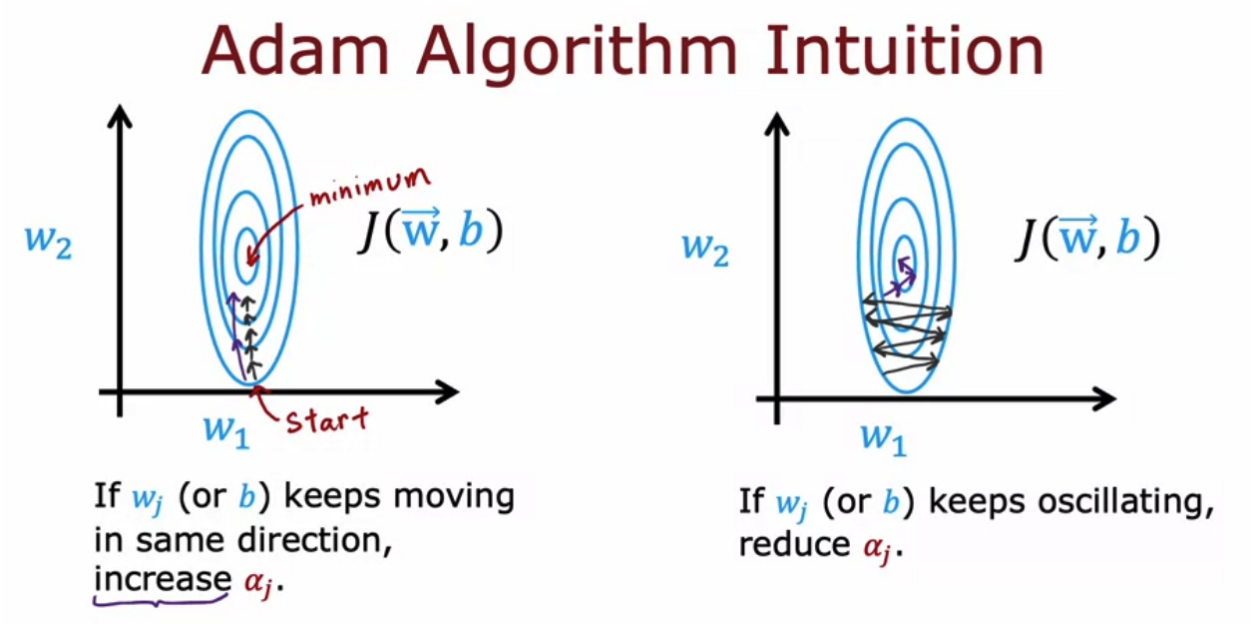
predict

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

Multi-label classification: single input maps to potentially more than one label. One can build multiple NNs to detect each label or have an output layer with multiple nodes, one for each label, using a sigmoid activation function.

Adam algorithm is an advance of gradient descent, increasing the learning rate if the steps are going in the same direction or decreasing the learning rate if the

steps are going all over the place. It uses different learning rates for each parameter of the model. It's become the standard method.



MNIST Adam

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')
])
```

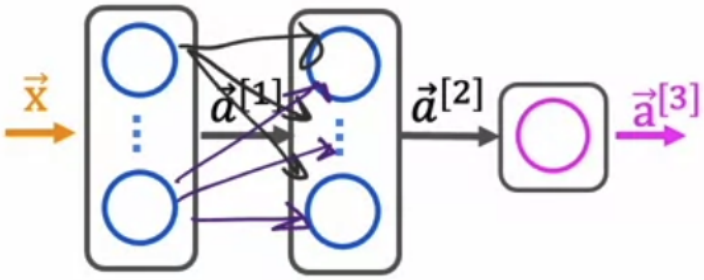
compile

```
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)
```

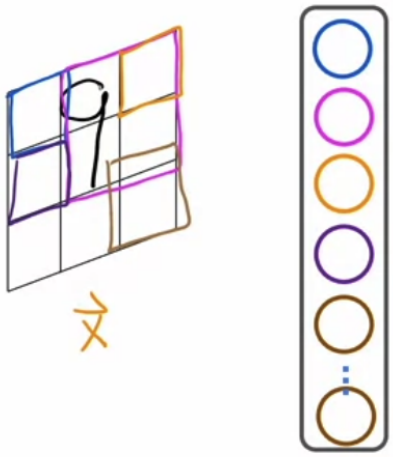
Dense Layer



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

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

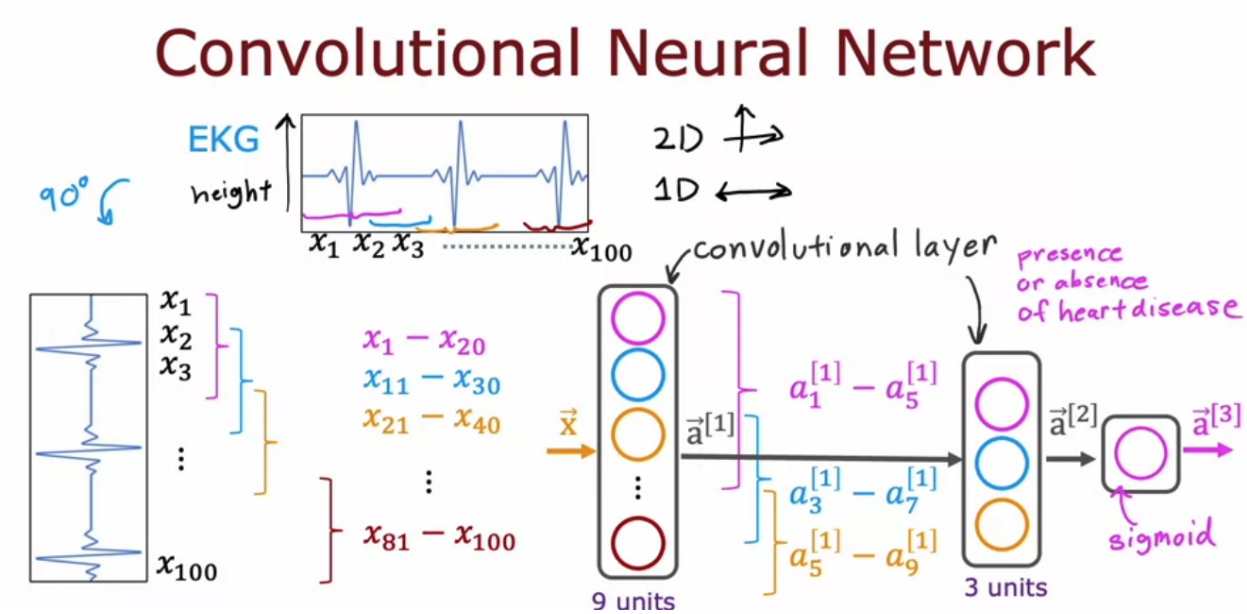
Convolutional Layer



Each neuron only looks at part of the previous layer's outputs.

Why?

- Faster computation
- Need less training data (less prone to overfitting)


Convolutional Neural Network
EKG
height
90°
2D
1D
x_1 x_2 x_3 ..... x_{100}
convolutional layer
presence or absence of heart disease
x_1
x_2
x_3
x_{100}
$x_1 - x_{20}$
$x_{11} - x_{30}$
$x_{21} - x_{40}$
$x_{81} - x_{100}$
$\vec{x}$
$\vec{a}^{[1]}$
$a_1^{[1]} - a_5^{[1]}$
$a_3^{[1]} - a_7^{[1]}$
$a_5^{[1]} - a_9^{[1]}$
$\vec{a}^{[2]}$
$\vec{a}^{[3]}$
sigmoid
9 units
3 units