

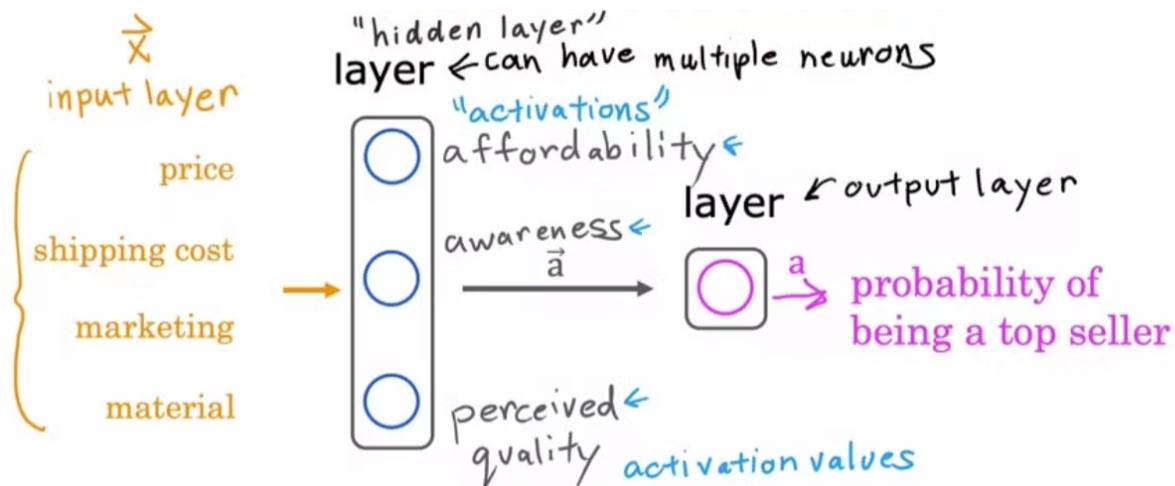
Advanced Learning Algorithms

Origins: algorithms that try to mimic the brain

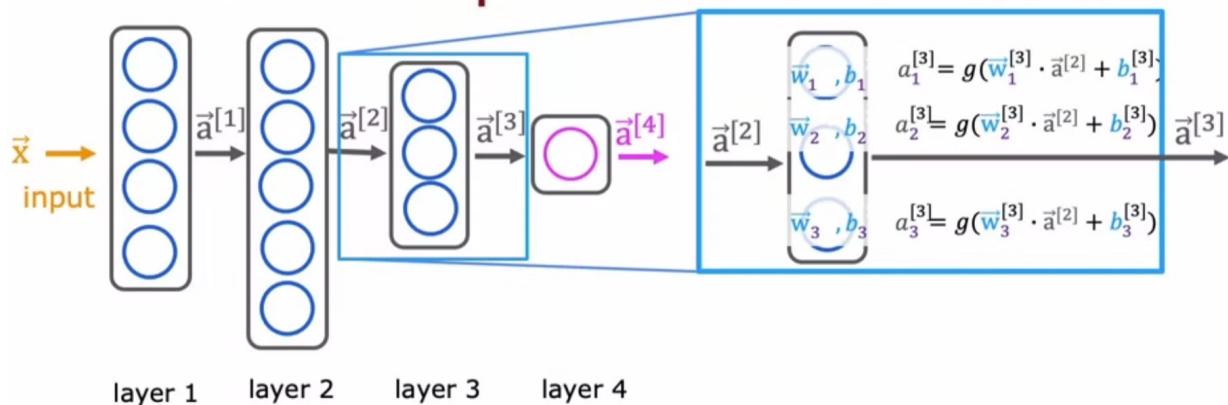
Ressurge from around 2005 under the term *deep learning*, as the amount of data rapidly increased. The rise of GPUs was also a major force.

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.

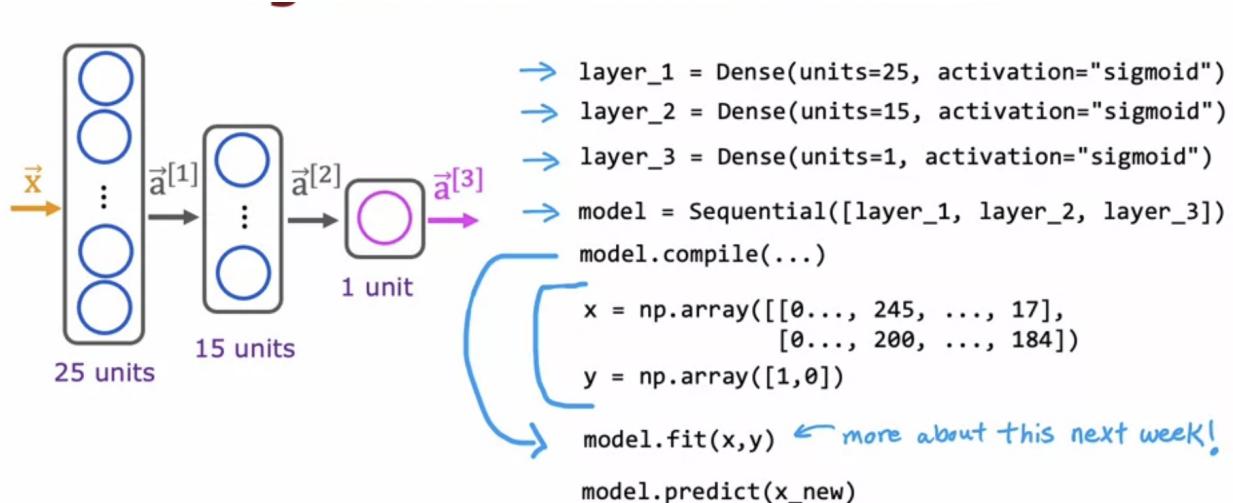
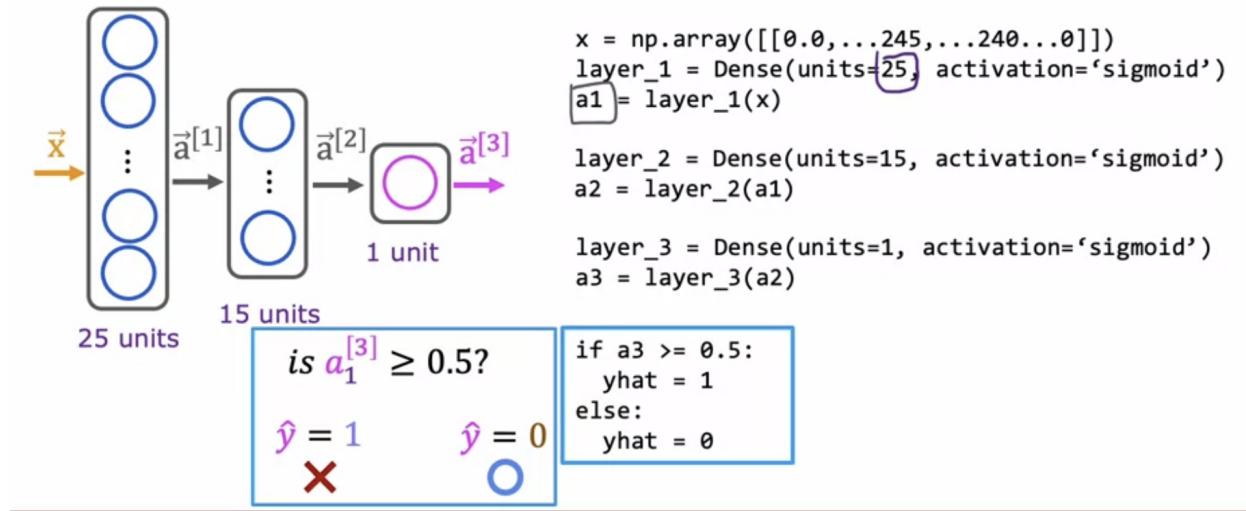


The activation function g could be the sigmoid function or other functions.

Forward propagation to make a prediction (inference).

TensorFlow

Model for digit classification



Vectorized implementation makes neural networks run much faster!

Dense layer vectorized

$A^T = [200 \quad 17]$

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

$B = [-1 \quad 1 \quad 2]$

$Z = A^T W + B$

$A = g(Z)$

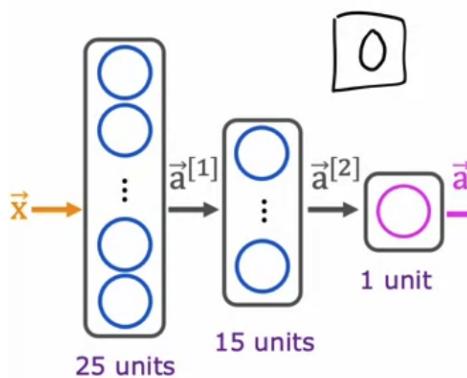
$A^2 = \begin{bmatrix} 165 & -531 & 900 \end{bmatrix}$

```

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]]

```

Train a Neural Network in TensorFlow



```

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()) (2)
model.fit(X, Y, epochs=100) (3)
    epochs: number of steps
    in gradient descent

```

Model Training Steps

TensorFlow

①	specify how to compute output given input x and parameters w, b (define model) $f_{\bar{w}, \bar{b}}(\vec{x}) = ?$	logistic regression $z = np.dot(w, x) + b$ $f_x = 1 / (1 + np.exp(-z))$	neural network $\text{model} = Sequential([\text{Dense}(...), \text{Dense}(...), \text{Dense}(...)])$
②	specify loss and cost $L(f_{\bar{w}, \bar{b}}(\vec{x}), y)$ 1 example $J(\bar{w}, \bar{b}) = \frac{1}{m} \sum_{i=1}^m L(f_{\bar{w}, \bar{b}}(\vec{x}^{(i)}), y^{(i)})$	logistic loss $\text{loss} = -y * np.log(f_x) - (1-y) * np.log(1-f_x)$	binary cross entropy $\text{model.compile(loss=BinaryCrossentropy())}$
③	Train on data to minimize $J(\bar{w}, \bar{b})$	$w = w - \alpha * \frac{\partial J}{\partial w}$ $b = b - \alpha * \frac{\partial J}{\partial b}$	$\text{model.fit(X, y, epochs=100)}$

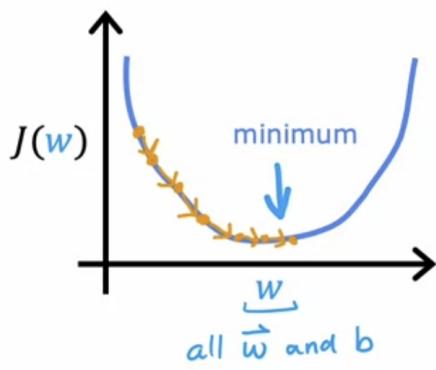
2. Loss and cost functions

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

3. Gradient descent



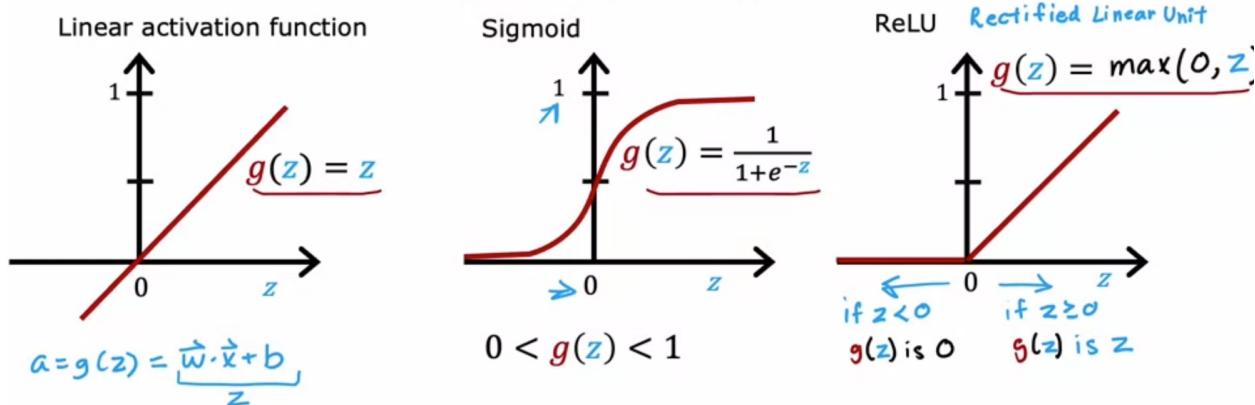
```
repeat {
     $w_j^{[l]} = w_j^{[l]} - \alpha \frac{\partial}{\partial w_j} J(\vec{w}, b)$ 
     $b_j^{[l]} = b_j^{[l]} - \alpha \frac{\partial}{\partial b_j} J(\vec{w}, b)$ 
}
```

} Compute derivatives for gradient descent using "back propagation"

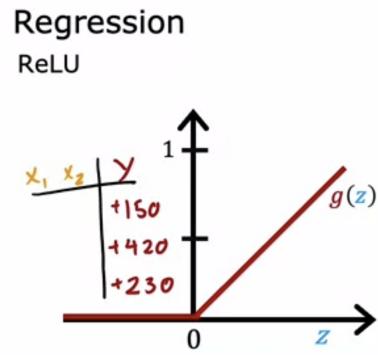
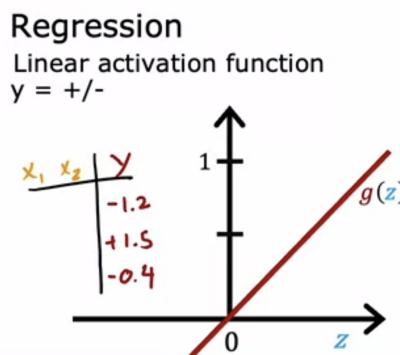
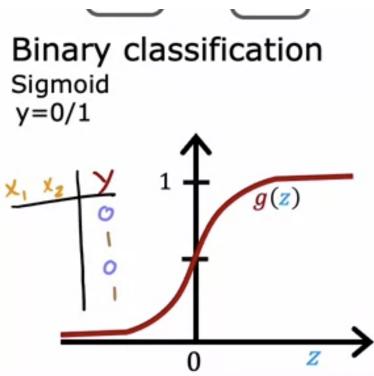
```
model.fit(X, y, epochs=100)
```

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

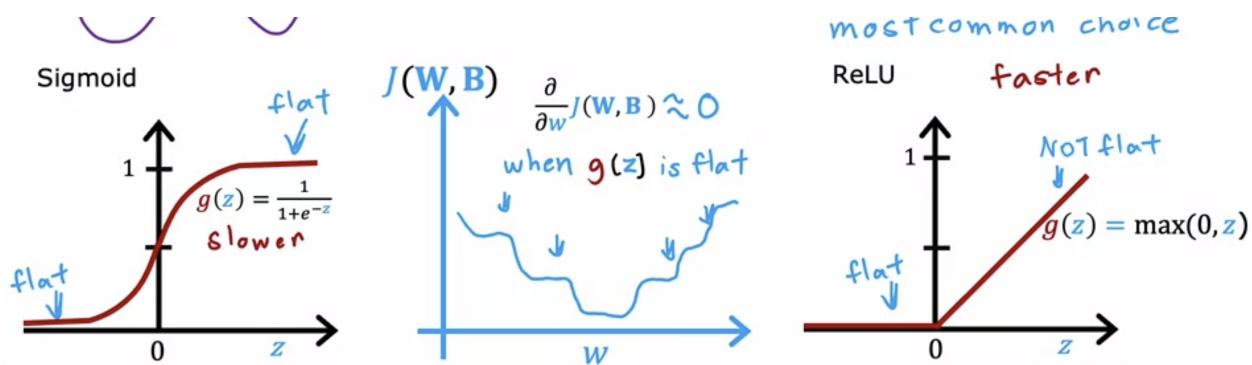
$$g(z) = \max(0, z)$$



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) $y=1, 2, 3, 4$

$$\text{X } z_1 = \vec{w}_1 \cdot \vec{x} + b_1 \quad a_1 = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}}$$

$\text{X} \quad \text{O} \quad \square \quad \Delta$
 $= P(y = 1|\vec{x})$

$$\text{O } z_2 = \vec{w}_2 \cdot \vec{x} + b_2 \quad a_2 = \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}}$$

$= P(y = 2|\vec{x})$

$$\square z_3 = \vec{w}_3 \cdot \vec{x} + b_3 \quad a_3 = \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}}$$

$= P(y = 3|\vec{x})$

$$\Delta z_4 = \vec{w}_4 \cdot \vec{x} + b_4 \quad a_4 = \frac{e^{z_4}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}}$$

$= P(y = 4|\vec{x})$

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

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

parameters w_1, w_2, \dots, w_N
 $e^{z_j} \quad b_1, b_2, \dots, b_N$

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

Cost

Logistic regression

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

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

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

$$\text{loss} = -y \underbrace{\log a_1}_{\text{if } y=1} - (1-y) \underbrace{\log(1-a_1)}_{\text{if } y=0}$$

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

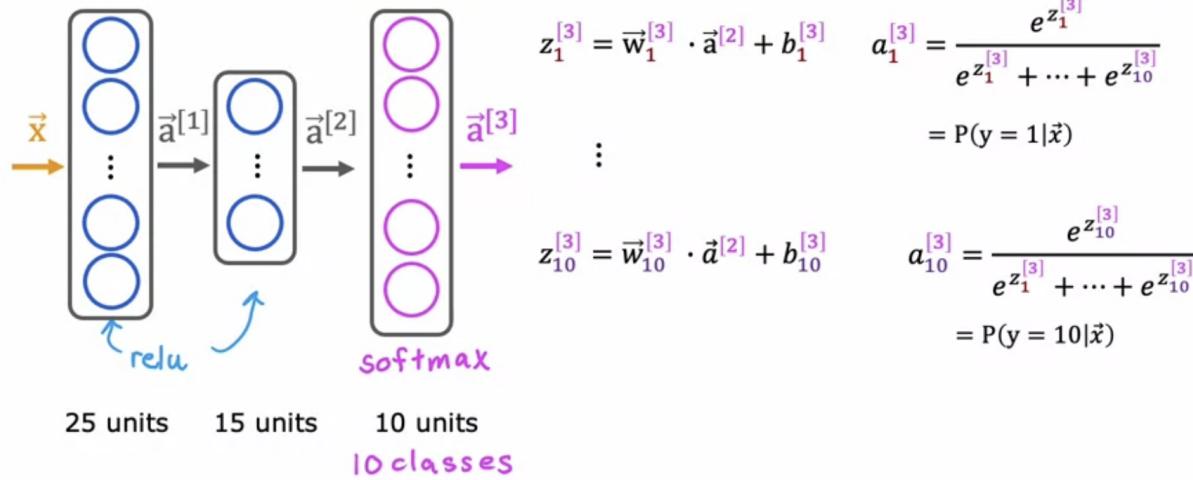
Softmax regression

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

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

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

Neural Network with Softmax output



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

② specify loss and cost

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

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

Numerical Roundoff Errors

More numerically accurate implementation of logistic loss:

$$| + \frac{1}{10,000} \quad | - \frac{1}{10,000}$$

Logistic regression:

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

Original loss

$$\text{loss} = -y \log(a) - (1-y) \log(1-a)$$

model.compile(loss=BinaryCrossEntropy()) ↴

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

More accurate loss (in code)

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

More numerically accurate implementation of softmax

Softmax regression

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

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

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

model.compile(loss=SparseCategoricalCrossEntropy())

```

More Accurate

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

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

MNIST (more numerically accurate)

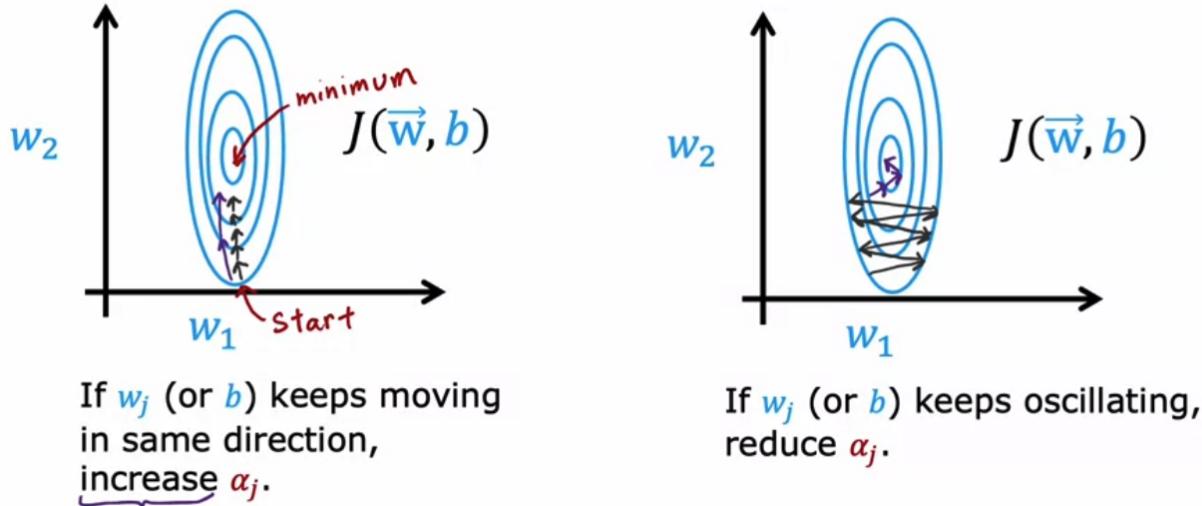
```
model import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
model = Sequential([
    Dense(units=25, activation='relu'),
    Dense(units=15, activation='relu'),
    Dense(units=10, activation='linear') ])
loss from tensorflow.keras.losses import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

Adam Algorithm Intuition



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')
])
```

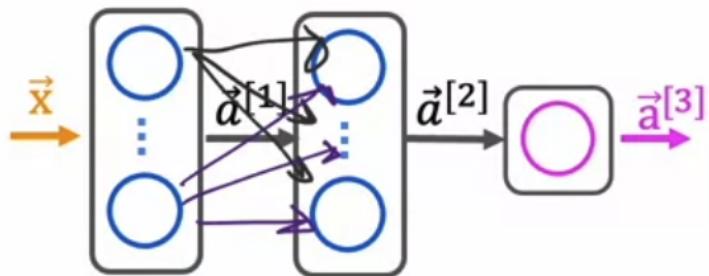
compile

```
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)
```

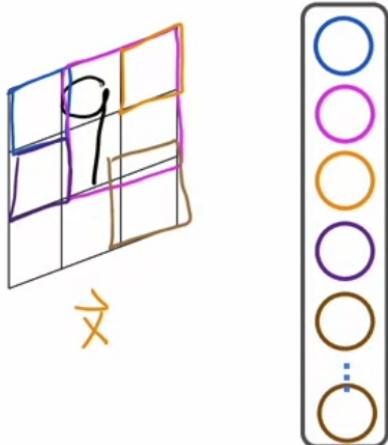
Dense Layer



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

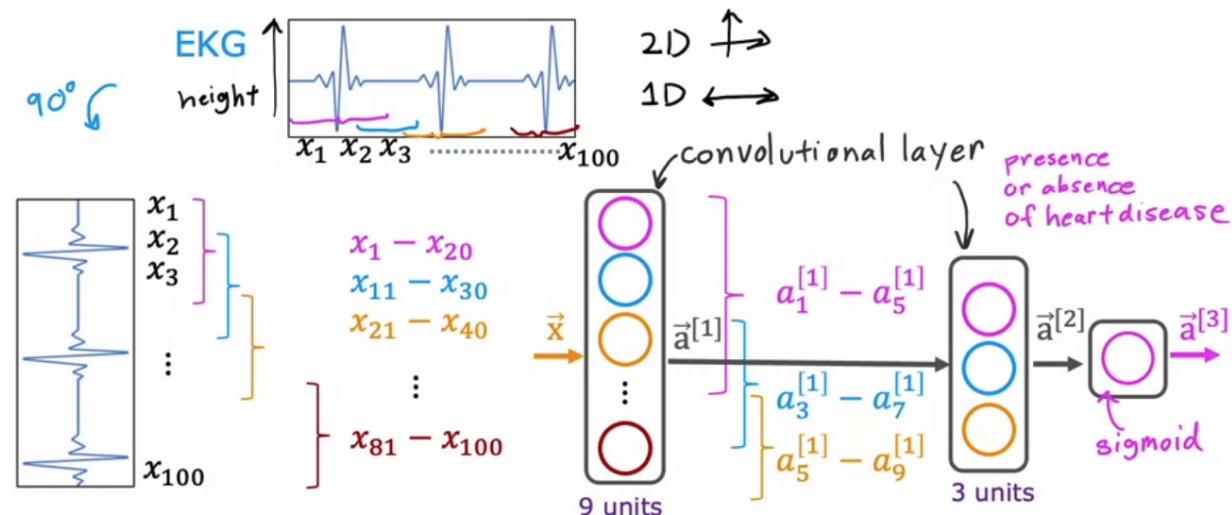


Each neuron only looks at part of the previous layer's outputs.

Why?

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

Convolutional Neural Network



Evaluating a Model

Use test set to see how well the model generalizes to unseen data. Training error being small does not guarantee a good model.

Train/test procedure for linear regression (with squared error cost)

Fit parameters by minimizing cost function $J(\vec{w}, b)$

$$\rightarrow J(\vec{w}, b) = \left[\frac{1}{2m_{train}} \sum_{i=1}^{m_{train}} (f_{\vec{w}, b}(\vec{x}^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m_{train}} \sum_{j=1}^n w_j^2 \right]$$

Compute test error:

$$J_{test}(\vec{w}, b) = \frac{1}{2m_{test}} \left[\sum_{i=1}^{m_{test}} (f_{\vec{w}, b}(\vec{x}_{test}^{(i)}) - y_{test}^{(i)})^2 \right] \quad \cancel{\sum_{j=1}^n w_j^2}$$

Compute training error:

$$J_{train}(\vec{w}, b) = \frac{1}{2m_{train}} \left[\sum_{i=1}^{m_{train}} (f_{\vec{w}, b}(\vec{x}_{train}^{(i)}) - y_{train}^{(i)})^2 \right]$$

Train/test procedure for classification problem

0 / 1

Fit parameters by minimizing $J(\vec{w}, b)$ to find $\vec{w} \cdot b$

E.g.,

$$J(\vec{w}, b) = -\frac{1}{m_{train}} \sum_{i=1}^{m_{train}} \underbrace{\left[y^{(i)} \log(f_{\vec{w}, b}(\vec{x}^{(i)})) + (1 - y^{(i)}) \log(1 - f_{\vec{w}, b}(\vec{x}^{(i)})) \right]}_{+ \frac{\lambda}{2m_{train}} \sum_{j=1}^n w_j^2}$$

Compute test error:

$$J_{test}(\vec{w}, b) = -\frac{1}{m_{test}} \sum_{i=1}^{m_{test}} \underbrace{\left[y_{test}^{(i)} \log(f_{\vec{w}, b}(\vec{x}_{test}^{(i)})) + (1 - y_{test}^{(i)}) \log(1 - f_{\vec{w}, b}(\vec{x}_{test}^{(i)})) \right]}$$

Compute train error:

$$J_{train}(\vec{w}, b) = -\frac{1}{m_{train}} \sum_{i=1}^{m_{train}} \left[y_{train}^{(i)} \log(f_{\vec{w}, b}(\vec{x}_{train}^{(i)})) + (1 - y_{train}^{(i)}) \log(1 - f_{\vec{w}, b}(\vec{x}_{train}^{(i)})) \right]$$

Model selection (choosing a model)

- $d=1$ 1. $f_{\vec{w}, b}(\vec{x}) = w_1 x + b \rightarrow w^{<1>} , b^{<1>} \rightarrow J_{test}(w^{<1>} , b^{<1>})$
- $d=2$ 2. $f_{\vec{w}, b}(\vec{x}) = w_1 x + w_2 x^2 + b \rightarrow w^{<2>} , b^{<2>} \rightarrow J_{test}(w^{<2>} , b^{<2>})$
- $d=3$ 3. $f_{\vec{w}, b}(\vec{x}) = w_1 x + w_2 x^2 + w_3 x^3 + b \rightarrow w^{<3>} , b^{<3>} \rightarrow J_{test}(w^{<3>} , b^{<3>})$
- \vdots
- $d=10$ 10. $f_{\vec{w}, b}(\vec{x}) = w_1 x + w_2 x^2 + \dots + w_{10} x^{10} + b \rightarrow J_{test}(w^{<10>} , b^{<10>})$

Choose $w_1 x + \dots + w_5 x^5 + b \quad d=5 \quad J_{test}(w^{<5>} , b^{<5>})$

How well does the model perform? Report test set error $J_{test}(w^{<5>} , b^{<5>})$?

The problem: $J_{test}(w^{<5>} , b^{<5>})$ is likely to be an optimistic estimate of generalization error (ie. $J_{test}(w^{<5>} , b^{<5>}) <$ generalization error). Because an extra parameter d (degree of polynomial) was chosen using the test set.

w, b are overly optimistic estimate of generalization error on training data.

Training/cross validation/test set

size	price			
2104	400			
1600	330			
2400	369			
1416	232			
3000	540			
1985	300			
1534	315			
1427	199			
1380	212			
1494	243			

training set
 60% → $(x^{(1)}, y^{(1)})$
 \vdots
 $(x^{(m_{train})}, y^{(m_{train})})$
 $M_{train} = 6$

cross validation → $(x_{cv}^{(1)}, y_{cv}^{(1)})$
 \vdots
 $(x_{cv}^{(m_{cv})}, y_{cv}^{(m_{cv})})$
 $M_{cv} = 2$

test set
 20% → $(x_{test}^{(1)}, y_{test}^{(1)})$
 \vdots
 $(x_{test}^{(m_{test})}, y_{test}^{(m_{test})})$
 $M_{test} = 2$

Cross validation is now used to test the validity of the different candidate models.

Training/cross validation/test set

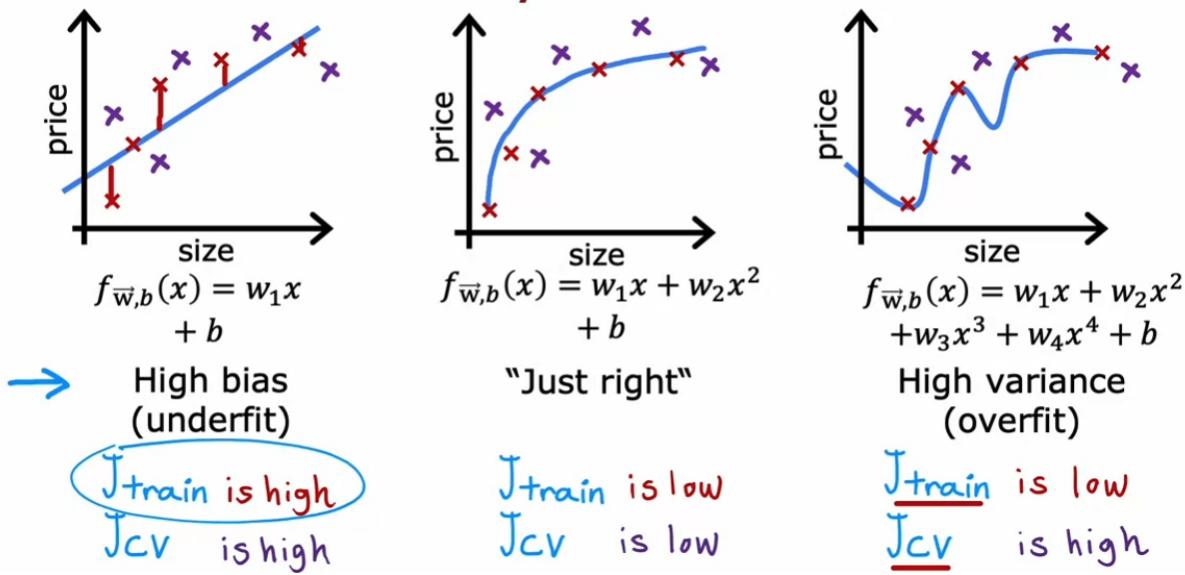
Training error: $J_{train}(\vec{w}, b) = \frac{1}{2m_{train}} \left[\sum_{i=1}^{m_{train}} (f_{\vec{w}, b}(\vec{x}^{(i)}) - y^{(i)})^2 \right]$

Cross validation error: $J_{cv}(\vec{w}, b) = \frac{1}{2m_{cv}} \left[\sum_{i=1}^{m_{cv}} (f_{\vec{w}, b}(\vec{x}_{cv}^{(i)}) - y_{cv}^{(i)})^2 \right]$ (validation error, dev error)

Test error: $J_{test}(\vec{w}, b) = \frac{1}{2m_{test}} \left[\sum_{i=1}^{m_{test}} (f_{\vec{w}, b}(\vec{x}_{test}^{(i)}) - y_{test}^{(i)})^2 \right]$

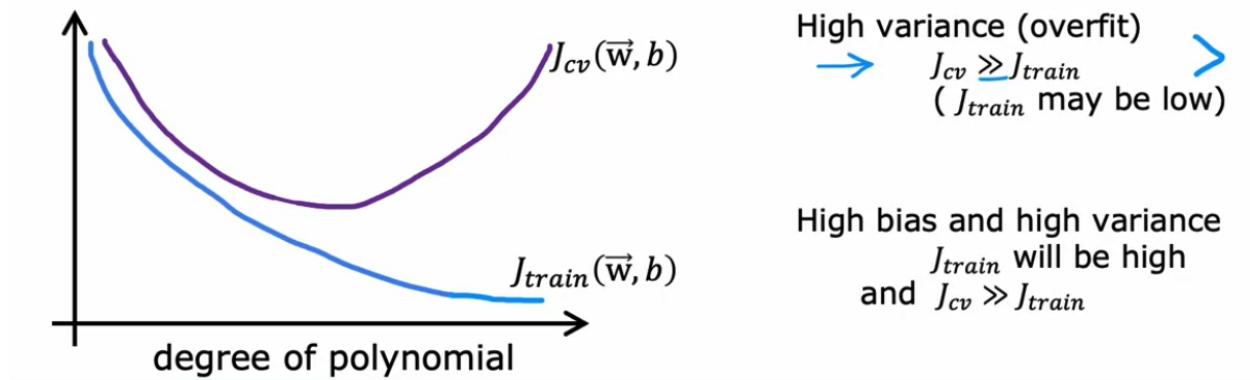
We pick the model with lowest cross validation error and estimate the generalization error using the test set! This procedure can also be applied to choose between different neural network architectures.

Bias/variance



Diagnosing bias and variance

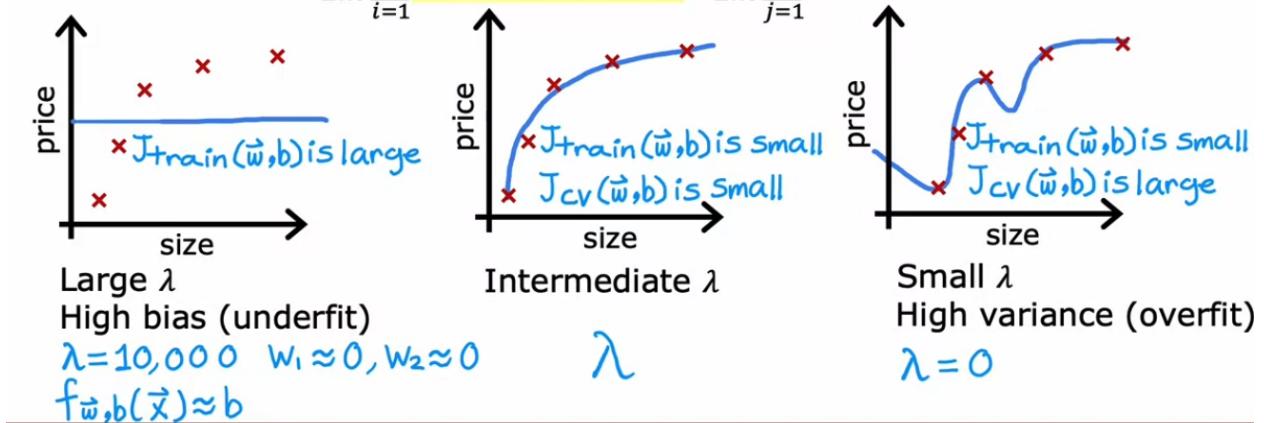
How do you tell if your algorithm has a bias or variance problem?



Linear regression with regularization

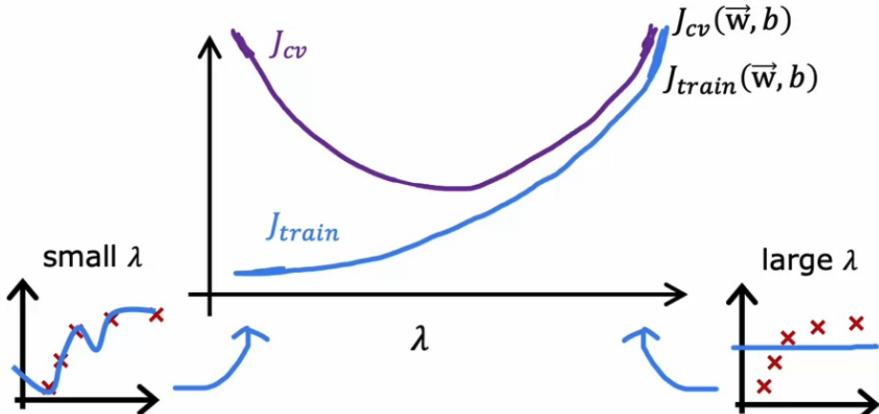
Model: $f_{\vec{w}, b}(x) = \underline{w_1}x + \underline{w_2}x^2 + \underline{w_3}x^3 + \underline{w_4}x^4 + b$

$$J(\vec{w}, b) = \frac{1}{2m} \sum_{i=1}^m (f_{\vec{w}, b}(\vec{x}^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m} \sum_{j=1}^n w_j^2$$



Bias and variance as a function of regularization parameter λ

$$J(\vec{w}, b) = \frac{1}{2m} \sum_{i=1}^m (f_{\vec{w}, b}(\vec{x}^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m} \sum_{j=1}^n w_j^2$$



Establishing a baseline level of performance

What is the level of error you can reasonably hope to get to?

- Human level performance
 - Competing algorithms performance
 - Guess based on experience

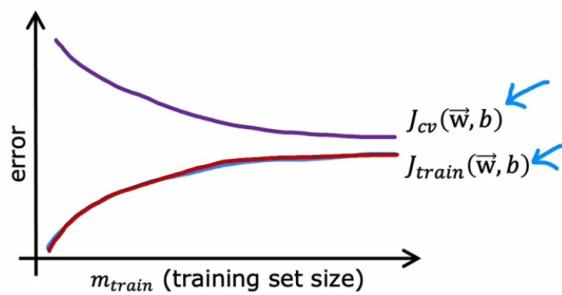
Bias/variance examples

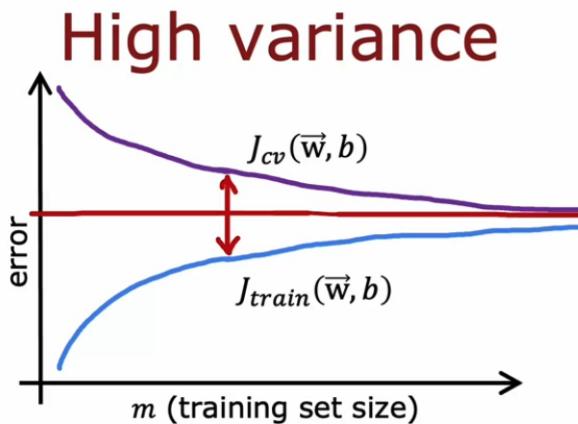
Baseline performance	:	10.6%	↑ 0.2%	10.6%	↑ 4.4%
Training error (J_{train})	:	10.8%	↓ 15.0%	15.0%	↓ 0.5%
Cross validation error (J_{cv})	:	14.8%	↓ 4.0%	15.5%	↓ 0.5%

Learning curves

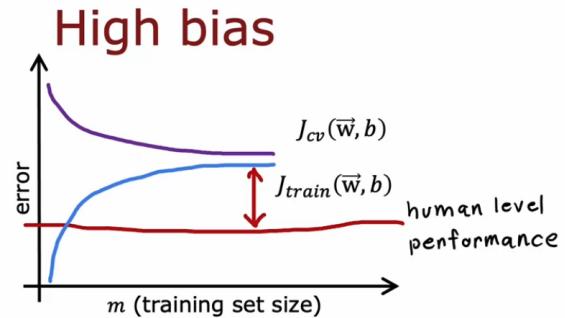
J_{train} = training error

J_{cv} = cross validation error





With high variance algorithm, adding more training data will likely help the performance.



With high bias algorithm, adding more training data will not help the performance much.

Debugging a learning algorithm

You've implemented regularized linear regression on housing prices

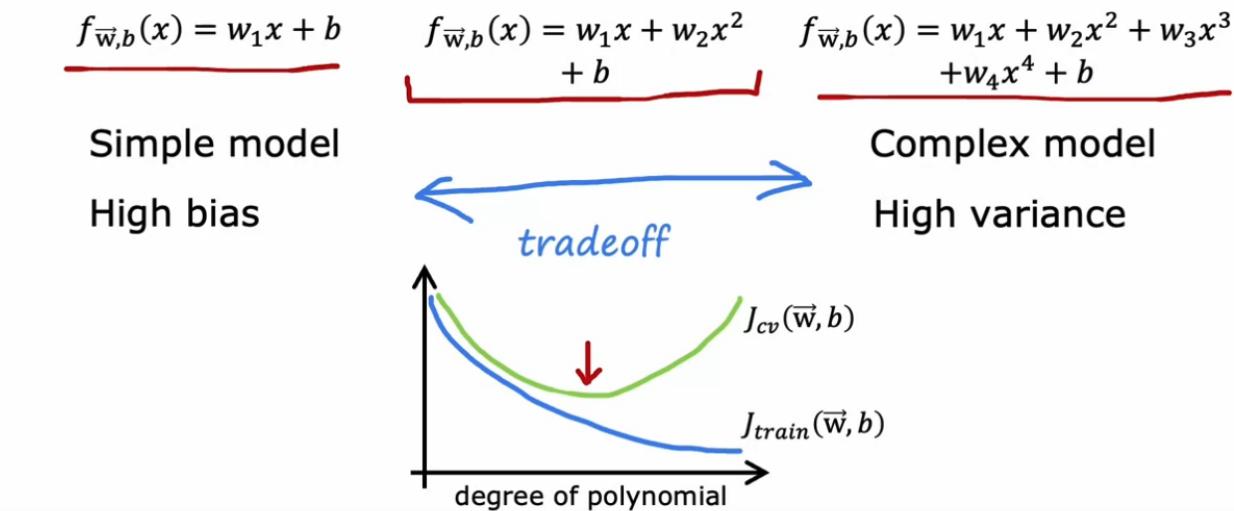
$$J(\vec{w}, b) = \frac{1}{2m} \sum_{i=1}^m (f_{\vec{w}, b}(\vec{x}^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m} \sum_{j=1}^n w_j^2$$

But it makes unacceptably large errors in predictions. What do you try next?

- Get more training examples
- Try smaller sets of features $x, x^2, \cancel{x}, \cancel{x^2}, \cancel{x^3}, \dots$
- Try getting additional features
- Try adding polynomial features $(x_1^2, x_2^2, x_1 x_2, \text{etc})$
- Try decreasing λ
- Try increasing λ

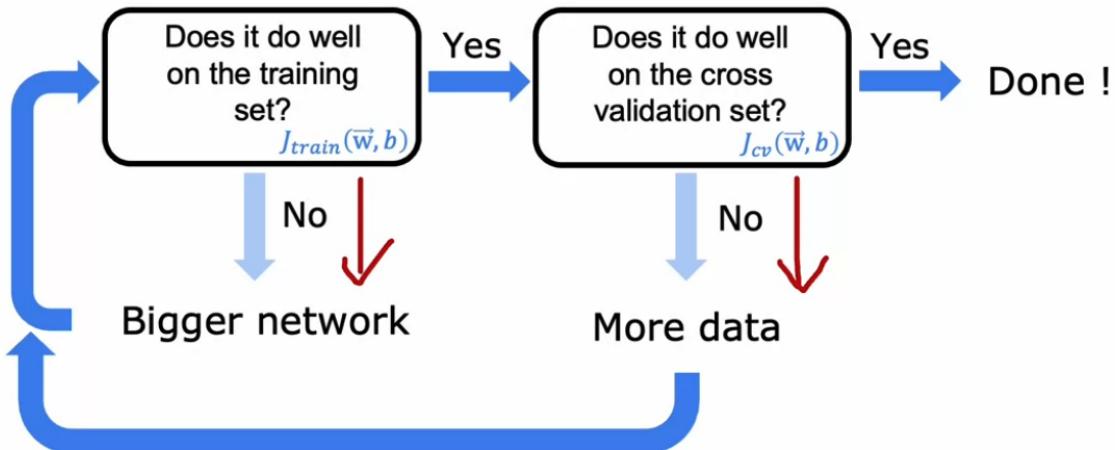
fixes high variance
fixes high variance
fixes high bias
fixes high bias
fixes high bias
fixes high variance

The bias variance tradeoff

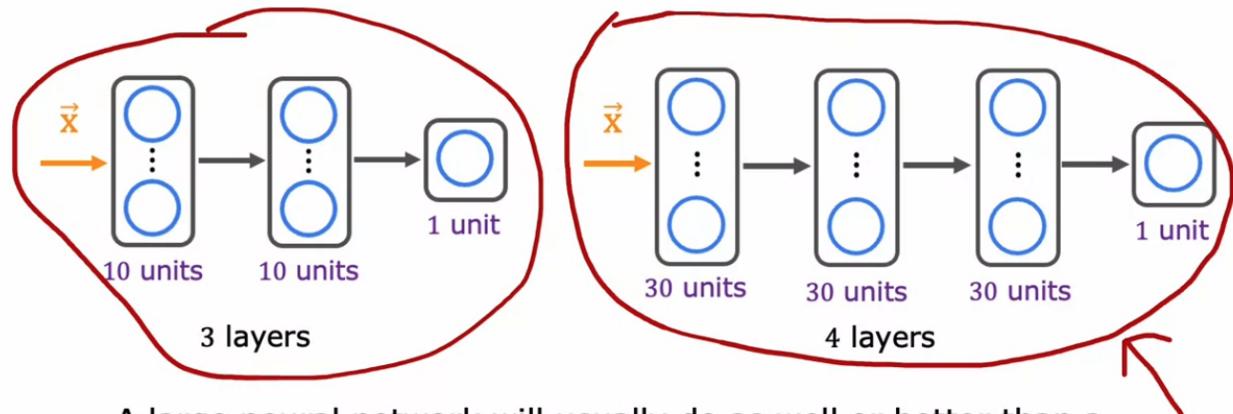


Neural networks and bias variance

Large neural networks are low bias machines



Neural networks and regularization



A large neural network will usually do as well or better than a smaller one so long as regularization is chosen appropriately.

Neural network regularization

$$J(\mathbf{W}, \mathbf{B}) = \frac{1}{m} \sum_{i=1}^m L(f(\vec{x}^{(i)}), y^{(i)}) + \frac{\lambda}{2m} \sum_{\text{all weights } \mathbf{W}} (w^2)$$

Unregularized MNIST model

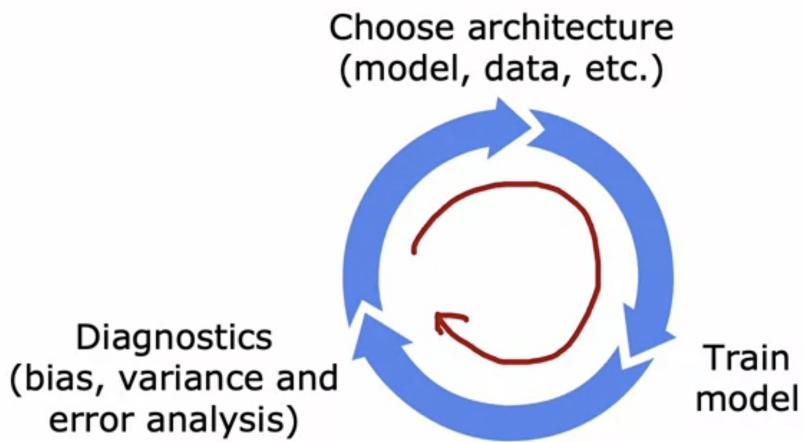
```
layer_1 = Dense(units=25, activation="relu")
layer_2 = Dense(units=15, activation="relu")
layer_3 = Dense(units=1, activation="sigmoid")
model = Sequential([layer_1, layer_2, layer_3])
```

Regularized MNIST model

```
layer_1 = Dense(units=25, activation="relu", kernel_regularizer=L2(0.01))
layer_2 = Dense(units=15, activation="relu", kernel_regularizer=L2(0.01))
layer_3 = Dense(units=1, activation="sigmoid", kernel_regularizer=L2(0.01))
model = Sequential([layer_1, layer_2, layer_3])
```

It hardly ever hurts to have a larger NN, so long as you regularize it properly.

Iterative loop of ML development



Avoid spending time in issues that are not very common by manually inspecting counts of categories of issues. We might then spend time collecting more data or building features related to a specific type of issue rather than another.

Error analysis

$m_{cv} = 500$ examples in cross validation set.

Algorithm misclassifies 100 of them.

Manually examine 100 examples and categorize them based on common traits.

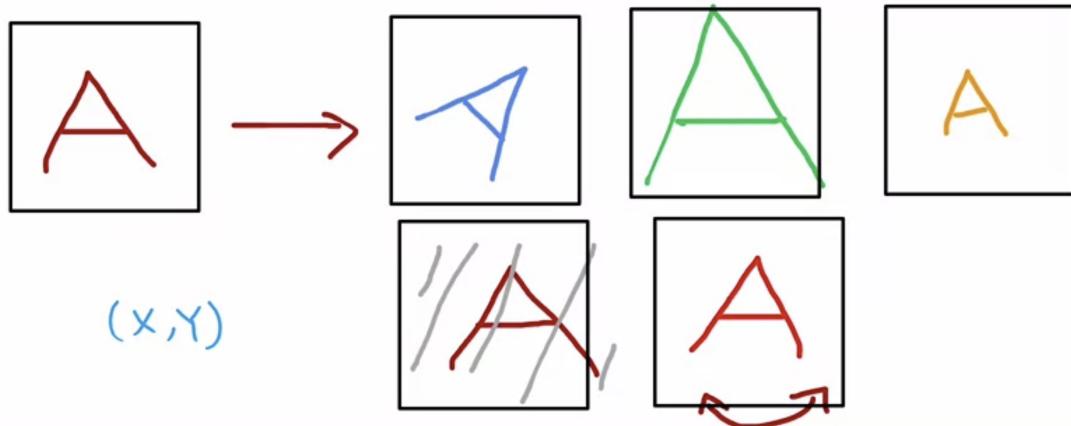
- Pharma: 21
- Deliberate misspellings (w4tches, med1cine): 3
- Unusual email routing: 7
- Steal passwords (phishing): 18
- Spam message in embedded image: 5

We can also artificially increase the size of the training data, without getting brand new examples, using data augmentation. These distortions should still be

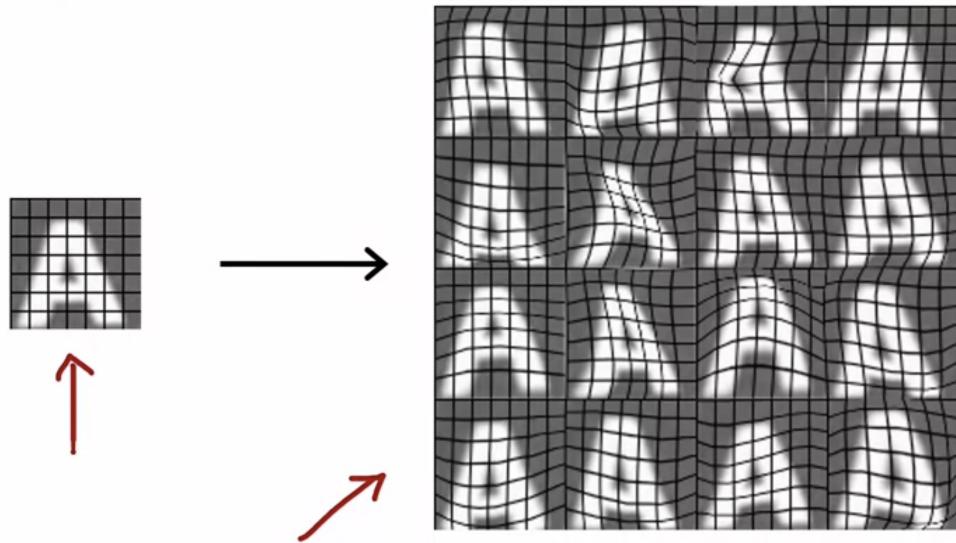
representative of what you'd get in the test set, or they wouldn't aid in performance.

Data augmentation

Augmentation: modifying an existing training example to create a new training example.



Data augmentation by introducing distortions

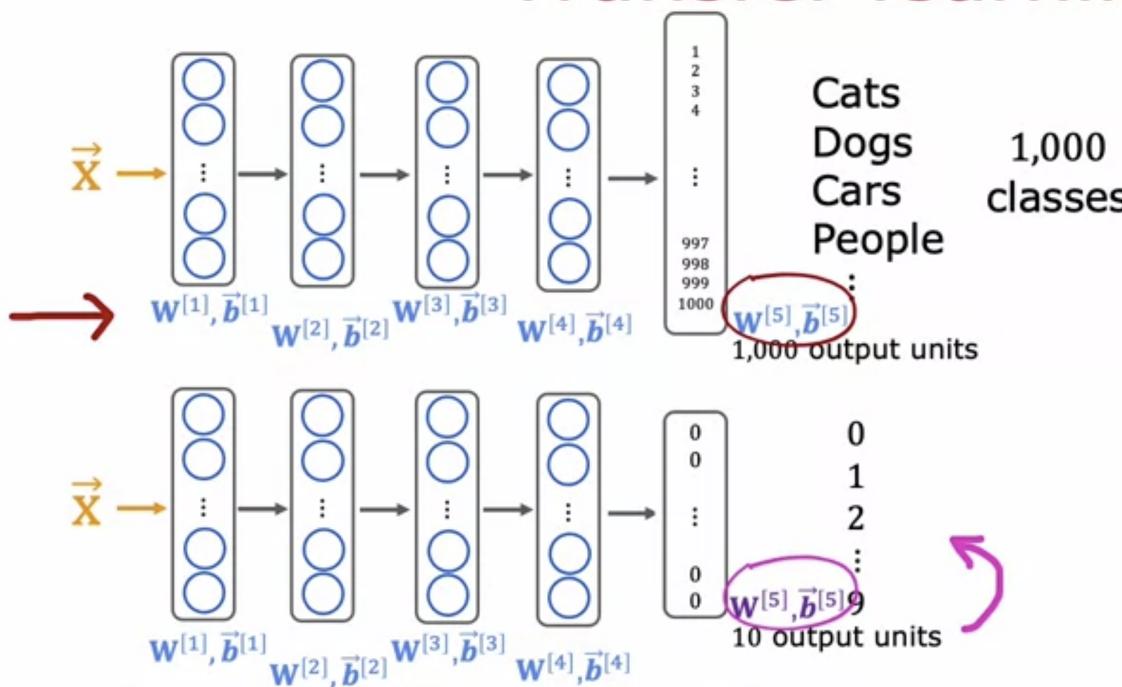


For speech data, we can add background noise to the original audio, for example.

We can also use data synthesis, for example, for computer vision tasks.

When we don't have much data for a particular problem, we can apply **transfer learning**. The idea is to use data from another similar problem (same input type), train a NN and then re-use the model, adjusting its output layer. We can then use the weights from the first NN and train the new NN to get the weights of the output layer. Or we can retrain the whole NN, using the original NN parameters as initial parameters. e.g. from classifying digits 0-9 to classifying animals. We call these steps: **supervised pre-training and fine-tuning**. We can also use NNs that are available online and fine-tune for our problem!

Transfer learning



Option 1: only train output layers parameters.

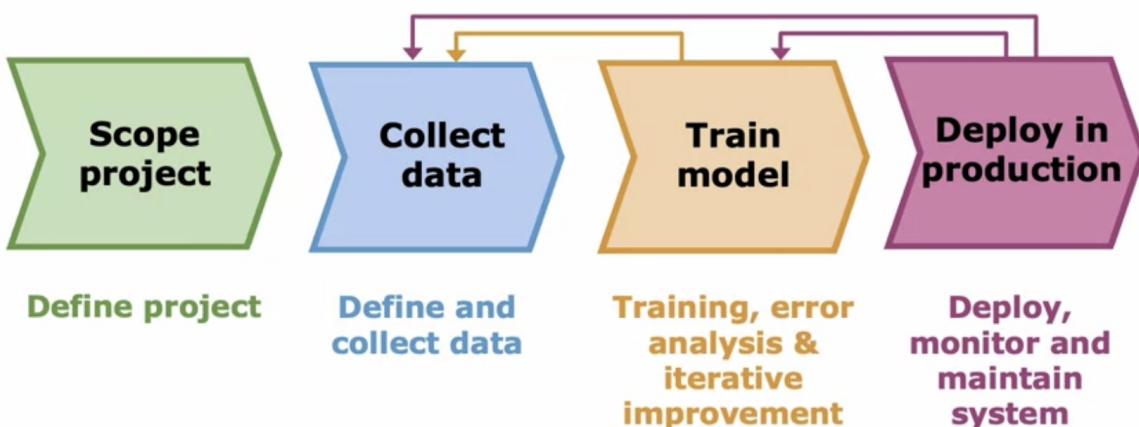
Option 2: train all parameters.

It's found that the fine-tuning requires a lot less examples if the pre-training was done with a large dataset.

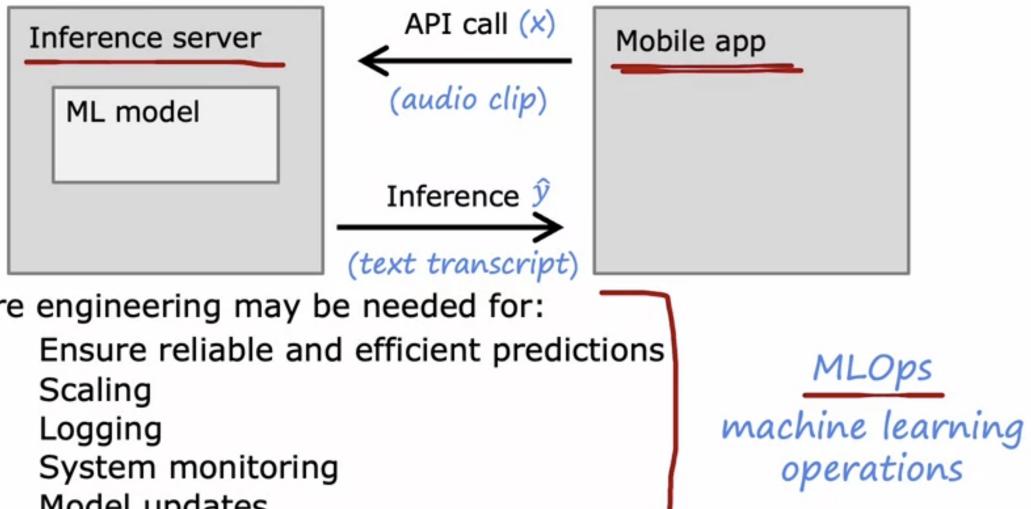
Transfer learning summary

1. Download neural network parameters pretrained on a large dataset with same input type (e.g., images, audio, text) as your application (or train your own).
1 million images
2. Further train (fine tune) the network on your own data.
1000 images
50 images

Full cycle of a machine learning project



Deployment



For skewed datasets specially, it's advised to use other metrics, such as precision and recall.

Precision/recall

$y = 1$ in presence of rare class we want to detect.

		Actual Class 1	0
Predicted Class	1	True positive 15	False positive 5
	0	False negative 10	True negative 70
	↓	25	↓ 75

Precision:

(of all patients where we predicted $y = 1$, what fraction actually have the rare disease?)

$$\frac{\text{True positives}}{\#\text{predicted positive}} = \frac{\text{True positives}}{\text{True pos} + \text{False pos}} = \frac{15}{15+5} = 0.75$$

Recall:

(of all patients that actually have the rare disease, what fraction did we correctly detect as having it?)

$$\frac{\text{True positives}}{\#\text{actual positive}} = \frac{\text{True positives}}{\text{True pos} + \text{False neg}} = \frac{15}{15+10} = 0.6$$

The threshold used for classification is based on a tradeoff between precision and recall, specific to the problem. If we want high precision, we'd increase the threshold. Else, if we want high recall, we'd lower the threshold.

Trading off precision and recall

- Logistic regression: $0 < f_{\vec{w}, b}(\vec{x}) < 1$
- Predict 1 if $f_{\vec{w}, b}(\vec{x}) \geq 0.5$ (0.7, 0.1, 0.3)
 - Predict 0 if $f_{\vec{w}, b}(\vec{x}) < 0.5$ (0.7, 0.1, 0.3)

$$\text{precision} = \frac{\text{true positives}}{\text{total predicted positive}}$$

$$\text{recall} = \frac{\text{true positives}}{\text{total actual positive}}$$

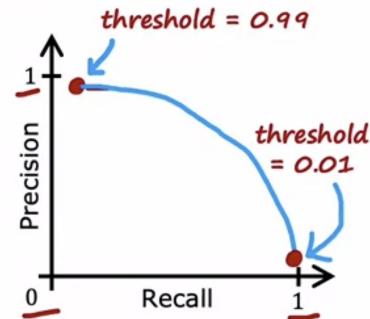
Suppose we want to predict $y = 1$ (rare disease) only if very confident.

higher precision, lower recall

Suppose we want to avoid missing too many case of rare disease (when in doubt predict $y = 1$)

lower precision, higher recall

More generally predict 1 if: $f_{\vec{w}, b}(\vec{x}) \geq \text{threshold}$.



F1 score

How to compare precision/recall numbers?

	Precision (P)	Recall (R)	Average	F ₁ score
Algorithm 1	0.5	0.4	0.45	0.444
Algorithm 2	0.7	0.1	0.4	0.175
Algorithm 3	0.02	1.0	0.501	0.0392

`print("y=1")`

Average = $\frac{P+R}{2}$

$$F_1 \text{ score} = \frac{1}{\frac{1}{2} \left(\frac{1}{P} + \frac{1}{R} \right)} = 2 \frac{PR}{P+R}$$

The F1 score is the harmonic mean of precision and recall and it'll give more emphasis on the one that is very small. This is because, if any of the two is too small, then the algorithm is not a very good one.