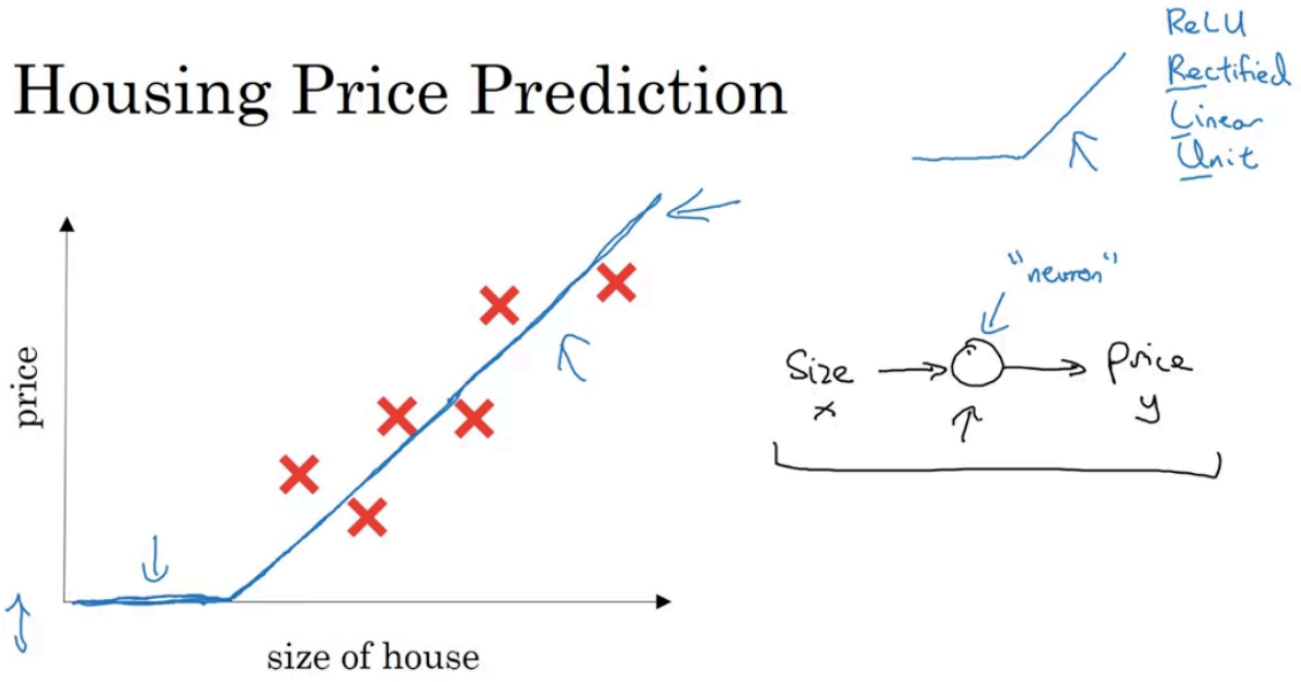
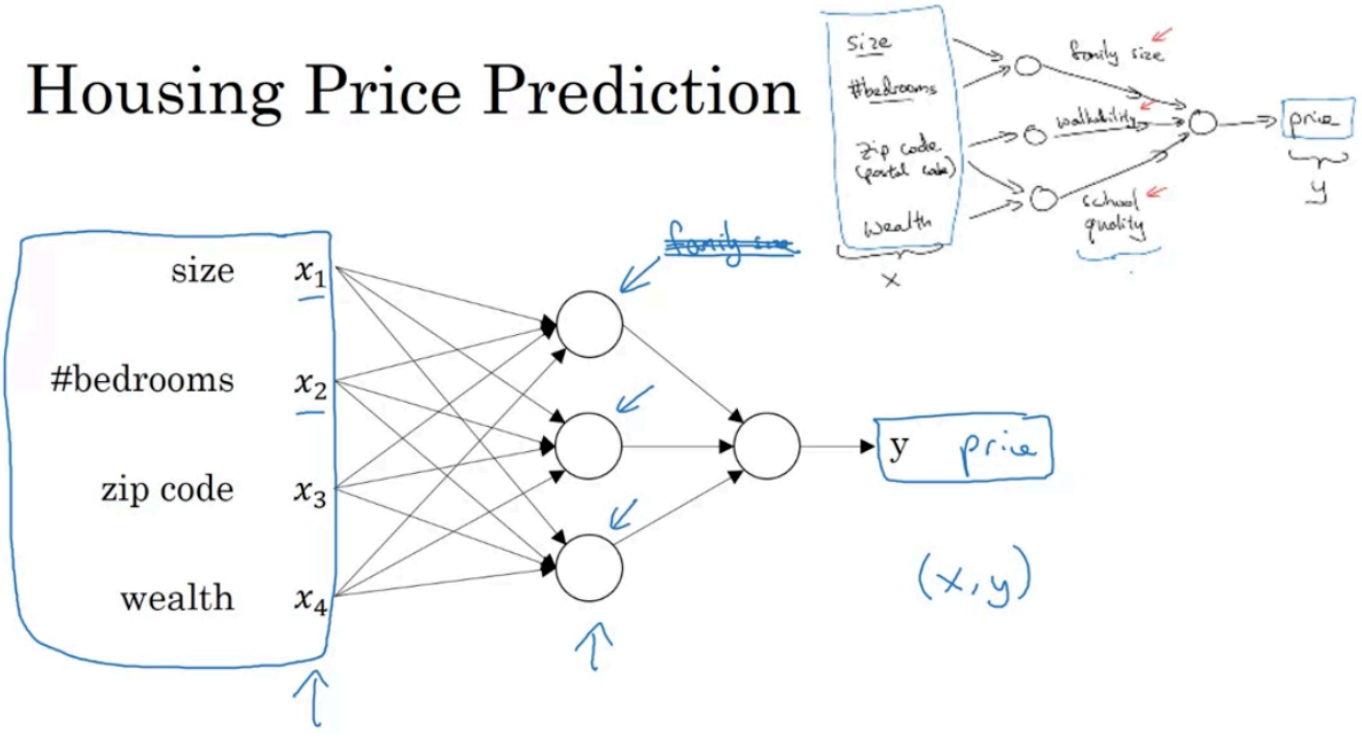
Neural Networks & Deep Learning

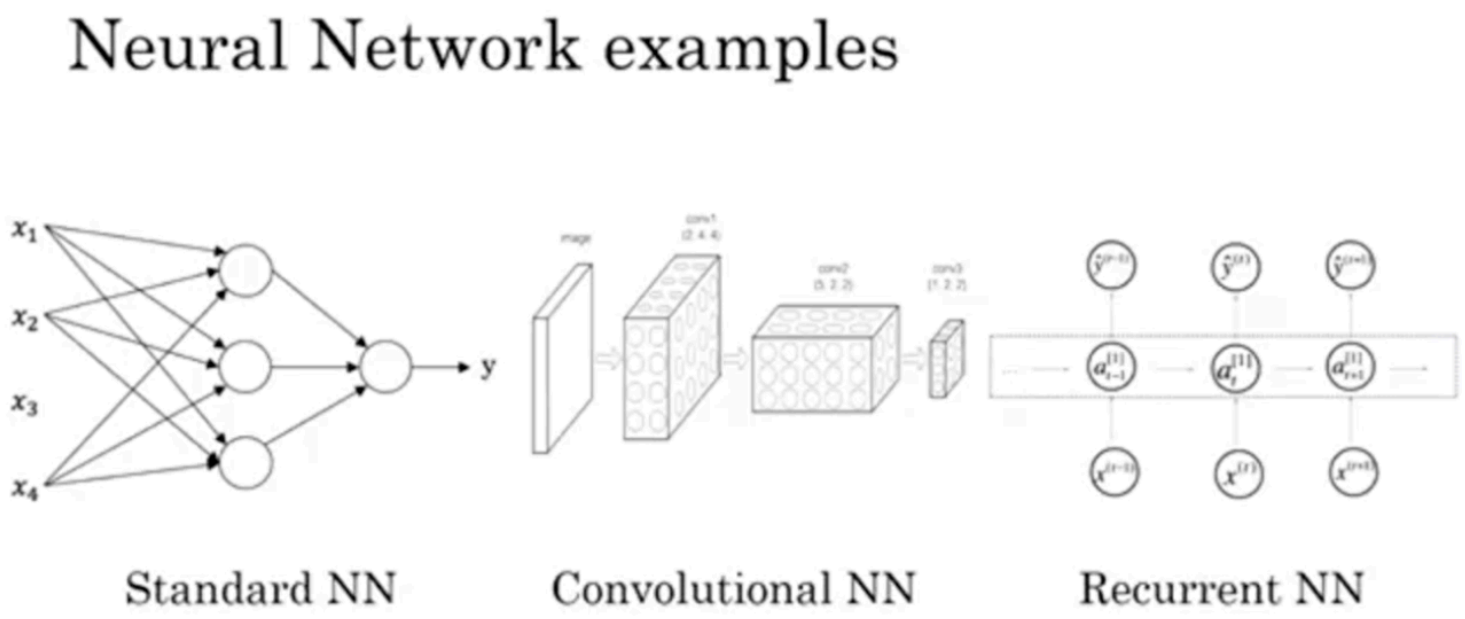
What to learn?

NN, DL
Improvements: Hyperparameters, Regularization, Optimization
Structuring projects
Convolutional NNs, CNNs
Sequence Models, NLP, RNN, LSTM





Relatively standart for real estate: Standart NN
Image: CNN
Sequence Data: (Audio) One dim. type of sequence data: RNN
Image + Radar: Custom/Hybrid NNs



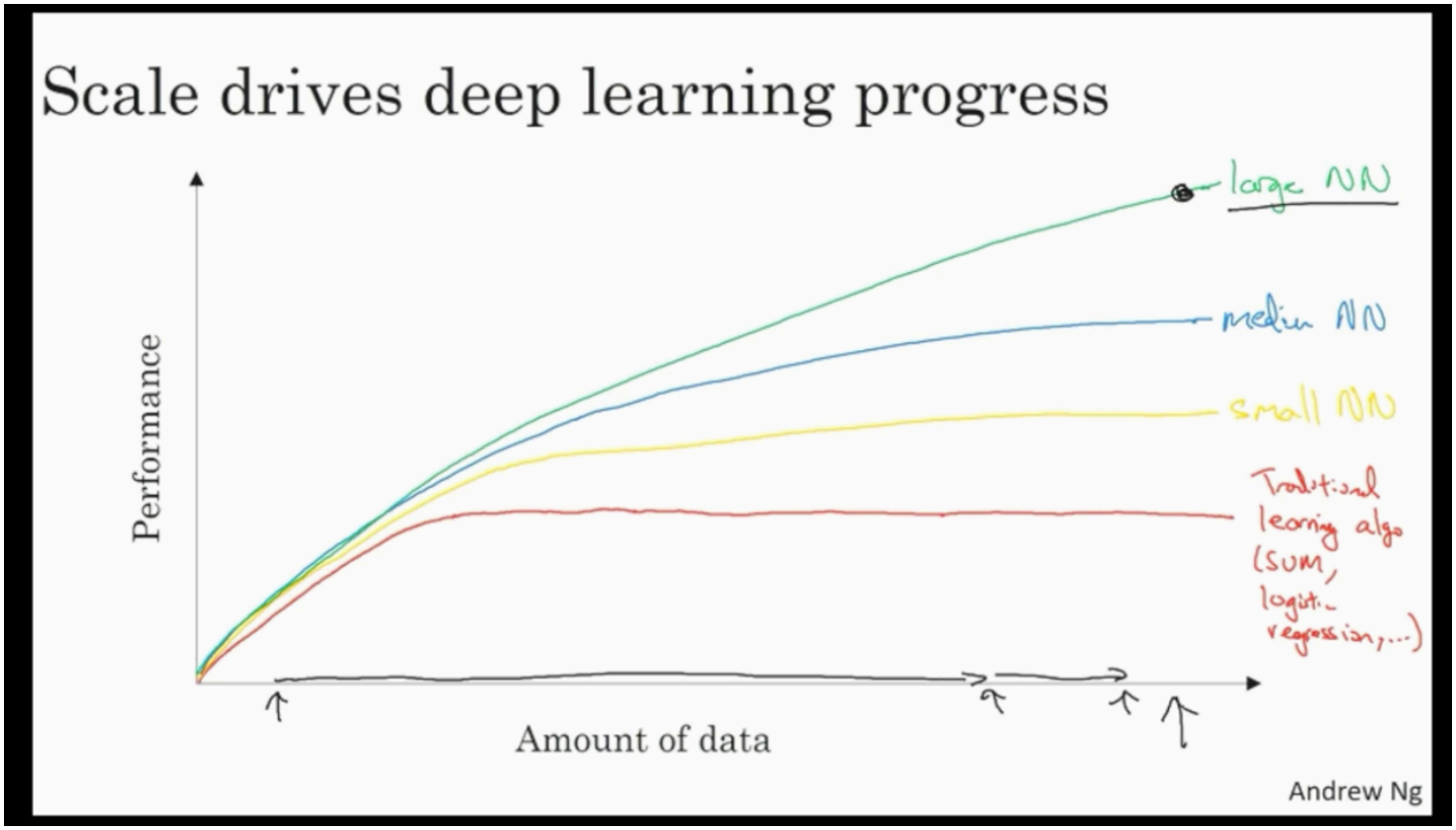
Structured Data: Databases -> clear labels, size, bedroom number, price, age, id...
Unstructured Data: Audio, Images, Text

Why now?

Scale drives DL
Traditional learning algo: Support Vector Machine, Logistic Regression

- learning curve flattens (plateus after some amount of data)
  We continue collecting data

For good modeling (NN): Big models & big data



just improving scale on both ends helps (mostly until the hitting a limit)

for small data, the performance of model architectures are not clearly listed. Depends on the specific situation

Data - Computation - Algorithm (mostly, enabled the NNs run faster)

Sigmoid fnc (parameters move too slow) -> ReLU (rectified last unit)
makes the gradient descent much faster

- Iterative process: Idea -> Code -> *Experiment* -> Repeat

10 min -> Day -> Month

Geoffrey Hinton: Godfather of DL

Binary Classification

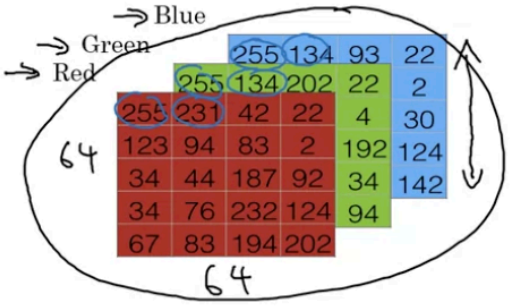
Binary Classification

64

64

1 (cat) vs 0 (non cat)

y



$$x = \begin{bmatrix} 255 \\ 231 \\ \vdots \\ 255 \\ 134 \\ \vdots \end{bmatrix}$$

$64 \times 64 \times 3 = 12288$

$n = n_x = 12288$

$x \longrightarrow y$

Andrew Ng

Notation

(x, y) $\quad x \in \mathbb{R}^{n_x}$, $y \in \{0, 1\}$

m training examples: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$

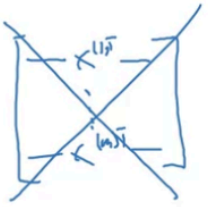
$M = M_{train}$ $\qquad M_{test}$ = #test examples.

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$$

n_x

m

$X \in \mathbb{R}^{n_x \times m}$ $\qquad X.shape = (n_x, m)$



$Y = [\, y^{(1)} \; y^{(2)} \; \cdots \; y^{(m)} \,]$

$Y \in \mathbb{R}^{1 \times m}$

$Y.shape = (1, m)$

Andrew Ng

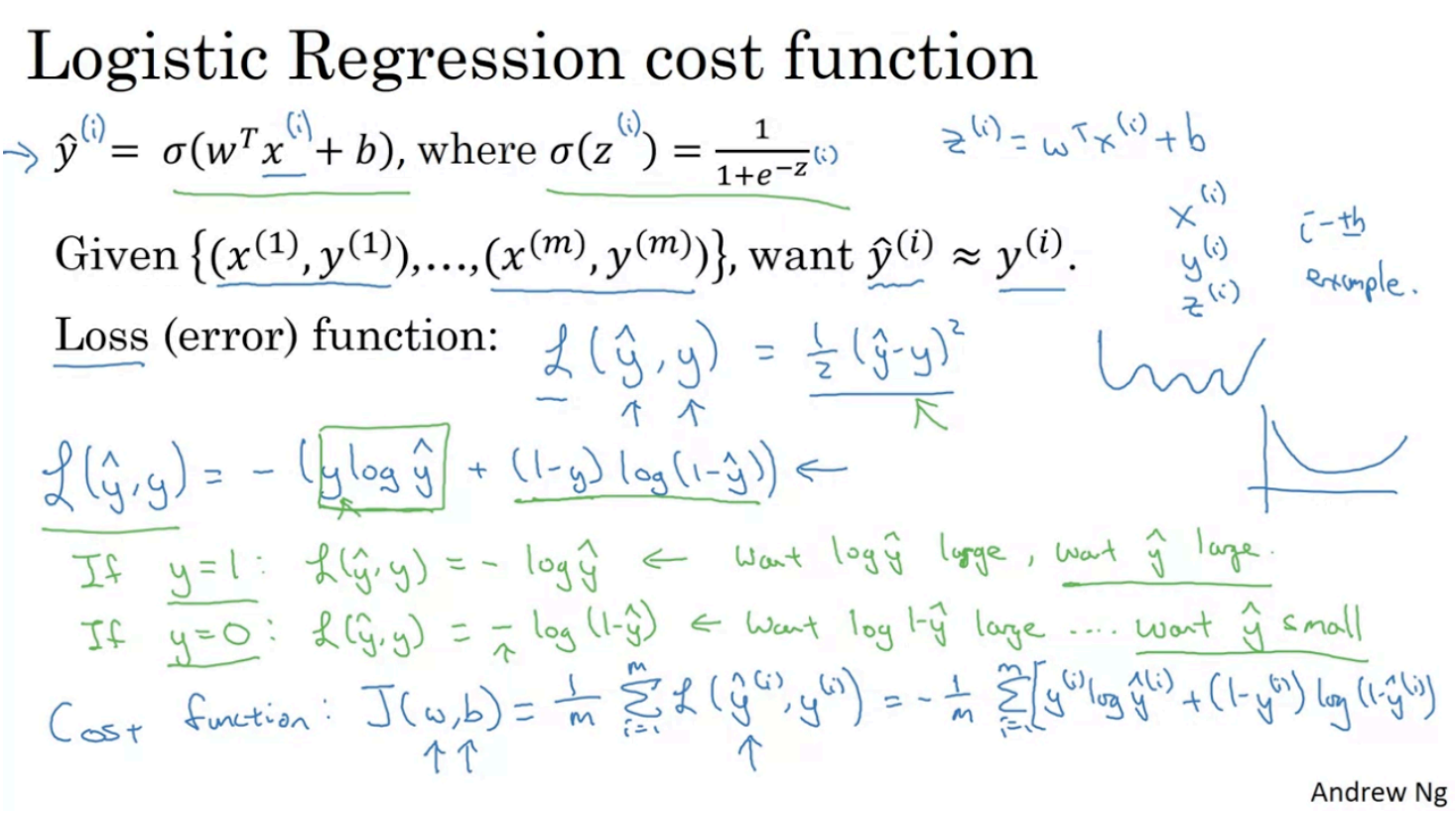
Logistic Regression

given x - want y'

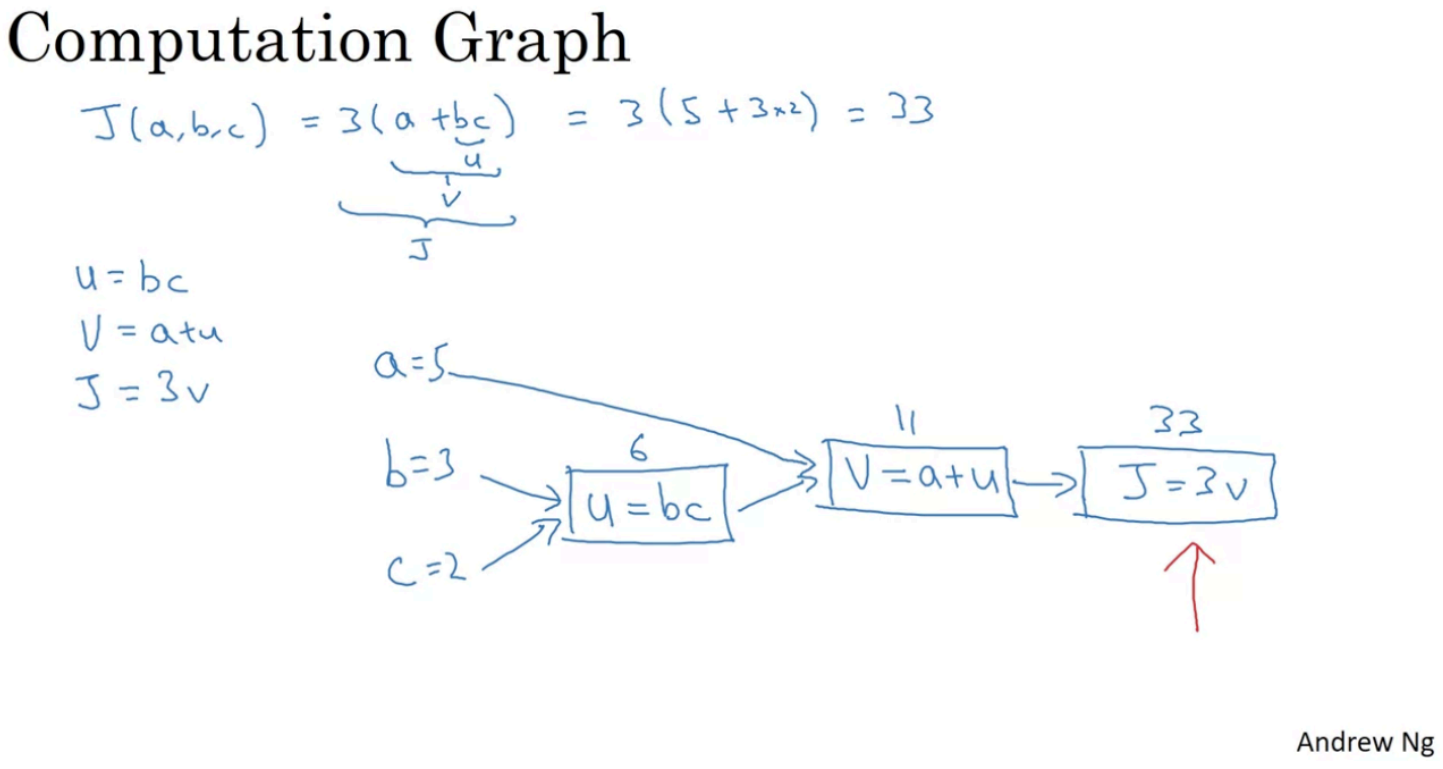
x picture, chance of cat

linear regression is not a good choice

sigmoid fnc



Check: Gradient Descent, Derivatives, etc.



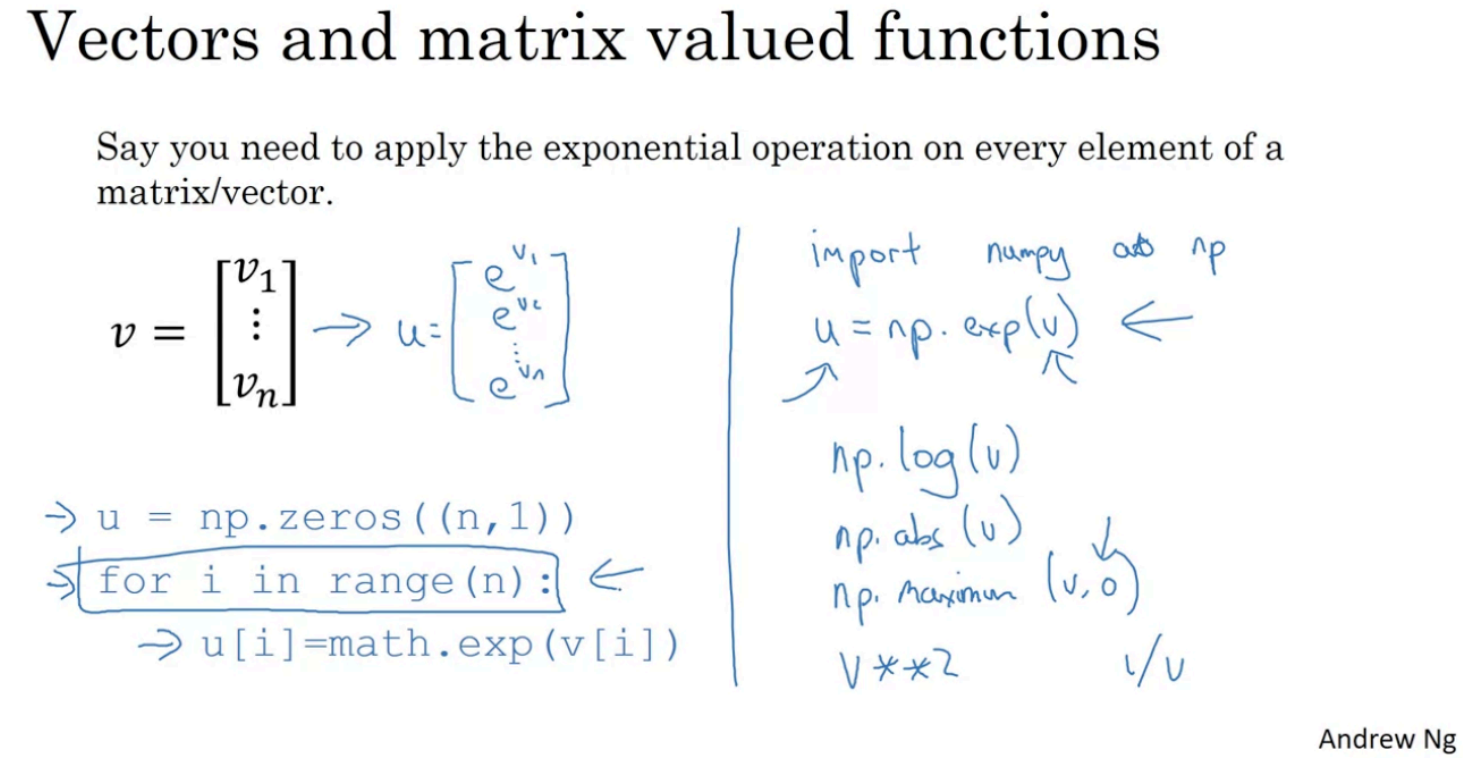
Logistic Regression derivatives
Gradient Descent on m Examples

Vectorization

for DL, vectorization is key
even in a small demonstration, for loop is 300x slower compared to vectorized operations.

SIMD: Single instruction, multiple data. Both works for CPU & GPU

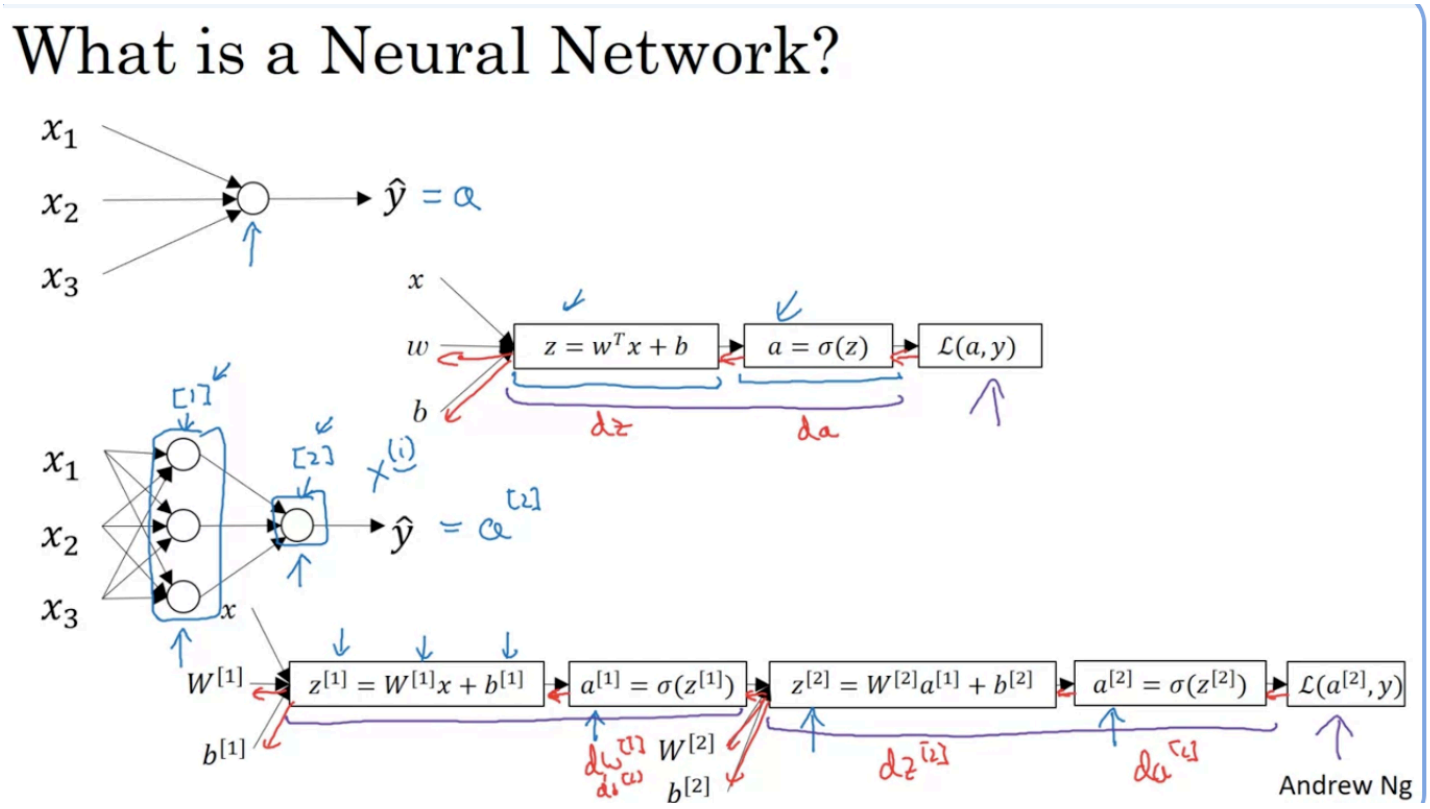
whenever possible, avoid explicit for loops.



Vectorizing Logistic Regression & Gradient Descent...

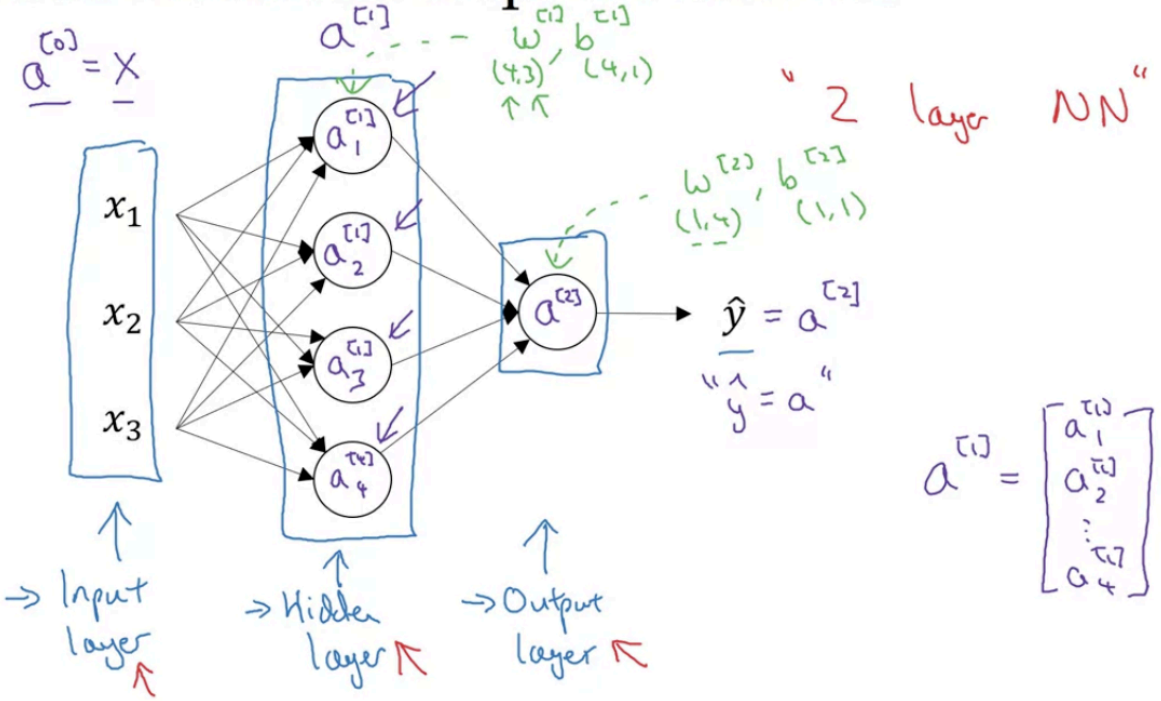
Broadcasting in Python

NN Architecture



Single Hidden Layer example
Input Layer -> Hidden Layer -> Output Layer

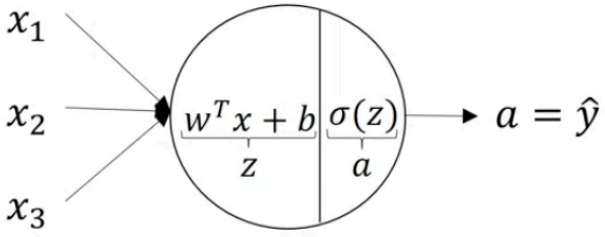
Neural Network Representation



Andrew Ng

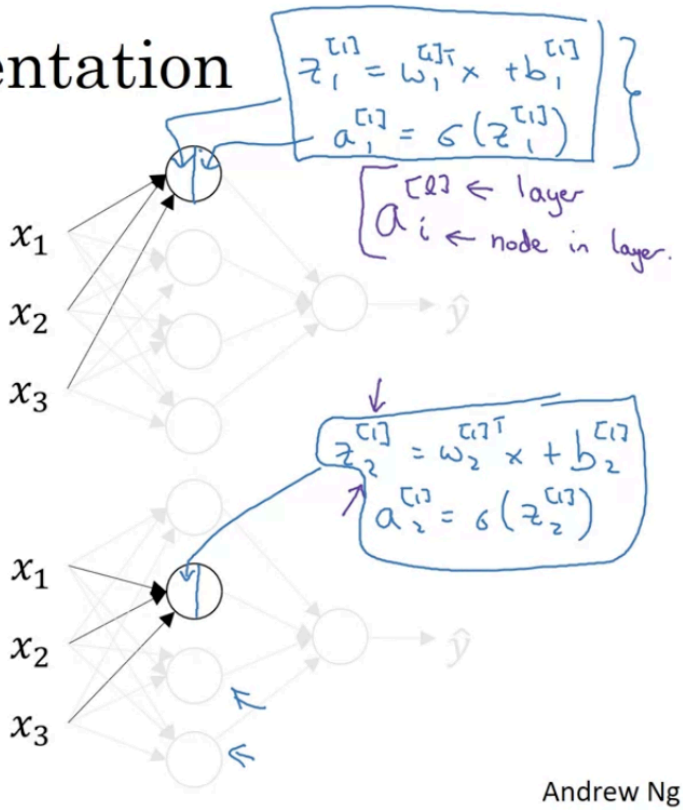
input layer is not counted when referencing the # of layers in NN

Neural Network Representation



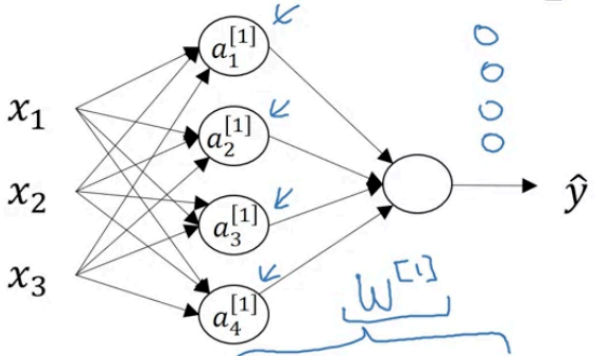
$$z = w^T x + b$$

$$a = \sigma(z)$$



Andrew Ng

Neural Network Representation



$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}, \quad a_1^{[1]} = \sigma(z_1^{[1]})$$

$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}, \quad a_2^{[1]} = \sigma(z_2^{[1]})$$

$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]}, \quad a_3^{[1]} = \sigma(z_3^{[1]})$$

$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]}, \quad a_4^{[1]} = \sigma(z_4^{[1]})$$

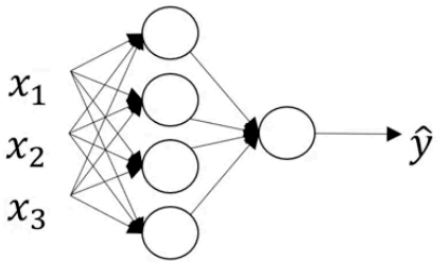
$$z^{[1]} = \begin{bmatrix} — w_1^{[1]T} — \\ — w_2^{[1]T} — \\ — w_3^{[1]T} — \\ — w_4^{[1]T} — \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} w_1^{[1]T} x + b_1^{[1]} \\ w_2^{[1]T} x + b_2^{[1]} \\ w_3^{[1]T} x + b_3^{[1]} \\ w_4^{[1]T} x + b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}$$

$W^{[1]}$ (4, 3) $\quad b^{[1]}$ (4, 1)

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ \vdots \\ a_4^{[1]} \end{bmatrix} = \sigma(z^{[1]})$$

Andrew Ng

Vectorizing across multiple examples



$$z^{[1]} = W^{[1]}x + b^{[1]}$$
$$a^{[1]} = \sigma(z^{[1]})$$
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$a^{[2]} = \sigma(z^{[2]})$$

$x \rightarrow a^{[2]} = \hat{y}$

$x^{(1)} \rightarrow a^{[2](1)} = \hat{y}^{(1)}$

$x^{(2)} \rightarrow a^{[2](2)} = \hat{y}^{(2)}$

$x^{(m)} \rightarrow a^{[2](m)} = \hat{y}^{(m)}$

$a^{[2](i)}$: activation vector, layer 2, example i

for $i = 1$ to m,

$z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]}$

$a^{[1](i)} = \sigma(z^{[1](i)})$

$z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]}$

$a^{[2](i)} = \sigma(z^{[2](i)})$

Andrew Ng

Vectorizing across multiple examples

```
for i = 1 to m:
```

$z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]}$

$a^{[1](i)} = \sigma(z^{[1](i)})$

$z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]}$

$a^{[2](i)} = \sigma(z^{[2](i)})$

$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$ (n_x, m)

$Z^{[1]} = \begin{bmatrix} | & | & & | \\ z^{[1](1)} & z^{[1](2)} & \cdots & z^{[1](m)} \\ | & | & & | \end{bmatrix}$

$A^{[1]} = \begin{bmatrix} | & | & & | \\ a^{[1](1)} & a^{[1](2)} & \cdots & a^{[1](m)} \\ | & | & & | \end{bmatrix}$

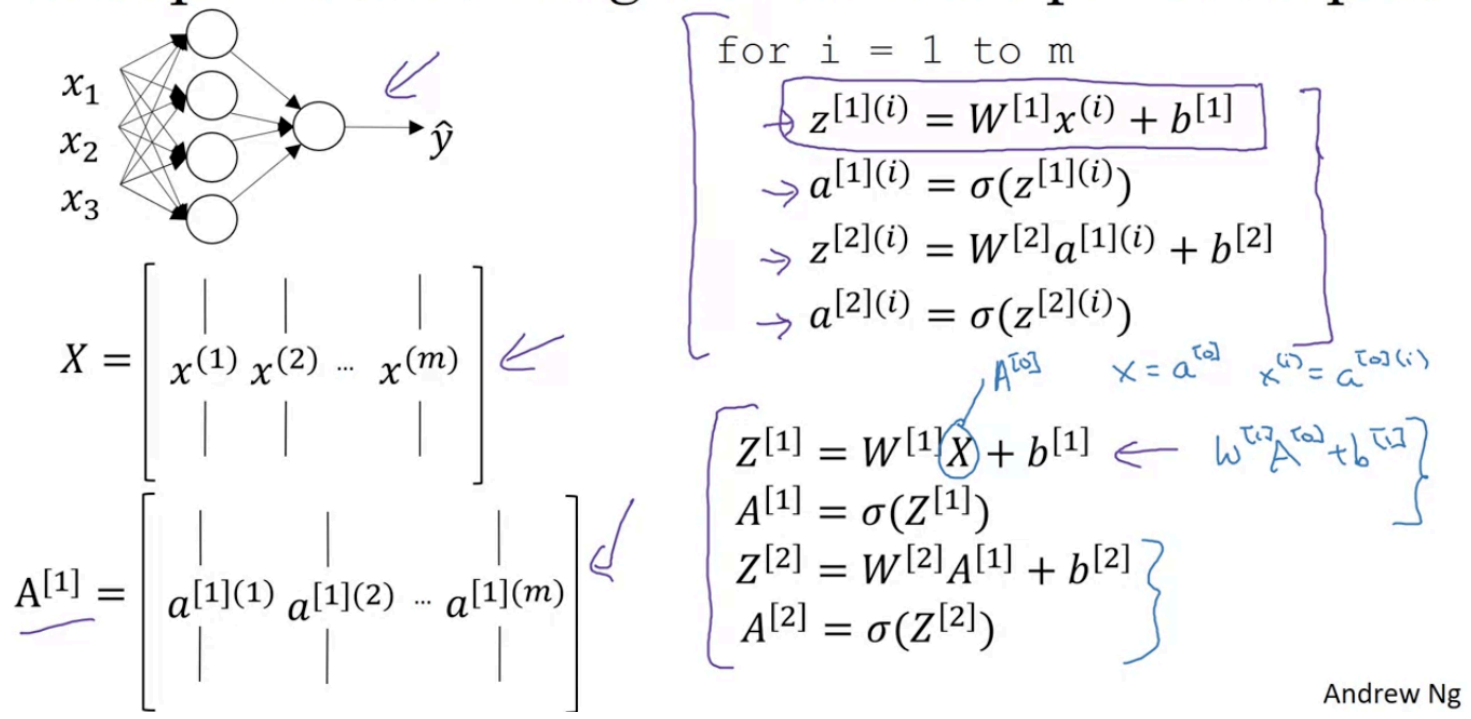
$Z^{[1]} = W^{[1]}X + b^{[1]}$

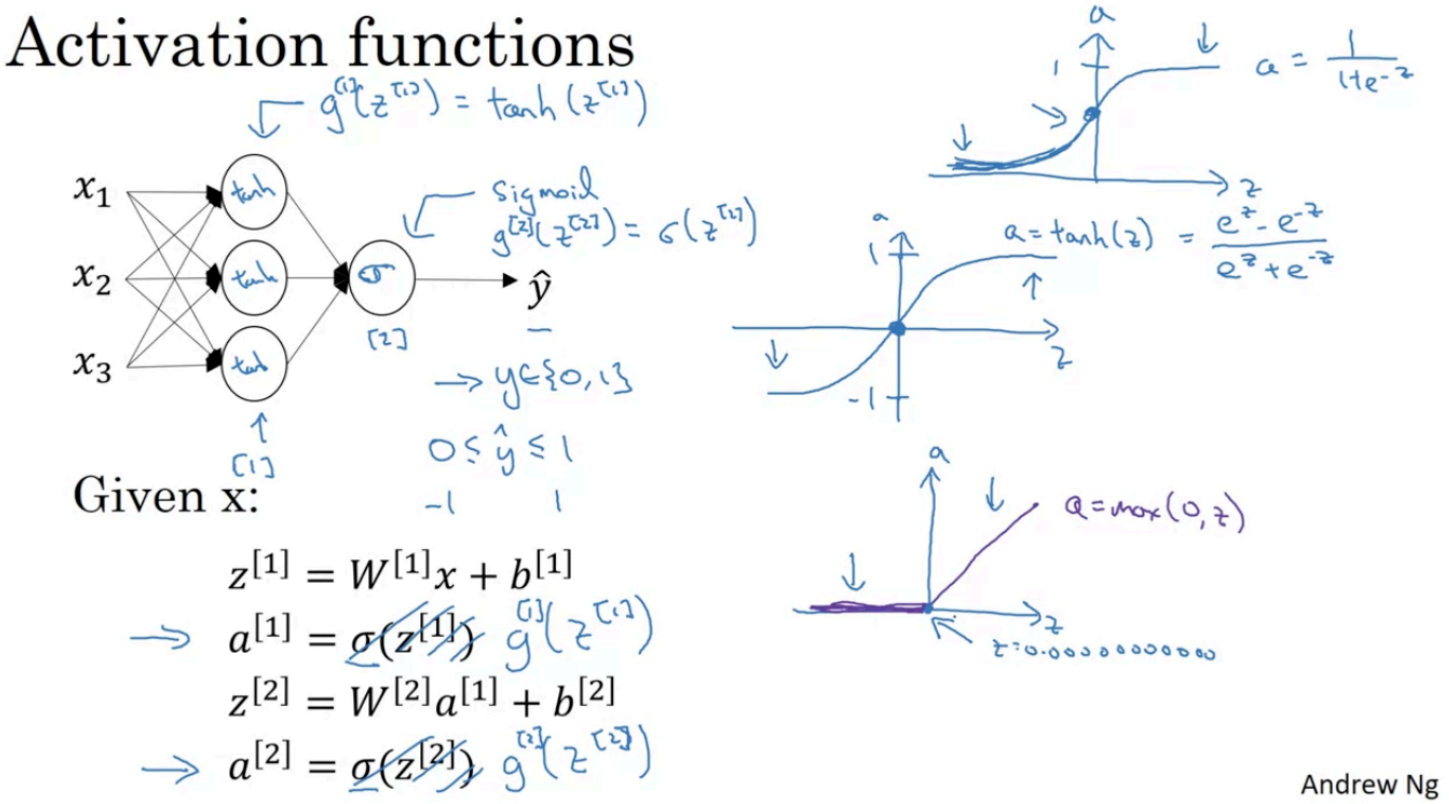
$A^{[1]} = \sigma(Z^{[1]})$

$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$

$A^{[2]} = \sigma(Z^{[2]})$

Andrew Ng





Activation Function almost always works better than the sigmoid function
since mean of -1,1 mostly always becomes nearer to 0
tanh is superior of activation, not practical to use activation nearly for all cases
0-1 value: sigmoid (never use)
ReLU: default for most cases
downside: derivative is 0, when value is negative
to neglect this downside: Leaky ReLU

Why do you need Non-Linear Activation Functions?

linear hidden layer is useless,
it is just a logistic regression

Non-linear activation functions are crucial for neural networks because they enable the network to compute more complex and interesting functions. Without them, the network's output would remain a linear function of the input, regardless of the number of hidden layers.

1. **Linear Activation Limitation:**

If the activation function is linear (or identity, where $g(z) = z$), the neural network simply performs linear operations. This results in the output being just a linear transformation of the input:

$a^{(1)} = W^{(1)}x + b^{(1)}$
$a^{(2)} = W^{(2)}a^{(1)} + b^{(2)} = W{\prime} x + b{\prime}$

This makes the hidden layers redundant, as the entire network simplifies to a linear model.

2. **Deep Networks and Linear Activation:**

Even with many hidden layers, if all activations are linear, the overall network still computes a linear function. Therefore, the depth of the network does not increase its expressive power. The text emphasizes that without non-linearity, you might as well not have hidden layers.

3. **Expressiveness of Non-Linear Activation:**

Non-linear activation functions (like **ReLU, tanh, sigmoid**) allow the network to compute more complex, non-linear functions. This is essential for learning complicated patterns and relationships in data.

4. **Comparison to Logistic Regression:**

A neural network with linear hidden layers and a sigmoid output is no more powerful than logistic regression. This highlights the importance of non-linearity in hidden layers to enhance the network's capacity.

5. **When Linear Activation is Useful:**

• **Output Layer for Regression:** Linear activation is useful in the output layer for regression tasks, where the target output is a continuous value (e.g., predicting housing prices).

• For hidden layers, non-linear functions are preferred to introduce the complexity needed for learning diverse patterns.

Derivatives of Activation Functions - Back Propagation

Sigmoid
tanh
ReLU and Leaky ReLU

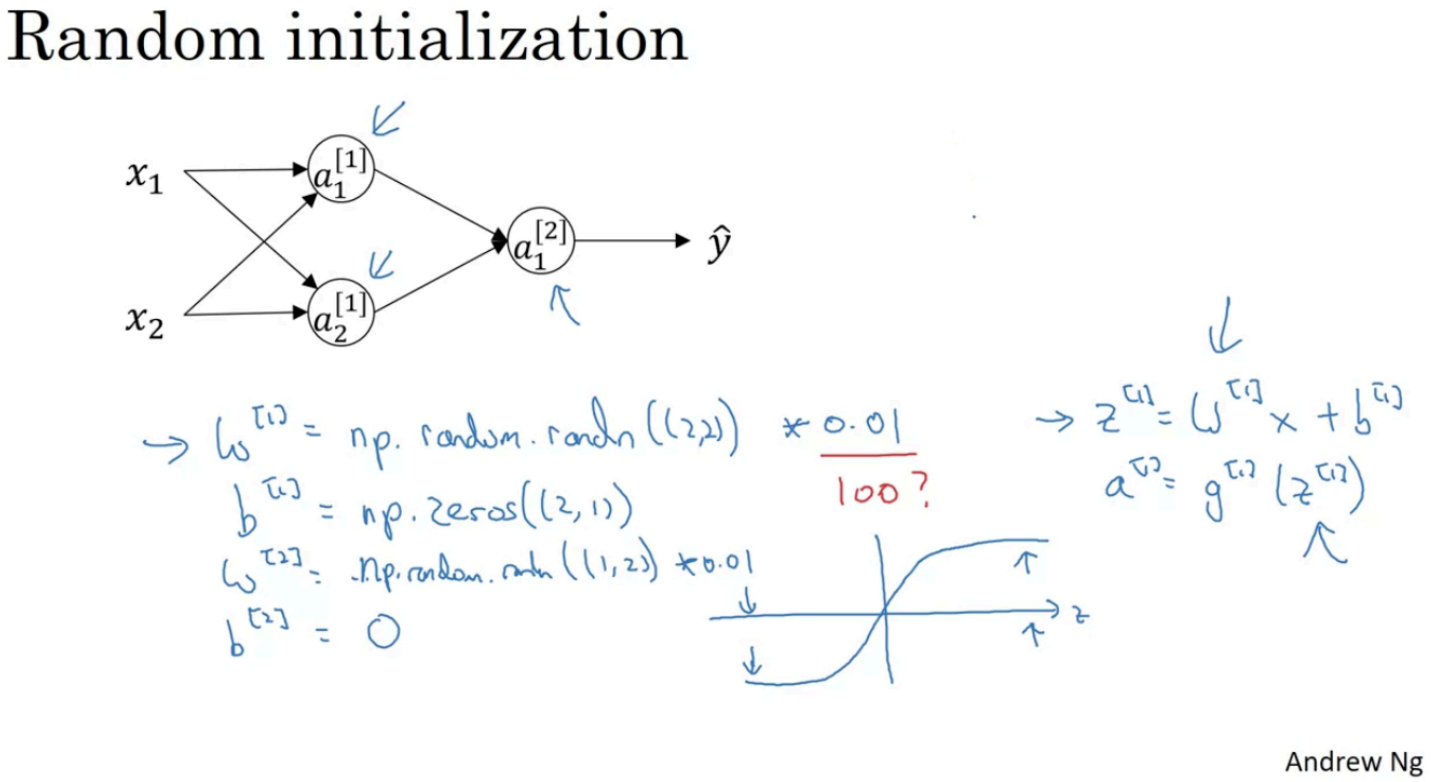
Gradient Descent for NNs

for gradient descent start with random values for parameters

Random Initialization

when started with zero, different neurons are initialized in the same way and compute the same operations.

How big?

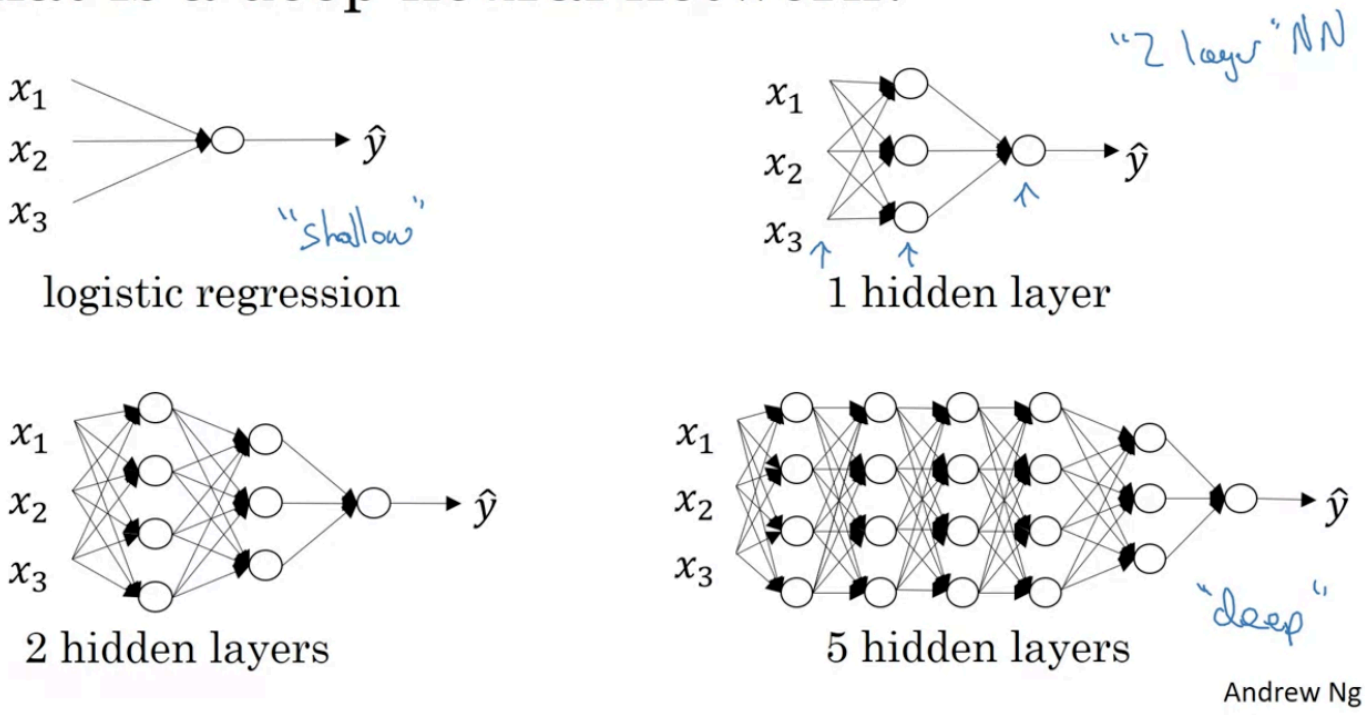


when w is too large, initialized at plateau points

Deep L-layer NN

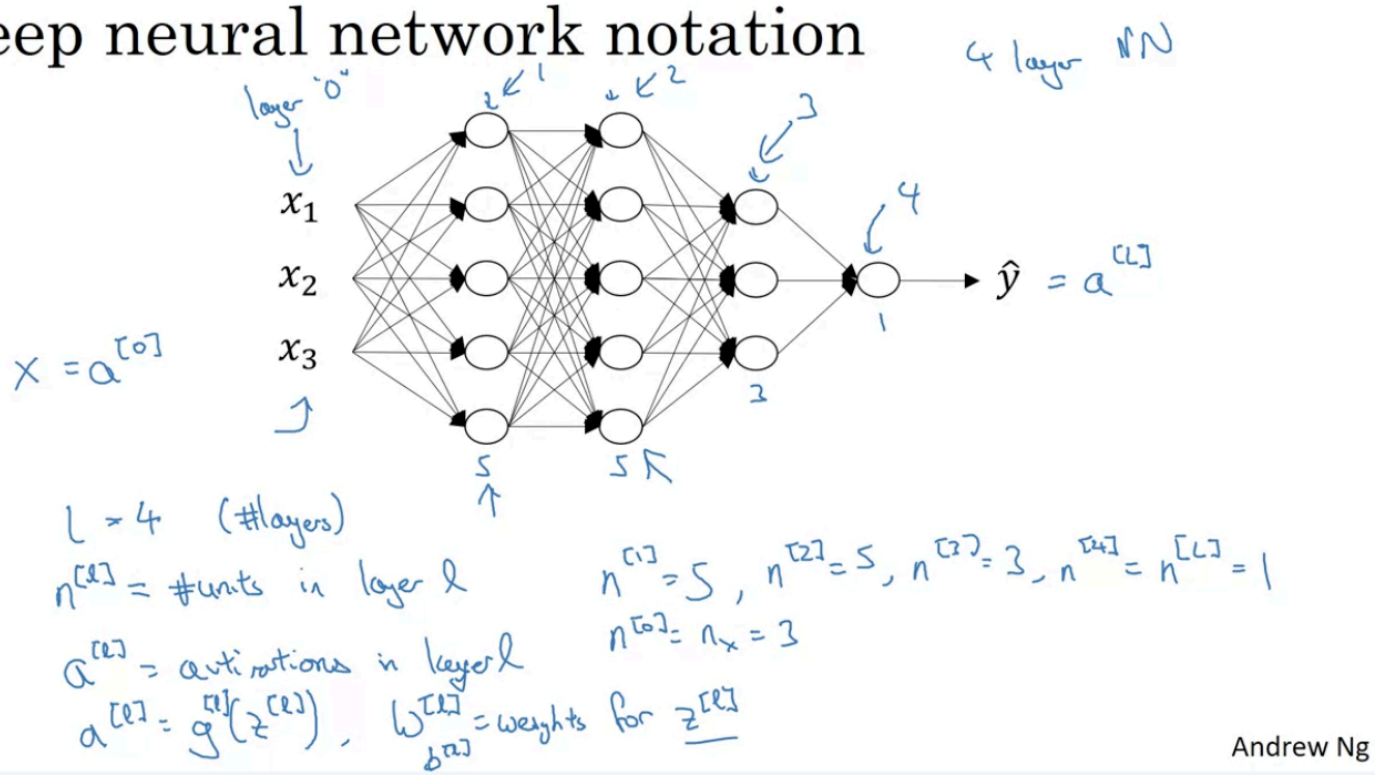
Deep NN ?

What is a deep neural network?



'# of hidden layers is another 'hyperparameter' for some cases

Deep neural network notation

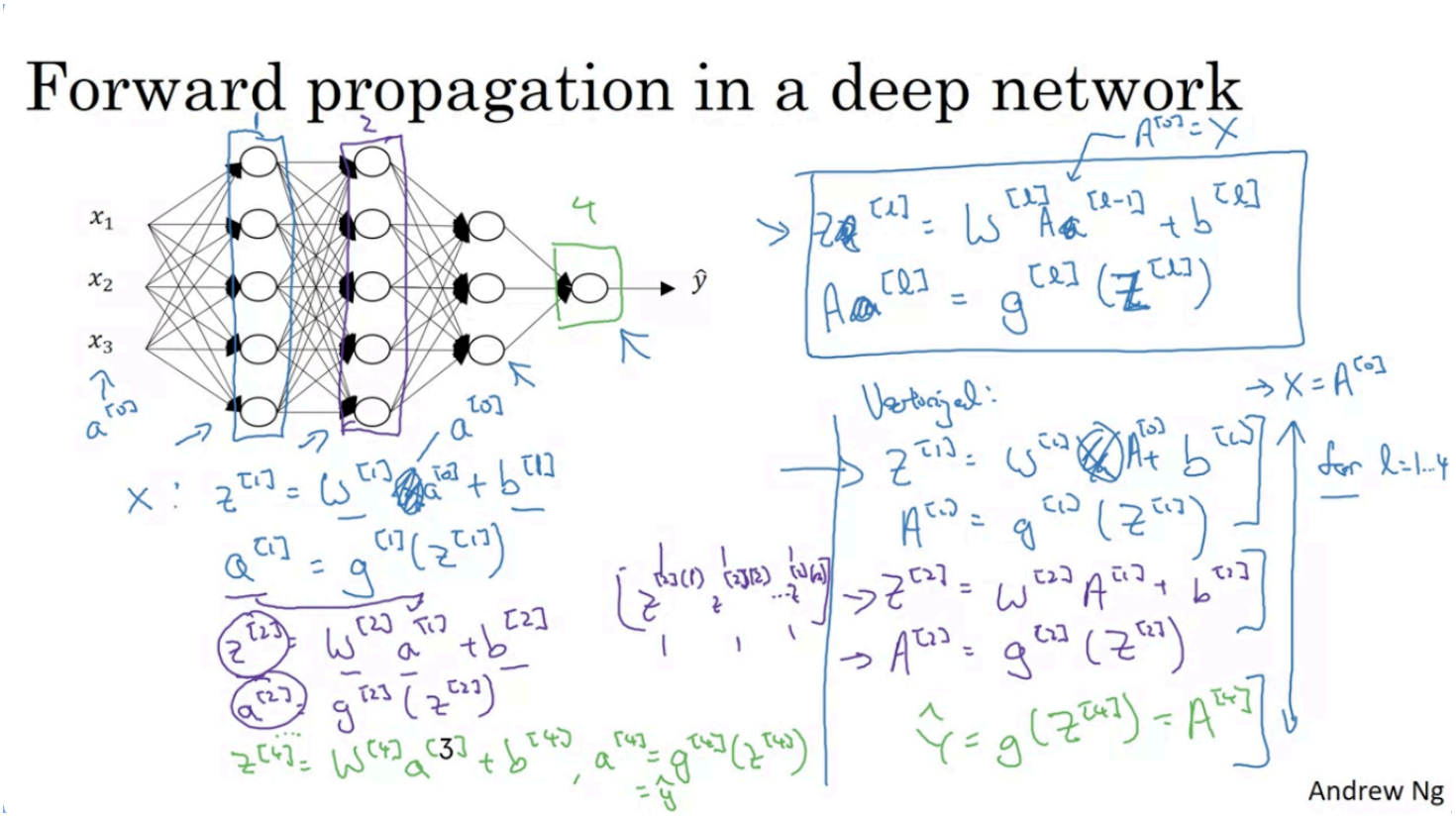


- **Layer "0"** (input layer)
- x_1, x_2, x_3 (input features)
- X = a^{[0]}
- **4 layer NN** (neural network)
- l = 4 (# layers)
- n^{[l]} = # units in layer l
- a^{[l]} = activations in layer l
- a^{[l]} = g(z^{[l]})

• W^{[l]} , b^{[l]} = weights for z^{[l]}
• n^{[1]} = 5, n^{[2]} = 5, n^{[3]} = 3, n^{[4]} = 1
• n^{[0]} = n_x = 3

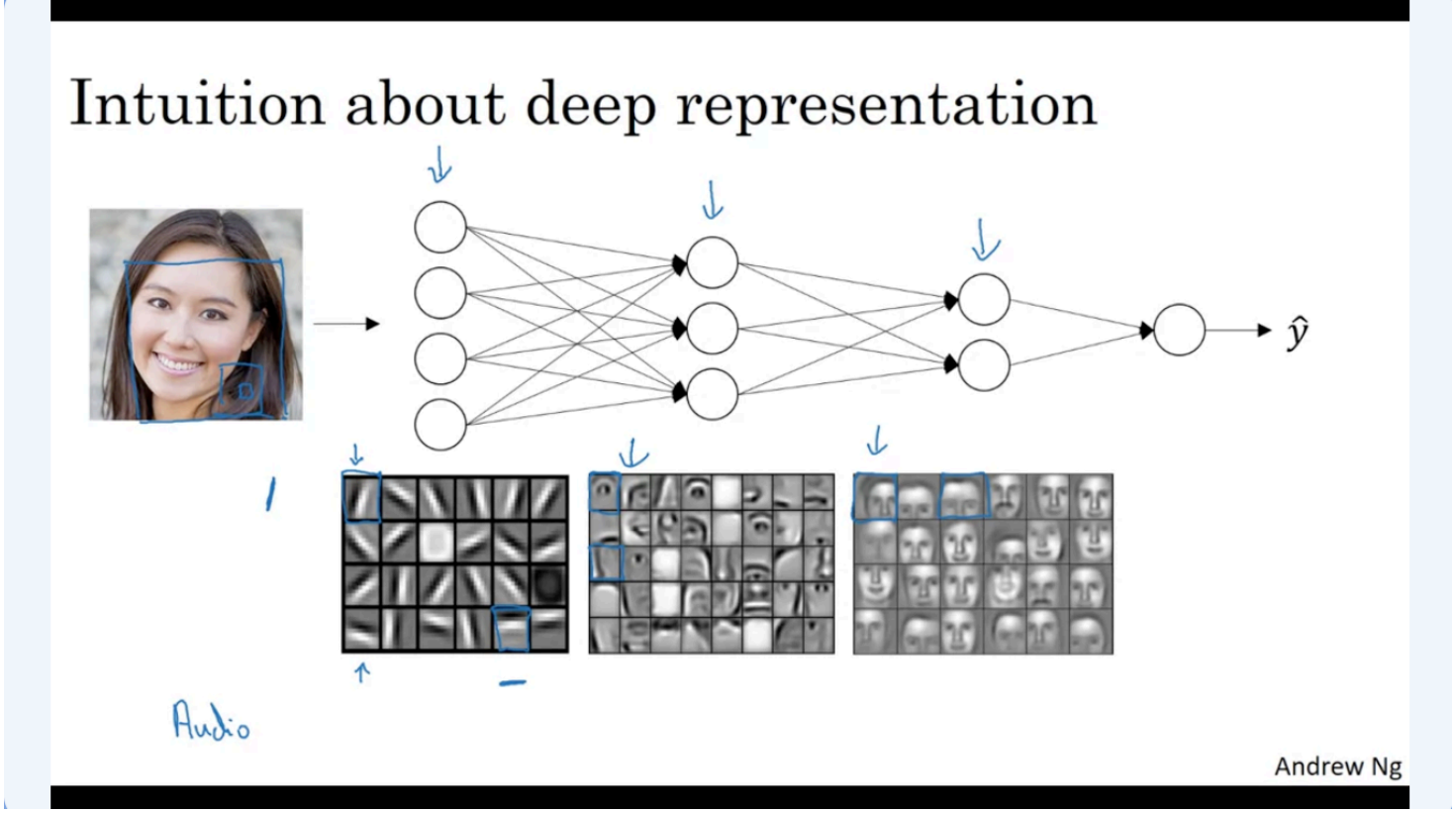
Neural Network Visualization (Text Representation):
• **Layer 0 (Input Layer):**
• 3 input features x_1, x_2, x_3
• n^{[0]} = 3 (number of neurons)
• **Layer 1 (Hidden Layer 1):**
• 5 neurons
• n^{[1]} = 5
• **Layer 2 (Hidden Layer 2):**
• 5 neurons
• n^{[2]} = 5
• **Layer 3 (Hidden Layer 3):**
• 3 neurons
• n^{[3]} = 3
• **Layer 4 (Output Layer):**
• 1 neuron (output)
• n^{[4]} = 1
• **Total Layers (l):**
• l = 4

Forward Propagation in a DNN



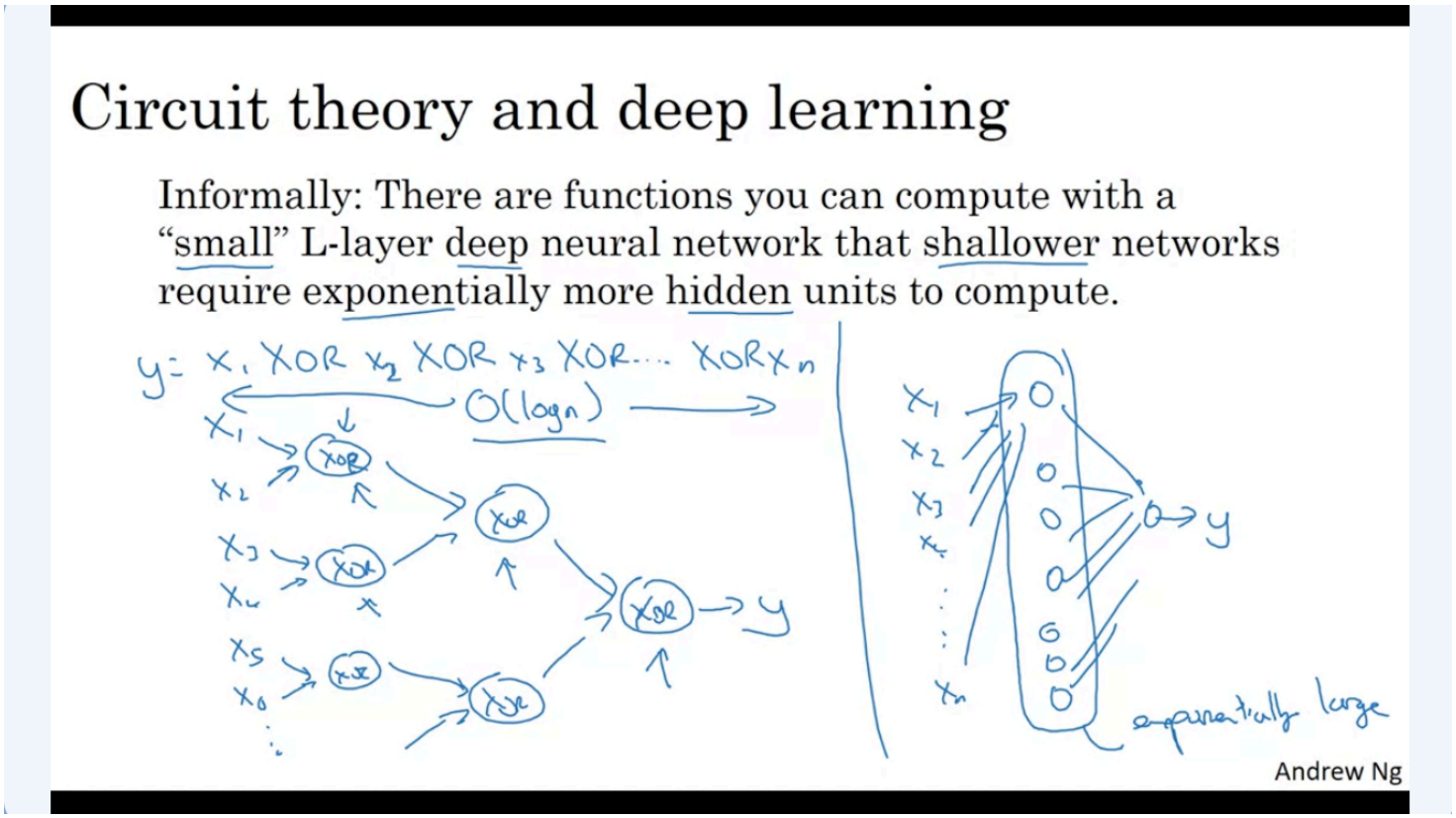
Get your Matrix Dimensions right

Why deep networks?
that famous representation



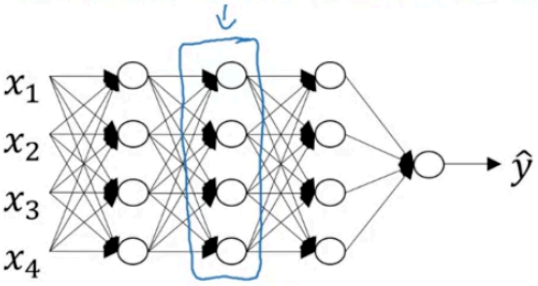
ex: for audio - low level - audio waveform (different sounds) - words - sentences/phrases

Circuit Theory



Building blocks of Deep NNs:

Forward and backward functions



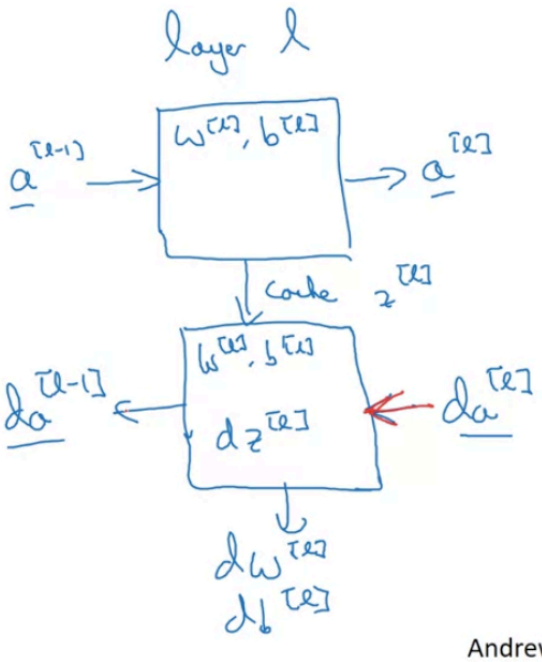
Layer l: $W^{[l]}, b^{[l]}$

→ Forward: Input $a^{[l-1]}$, output $a^{[l]}$

$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$ cache $z^{[l]}$

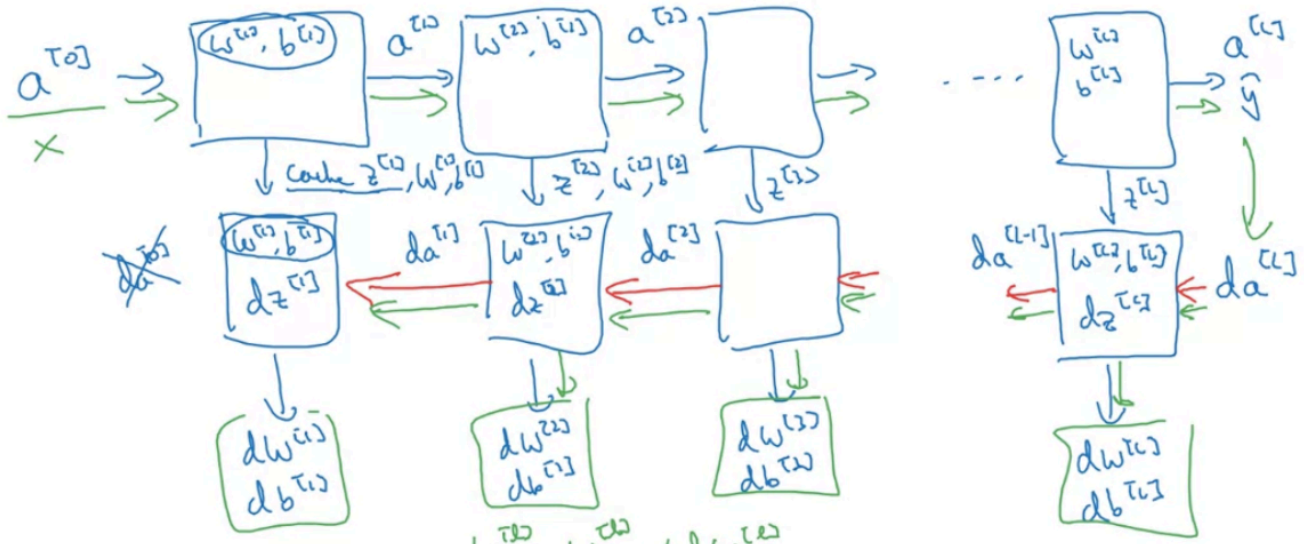
$a^{[l]} = g^{[l]}(z^{[l]})$

→ Backward: Input $da^{[l]}$, cache($z^{[l]}$), output $da^{[l-1]}$, $dW^{[l]}$, $db^{[l]}$



Andrew Ng

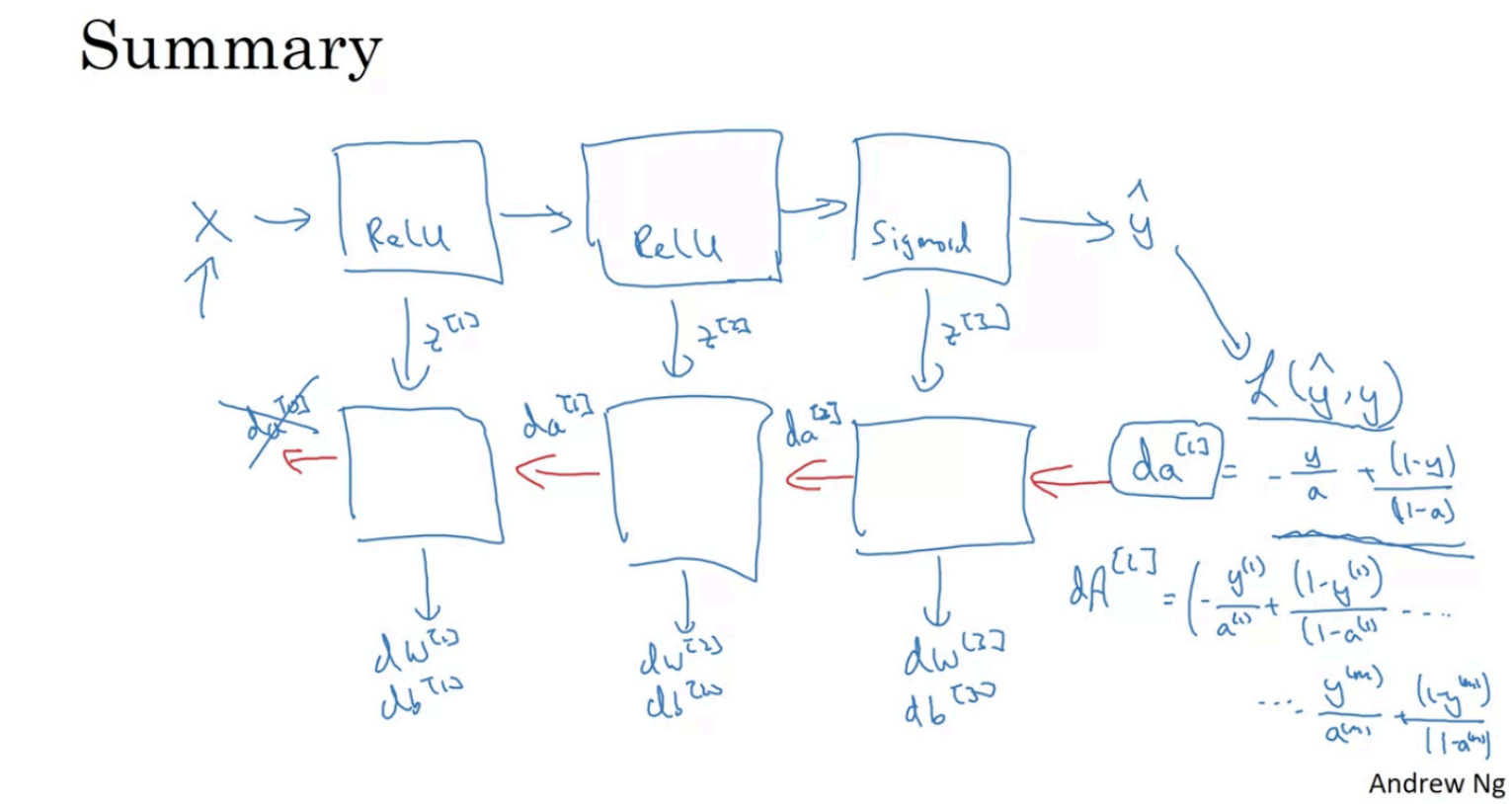
Forward and backward functions



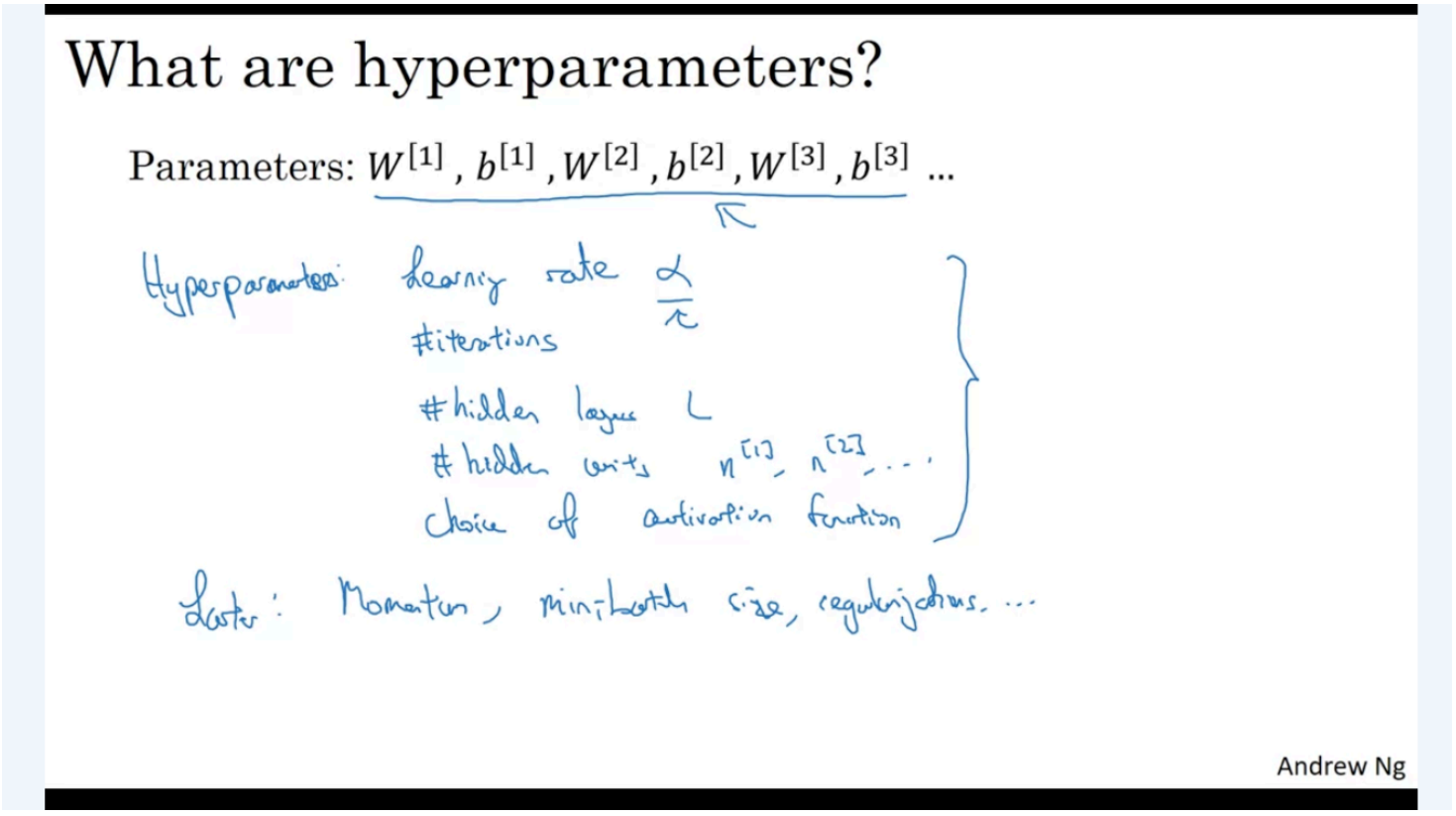
$W^{[l]} := W^{[l]} - \alpha dW^{[l]}$

$b^{[l]} := b^{[l]} - \alpha db^{[l]}$

Andrew Ng



Parameters & Hyperparameters ?



Deep Learning and Brain ?

much more simplex, can not even understand a single neuron yet
it just *resembles*

Summary of Neural Network Training and Inference (Running the Model):

1. Training a Neural Network (Step-by-Step):

Training involves **forward propagation, loss computation, backward propagation, and parameter update**. This process is repeated for multiple iterations (epochs) to minimize the cost function.

Step 1: Forward Propagation (Compute Predictions)

Purpose: Calculate predictions by passing input data through the network layer by layer.

Steps (per layer):

1. **Linear Step:**

• Linear output before activation
• Weights for layer
• Activations from the previous layer (or input for the first layer)
• Bias vector for layer

2. **Activation Step:**

• is the activation function:
• **ReLU** for hidden layers:
• **Sigmoid** for the output layer (for binary classification):

Step 2: Compute Cost (Loss Function)

Purpose: Measure how far predictions are from the true labels.

Cost Function (Binary Cross-Entropy):
• – Number of examples
• – True labels
• – Predicted output from the final layer

Step 3: Backward Propagation (Compute Gradients)

Purpose: Calculate gradients to adjust weights and biases to minimize the loss.

Steps (per layer, from last to first):

1. **Output Layer Gradient (Final Layer):**
   • Gradient of the loss with respect to .
2. **Hidden Layer Gradients (for each layer from to 1):**
   • – Derivative of the activation function (ReLU or sigmoid).
3. **Gradients for Weights and Biases:**

Step 4: Update Parameters (Gradient Descent Step)

Purpose: Adjust weights and biases using gradients to minimize the cost.
• – Learning rate (controls the size of updates).

2. Running the Neural Network (After Training - Inference):

Now that the model is trained, inference involves **forward propagation only**.

Steps for Inference (Forward Propagation):

1. **Input Data** – Pass input through the network.
2. **Linear and Activation Steps (Layer by Layer):**
   • is **ReLU** for hidden layers and **sigmoid** for the final layer.
3. **Final Output:**
   • – Prediction (probability between 0 and 1 for binary classification).
4. **Thresholding for Binary Classification:**

Order of Operations (Training Phase):

1. **Forward Propagation (Layer by Layer):**
   • Linear Step:
   • Activation:
2. **Compute Cost**
3. **Backward Propagation (Layer by Layer, Reverse Order):**
   • Compute for output layer
   • Calculate gradients
4. **Parameter Update:**
5. **Repeat for Multiple Iterations**

Order of Operations (Inference Phase - Running the Model):

1. **Input Data:**
2. **Forward Propagation (Layer by Layer):**
3. **Final Prediction:**
   • (Output Layer)
4. **Thresholding for Binary Classification:**

Key Concepts Simplified:
• **Forward Propagation:** Pass input through the network to compute predictions.
• **Cost Function:** Measures how good/bad the predictions are.
• **Backward Propagation:** Calculate gradients to improve weights/biases.
• **Parameter Update:** Adjust weights/biases to reduce the cost.
• **Repeat:** Continue this process until the model converges (low cost).