WEEK 2 NEURAL NETWORK TRAINING

This week includes: NN training in Tensorflow, multiclass classification, multi-label classification, adam and **backpropagation**.

First week we learned how to carry out **inferencing**. Now we **train**.

How do we adjust the the parameters of the neural network?

What does **model.compile** and **model.fit** do exactly?

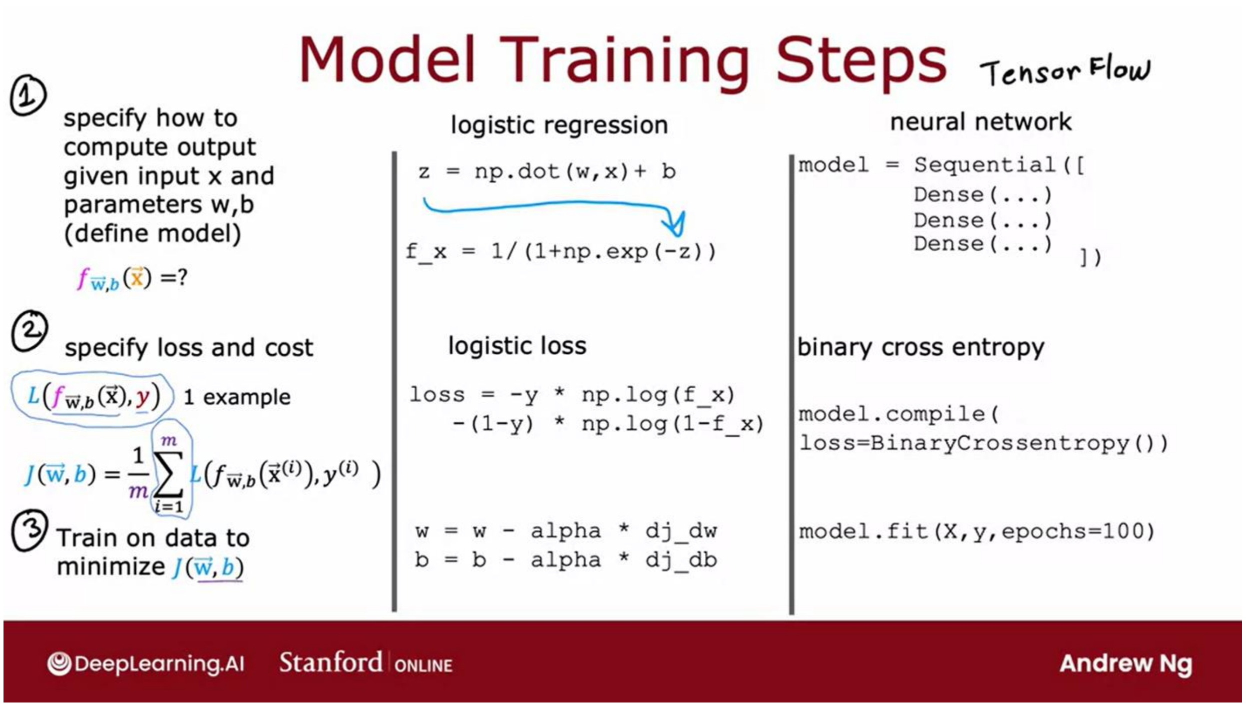
CONCEPTUAL MENTAL FRAMEWORK OF YOURS NEEDS TO DEEPLY UNDERSTAND THESE.

MODEL TRAINING STEPS

In the first course, say for logistic regression, we followed these steps:

1) Figure out what our function f(w, b) = x is. Engineer features if needed.
2) Specify LOSS and COST functions. (cost is average loss)
3) Train to minimize J(w, b). **Use an optimization algorithm to do so.**

Neural network requires these 3 steps as well.



Model Training Steps

Tensor Flow

	logistic regression	neural network

① specify how to compute output given input x and parameters w,b (define model) $f_{\vec{w},b}(\vec{x}) = ?$	`z = np.dot(w,x)+ b`
	`f_x = 1/(1+np.exp(-z))`
		`model = Sequential([`
		`Dense(...)`
		`Dense(...)`
		`Dense(...)`
		`])`

② specify loss and cost $L(f_{\vec{w},b}(\vec{x}), y)$ 1 example
$J(\vec{w}, b) = \frac{1}{m}\sum_{i=1}^{m} L(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)})$
	logistic loss
	`loss = -y * np.log(f_x)`
	`-(1-y) * np.log(1-f_x)`
		binary cross entropy
		`model.compile(`
		`loss=BinaryCrossentropy())`

③ Train on data to minimize $J(\vec{w}, b)$	`w = w - alpha * dj_dw`
	`b = b - alpha * dj_db`
		`model.fit(X,y,epochs=100)`

Neural networks:

1) Model creating is what you do with “Sequential” and “Dense”.
2) Choose loss and cost functions: In binary classification, we use “**binary cross entropy loss function**”. “**model.compile(loss = BinaryCrossentropy())**”
   Minimizing cost function will result in fitting the neural network to your data.
   If you want to solve a regression problem:
   You can “**model.compile(loss = MeanSquaredError())**”

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

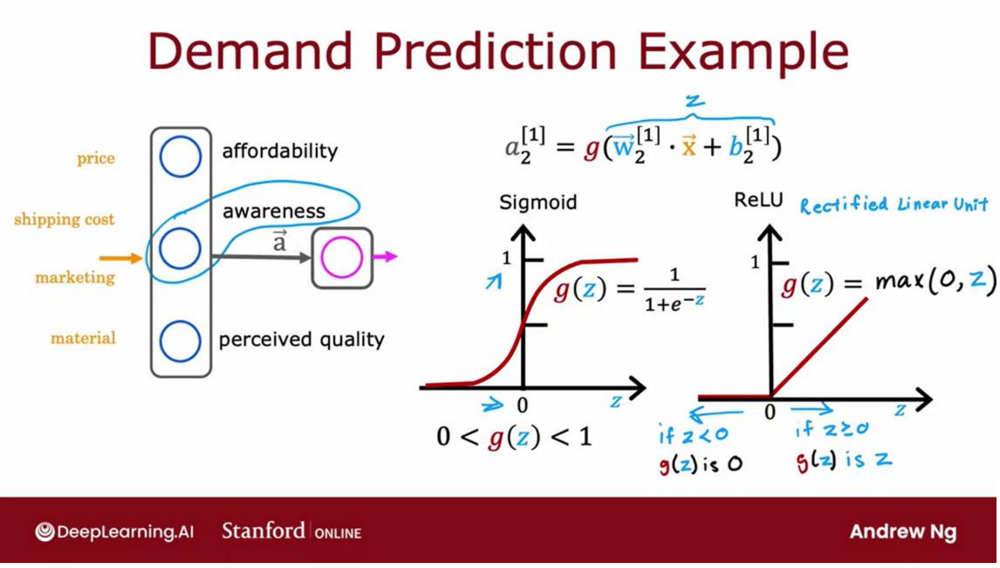
$\mathbf{W}^{[1]}, \mathbf{W}^{[2]}, \mathbf{W}^{[3]}$ $\quad \vec{b}^{[1]}, \vec{b}^{[2]}, \vec{b}^{[3]}$ $\quad f_{W,B}(\vec{x})$

*****IN NEURAL NETWORKS, COST FUNCTION IS A FUNCTION OF ALL THE PARAMETERS OF YOUR NEURAL NETWORK. THEREFORE IT’S PARAMETERS ARE MATRIXES.**

3) Last step is to minimize J, which means running an opt. alg. For parameter updates, we need to compute partial derivatives. **THIS IS DONE BY USING “BACKPROPAGATION”.**

Activation functions

This selection improves the network.



Awareness might not be a binary value. It doesn’t have to be between 0 and 1. This can differ depending on the problem we’re solving.

ReLU is prettty common.

Later on we’ll learn “softmax”.

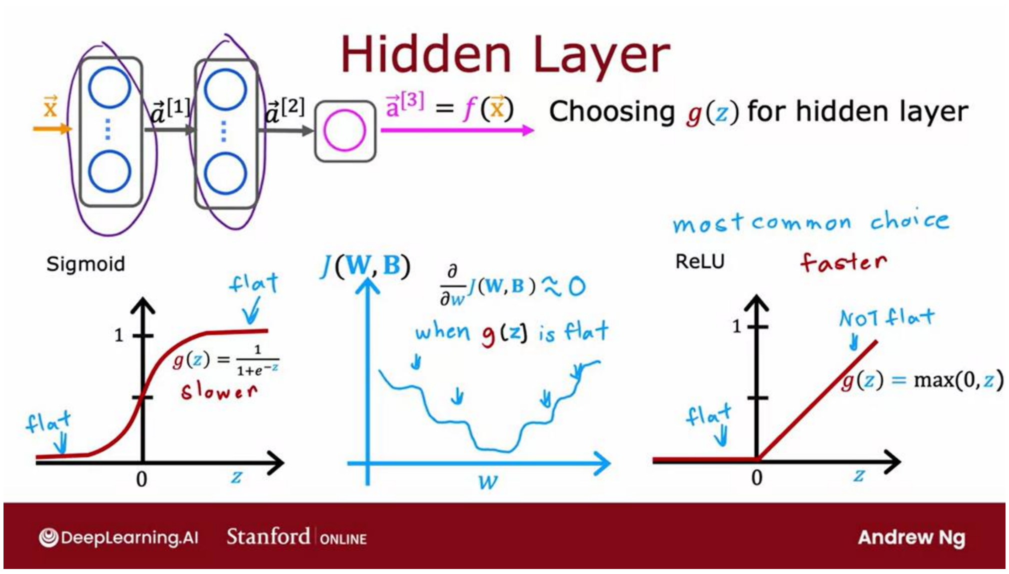
How do i choose?

Depending on the form of our prediction, act. fun. of the output layer can be chosen naturally. For binary classification, sigmoid is the choice. For regression, “linear activation function” can be used. If y can only get non-negative number, choose ReLU.

**What about hidden layers?

ReLU is by far the most common choice. Development made sigmoid chosen hardly ever, except in output layer, which we use in case of a binary classification problem. But why?

ReLU is faster. But more importantly; relu has 1 flat side while sigmoid has two. This can be a problem in optimization.

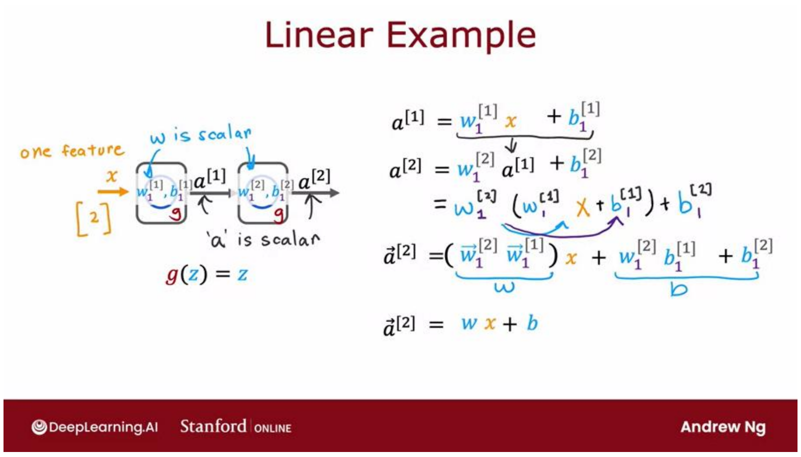


CONCLUSION: Output → choose   Hidden → just use ReLU

There are others like**: tanh, leakyrelu, swich…**

Why activation functions are used:

If every neuron's activation is linear, then it would only solve linear regression. Which means we did not gain anything from using a neural network. (We solved it in a singular model in the first course) Activation functions grant us the ability to solve more complex problems, via magic of math.



Linear Example
one feature
w is scalar
x
$[2]$
$w_1^{[1]}, b_1^{[1]}$
$a^{[1]}$
$w_1^{[2]}, b_1^{[2]}$
$a^{[2]}$
g
'a' is scalar
$g(z) = z$
$a^{[1]} = w_1^{[1]} x + b_1^{[1]}$
$a^{[2]} = w_1^{[2]} a^{[1]} + b_1^{[2]}$
$= w_1^{[2]} (w_1^{[1]} x + b_1^{[1]}) + b_1^{[2]}$
$\vec{a}^{[2]} = \underbrace{(\vec{w}_1^{[2]} \vec{w}_1^{[1]})}_{w} x + \underbrace{w_1^{[2]} b_1^{[1]} + b_1^{[2]}}_{b}$
$\vec{a}^{[2]} = w x + b$
DeepLearning.AI
Stanford ONLINE
Andrew Ng

Result is same. Just use linear regression model. Linear function of a linear function is still linear.

NOTE: Hidden layers and output layer -> linear → becomes linear regression

Hidden layers linear, output layer -> sigmoid → becomes logistic regression

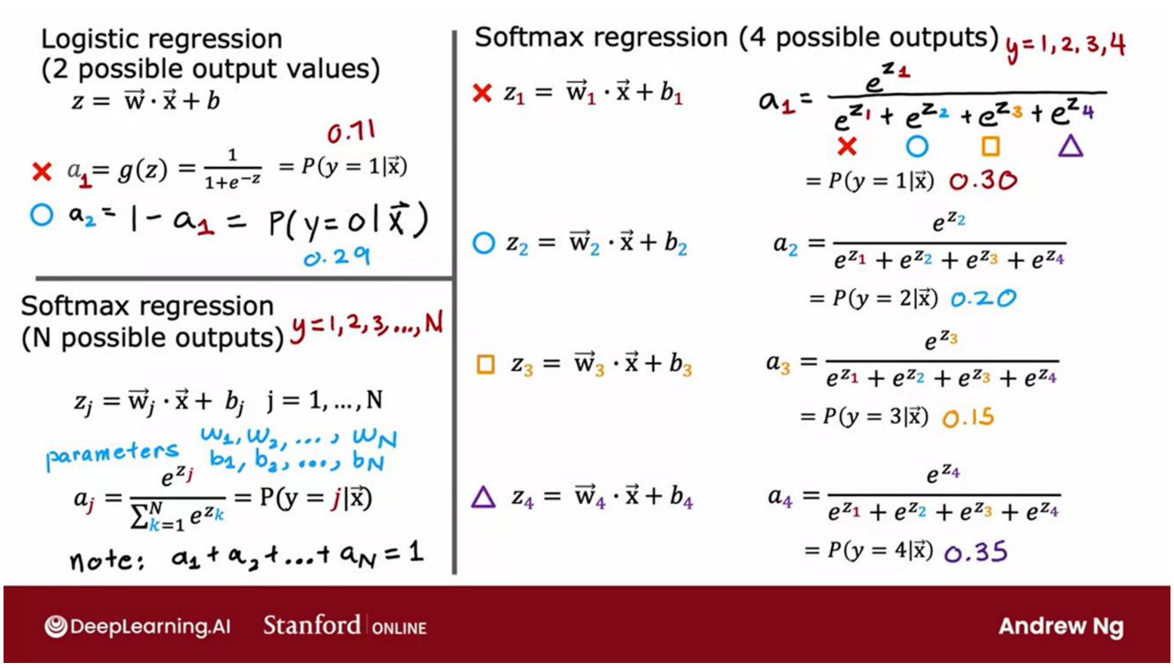
SUGGESTION: Dont use linear activation in hidden layers. **Use ReLU instead.**

Multiclass Classification

0-1 identifiying becomes digit or letter recognition.

In graph, there are multiple clusters with different labels. Y will have probabilities of being every one of these classes. Decision boundary will be different.

SOFTMAX (softmax regression algorithm, generalization of log reg)



Logistic regression
(2 possible output values)

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

× $a_1 = g(z) = \frac{1}{1+e^{-z}} = P(y=1|\vec{x})$ 0.71

○ $a_2 = 1 - a_1 = P(y=0|\vec{x})$ 0.29

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 $w_1, w_2, \dots, w_N$; $b_1, b_2, \dots, b_N$

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

note: $a_1 + a_2 + \dots + a_N = 1$

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

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

$a_1 = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}}$
× ○ □ △

$= P(y=1|\vec{x})$ 0.30

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

$a_2 = \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}}$
$= P(y=2|\vec{x})$ 0.20

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

$a_3 = \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}}$
$= P(y=3|\vec{x})$ 0.15

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

$a_4 = \frac{e^{z_4}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}}$
$= P(y=4|\vec{x})$ 0.35

DeepLearning.AI Stanford ONLINE Andrew Ng

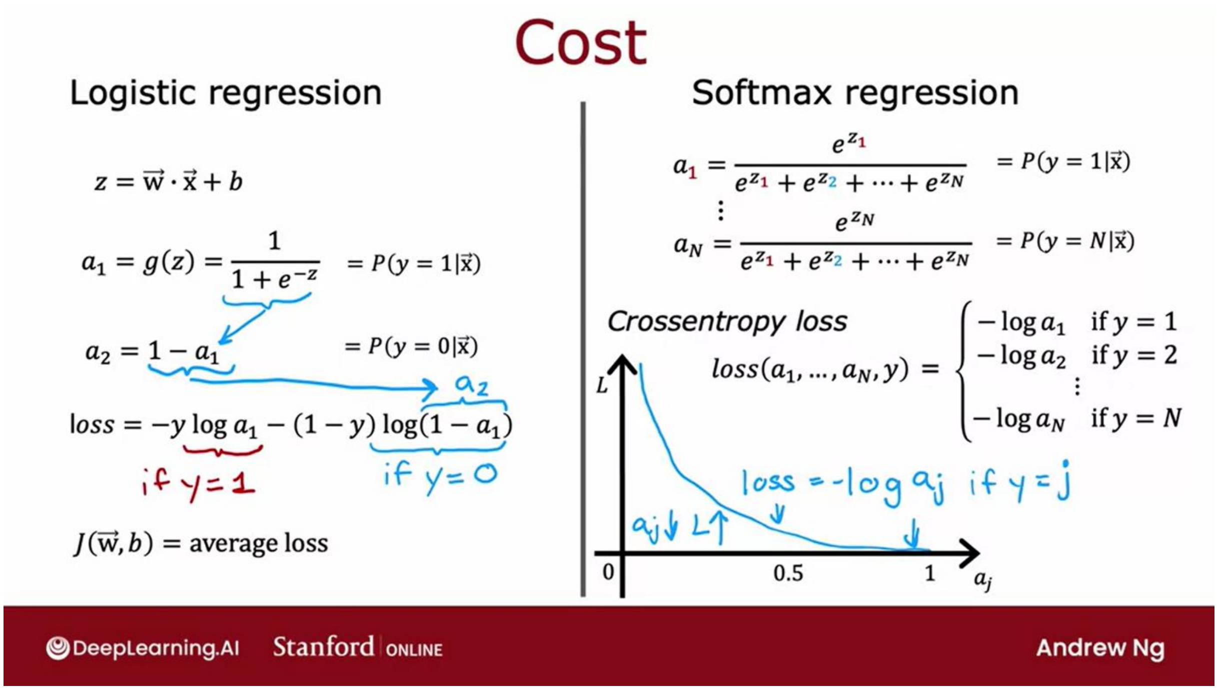
It's generalization of log. reg. which was binary.

If you use it with N = 2, it simplifies to logistic regression.

Way to go: Just compute the label the same as log reg but this time every label has it's own parameters. Every computation gives probability of that class. **But** we need to find those parameters somehow. That's the extra.

COST FUNCTION FOR SOFTMAX

For log reg, we used binary cross entropy formula.



Cost

Logistic regression

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

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

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

$\text{loss} = -y \underbrace{\log a_1}_{\text{if } y=1} - (1-y) \underbrace{\log(\overbrace{1-a_1}^{a_2})}_{\text{if } y=0}$

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

Softmax regression

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

$\vdots$

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

Crossentropy loss

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

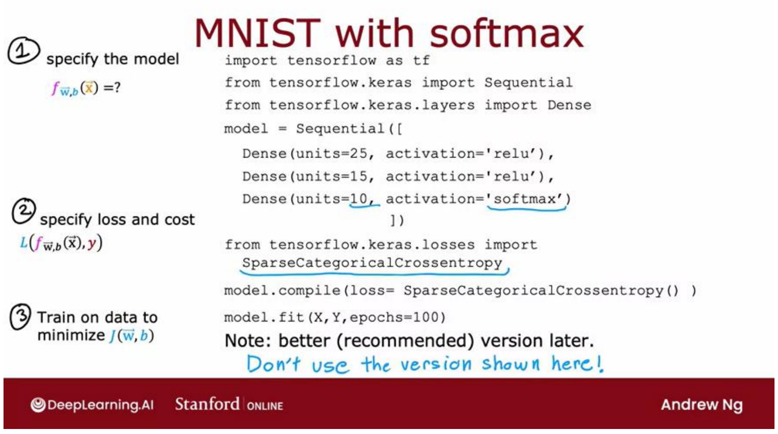
DeepLearning.AI Stanford ONLINE Andrew Ng

It’s the same as logistic regression, just with more situations, so we leave the function piecewise. Loss is computed via the true label of the sample.

Neural network with SOFTMAX

Our output layer will be a softmax layer: It’ll have multiple outputs. Every procedure is run the same way as before in hidden layers. For every output unit, z_i and a_i will be computed.

In every computation, x is activation from the previous layer and w, b are the parameters of that unit. With softmax, a_i is a function of z_1, z_2 … and z_10. It is affected by others as well. So a_3(z_1, z_2…) = g(z_1, z_2…)



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)
```

Note: better (recommended) version later.
Don't use the version shown here!

DeepLearning.AI Stanford ONLINE Andrew Ng

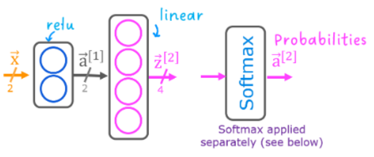
NOTE: SparseCategoricalCrossentropy (loss function), sparse means y can only take one of these values. Meaning they are disjoint classes.

NOTES FROM LABS

2.2 Model

This lab will use a 2-layer network as shown. Unlike the binary classification networks, this network has four outputs, one for each class. Given an input example, the output with the highest value is the predicted class of the input.

Below is an example of how to construct this network in Tensorflow. Notice the output layer uses a `linear` rather than a `softmax` activation. While it is possible to include the softmax in the output layer, it is more numerically stable if linear outputs are passed to the loss function during training. If the model is used to predict probabilities, the softmax can be applied at that point.



```
tf.random.set_seed(1234)  # applied to achieve consistent results
model = Sequential(
    [
        Dense(2, activation = 'relu',   name = "L1"),
        Dense(4, activation = 'linear', name = "L2")
    ]
)
```

The statements below compile and train the network. Setting `from_logits=True` as an argument to the loss function specifies that the output activation was linear rather than a softmax.

```
model.compile(
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    optimizer=tf.keras.optimizers.Adam(0.01),
)
```

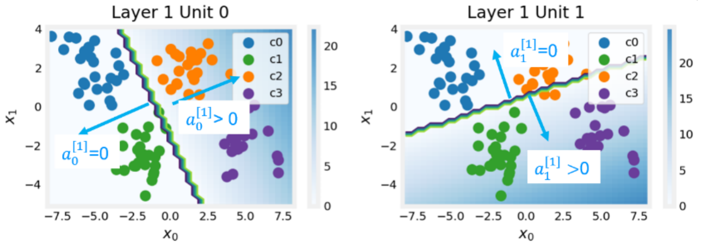
- Use softmax in the end because linear layer outputs numbers, not probabilities. Then add “from_logits” to loss’ parameters.

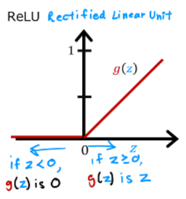
Explanation

Layer 1

These plots show the function of Units 0 and 1 in the first layer of the network. The inputs are (x_0, x_1) on the axis. The output of the unit is represented by the color of the background. This is indicated by the color bar on the right of each graph. Notice that since these units are using a ReLu, the outputs do not necessarily fall between 0 and 1 and in this case are greater than 20 at their peaks. The contour lines in this graph show the transition point between the output, $a^{[1]}_j$ being zero and non-zero. Recall the graph for a ReLu : The contour line in the graph is the inflection point in the ReLu.

$a^{[1]}_0 = relu(w_{00}x_0 + w_{01}x_1)$ $\qquad$ $a^{[1]}_1 = relu(w_{10}x_0 + w_{11}x_1)$



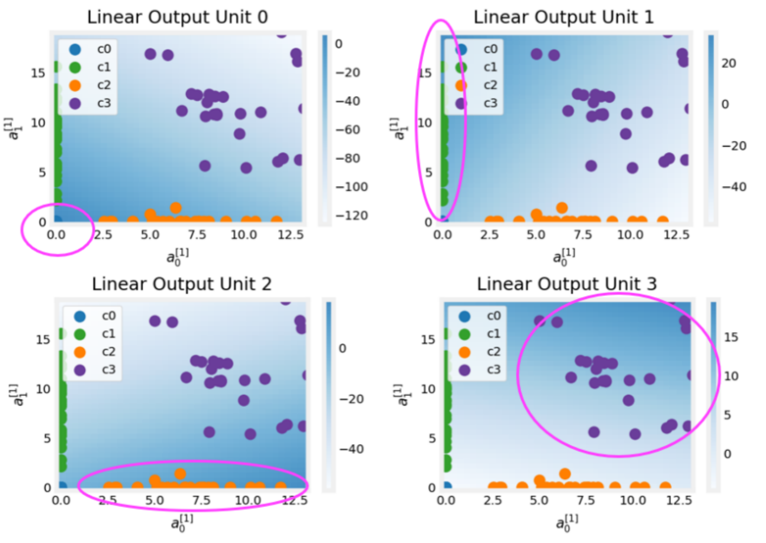


Unit 0 has separated classes 0 and 1 from classes 2 and 3. Points to the left of the line (classes 0 and 1) will output zero, while points to the right will output a value greater than zero.

Unit 1 has separated classes 0 and 2 from classes 1 and 3. Points above the line (classes 0 and 2) will output a zero, while points below will output a value greater than zero. Let's see how this works out in the next layer!

- Used a layer with 2 units. Each unit did it’s own job.

Layer 2, the output layer



The dots in these graphs are the training examples translated by the first layer. One way to think of this is the first layer has created a new set of features for evaluation by the 2nd layer. The axes in these plots are the outputs of the previous layer $a_0^{[1]}$ and $a_1^{[1]}$. As predicted above, classes 0 and 1 (blue and green) have $a_0^{[1]} = 0$ while classes 0 and 2 (blue and orange) have $a_1^{[1]} = 0$.
Once again, the intensity of the background color indicates the highest values.
Unit 0 will produce its maximum value for values near (0,0), where class 0 (blue) has been mapped.
Unit 1 produces its highest values in the upper left corner selecting class 1 (green).
Unit 2 targets the lower right corner where class 2 (orange) resides.
Unit 3 produces its highest values in the upper right selecting our final class (purple).

One other aspect that is not obvious from the graphs is that the values have been coordinated between the units. It is not sufficient for a unit to produce a maximum value for the class it is selecting for, it must also be the highest value of all the units for points in that class. This is done by the implied softmax function that is part of the loss function (`SparseCategoricalCrossEntropy`). Unlike other activation functions, the softmax works across all the outputs.

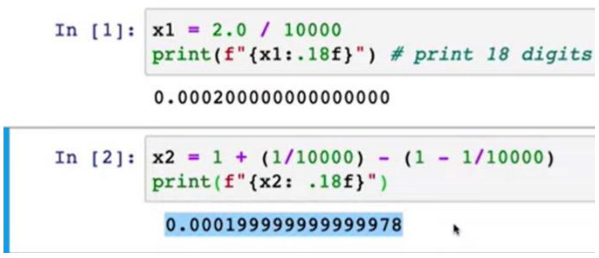
You can successfully use neural networks without knowing the details of what each unit is up to. Hopefully, this example has provided some intuition about what is happening under the hood.

Now back to lectures.

IMPROVEMENTS FOR SOFTMAX (AND RECOMMENDED WAY OF CODING)

What can go wrong with the previous implementation?

ROUNDOFF ERROR: Depending on the way of computing, roundoff errors might happen. This situation can happen in softmax implementation.



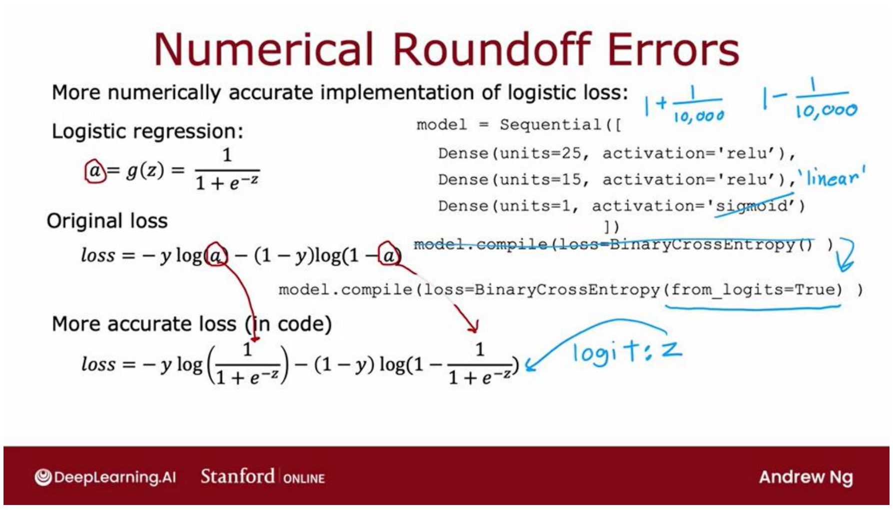
```
In [1]: x1 = 2.0 / 10000
        print(f"{x1:.18f}") # print 18 digits

        0.000200000000000000

In [2]: x2 = 1 + (1/10000) - (1 - 1/10000)
        print(f"{x2: .18f}")

        0.000199999999999978
```

**To compute more numercally accurate logistic loss: REMOVE INTERMEDIATE VALUE



Numerical Roundoff Errors

More numerically accurate implementation of logistic loss:

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

Logistic regression:

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

Original loss

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

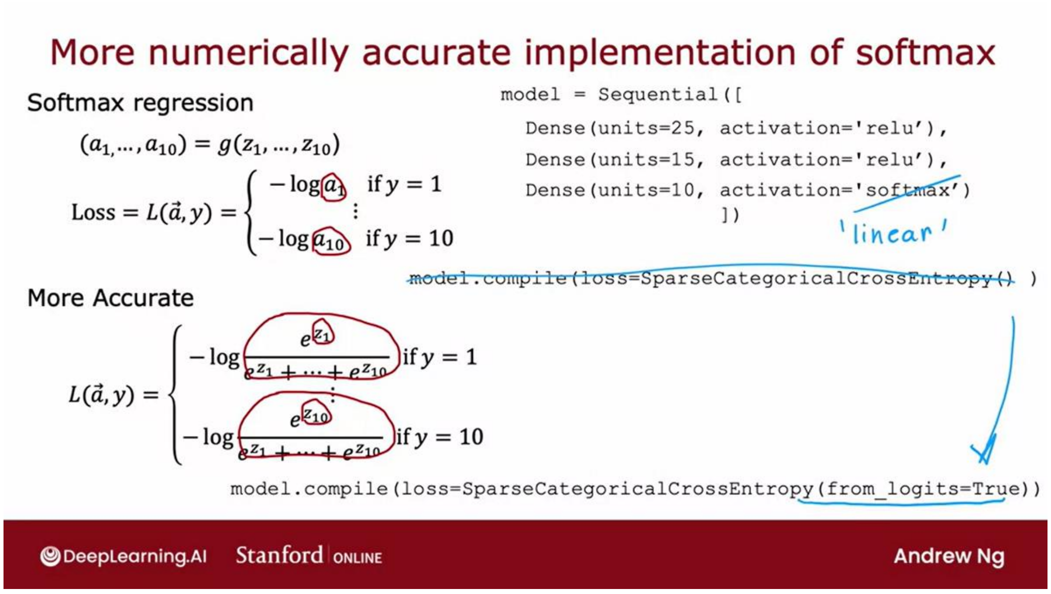
More accurate loss (in code)

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

```
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) )
```

'linear'

DeepLearning.AI Stanford ONLINE Andrew Ng



More numerically accurate implementation of softmax

Softmax regression

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

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

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

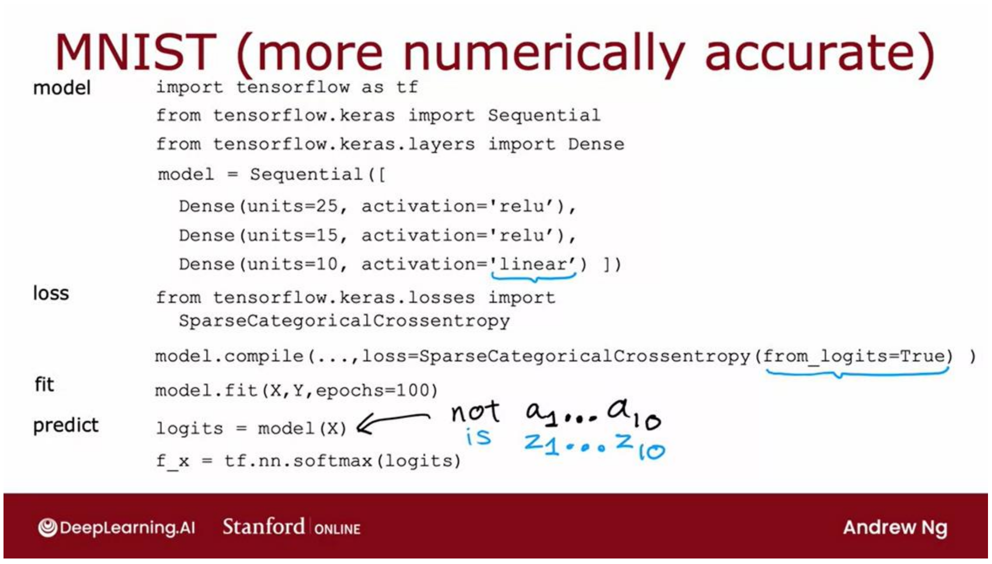
'linear'

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

DeepLearning.AI Stanford ONLINE Andrew Ng

***We'll apply the same rule to softmax regression:

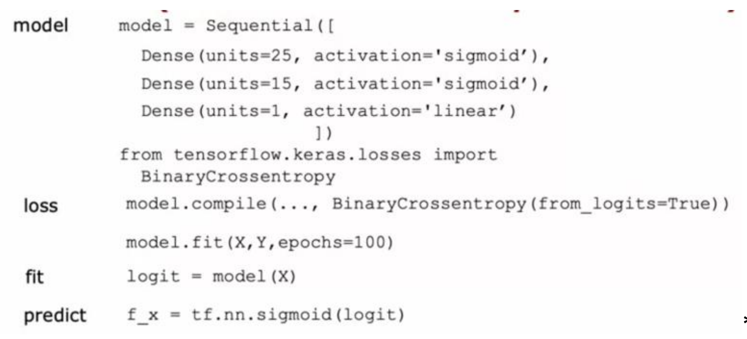
If z is very small or very large, e^z becomes so much bigger or smaller.



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)
```

In the example to the left, final layer is now linear activated. **In this form, model returns z, not a. So we need to take these z's and put them into a softmax function.**



```
model     model = Sequential([
            Dense(units=25, activation='sigmoid'),
            Dense(units=15, activation='sigmoid'),
            Dense(units=1, activation='linear')
                              ])
          from tensorflow.keras.losses import
            BinaryCrossentropy
loss      model.compile(..., BinaryCrossentropy(from_logits=True))
          model.fit(X,Y,epochs=100)
fit       logit = model(X)
predict   f_x = tf.nn.sigmoid(logit)
```

***Same applies to logistic regression.

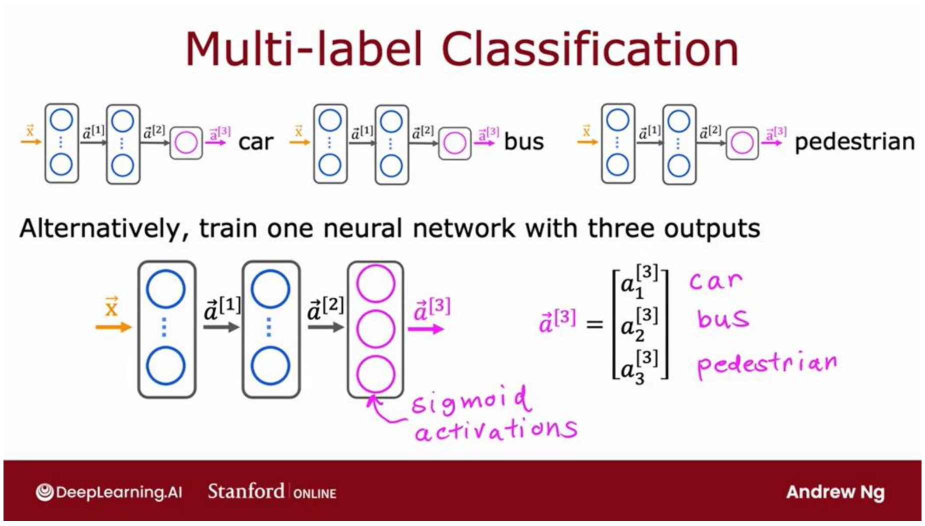
In summary: We have 2 ways: compute from logits and then take the result into softmax/sigmoid function, or do the procedure in the normal way.

Watch these again maybe

MULTI LABEL CLASSIFICATION

Sample belongs to multiple labels. Populate yourself further.

Eaxmple: What do we have in this image? Car, bus, person etc.


Multi-label Classification
$\vec{x}$
$\vec{a}^{[1]}$
$\vec{a}^{[2]}$
$\vec{a}^{[3]}$
car
bus
pedestrian
Alternatively, train one neural network with three outputs
sigmoid activations
$\vec{a}^{[3]} = \begin{bmatrix} a_1^{[3]} \\ a_2^{[3]} \\ a_3^{[3]} \end{bmatrix}$
car
bus
pedestrian
DeepLearning.AI
Stanford ONLINE
Andrew Ng

In this problem, target value is a vector, say 0 or 1 or every label. You can approach this as three different problems. Use a neural network for all labels. **Alternative,** train 1 NN with 3 outputs. You can use sigmoid for all three.

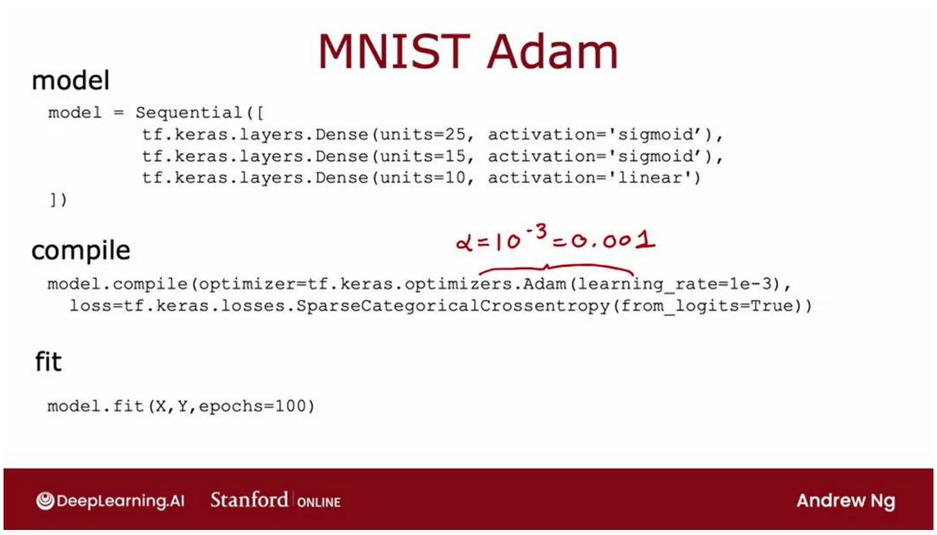
- **The default size of a batch in Tensorflow is 32.**

So if you have 5000 training examples, you'd have 157 batches.

Next: Additional NN Concepts

GD was the foundation but today we have better optimization algorithms. Can we have an algorithm that increases the learning rate when we are going in the same direction? In contrast, if we are oscillating back and forth, we need to take smaller steps. **ADAM does this.**

Adaptive Moment Estimation: It adjust the learning rate automatically.



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

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

```
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)
```

DeepLearning.AI Stanford ONLINE Andrew Ng

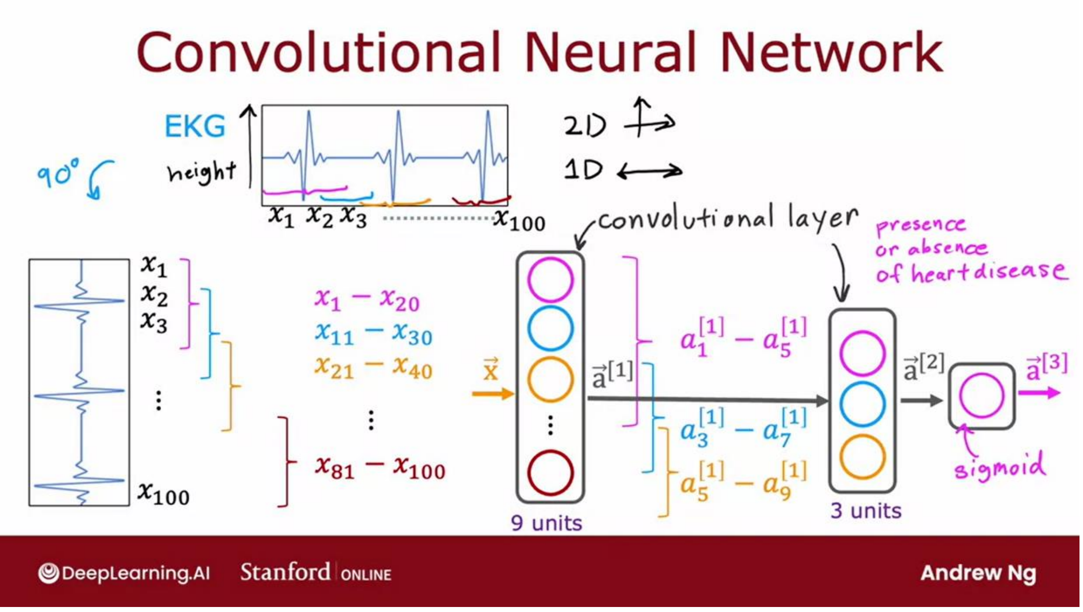
Works faster than GD. It’s a safe choice, also robust to noise.

MORE!!! Alternative Layer Types (other than Dense)

Dense: Every neuron in this layer is tied to every neuron of the next layer. They have a weight between them.

Convolutional layer: Imagine the first layer in digit recognition with a 3x3 image. First neuron looks at one pixel, second neuron maybe looks at 4 pixels… Every neuron looks at some of the pixels. IT GIVES YOU FASTER COMPUTATION AND NEEDS LESS DATA (ALSO PRONE TO OVERFITTING).

John Macoun found out these and popularized them. It’s used in image interpreting.



Convolutional Neural Network

DeepLearning.AI Stanford ONLINE Andrew Ng

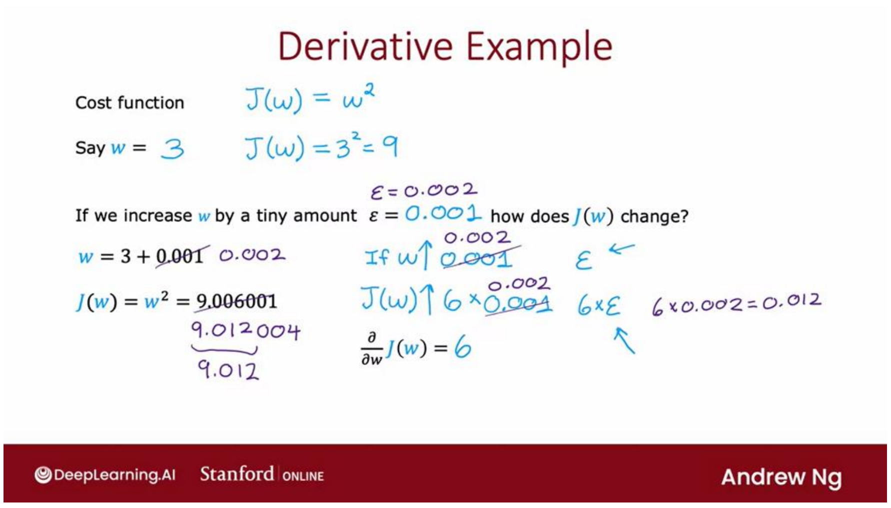
With CNN’s, there are multiple new choices to make: How many neurons should i use, how many past neurons should a neuron examine?

It can be more effective than Dense in some applications.

Others: Transformer, LTSM …

Backpropagation

Computes derivatives of the cost function.



Derivative Example

Cost function $J(w) = w^2$

Say $w = 3$ $J(w) = 3^2 = 9$

If we increase w by a tiny amount $\varepsilon = 0.001$ $\varepsilon = 0.002$ how does $J(w)$ change?

$w = 3 + \cancel{0.001}$ 0.002 If $w \uparrow \overset{0.002}{\cancel{0.001}}$ $\varepsilon \leftarrow$

$J(w) = w^2 = \cancel{9.006001}$ $\underbrace{9.012}_{9.012}004$ $J(w) \uparrow 6 \times \overset{0.002}{\cancel{0.001}}$ $6 \times \varepsilon$ $6 \times 0.002 = 0.012$

$\frac{\partial}{\partial w} J(w) = 6$

DeepLearning.AI Stanford ONLINE Andrew Ng

Basic presentation of derivative.

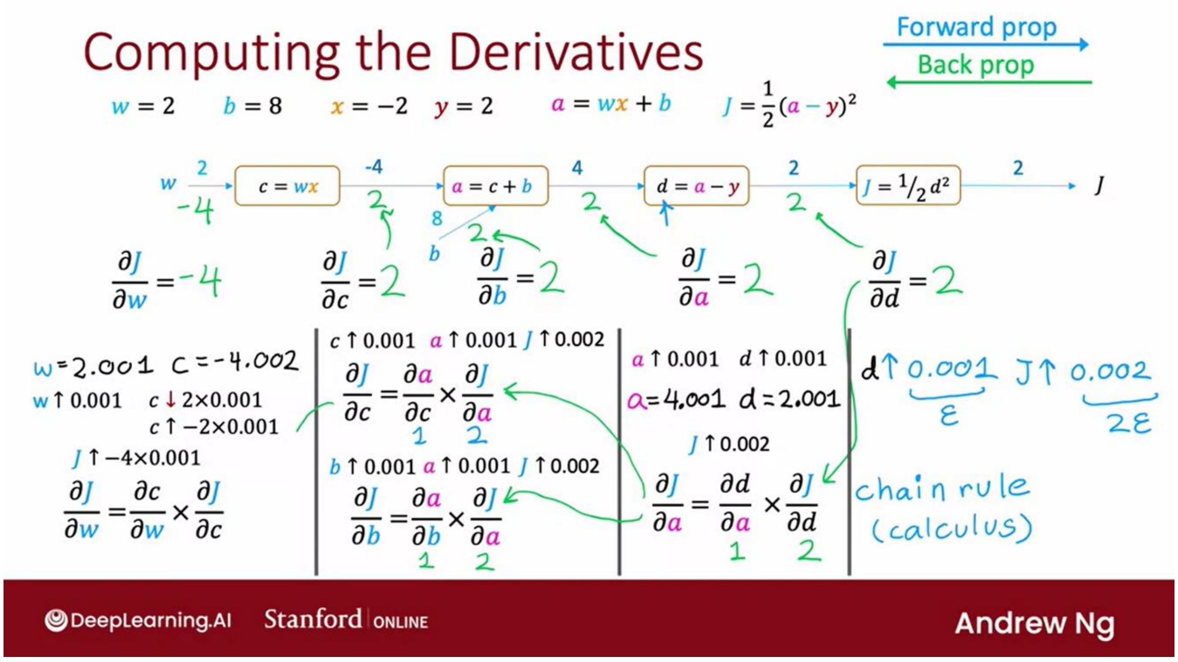
*Can compute derivatives with **“sympy”**.

Informal Definition: If w increases e and J increases k*e, then k is the derivative.

Next: computing derivatives in a neural network → Computation graph

Graph here refers to literal meaning of graph in computer science (consists of nodes).

*****MAIN IDEA:** Start from last node, go back till the first node while computing derivatives of every nodes derivative (actually derivative of the variable inside that node). Make it to w like this.



Computing the Derivatives

Forward prop →

← Back prop

$w = 2$ $b = 8$ $x = -2$ $y = 2$ $a = wx + b$ $J = \frac{1}{2}(a-y)^2$

$$\begin{tikzcd} w \arrow[r, "2"', "-4"] & \boxed{c = wx} \arrow[r, "-4", "2"'] & \boxed{a = c + b} \arrow[r, "4", "2"'] & \boxed{d = a - y} \arrow[r, "2", "2"'] & \boxed{J = \frac{1}{2}d^2} \arrow[r, "2"] & J \\ & b \arrow[ur, "8"] & & & & \end{tikzcd}$$

$\frac{\partial J}{\partial w} = -4$ $\frac{\partial J}{\partial c} = 2$ $\frac{\partial J}{\partial b} = 2$ $\frac{\partial J}{\partial a} = 2$ $\frac{\partial J}{\partial d} = 2$

$w = 2.001$ $c = -4.002$
$w \uparrow 0.001$ $c \downarrow 2\times0.001$
$c \uparrow -2\times0.001$
$J \uparrow -4\times0.001$
$\frac{\partial J}{\partial w} = \frac{\partial c}{\partial w} \times \frac{\partial J}{\partial c}$

$c \uparrow 0.001$ $a \uparrow 0.001$ $J \uparrow 0.002$
$\frac{\partial J}{\partial c} = \underset{1}{\frac{\partial a}{\partial c}} \times \underset{2}{\frac{\partial J}{\partial a}}$
$b \uparrow 0.001$ $a \uparrow 0.001$ $J \uparrow 0.002$
$\frac{\partial J}{\partial b} = \underset{1}{\frac{\partial a}{\partial b}} \times \underset{2}{\frac{\partial J}{\partial a}}$

$a \uparrow 0.001$ $d \uparrow 0.001$
$a = 4.001$ $d = 2.001$
$J \uparrow 0.002$
$\frac{\partial J}{\partial a} = \underset{1}{\frac{\partial d}{\partial a}} \times \underset{2}{\frac{\partial J}{\partial d}}$

$d \uparrow \underbrace{0.001}_{\varepsilon}$ $J \uparrow \underbrace{0.002}_{2\varepsilon}$
chain rule
(calculus)

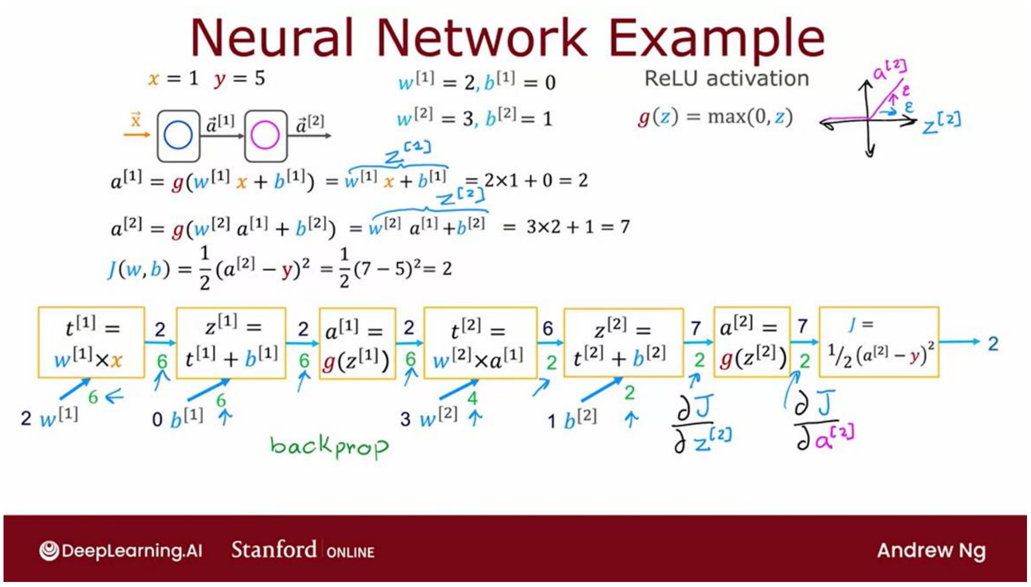
DeepLearning.AI Stanford ONLINE Andrew Ng

Frameworks like Tensorflow use this.

IT USES CHAIN RULE OF DERIVATION.

*****N nodes, P parameters → compute derivatives in** roughly **N+P steps rather than N*P steps.**

Large neural networks benefit so much from it.



Neural Network Example

$x = 1 \quad y = 5$

$w^{[1]} = 2, b^{[1]} = 0$

$w^{[2]} = 3, b^{[2]} = 1$

ReLU activation

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

$$a^{[1]} = g(w^{[1]}x + b^{[1]}) = \overbrace{w^{[1]}x + b^{[1]}}^{z^{[1]}} = 2 \times 1 + 0 = 2$$

$$a^{[2]} = g(w^{[2]}a^{[1]} + b^{[2]}) = \overbrace{w^{[2]}a^{[1]} + b^{[2]}}^{z^{[2]}} = 3 \times 2 + 1 = 7$$

$$J(w, b) = \frac{1}{2}(a^{[2]} - y)^2 = \frac{1}{2}(7 - 5)^2 = 2$$

$\vec{x}$
$\vec{a}^{[1]}$
$\vec{a}^{[2]}$
$a^{[2]}$
ε
ε
$z^{[2]}$
$t^{[1]} = w^{[1]} \times x$
$z^{[1]} = t^{[1]} + b^{[1]}$
$a^{[1]} = g(z^{[1]})$
$t^{[2]} = w^{[2]} \times a^{[1]}$
$z^{[2]} = t^{[2]} + b^{[2]}$
$a^{[2]} = g(z^{[2]})$
$J = 1/2\,(a^{[2]} - y)^2$
2 6 2 6 2 6 6 2 7 2 7 2 2
2 $w^{[1]}$ 6
0 $b^{[1]}$ 6
3 $w^{[2]}$ 4
1 $b^{[2]}$ 2
$\frac{\partial J}{\partial z^{[2]}}$
$\frac{\partial J}{\partial a^{[2]}}$
backprop

DeepLearning.AI Stanford ONLINE Andrew Ng

Example with 2 layers.

Compute these derivatives and feed them to your optimization algorithm.