

# Neural Networks

Keras for Deep Learning Research



### Feedback is greatly appreciated!



#### Overview

- Neural Network
- Backpropagation
- Gradient Descent (Optimization Algorithm)
- Cost/Loss Functions
- Activation Function
- Learning Rate

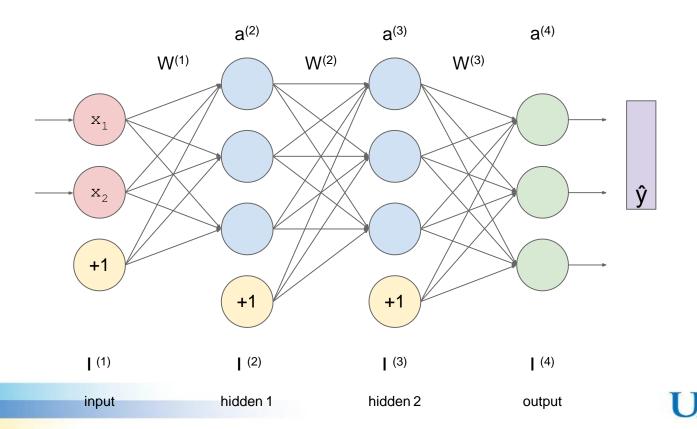


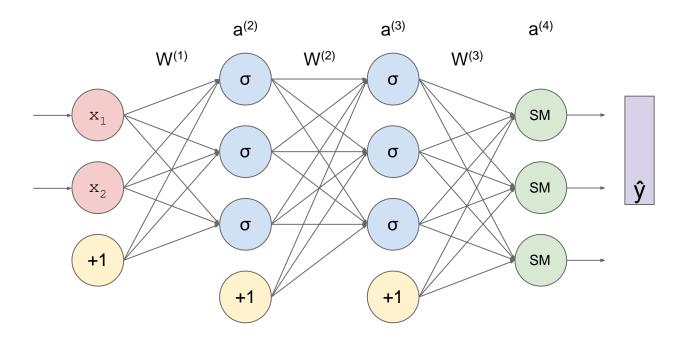
### Artificial Neural Network

- What are artificial neural network?
  - A combination of a training method
  - An optimization method
- Two phases cycle:
  - Propagation
  - Weight update



#### Feed forward neural network





: output vector

 $\sigma$ : sigmoid (logistic) function

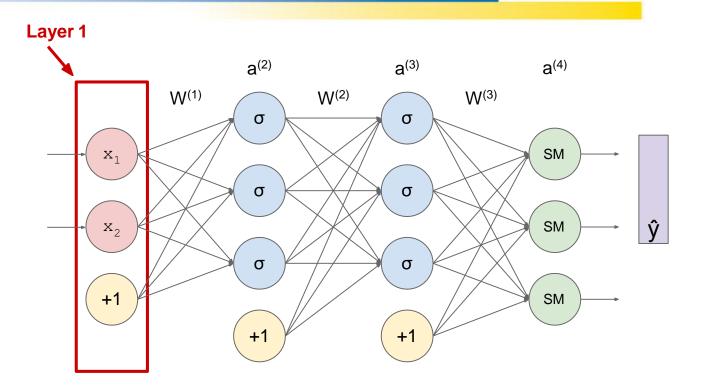
SM: Softmax function

+1: bias (constant) unit

a<sup>(I)</sup>: activation vector for layer I

W<sup>(l)</sup>: weight matrix for layer l





output vector SM: S

 $\sigma$ : sigmoid (logistic) function

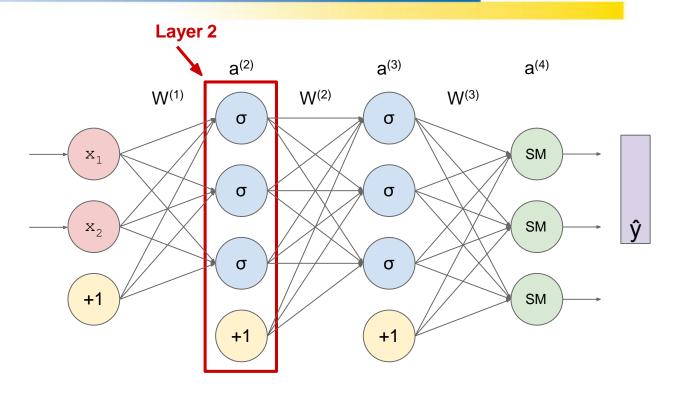
SM: Softmax function

+1: bias (constant) unit

a<sup>(I)</sup>: activation vector for layer I

W<sup>(l)</sup>: weight matrix for layer I





ŷ: output vector

 $\sigma$ : sigmoid (logistic) function

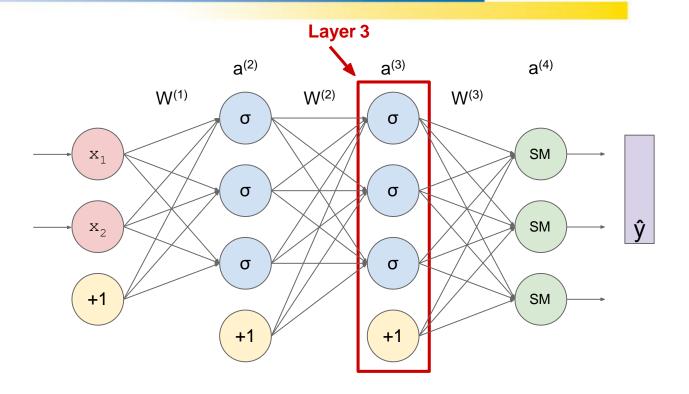
SM: Softmax function

+1: bias (constant) unit

a(I): activation vector for layer I

W<sup>(l)</sup>: weight matrix for layer I





: output vector

σ: sigmoid (logistic) function

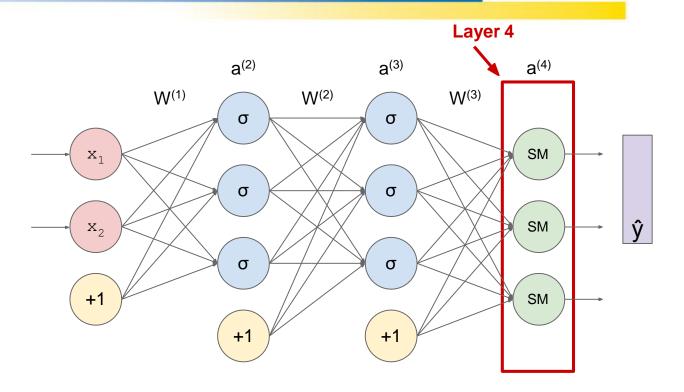
SM: Softmax function

+1: bias (constant) unit

a<sup>(I)</sup>: activation vector for layer I

W<sup>(l)</sup>: weight matrix for layer I





: output vector

 $\sigma$ : sigmoid (logistic) function

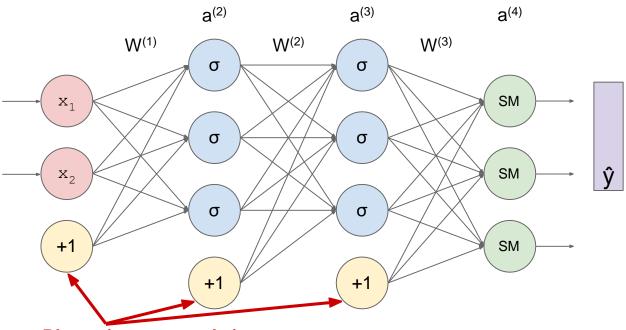
SM: Softmax function

+1: bias (constant) unit

a<sup>(l)</sup>: activation vector for layer l

W<sup>(I)</sup>: weight matrix for layer I





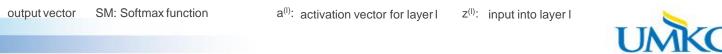
**Biases** (constant units)

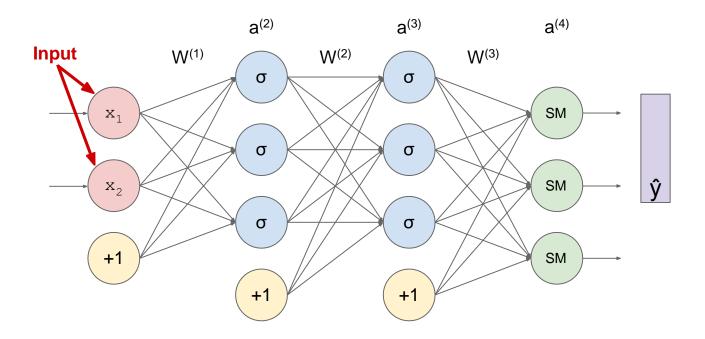
x<sub>i</sub>: input value

 $\sigma$ : sigmoid (logistic) function

+1: bias (constant) unit

W<sup>(l)</sup>: weight matrix for layer I





output vector

 $\sigma$ : sigmoid (logistic) function

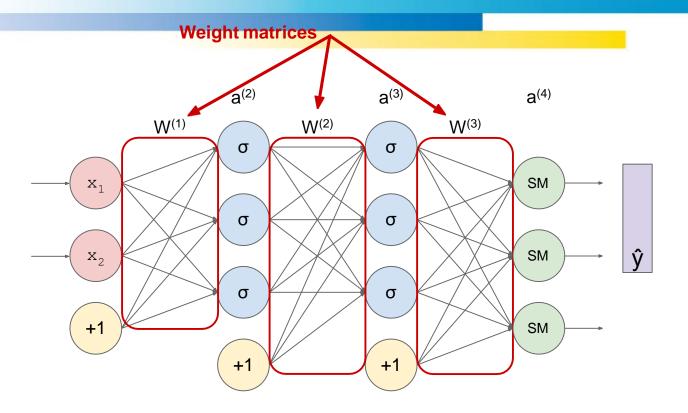
SM: Softmax function

+1: bias (constant) unit

a<sup>(I)</sup>: activation vector for layer I

W<sup>(l)</sup>: weight matrix for layer l





output vector

value σ: sigmoid (logistic) function

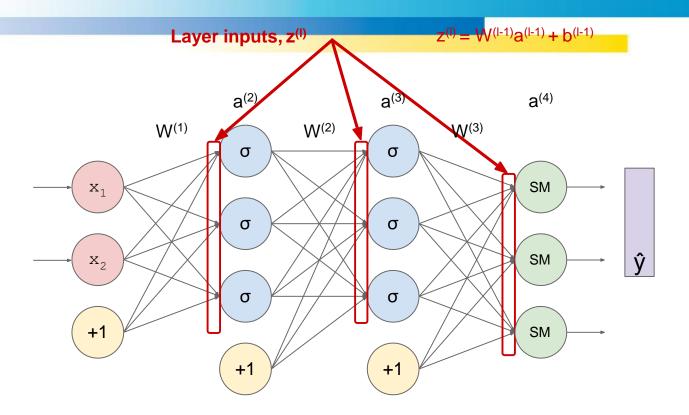
SM: Softmax function

+1: bias (constant) unit

a<sup>(l)</sup>: activation vector for layer l

W<sup>(l)</sup>: weight matrix for layer I





ŷ: output vector

 $\sigma$ : sigmoid (logistic) function

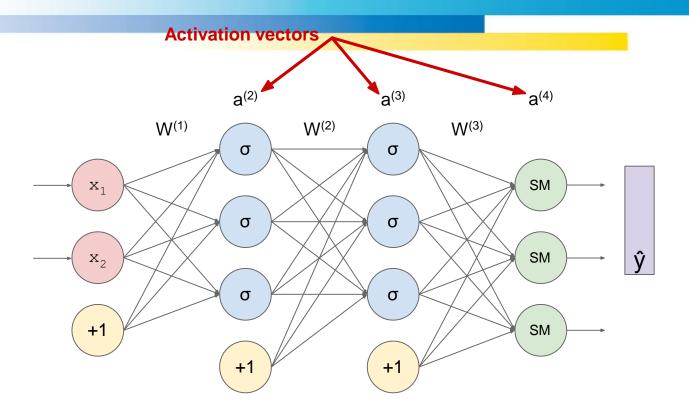
SM: Softmax function

+1: bias (constant) unit

a<sup>(I)</sup>: activation vector for layer I

W<sup>(l)</sup>: weight matrix for layer I





ŷ: output vector

σ: sigmoid (logistic) function

SM: Softmax function

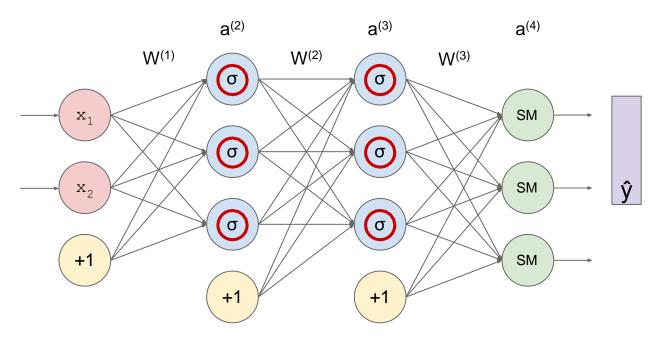
+1: bias (constant) unit

a<sup>(l)</sup>: activation vector for layer l

W<sup>(l)</sup>: weight matrix for layer I



#### Sigmoid activation function



x<sub>i</sub>: input value

ŷ: output vector

 $\sigma{:}\quad \text{sigmoid (logistic) function}$ 

SM: Softmax function

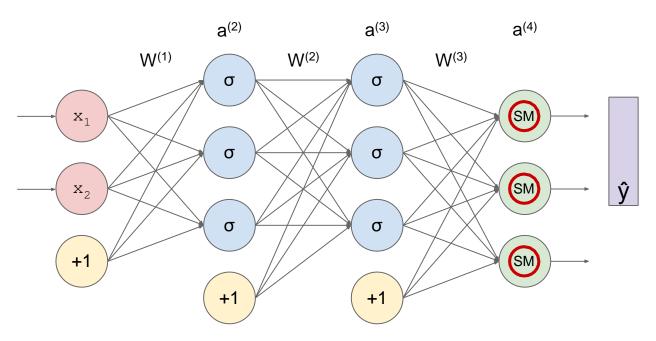
+1: bias (constant) unit

a<sup>(l)</sup>: activation vector for layer l

W<sup>(l)</sup>: weight matrix for layer I



#### **Softmax activation function**



x<sub>i</sub>: input value

: output vector

 $\sigma$ : sigmoid (logistic) function

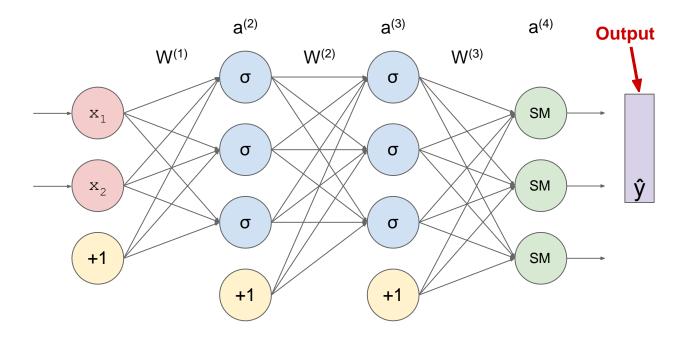
SM: Softmax function

+1: bias (constant) unit

a<sup>(l)</sup>: activation vector for layer l

W<sup>(l)</sup>: weight matrix for layer I





input value

σ: sigmoid (logistic) function

W<sup>(l)</sup>: weight matrix for layer l

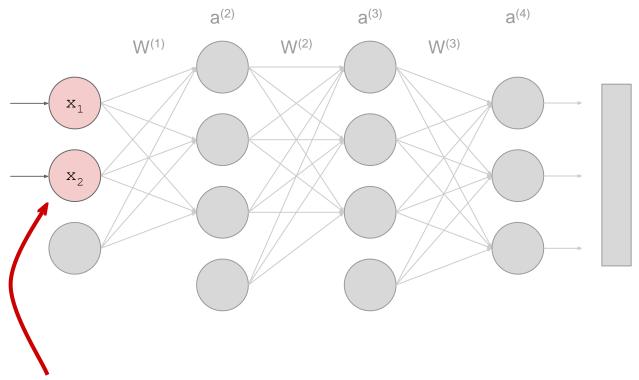
output vector

SM: Softmax function

a<sup>(l)</sup>: activation vector for layer l

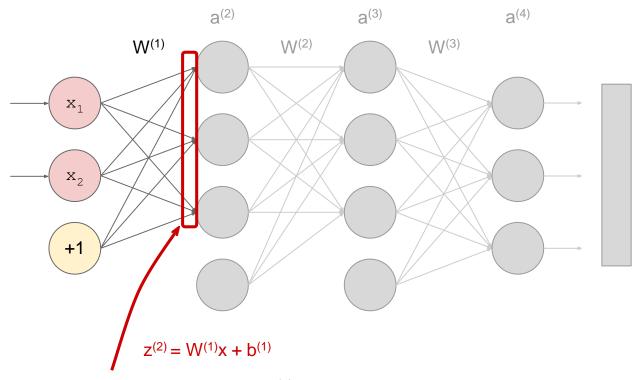
+1: bias (constant) unit





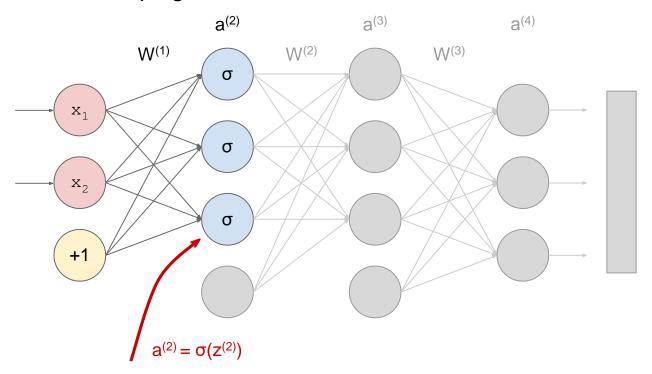
Input vector is passed into the network





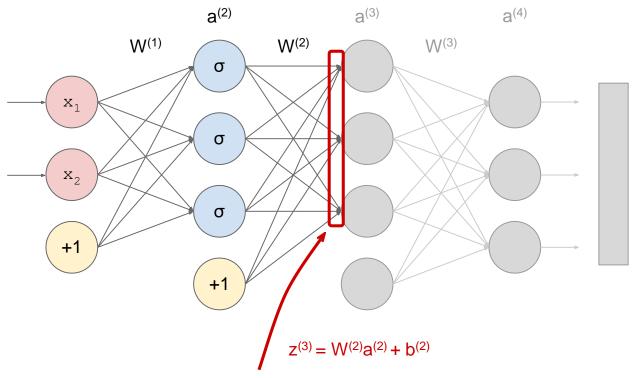
Input is multiplied with  $W^{(1)}$  weight matrix and added with layer 1biases to calculate  $z^{(2)}$ 





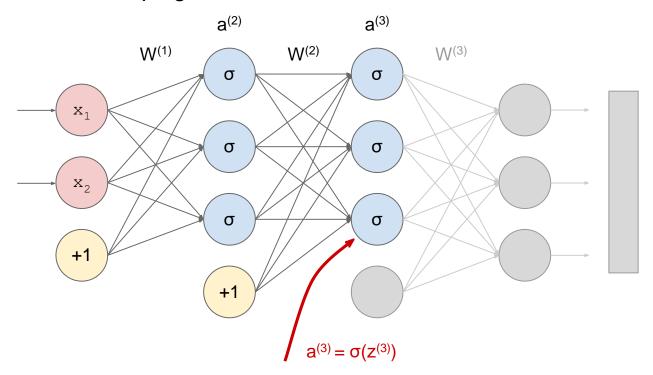
Activation value for the second layer is calculated by passing  $z^{(2)}$  into some function. In this case, the sigmoid function.





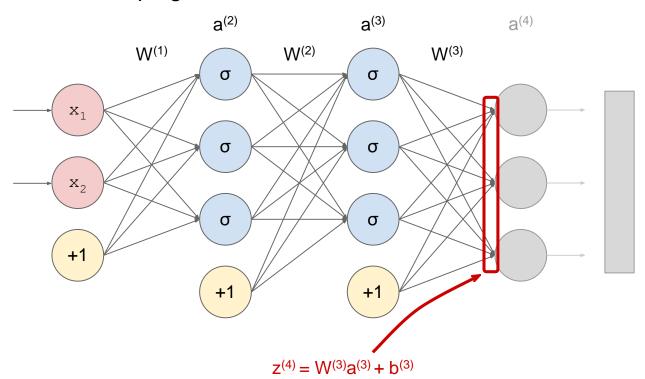
 $z^{(3)}$  is calculated by multiplying  $a^{(2)}$  vector with  $W^{(2)}$  weight matrix and adding layer 2 biases



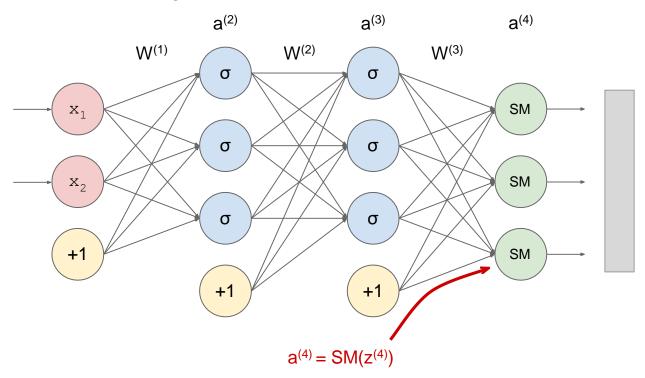


Similar to previous layer,  $a^{(3)}$  is calculated by passing  $z^{(3)}$  into the sigmoid function



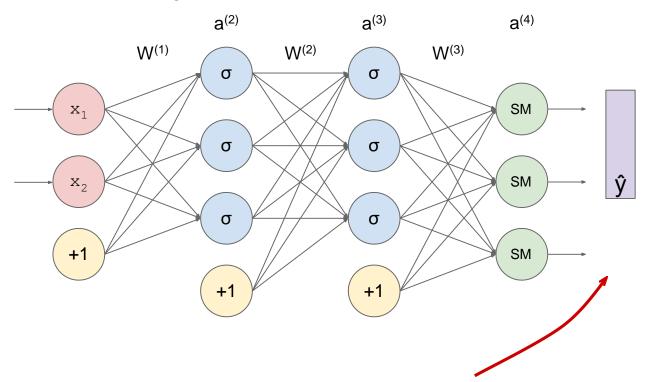


 $z^{(4)}$  is calculated by multiplying  $a^{(3)}$  vector with  $W^{(3)}$  weight matrix and adding layer 3 biases



For the final layer, we calculate  $a^{(4)}$  by passing  $z^{(4)}$  into the Softmax function

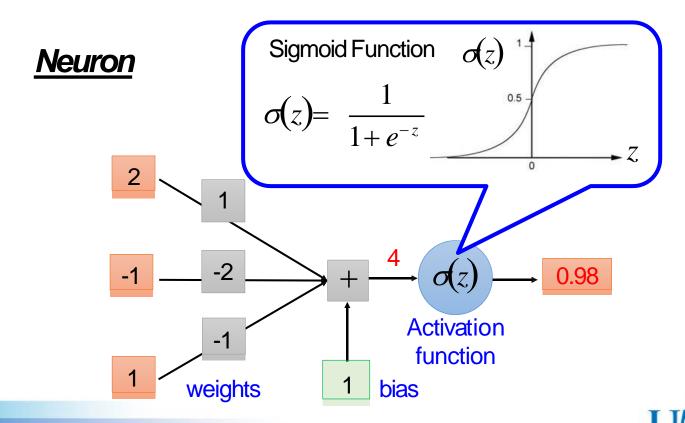




We then make our prediction based on the final layer's output



### Neural Network



$$z^{(2)} = W^{(1)}x + b^{(1)}$$

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

$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)}$$

$$a^{(3)} = \sigma(z^{(3)})$$

$$z^{(4)} = W^{(3)}a^{(3)} + b^{(3)}$$

$$a^{(4)} = \hat{y} = SM(z^{(4)})$$



#### Goal:

Find which direction to shift weights

#### How:

Find partial derivatives of the cost with respect to weight matrices

#### How (again):

Chain rule



### Cost/ Loss Functions

- We can use a cost function to measure how far off we are from the expected value.
- We'll use the following variables:
  - y to represent the true value
  - ŷ to represent the prediction value



#### **Back Propagation**

Want:

 $\partial cost$ 

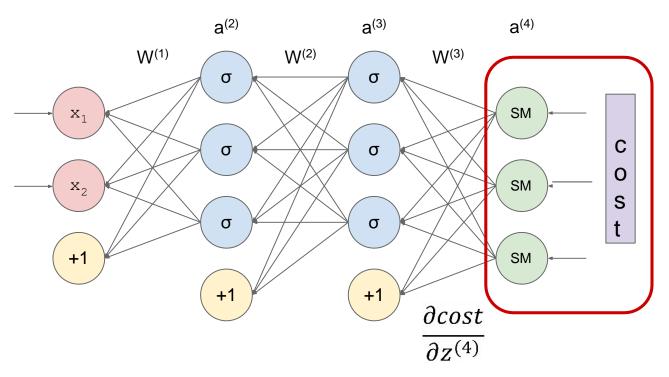
 $\partial cost$ 

 $\partial W^{(1)}$ 

∂cost

 $\partial W^{(2)}$ 

 $\partial W^{(3)}$ 



input value

σ: sigmoid (logistic) function

W<sup>(l)</sup>: weight matrix for layer I

output vector

SM: Softmax function

a(I): activation vector for layer I

+1: bias (constant) unit



$$\frac{\partial z^{(4)}}{\partial W^{(3)}} = a^{(3)}$$

$$\frac{\partial cost}{\partial W^{(3)}} = \frac{\partial cost}{\partial z^{(4)}} \cdot \frac{\partial z^{(4)}}{\partial W^{(3)}} = \delta^{(4)} a^{(3)}$$



#### **Back Propagation**

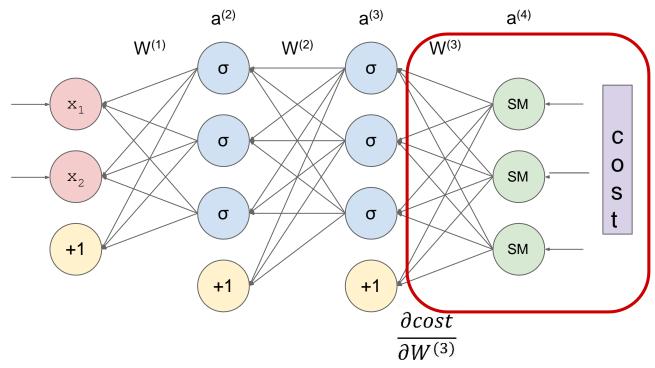
∂cos. Want:

 $\partial cost$ 

 $\partial cost$ 

 $\partial W^{(2)}$ 

 $\partial W^{(3)}$ 



input value

σ: sigmoid (logistic) function

+1: bias (constant) unit a(I): activation vector for layer I W<sup>(l)</sup>: weight matrix for layer I

z<sup>(l)</sup>: input into layer l



SM: Softmax function



$$\frac{\partial z^{(4)}}{\partial a^{(3)}} = W^{(3)}$$

$$\frac{\partial a^{(3)}}{\partial z^{(3)}} = a^{(3)}(1 - a^{(3)})$$

$$\frac{\partial cost}{\partial z^{(3)}} = \frac{\partial cost}{\partial z^{(4)}} \cdot \frac{\partial z^{(4)}}{\partial a^{(3)}} \cdot \frac{\partial a^{(3)}}{\partial z^{(3)}} = \delta^{(3)}$$



#### **Back Propagation**

Want:  $\frac{\partial \cos c}{\partial x}$ 

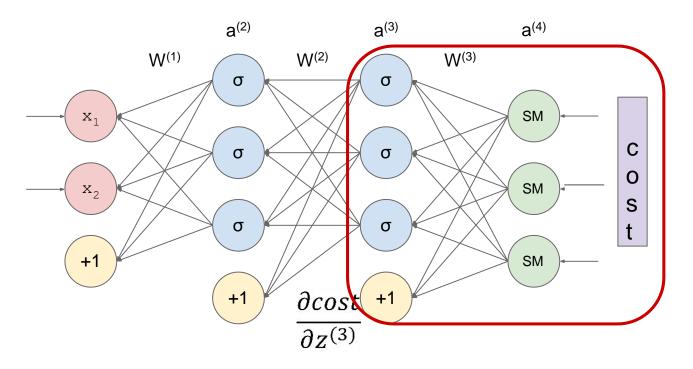
∂cost

∂cost

(1)

 $\partial W^{(2)}$ 

 $\partial W^{(3)}$ 



x<sub>i</sub>: input value

σ: sigmoid (logistic) function

+1: bias (constant) unit

W<sup>(I)</sup>: weight matrix for layer I

ŷ: output vector

SM: Softmax function

a<sup>(I)</sup>: activation vector for layer I



$$\frac{\partial cost}{\partial z^{(3)}} = \frac{\partial cost}{\partial z^{(4)}} \cdot \frac{\partial z^{(4)}}{\partial a^{(3)}} \cdot \frac{\partial a^{(3)}}{\partial z^{(3)}} = \delta^{(3)}$$

$$\frac{\partial cost}{\partial W^{(2)}} = \delta^{(3)} a^{(2)}$$

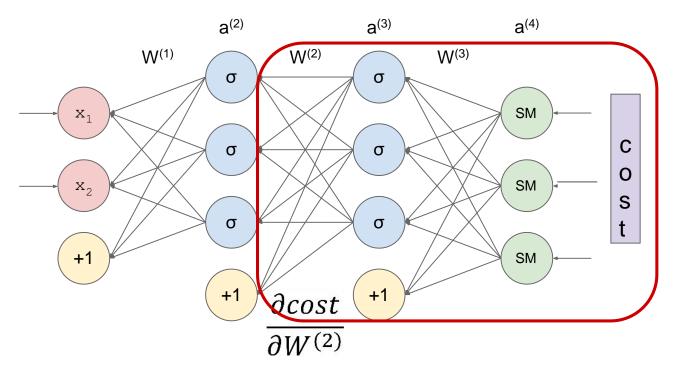


#### **Back Propagation**

Want:

dcost

 $\partial cost$  $\partial W^{(3)}$ 



input value

σ: sigmoid (logistic) function

+1: bias (constant) unit

W<sup>(l)</sup>: weight matrix for layer I

output vector

SM: Softmax function

a<sup>(l)</sup>: activation vector for layer l

z<sup>(l)</sup>: input into layer l



#### **Back Propagation**

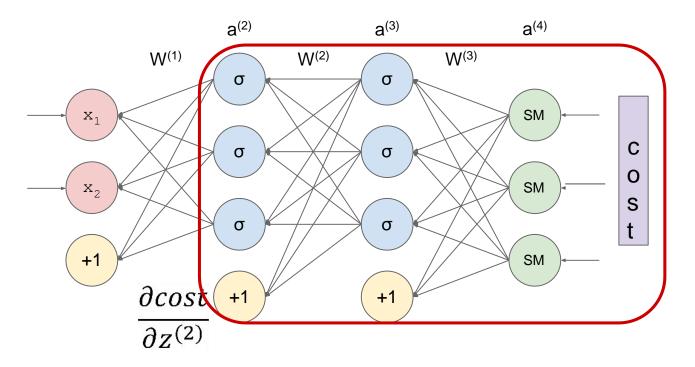
Want:  $\frac{\partial \cos x}{\partial x}$ 

dcos.

∂cost

 $\partial W^{(3)}$ 

 $\partial \overline{W}^{(2)}$   $\overline{\partial}$ 



x<sub>i</sub>: input value

ŷ: output vector

σ: sigmoid (logistic) function

SM: Softmax function

+1: bias (constant) unit

a<sup>(l)</sup>: activation vector for layer l

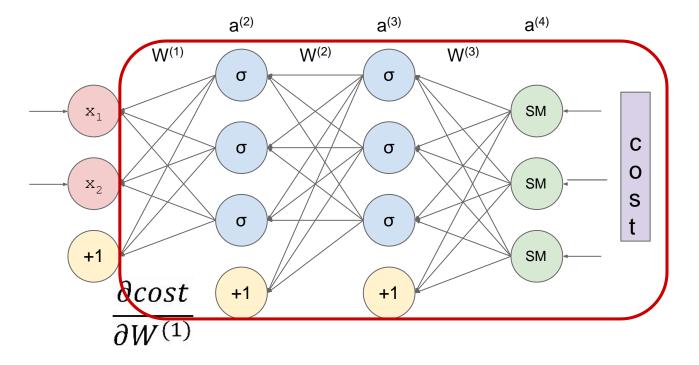
W<sup>(l)</sup>: weight matrix for layer I

z(I): input into layer I



#### **Back Propagation**

Want:



input value

output vector

σ: sigmoid (logistic) function

SM: Softmax function

+1: bias (constant) unit

a<sup>(l)</sup>: activation vector for layer l

W<sup>(l)</sup>: weight matrix for layer I

z<sup>(l)</sup>: input into layer l



$$\delta^{(l)} = \frac{\partial cost}{\partial z^{(l)}}$$

$$\frac{\partial cost}{\partial W^{(l)}} = \delta^{(l+1)} a^{(l)}$$

$$\delta^{(l)} = \delta^{(l+1)} W^{(l)} a'^{(l)}$$



## As programmers...

• How do we **NOT** do this ourselves?

We're lazy by trade.



## **Automatic Differentiation**



#### Auto-Differentiation: Idea

- Use functions that have easy-to-compute derivatives
- Compose these functions to create more complex super-model
- Use the chain rule to get partial derivatives of the model



# What is Backpropagation

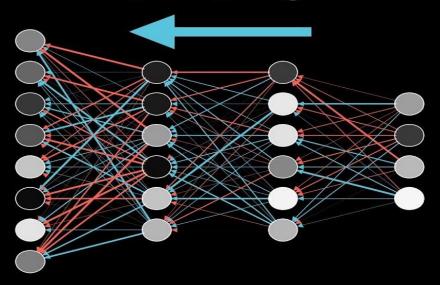
#### 1. Definition

 Backpropagation is an algorithm for supervised learning of artificial neural networks using gradient descent.



## Backpropagation

# Backpropagation





# Backpropagation: Algorithm

The following steps is the recursive definition of algorithm:

- 1. Initialize weights and biases in neural network
- 2. Propagate input forward (by applying activation function)
- 3. Calculate the output for every neuron
- 4. Calculate the error at the outputs
- 5. Backpropagate the error

$$\frac{\partial J}{\partial W} = \frac{1}{n} \sum_{i}^{n} x^{(i)} \left( W x^{(i)} + b - y^{(i)} \right)$$

6. use the error signals to compute weight adjustments.

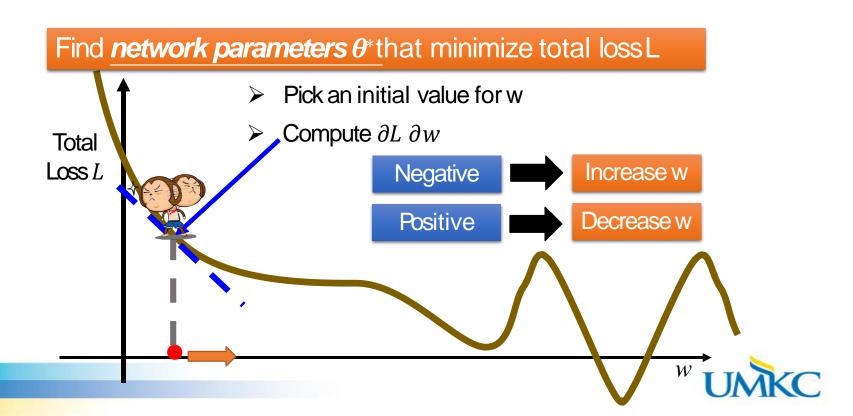


Network parameters 
$$\theta = \{ w_1, w_2, \dots, b_1, b_2, \} \cdots$$

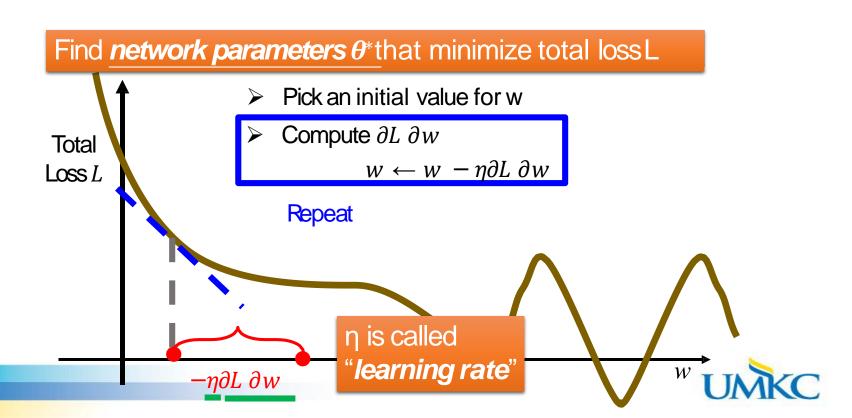
#### Find *network parameters* $\theta^*$ that minimize total loss L



Network parameters 
$$\theta = \{ w_1, w_2, \dots, b_1, b_2, \} \dots$$

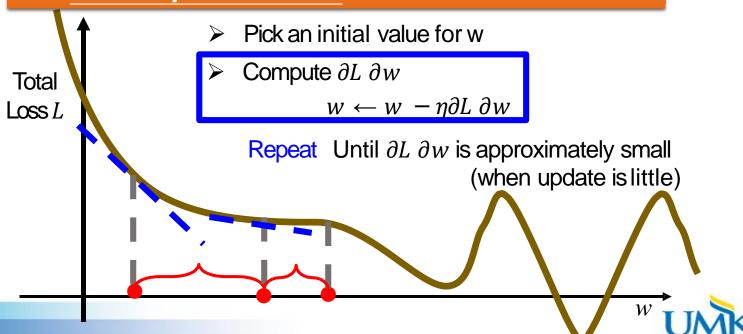


Network parameters 
$$\theta = \{ w_1, w_2, \dots, b_1, b_2, \} \dots$$



Network parameters 
$$\theta = \{ w_1, w_2, \dots, b_1, b_2, \} \dots$$

#### Find *network parameters* $\theta^*$ that minimize total loss L



# Learning Rate

- Learning rate definition:
  - learning rate determines **how fast** weights change.
- Learning rate = Trade-Off
  - Large values for ratio => Fasttraining
  - Lower ratios => Accurate training
- Question: How do *you* choose the learning rate?



## Cost/ Loss Functions

- We can use a cost function to measure how far off we are from the expected value.
- We'll use the following variables:
  - y to represent the true value
  - ŷ to represent the prediction value



# Common types of loss functions (1)

- Loss functions depend on the type of task:
  - Regression: the network predicts continuous, numeric variables
    - Example: Length of fishes in images, temperature from latitude/longitude or housing prices
    - Absolute value, square error

$$\mathcal{L} = \frac{1}{n} \sum_{i=1}^{n} (y_i - f_{\theta}(x_i))^2$$

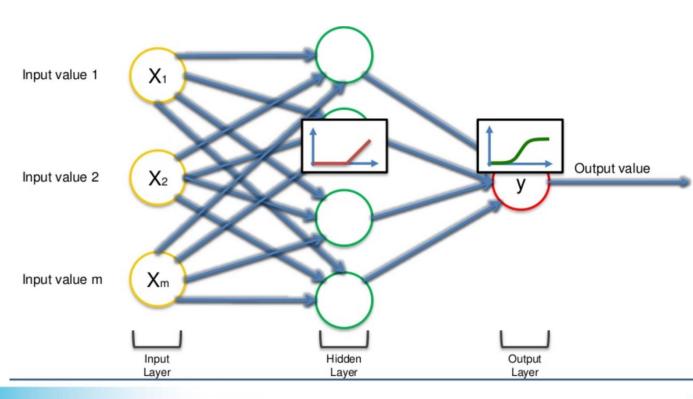


# Common types of loss functions (2)

- Loss functions depend on the type of task:
  - Classification: the network predicts categorical variables (fixed number of classes)
    - Example: classify email as spam, predict student grades from essays.
    - hinge loss, Cross-entropy loss



## Activation function





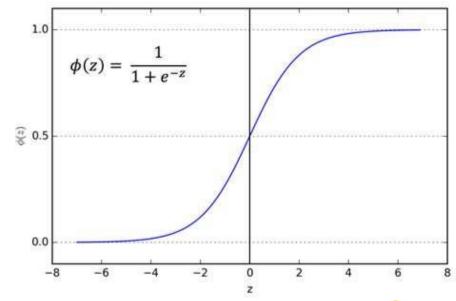
### **Activation Function**

- They define the output of that node given an input or set of inputs
- They decide whether a neuron should be activated or not
- The activation function can be divided in two types:
  - Linear activation function
  - Non-Linear Activation Function
    - Sigmoid
    - Tanh
    - Relu



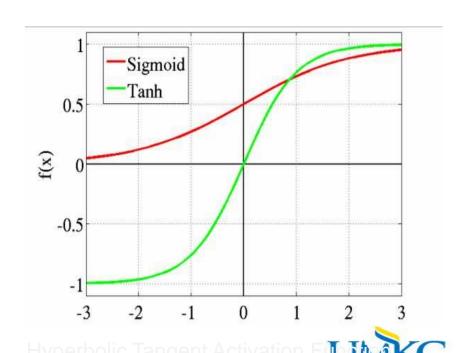
## Non-Linear Activation Function - Sigmoid

- It exists between (0 to 1).
- Used for models where we have to predict the probability as an output, sigmoid is the right choice.
- The function is differentiable, we can find the slope of the sigmoid curve at any two points.



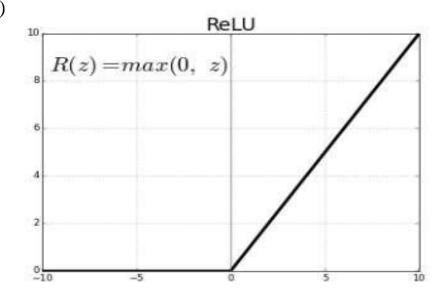
## Non-Linear Activation Function - Tanh

- The range of the tanh function is from (-1 to 1). Tanh is also sigmoidal
- Negative inputs will be mapped strongly negative and the zero inputs will be mapped near zero in the tanh graph.
- It is differentiable
- The tanh function is mainly used classification between two classes



### Non-Linear Activation Function - ReLU (Rectified Linear Unit)

- The ReLU is half rectified (from bottom). f(z) is zero when z is less than zero and f(z) is equal to z when z is above or equal to zero.
- **Range**: [ 0 to infinity)
- **Issue**: Negative values become zero immediately which decreases the ability of the model to fit or train from the data properly



# Deep Learning Glossary

- **Input**: the first layer
- Output: what we want to compute
- Weight update: updating weight to decrease the cost function
- Bias: added to increase the flexibility of the model to fit the data
- Hidden layers: the neurons for the intermediate step
- Forward propagation: forwarding input through the neural network in order to generate network output value(s)
- Cost function: the difference between the targeted and actual output
- Activation function: They introduce non-linear properties to our Network
- Backward propagation: backward pass



# Deep Learning Glossary

- Learning rate: a hyper-parameter that controls how much we are adjusting the weights of our **network** with respect the loss gradient
- Epochs: one forward pass and one backward pass of *all* the training examples
- Batch size: the number of training examples in one forward/backward pass
- Dropout: next class
- Regularization: next class



# To recap: Keras Programming steps

- Importing Sequential class from keras.models
- Stacking layers using .add() method
- Configure learning process using .compile() method
- Train the model on train dataset using .fit() method



# General layers

• Dense: implements the operation

```
output = activation(dot(input, weight) + bias)
```

- keras.layers.core.Dense(units, activation=None, use\_bias=True, kernel\_initializer='glorot\_uniform', bias\_initializer='zeros', kernel\_regularizer=None, bias\_regularizer=None, activit y\_regularizer=None, kernel\_constraint=None, bias\_constraint=None)
- Activations:
  - you can add one more new layer: model.add(Activation('tanh'))
  - or Activation argument in all forwarded layer: model.add(Dense(64, activation='tanh'))



# Optimizers available in Keras

- How do we find the "best set of parameters (weights and biases)" for the given network?
- Optimization
  - They vary in the speed of convergence,
  - ability to avoid getting stuck in local minima
  - SGD –Stochastic gradient descent
  - SGD with momentum
  - Adam
  - AdaGrad
  - RMSprop
  - AdaDelta

```
sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True) model.compile(loss='mean_squared_error', optimizer=sgd)
```



## Loss functions available in Keras

- MSE –Mean square error: keras.losses.mean\_squared\_error(y\_true, y\_pred)
- MAE –Mean absolute error: keras.losses.mean\_absolute\_error(y\_true, y\_pred)

- Hinge: keras.losses.hinge(y\_true, y\_pred)
- Categorical\_crossEntropy: keras.losses.categorical\_crossentropy(y\_true, y\_pred)



### Initializations

- Initialization define the way to set the initial random weights of keras layers
- Init → model.add(Dense(64,init='uniform'))
- Uniform
- Lecun\_uniform
- Normal
- Zero
- One
- Glorot\_normal
- Glorot\_uniform



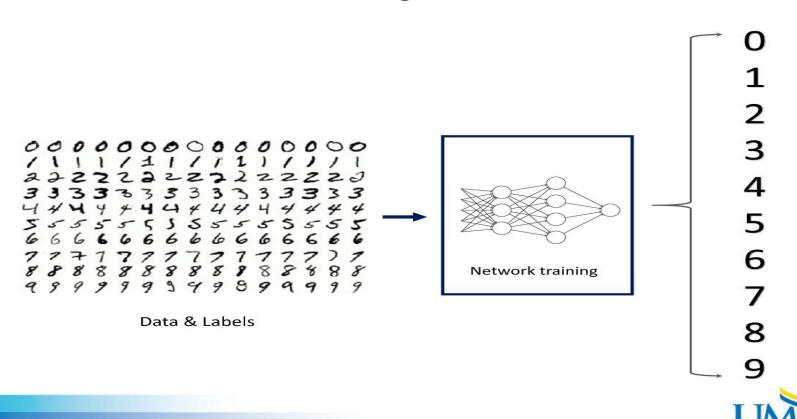
### **Evaluation** in keras

- Usage of metrics:
  - A metrics is a function that is used to judge the performance of your model

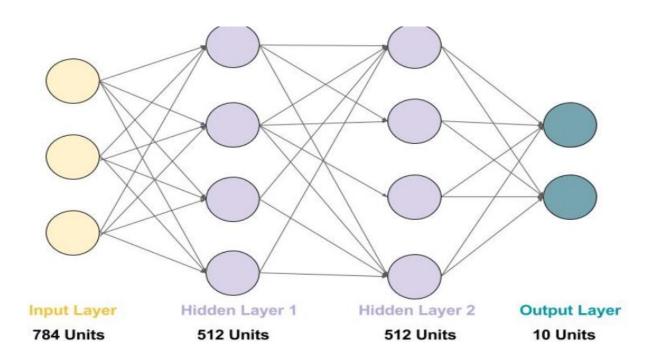
```
model.compile(loss='mean_squared_error', optimizer='sgd'
, metrics=['mae', 'acc'])
```



# Use case: Image classification



# Model for image classification





# Loading data and plotting the digit

```
(train_images,train_labels),(test_images, test_labels) = mnist.load_data()
#display the first image in the training data
plt.imshow(train_images[0,:,:],cmap='gray')
plt.title('Ground Truth : {}'.format(train_labels[0]))
# plt.show()
                                      Ground Truth: 5
```

25 -



# Processing the data

```
#1. convert each image of shape 28*28 to 784 dimensional which will be fed to the network
as a single feature
dimData = np.prod(train_images.shape[1:])
train_data = train_images.reshape(train_images.shape[0],dimData)
test_data = test_images.reshape(test_images.shape[0],dimData)
#convert data to float and scale values between 0 and 1
train data.astype('float')
test_data.astype('float')
#scale data
train data //=255
test data //=255
#change the labels frominteger to one-hot encoding
train_labels_one_hot = to_categorical(train_labels)
test labels one hot = to categorical(test labels)
```



# Creating the network & fitting



## **Evaluation**

```
[test_loss, test_acc] = model.evaluate(test_data, test_labels_one_hot)
print("Evaluation result on Test Data : Loss = {}, accuracy = {}".format(test_loss, test_acc))
```



### Inference on the model

- model.predict\_classes(test\_data[0].reshape(1,784))
- You can plot a test image with:
- plt.imshow(test\_data[0].reshape(28,28))



### Callbacks

- A callback is a set of functions to be applied at given stages of the training procedure.
- You can use callbacks to get a view on internal states and statistics of the model during training
- One of the default callbacks that is registered when training all deep learning models is the <u>History callback</u>
- It records training metrics for each epoch
- This includes the loss and the accuracy
- Metrics are stored in a dictionary in the history member of the object returned.
- # list all data in history
- print(history.history.keys()) → ['acc', 'loss', 'val\_acc', 'val\_loss']



#### References

- <a href="https://www.youtube.com/watch?v=aircAruvnKk&t=866s">https://www.youtube.com/watch?v=aircAruvnKk&t=866s</a>
- <a href="https://medium.com/@14prakash/back-propagation-is-very-simple-who-made-it-complicated-97b794c97e5c">https://medium.com/@14prakash/back-propagation-is-very-simple-who-made-it-complicated-97b794c97e5c</a>
- <a href="https://www.youtube.com/watch?v=aircAruvnKk">https://www.youtube.com/watch?v=aircAruvnKk</a>





