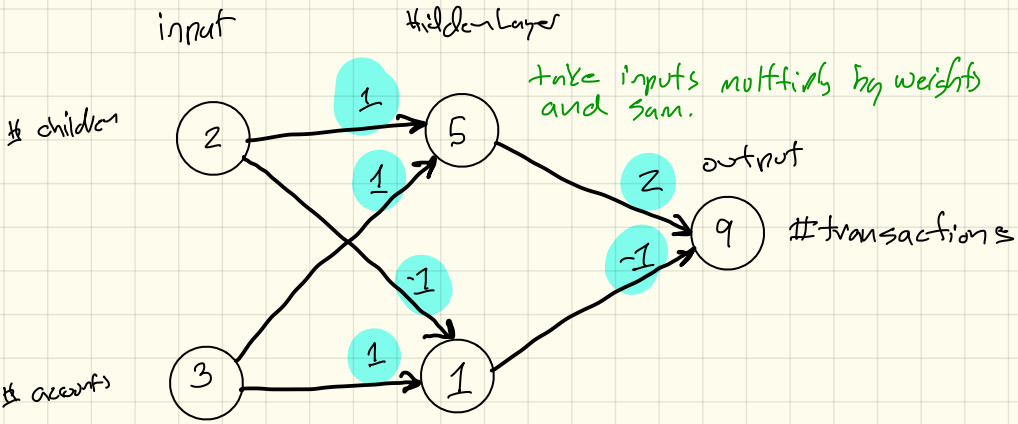
DataCamp Deep Learning
in python

Forward Propagation



take inputs multiply by weights and sum.

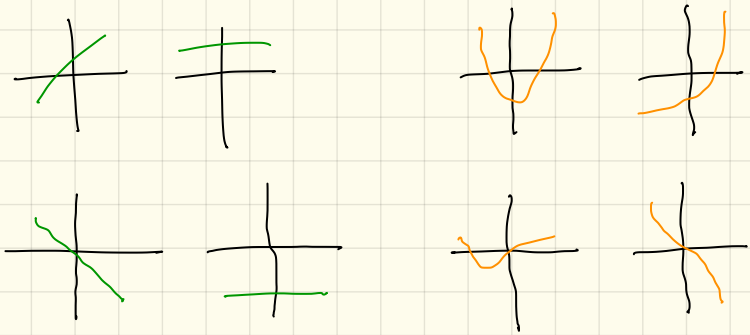
First Set of weights

multiply & add - this is the dot product.

Activation Functions

- adds non linear component

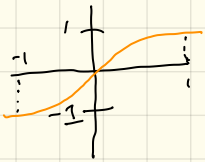
<u>linear functions</u> | <u>non linear functions</u>

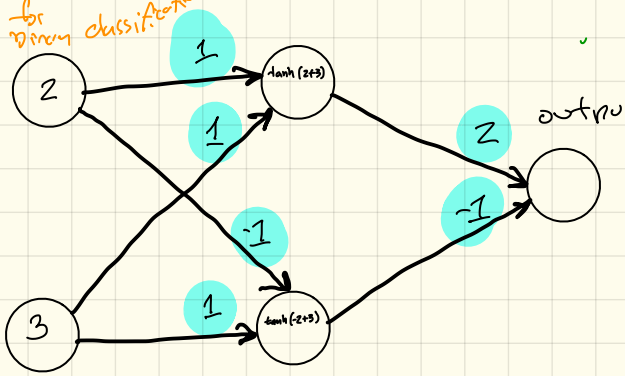


Activation functions - applied to node inputs to produce node outputs.

tanh was a popular activation function for a long time

hyperbolic tan which is analog of the tan function ← used for binary classification





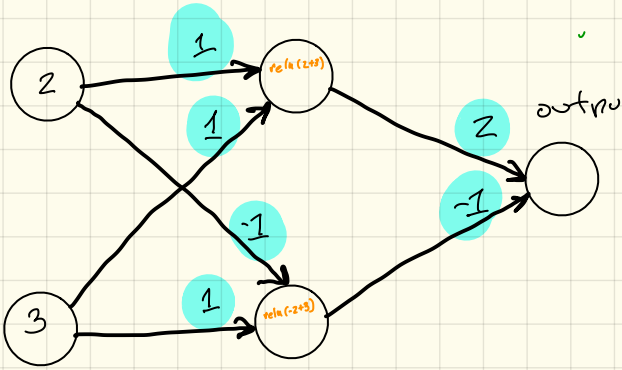
Similar to sigmoid

$$\text{sigmoid} = \frac{1}{1+e^{-x}}$$

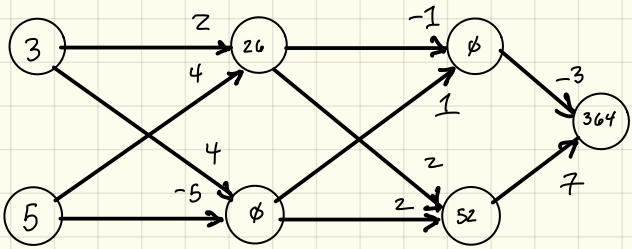
ReLu - Rectified Linear Activation

now Relu is used instead of tanh.

$R(x) = \max(0, x)$



Multiple Hidden Layers



use ReLU activation function

Calculate Nodes

$3 \cdot 2 + 4 \cdot 5 = 26 \rightarrow \text{Relu}(26) = 26$

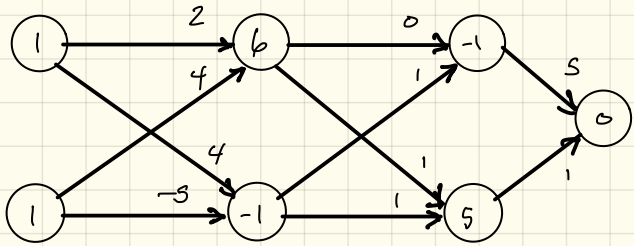
$3 \cdot 4 + 5(-5) = 12 - 25 \Rightarrow \text{Relu}(-13) = \phi$

$26 \cdot (-1) + \phi \cdot 1 = -26 \Rightarrow \text{Relu}(-26) = \phi$

$26(2) + (\phi)(2) = 52 \Rightarrow \text{Relu}(52) = 52$

$\phi \cdot (-3) + 52(7) = 364 \Rightarrow \text{Relu}(364) = 364$

Practice



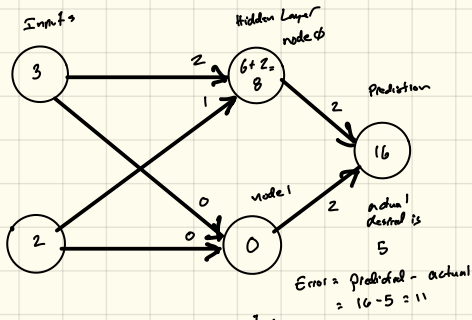
use identity activation function

$1 \cdot 2 + 1 \cdot 4 = 2 + 4 = 6$

$1 \cdot 4 - 1 \cdot 5 = -1$

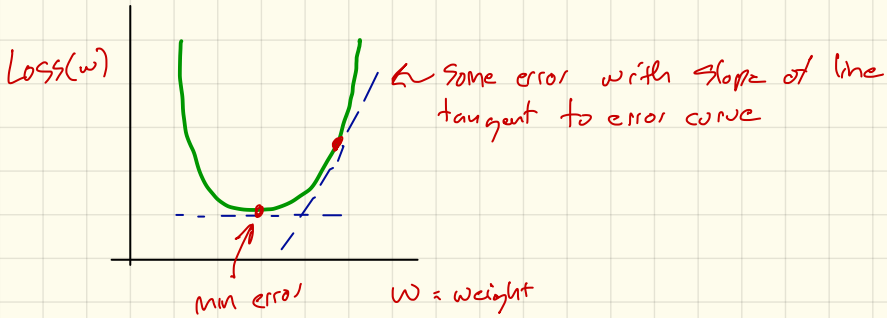
$5 + (-5) = 0$

Calculating Errors



change weight of output [node 0] to 2.01
then error increases

Consider gradient descent w/ one variable



positive slope : then decrease w to min error

Update each weight by subtracting (learning rate)* Slope

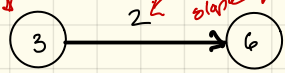
0.01

Slope Calculation Example

Value of node that feeds into weight

we feed into this node

not the slope



actual target value = 10

To calculate slope for a weight multiply:

1) Slope of loss function w.r.t. value of the node we feed into

from derivative of mean squared error?

Mean squared error →

2 * (predicted value − Actual value) = 2 * Error
= 2(-4)

2) The value of the node that feeds into our weight

~~3) Slope of the activation function w.r.t. value we feed into~~

no activation function in this example.

Slope of mean squared loss function =

$= 2 * (-4) * 3$

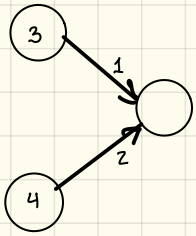
$= -24$

If learning rate was 0.01, the new weight would be

$2 - 0.01(-24) = 2.24$

new error is $3(2.24) = 6.72$ which is closer to 10

Network w/ 2 inputs and code to update weights




```
import numpy as np

weights = np.array([1,2])

input_data = np.array([3,4])

target = 6

learning_rate = 0.01

preds = (weights * input_data).sum()

error = preds - target

error -> 5
```

an array of slopes is called a gradient. This is where gradient descent comes from.

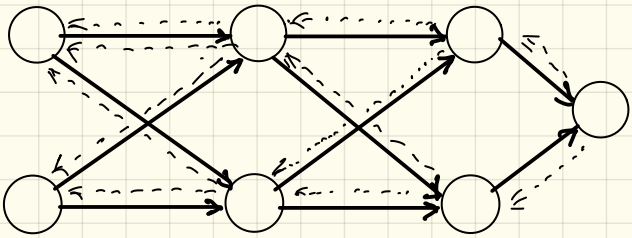
```
gradient = 2*input_data * error
```

$2*[3\ 4][5] = [30\ 40]$

```
gradient -> 30, 40

weights_updated = weights - learning_rate * gradient

preds_updated = (weights_updated * input_data).sum()

error_updated = preds_updated - target
```

error_updated -> -2.5

Back Propagation

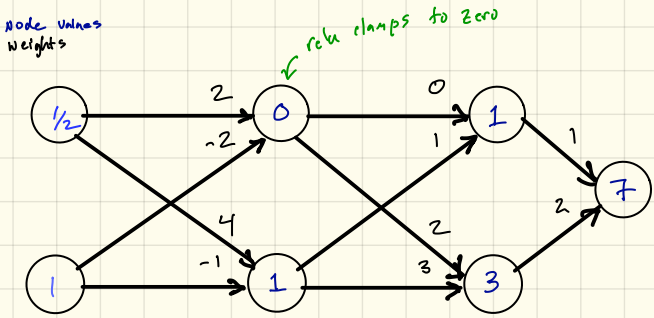


- allows gradient descent to update all weights in neural network by getting gradient for all weights
- Comes from chain rule of calculus
- Important to understand the process, but you will generally use a library that implements this

---

- Trying to estimate the slope of the loss function w.r.t. each weight
- Do forward propagation to calculate predictions and errors

Do this before back propagation



ReLU activation function

Actual Target value = 4

Go back one layer at a time

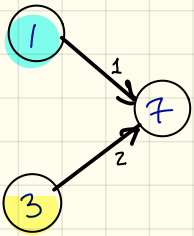
Gradient for weights is product of

1. node value feeding into that weight
2. slope of loss function w.r.t. node it feeds into — 2 × error
3. slope of activation function at the node it feeds into — 0 or 1 since relu

We reversed the order of 2 & 3

Need to also keep track of the slopes of the loss function w.r.t. node values.

Slope of node values are the sum of the slopes for all weights that come out of them.

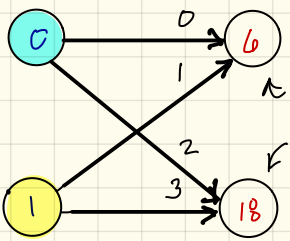


- Relu Activation function
- Actual Target Value = 4

Error = 3

Top weight's slope = 1 * 2*3 = 6
"gradient"

Bottom weights slope = 3 * 2*3 = 18



slope of calculated loss function

Current weight value		Gradient
0	not used to calculate gradient derivative	0
1		6
2		0
3		18

outputs node slope

relu slope

Top weight's slope = 0 * 6 * 1 = 0
&
= 1 * 6 * 1 = 6

Calculated slope of loss function

Bottom weights slope = 0 * 18 * 1 = 0
&
= 1 * 18 * 1 = 18

Its common to calculate slopes only on a subset of the data ('batch')

Use a different batch of data to calculate the next update.

Start over from beginning once all data is used.

Each time through the training data is called an epoch.

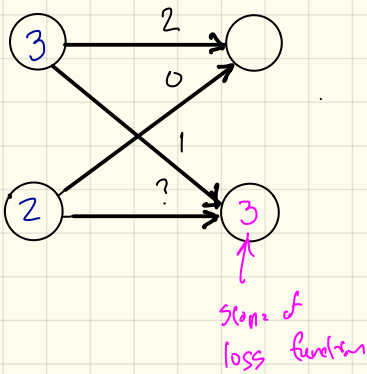
When slopes are calculated on one batch at a time its called:

Stochastic gradient descent.

(rather than gradient descent.)

Example

In figure, we have done forward propagation. node values calculated as part of forward propagation are shown in BLUE. Weights shown in black.



This network uses reLU activation function. So the slope of the activation function is 1 for any node receiving a positive value as input. Assume the node being examined had a positive value (so the activation function is 1.)

Layers after the question mark show the slopes calculated as part of the back propagation, rather than the forward-prop values. Those slope values are shown in PURPLE.

What is the slope needed to update weight "2"

$$2 \times 1 \times 3 = 6$$

Classification

- Categorical_loss_function
- Similar to log loss function. Lower is better
- Add metrics = ['accuracy'] to compile step for easy to understand diagnostics
- Output layer has seperate node for each possible outcome and uses 'softmax' activation

Cmd - R

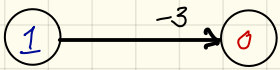
Spctl Kext- consent disable

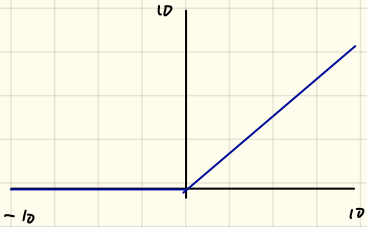
Updating model optimization

Stochastic Gradient Descent

```
def get_new_model(input_shape=input_shape):
    model = Sequential()
    model.add(Dense(100, activation='relu', input_shape=input_shape))
    model.add(Dense(100, activation='relu'))
    model.add(Dense(2, activation='softmax'))
    return(model)

lr_to_test = [0.0001, 0.01, 1]

for lr in lr_to_test:
    model = get_new_model()
    my_optimizer = SGD(lr=lr)
    model.compile(optimizer=my_optimizer, loss='categorical_crossentropy')
    model.fit(predictors, target)
```

The dying neuron problem





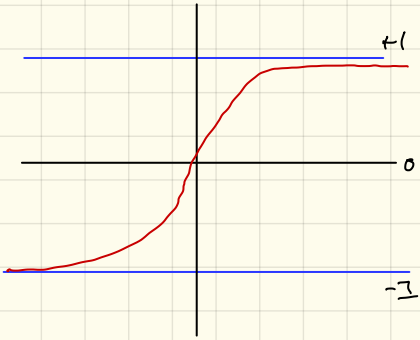
ReLU activation

$\max(0, x)$

Once a node starts getting negative inputs

- it may continue only getting negative inputs
- Contributes nothing to the model
- dead neuron

Vanishing Gradients



tanh function

- Occurs when many layers have very small slopes
  - due to being on flat part of tanh curve.
- In deep networks, updates to backprop were close to $\emptyset$

Validation in Deep Learning

- Commonly use validation split rather than cross validation
- Deep learning widely used on large datasets
- Single validation score is based on large amount of data and is reliable
- Repeated training from cross-validation would take long time

Model Validation in Keras

```
model.compile(optimizer='adam', loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(predictors, target, validation_split=0.3)
```

shows epochs
and
val_acc for each epoch

Early Stopping

```
from keras.callbacks import EarlyStopping

early_stopping_monitor = EarlyStopping(patience=2)
```

how many epochs the model can go without improvement before training stops

```
model.fit(predictors, target, validation_split=0.3,
          epochs=20, callbacks=[early_stopping_monitor])
```

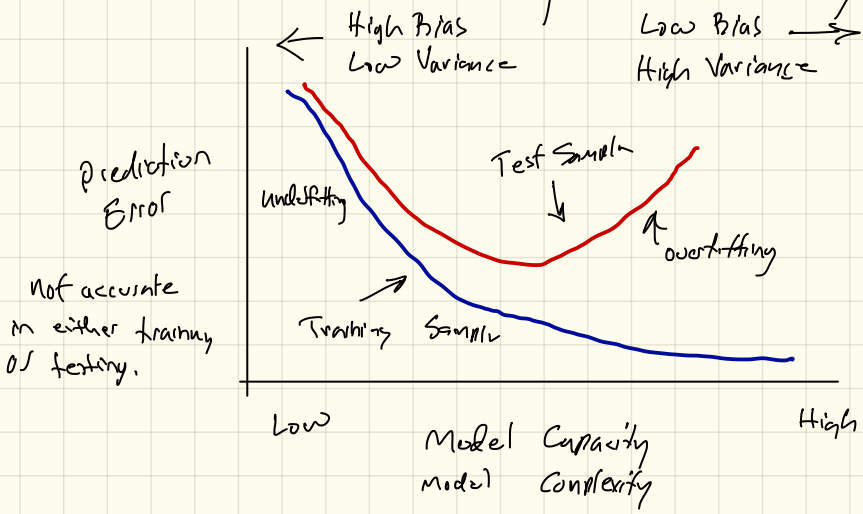
by default Keras trains for 10 epochs.

Thinking about model capacity

or

network capacity

Related to overfitting & underfitting



Workflow for optimizing model capacity

Start with small network

Get the validation score

Keep increasing capacity until validation score is no longer improving

Sequential Experiments

Hidden Layers	Nodes per layer	Mean Squared Error	Next Step
1	100	5.4	increase capacity
1	250	4.8	increase capacity
2	250	4.4	increase capacity
3	250	4.5	decrease capacity
3	200	4.3	Done

Recognizing handwritten Digits

MNIST dataset

28x28 grid flattened to 784 values for each image

Value in each part of array denotes darkness of that pixel

0 = DARK
255 = Light