Introduction to Deep Learning in Python

Sunday, 16 August 2020 12:59 PM

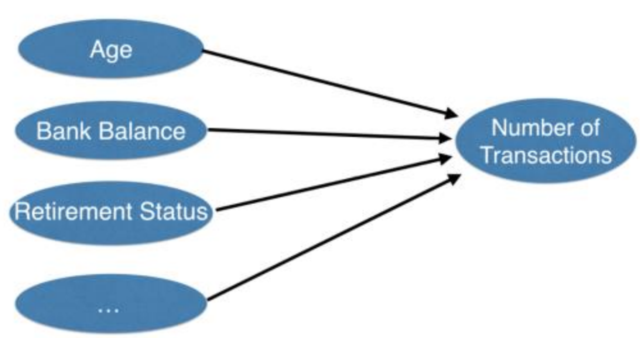
Basics of Deep learning and Neural Networks

Imagine you work for a bank

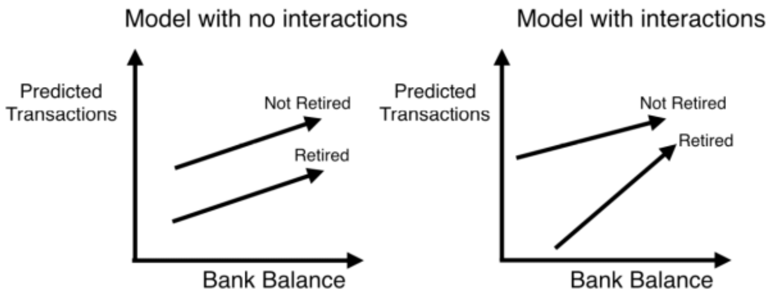
- You need to predict how many transactions each customer will make next year

Example as seen by linear regression

Example as seen by linear regression



Example as seen by linear regression



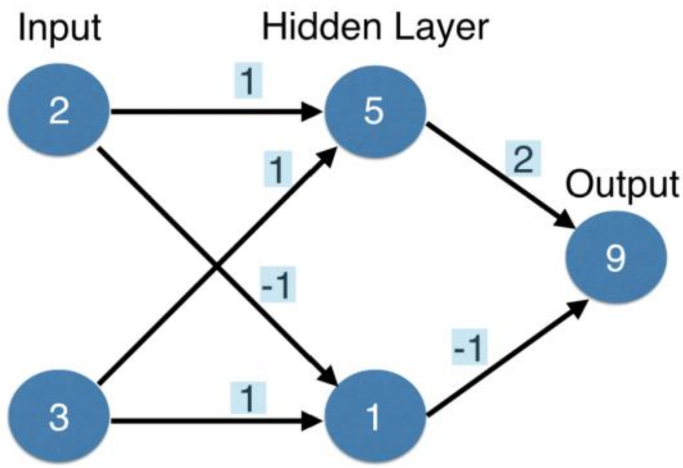
Interactions

- Neural networks account for interactions really well
- Deep learning uses especially powerful neural networks
  - Text
  - Images
  - Videos
  - Audio
  - Source code

Bank transactions example

- Make predictions based on:
  - Number of children
  - Number of existing accounts

Forward propagation



Forward propagation

- Multiply - add process
- Dot product
- Forward propagation for one data point at a time
- Output is the prediction for that data point

Forward propagation code

```
import numpy as np
input_data = np.array([2, 3])
weights = {'node_0': np.array([1, 1]),
           'node_1': np.array([-1, 1]),
           'output': np.array([2, -1])}
node_0_value = (input_data * weights['node_0']).sum()
node_1_value = (input_data * weights['node_1']).sum()
```

Forward propagation code

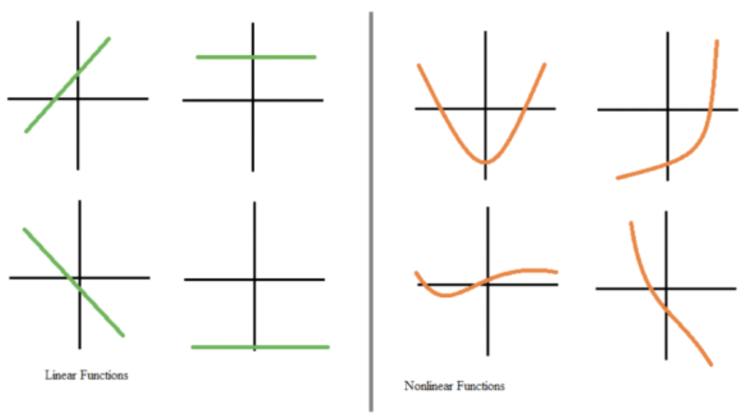
```
hidden_layer_values = np.array([node_0_value, node_1_value])
print(hidden_layer_values)
```

```
[5, 1]
```

```
output = (hidden_layer_values * weights['output']).sum()
print(output)
```

```
9
```

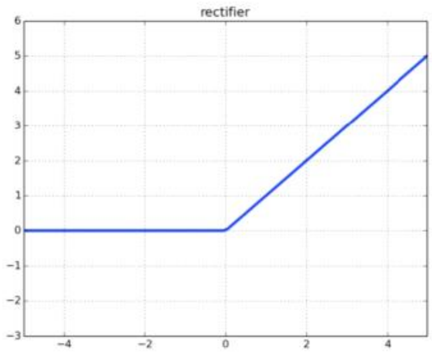
Linear vs Nonlinear Functions



Activation functions

- Applied to node inputs to produce node output

ReLU (Rectified Linear Activation)



$$RELU(x) = \begin{cases} 0 \; if \; x < 0 \\ x \; if \; x >= 0 \end{cases}$$

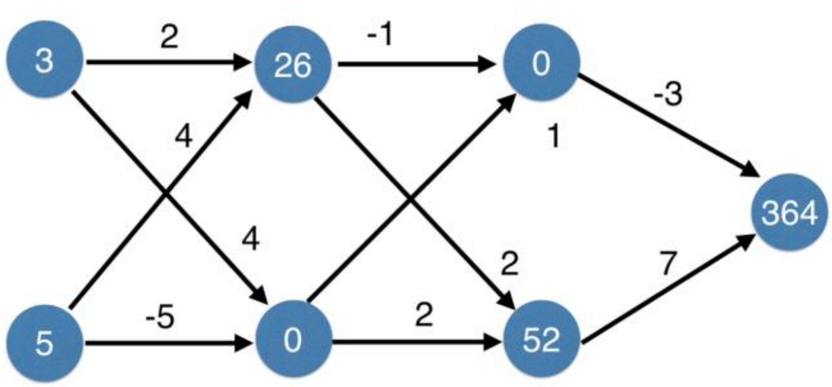
Activation functions

```
import numpy as np
input_data = np.array([-1, 2])
weights = {'node_0': np.array([3, 3]),
           'node_1': np.array([1, 5]),
           'output': np.array([2, -1])}
node_0_input = (input_data * weights['node_0']).sum()
node_0_output = np.tanh(node_0_input)
node_1_input = (input_data * weights['node_1']).sum()
node_1_output = np.tanh(node_1_input)
hidden_layer_outputs = np.array([node_0_output, node_1_output])
output = (hidden_layer_output * weights['output']).sum()
```

```
print(output)
```

```
1.2382242525694254
```

Multiple hidden layers

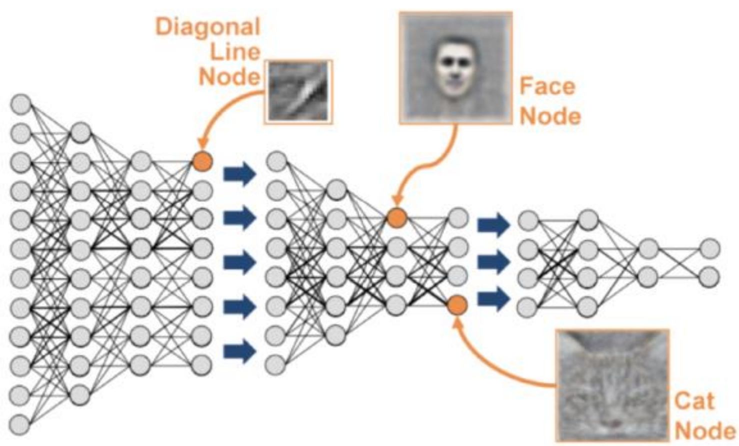


Calculate with ReLU Activation Function

Representation learning

- Deep networks internally build representations of patterns in the data
- Partially replace the need for feature engineering
- Subsequent layers build increasingly sophisticated representations of raw data

Representation learning

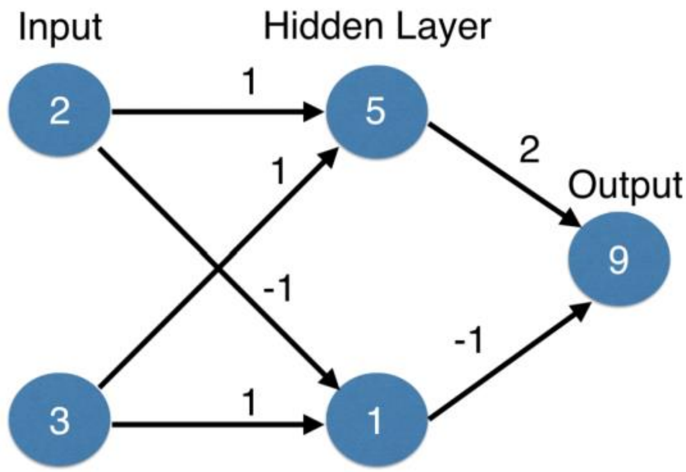


Deep learning

- Modeler doesn't need to specify the interactions
- When you train the model, the neural network gets weights that find the relevant patterns to make better predictions

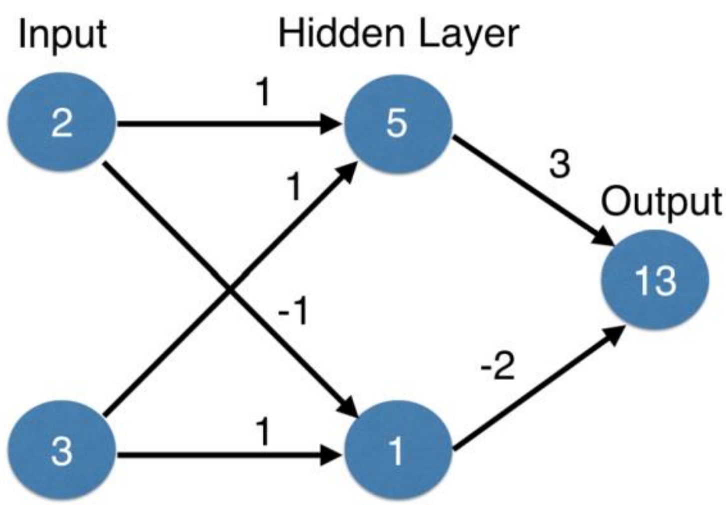
Optimizing Neural Networks with Backward Propagation

A baseline neural network



- Actual Value of Target: 13
- Error: Predicted - Actual = -4

A baseline neural network



- Actual Value of Target: 13
- Error: Predicted - Actual = 0

Predictions with multiple points

- Making accurate predictions gets harder with more points
- At any set of weights, there are many values of the error
- ... corresponding to the many points we make predictions for

Loss function

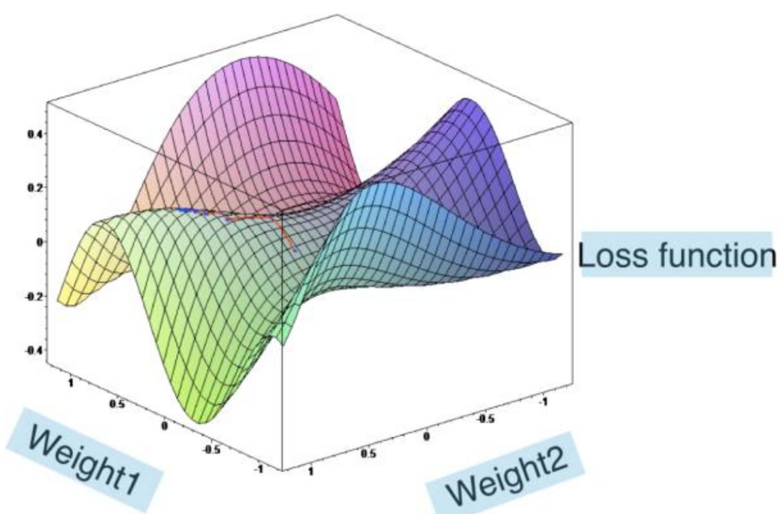
- Aggregates errors in predictions from many data points into single number
- Measure of model's predictive performance

Squared error loss function

Prediction	Actual	Error	Squared Error
10	20	-10	100
8	3	5	25
6	1	5	25

- Total Squared Error: 150
- Mean Squared Error: 50

Loss function



Loss function

- Lower loss function value means a better model
- Goal: Find the weights that give the lowest value for the loss function
- Gradient descent

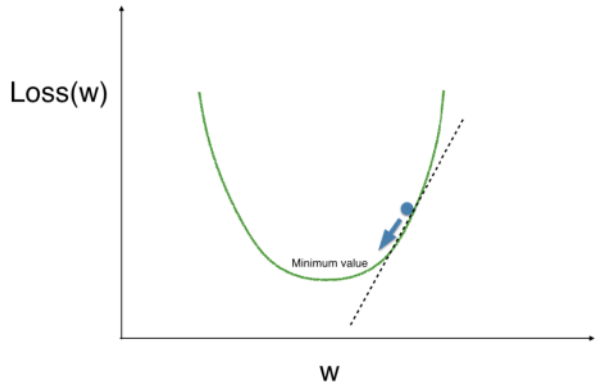
Gradient descent

- Imagine you are in a pitch dark field
- Want to find the lowest point
- Feel the ground to see how it slopes
- Take a small step downhill
- Repeat until it is uphill in every direction

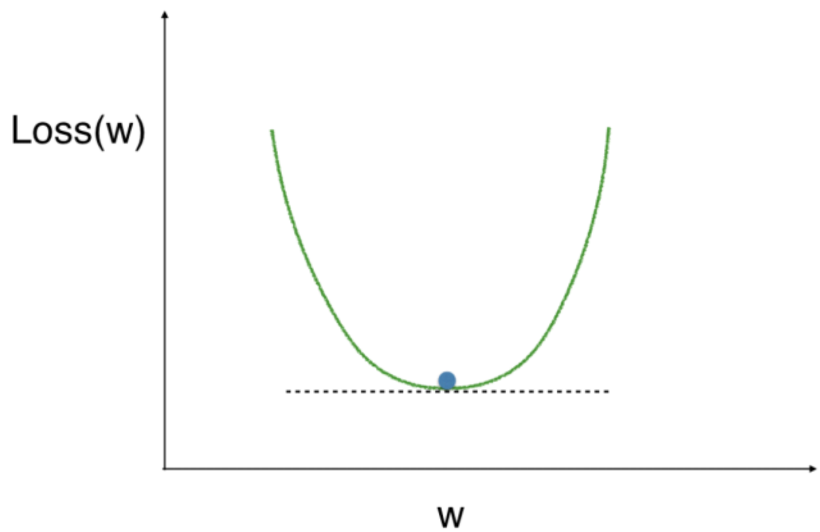
Gradient descent steps

- Start at random point
- Until you are somewhere flat:
  - Find the slope
  - Take a step downhill

Optimizing a model with a single weight



Gradient descent



Gradient descent

- If the slope is positive:
    - Going opposite the slope means moving to lower numbers
    - Subtract the slope from the current value
    - Too big a step might lead us astray
- Solution: learning rate
    - Update each weight by subtracting learning rate * slope

Slope calculation example

- To calculate the slope for a weight, need to multiply:
  - Slope of the loss function w.r.t value at the node we feed into
  - The value of the node that feeds into our weight
  - Slope of the activation function w.r.t value we feed into

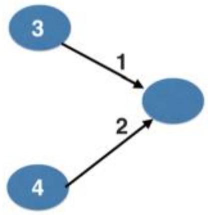
Slope calculation example

- Slope of mean-squared loss function w.r.t prediction:
  - 2 *(Predicted Value - Actual Value) = 2* Error
  - 2 * -4

Slope calculation example

- `2 * -4 * 3`
- `-24`
- If learning rate is `0.01` , the new weight would be
- `2 - 0.01(-24) = 2.24`

Network with two inputs affecting prediction



Code to calculate slopes and 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
print(error)
```

```
5
```

Code to calculate slopes and update weights

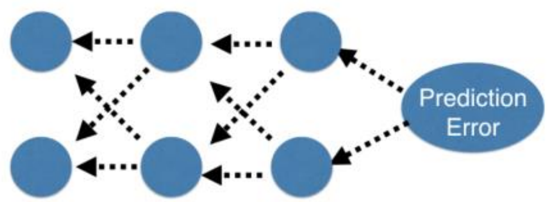
```
gradient = 2 * input_data * error
gradient
```

```
array([30, 40])
```

```
weights_updated = weights - learning_rate * gradient
preds_updated = (weights_updated * input_data).sum()
error_updated = preds_updated - target
print(error_updated)
```

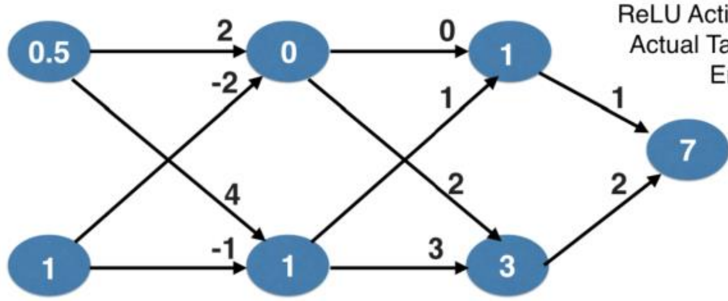
```
-2.5
```

Backpropagation



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

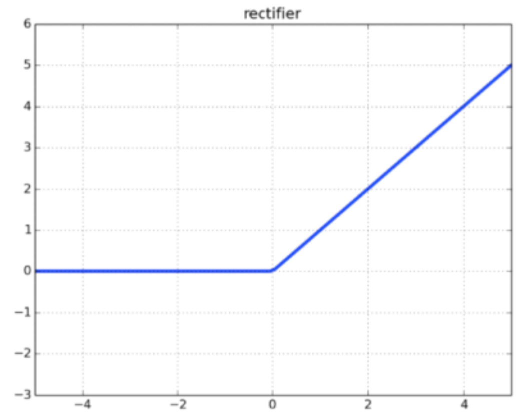
Backpropagation process



Backpropagation process

- Go back one layer at a time
- Gradients for weight is product of:
  1. Node value feeding into that weight
  2. Slope of loss function w.r.t node it feeds into
  3. Slope of activation function at the node it feeds into

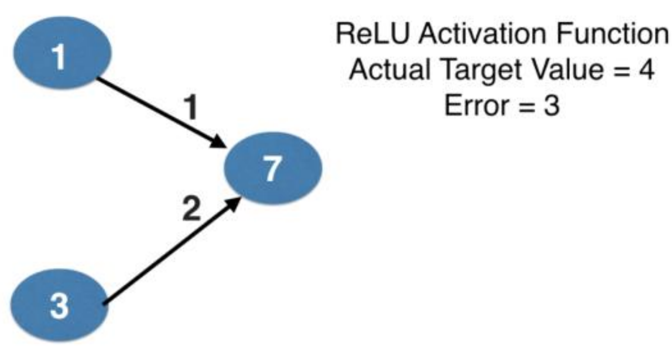
ReLU Activation Function



Backpropagation process

- 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

Backpropagation



We multiply 3 things.

The node values feeding into these weights

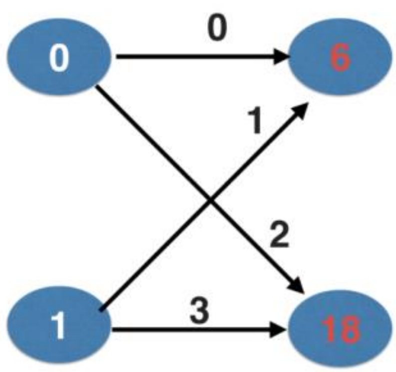
are 1 and 3.

The relevant slope for the output node is 2 times the error.

That's 6.

And the slope of the activation function is 1, since the output node is positive

Backpropagation



we have a slope for the top weight of 6, and a slope for the bottom weight of 18.

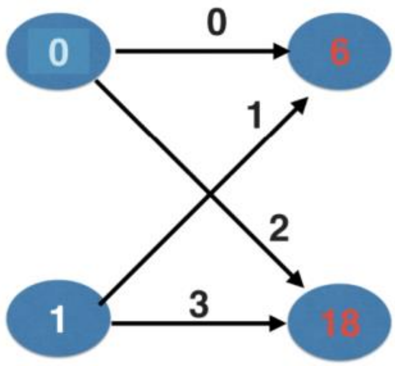
white to denotes node values, black to denote weight values, and the red shows the

calculated slopes of the loss function with respect to that node, which we just finished calculating.

Calculating slopes associated with any weight

- Gradients for weight is product of:
  1. Node value feeding into that weight
  2. Slope of activation function for the node being fed into
  3. Slope of loss function w.r.t output node

Backpropagation



Current Weight Value	Gradient
0	0
1	6
2	0
3	18

For the top weight going into the top node, we multiply

0 for the input node's value, which is in white.

Times 6 for the output node's slope, which is in red.

Times the derivative of the ReLU activation function.

That output node has a positive value for the input, so the ReLU activation has

a slope of 1.

0 times 6 times 1 is 0.

Backpropagation: Recap

- Start at some random set of weights
- Use forward propagation to make a prediction
- Use backward propagation to calculate the slope of the loss function w.r.t each weight
- Multiply that slope by the learning rate, and subtract from the current weights
- Keep going with that cycle until we get to a flat part

Stochastic gradient descent

- It is common to calculate slopes on only a subset of the data (a *batch*)
- Use a different batch of data to calculate the next update
- Start over from the 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: stochastic gradient descent

Model building steps

- Specify Architecture
- Compile
- Fit
- Predict

Model specification

```
import numpy as np
from keras.layers import Dense
from keras.models import Sequential

predictors = np.loadtxt('predictors_data.csv', delimiter=',')
n_cols = predictors.shape[1]

model = Sequential()
model.add(Dense(100, activation='relu', input_shape = (n_cols,)))
model.add(Dense(100, activation='relu'))
model.add(Dense(1))
```

Why you need to compile your model

- Specify the optimizer
  - Many options and mathematically complex
  - "Adam" is usually a good choice
- Loss function
  - "mean_squared_error" common for regression

Compiling a model

```
n_cols = predictors.shape[1]
model = Sequential()
model.add(Dense(100, activation='relu', input_shape=(n_cols,)))
model.add(Dense(100, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mean_squared_error')
```

What is fitting a model

- Applying backpropagation and gradient descent with your data to update the weights
- Scaling data before fitting can ease optimization

Fitting a model

```
n_cols = predictors.shape[1]
model = Sequential()
model.add(Dense(100, activation='relu', input_shape=(n_cols,)))
model.add(Dense(100, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mean_squared_error')
model.fit(predictors, target)
```

Classification

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

Quick look at the data

shot_clock	dribbles	touch_time	shot_dis	close_def_dis	shot_result
10.8	2	1.9	7.7	1.3	1
3.4	0	0.8	28.2	6.1	0
0	3	2.7	10.1	0.9	0
10.3	2	1.9	17.2	3.4	0

Transforming to categorical

shot_result
1
0
0
0

→

Outcome 0	Outcome 1
0	1
1	0
1	0
1	0

Classification

```
from keras.utils.np_utils import to_categorical

data = pd.read_csv('basketball_shot_log.csv')
predictors = data.drop(['shot_result'], axis=1).as_matrix()
target = to_categorical(data.shot_result)

model = Sequential()
model.add(Dense(100, activation='relu', input_shape = (n_cols,)))
model.add(Dense(100, activation='relu'))
model.add(Dense(100, activation='relu'))
model.add(Dense(2, activation='softmax'))
model.compile(optimizer='adam', loss='categorical_crossentropy',
           metrics=['accuracy'])
model.fit(predictors, target)
```

Using models

- Save
- Reload
- Make predictions

Saving, reloading and using your Model

```
from keras.models import load_model
model.save('model_file.h5')
my_model = load_model('my_model.h5')
predictions = my_model.predict(data_to_predict_with)
probability_true = predictions[:,1]
```

Fine tuning keras Model

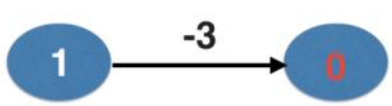
Why optimization is hard

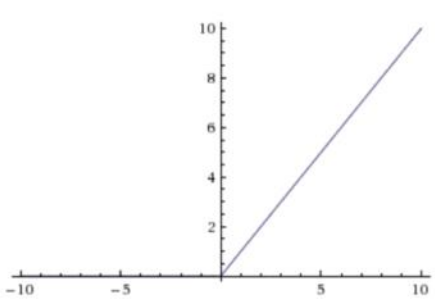
- Simultaneously optimizing 1000s of parameters with complex relationships
- Updates may not improve model meaningfully
- Updates too small (if learning rate is low) or too large (if learning rate is high)

Stochastic gradient descent

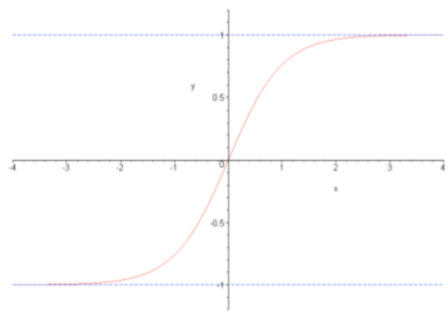
```
def get_new_model(input_shape = input_shape):
    model = Sequential()
    model.add(Dense(100, activation='relu', input_shape = input_shape))
    model.add(Dense(100, activation='relu'))
    model.add(Dense(2, activation='softmax'))
    return(model)

lr_to_test = [.000001, 0.01, 1]

# loop over learning rates
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





Vanishing gradients



tanh function

Vanishing gradients

- Occurs when many layers have very small slopes (e.g. due to being on flat part of tanh curve)
- In deep networks, updates to backprop were close to 0

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

Model validation

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

```
Epoch 1/10
89648/89648 [=====] - 3s - loss: 0.7552 - acc: 0.5775 - val_loss: 0.6969 - val_acc: 0.5561
Epoch 2/10
89648/89648 [=====] - 4s - loss: 0.6670 - acc: 0.6004 - val_loss: 0.6580 - val_acc: 0.6102
...
Epoch 8/10
89648/89648 [=====] - 5s - loss: 0.6578 - acc: 0.6125 - val_loss: 0.6594 - val_acc: 0.6037
Epoch 9/10
89648/89648 [=====] - 5s - loss: 0.6564 - acc: 0.6147 - val_loss: 0.6568 - val_acc: 0.6110
Epoch 10/10
89648/89648 [=====] - 5s - loss: 0.6555 - acc: 0.6158 - val_loss: 0.6557 - val_acc: 0.6126
```

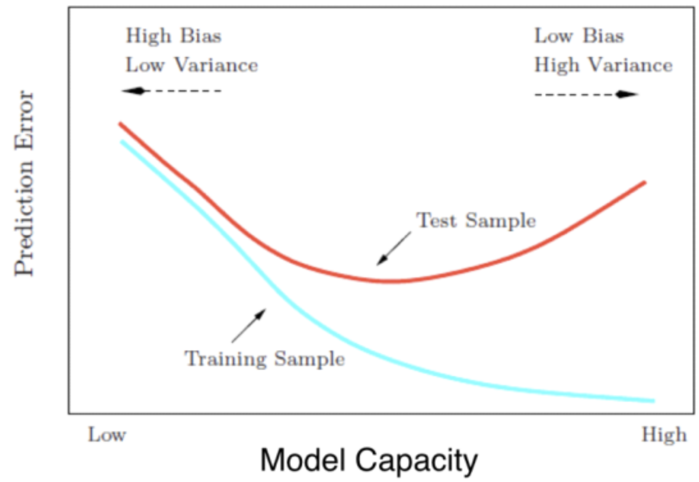
Early Stopping

```
from keras.callbacks import EarlyStopping

early_stopping_monitor = EarlyStopping(patience=2)

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

Experimentation

- Experiment with different architectures
- More layers
- Fewer layers
- Layers with more nodes
- Layers with fewer nodes
- Creating a great model requires experimentation

Overfitting



Workflow for optimizing model capacity

- Start with a small network
- Gradually increase capacity
- 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
- 28 x 28 grid flattened to 784 values for each image
- Value in each part of array denotes darkness of that pixel