Some practical considerations

Learning rate, parameter initialisation, batch-vs-stochastic gradient, etc.

In this Lecture

Practical considerations

- Learning rate & some hints to variants of SGD
- Batch processing
- Data normalization
- Parameter Initialization
- Parameter Regularization
- Dropouts
- Activation Functions
- (Backpropagation)

Optimization: Learning Rate

How far should we move?

The *step size* or *learning rate* defines how big a step we should take in the direction of the gradient

Optimization: Learning Rate

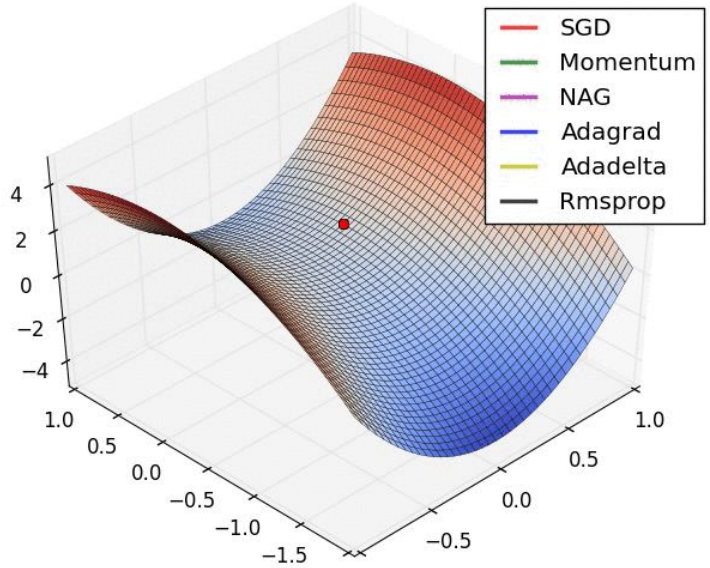
How far should we move?

The *step size* or *learning rate* defines how big a step we should take in the direction of the gradient

It must be well controlled - too small a step and it may take a long time to reach the bottom - too big a step and we may miss the minimum all together!

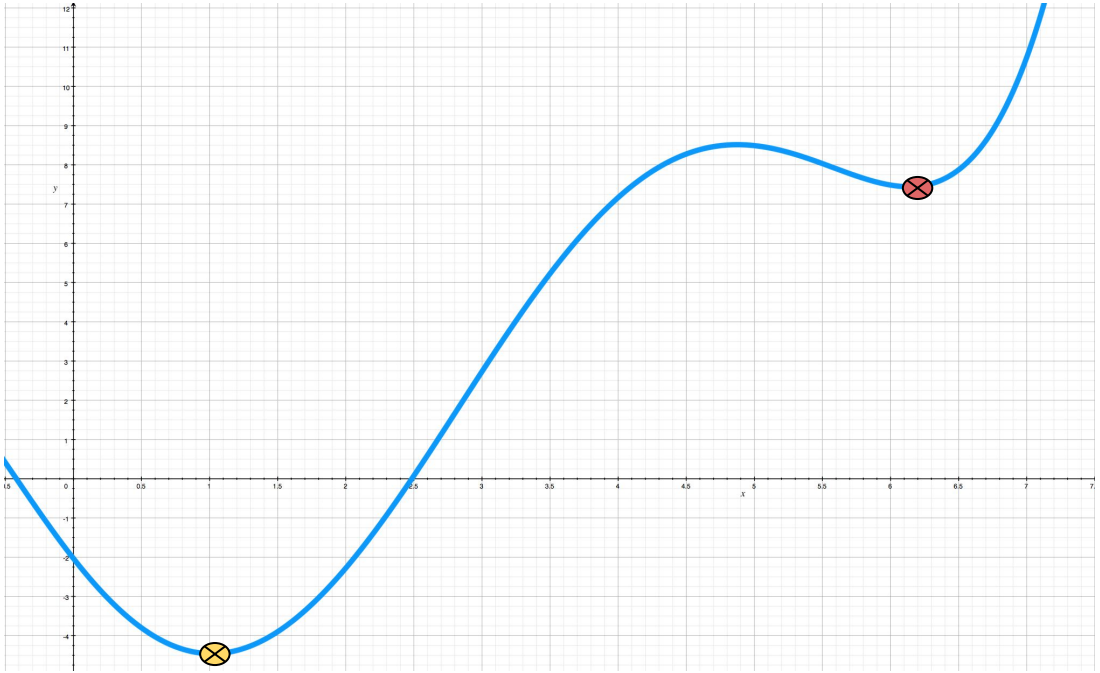
Optimization

Various optimization algorithms



Alec Radford (Reddit)

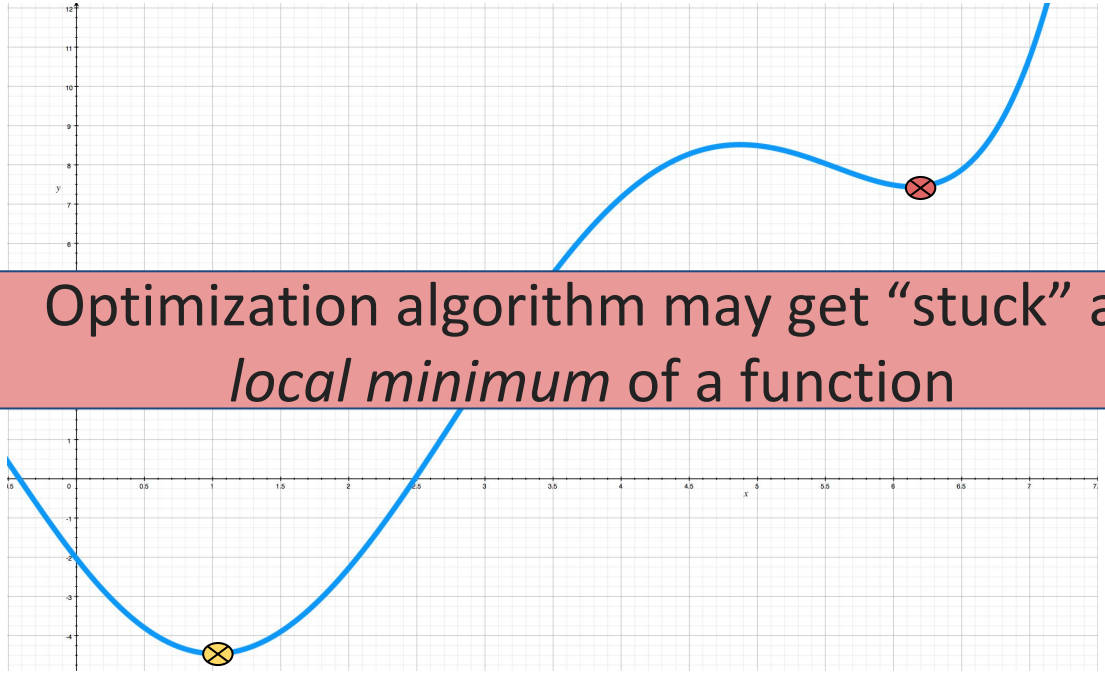
Local Minima



Minimum value of the function

Local minimum value of the function

Local Minima



Optimization algorithm may get “stuck” at *local minimum* of a function

Minimum value of the function

Local minimum value of the function

Learning rate in Keras

sgd = keras.optimizers.SGD(*lr=0.1*, *decay=1e-6*, momentum=0.9, nesterov=False/True)

model.compile(loss='mean_squared_error', optimizer=sgd)

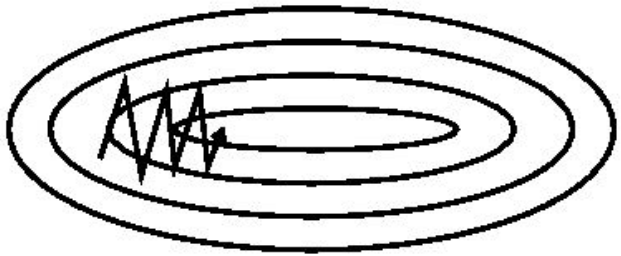
We use a constant learning rate and decay it after each iteration. Other ways is to use an adaptive way (speed up or slow down, even for each parameter): AdaGrad, RMSprop and **ADAM (state of the art now)**.

AdaGrad e.g. reduces the learning rate for parameters that got big updates and increases it for parameters with small updates.

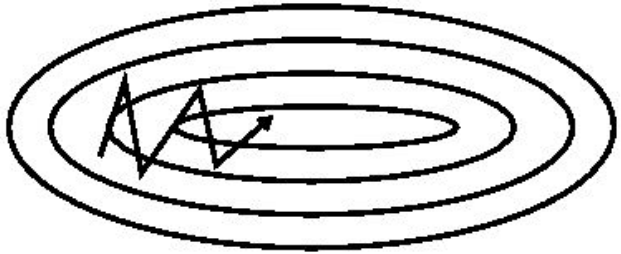
See these optimizers: https://keras.io/optimizers/

Also check this: https://deepnotes.io/sgd-momentum-adaptive

SGD without momentum



SGD with momentum



Batch Processing

Batch Processing

In previous implementations, we've computed the objective function and gradient for every instance, adjusted our parameters and then continued with the next instance.

Batch Processing

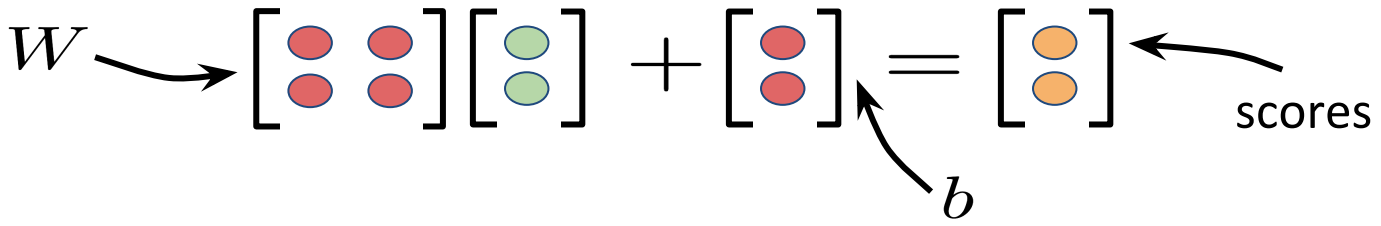
In previous implementations (jupyter notebooks), we've computed the objective function and gradient for every instance, adjusted our parameters and then continued with the next instance.

This approach is known as ***stochastic gradient descent*** (SGD).

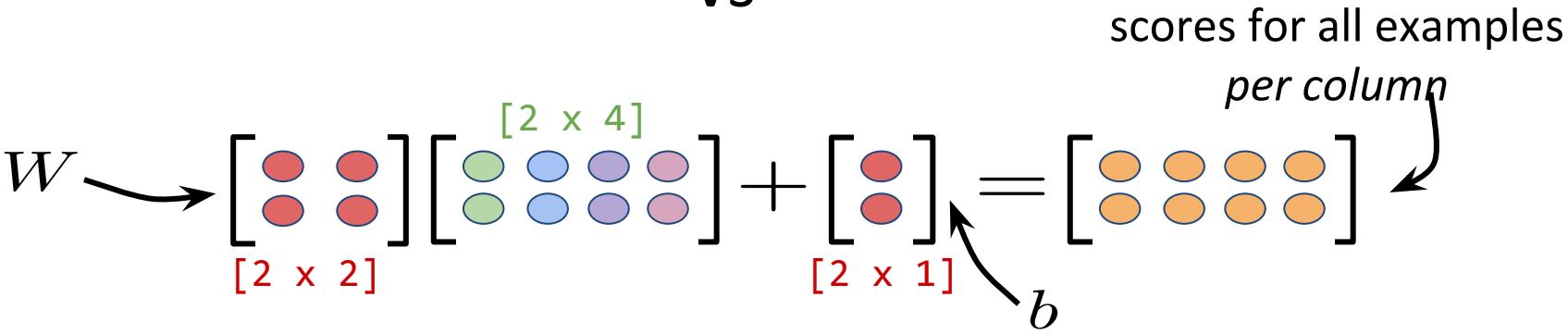
Batch Processing

Another approach is to accumulate the gradients over all instances, and then do a single update to the parameters - this approach is known as ***Batch gradient descent.***

Batch Processing



vs



Batch Processing

Batch gradient descent leads to more “stable” updates - the direction towards the optimal parameters is computed after looking at all examples, instead of just one!

Batch Processing

What if you had a few outliers (bad examples)

- SGD will cause the parameters to drift farther from their optimal values when the update loop goes over these outliers.
- Batch GD will drown out the effect of the outliers since there are many more good examples.

Batch Processing

But

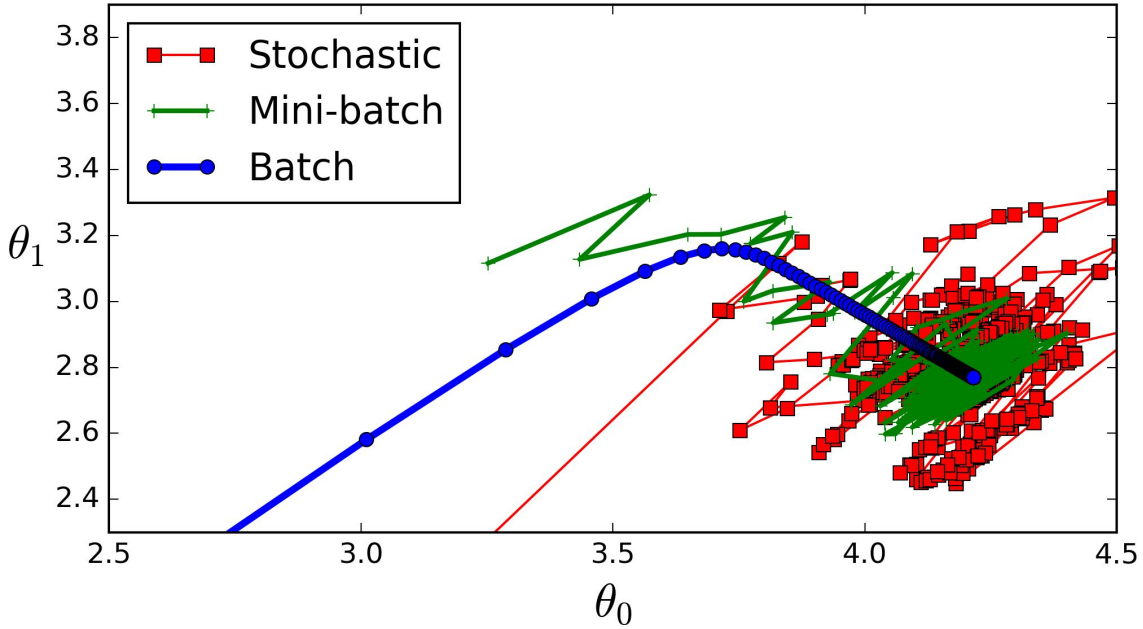
- Batch GD requires us to look over the entire dataset before making any progress - so it's much slower.
- The entire dataset may not even fit in memory, so making the overall code efficient would be more difficult.

Batch Processing

Solution

- **Minibatch SGD**: Perform updates after looking at a “minibatch” (e.g. 32 data points)
- Much faster than Batch GD, but largely avoids the issues with SGD.

Batch Processing



https://stats.stackexchange.com/a/153535

Batch size in Keras

model.fit(..., batch_size=128,...)

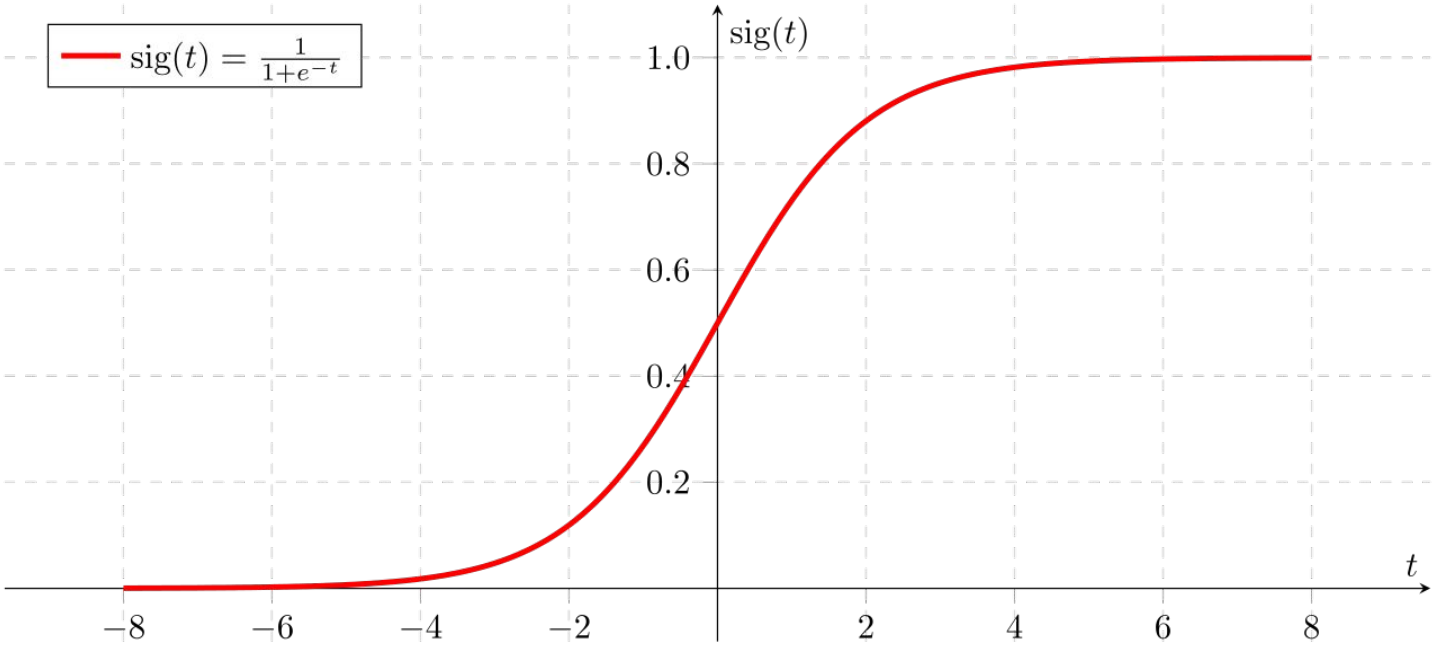
- **batch_size**: Integer or None. Number of samples per gradient update. If unspecified, it will default to 32.

Data Normalization

Data Normalization

- Features in the data come at different scales
- Large feature values when multiplied with random weights result in larger values
- Activation functions like sigmoid result in the saturation of these neurons
- The optimization stops working for those parameters since the gradient becomes very small or very close to zero

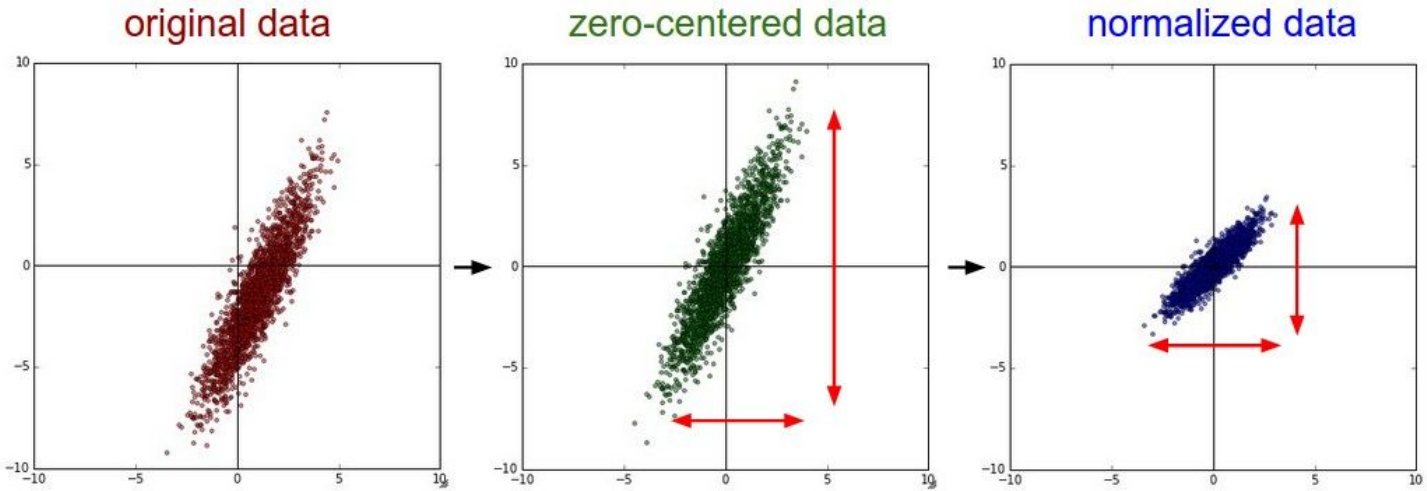
Data Normalization



Data Normalization

- Normalizing the data before training the model is one way to solve this problem
- A common method of data normalization is to first **zero center** the data by subtracting the **mean** across every *feature* in the data, and then **divide** the resulting value with **standard deviation**
- This results in the data having *unit variance*

Data Normalization



https://cs231n.github.io/neural-networks-2/#datapre

Data Normalization

```
from sklearn.preprocessing import StandardScaler

sc = StandardScaler()
sc.fit(X_train)
X_train_std = sc.transform(X_train)
X_test_std = sc.transform(X_test)
```

Parameter Initialization

Parameter Initialization

- We've learned that we need to move in the direction of the gradient to reach the minimum value for a given loss function
- But where do we start?

Parameter Initialization

- Initial values of `W` and `b` dictate where in the terrain we begin
- If we start near a minima, we can optimize very quickly - If we start too far, it may take a long time to find a good model
- We may even start near a local minimum and never find the global minimum for a given function

Parameter Initialization

- Zero initialization?
- Random initialization?
- Something more complicated?

Parameter Initialization

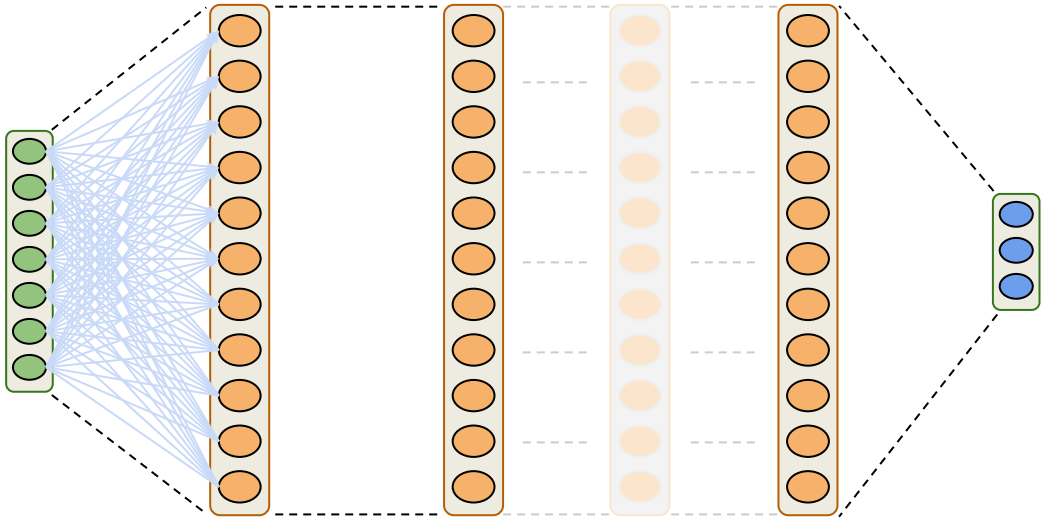
- Zero initialization
- Random initialization
- Something more complicated:
  - Xavier Initialization

Zero Initialization

Q: Why should we never use zero initialization with neural networks?

Zero Initialization

Q: Why should we never use zero initialization with neural networks?



Zero Initialization

Q: Why should we never use zero initialization with neural networks?

A: All the neurons see the same input - and with the same weight matrices, they will make the exact same decisions!

Random Initialization

- Randomly initialize weights close to zero
- Avoids symmetry in the network by having different random weights

Xavier Initialization

There is a better way to initialize weight matrices other than random numbers: **Xavier initialization**

Xavier Initialization

There is a better way to initialize weight matrices other than random numbers: **Xavier initialization**

Different for each layer - depends on the number of connections coming in and going out!

Xavier Initialization

Usually when we pick random numbers, we pick them from a uniform distribution

Xavier Initialization

Usually when we pick random numbers, we pick them from a uniform distribution

Xavier initialization says we should pick the random numbers from a distribution with zero mean and the following variance:

$$Var(W) = \frac{2}{n_{in} + n_{out}}$$

Xavier Initialization

Works very well in practice - was actually an enabler in training deeper networks at some point in time!

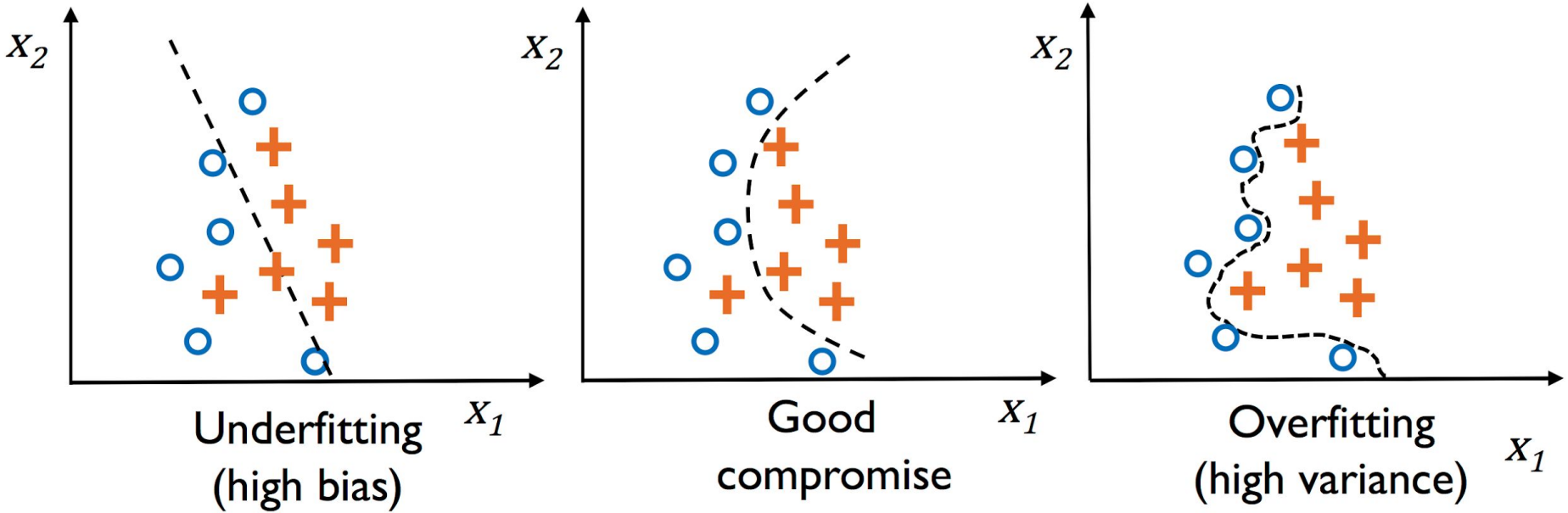
Initialization in Keras

```
model.add(Dense(64, init='uniform'))
```

glorot_normal is the Xavier initialization. There are more here: https://faroit.github.io/keras-docs/1.2.2/initializations/

Regularization

Parameter Regularizations
Overfitting/Underfitting



Parameter Regularizations Overfitting/Underfitting

Overfitting, i.e. there is high variance ⇒ likely that the system has too many parameters that lead to a model that is too complex given the underlying data.

Underfitting, i.e. there is high bias ⇒ our model is not complex enough to capture the pattern in the training data well and therefore suffers from low performance on unseen data.

L2 regularization

L2 Regularization

This helps to tackle the problem with collinearity (high correlation among features), filter out noise from data and prevent overfitting.

Idea: Introduce to the cost functions some extra weights to penalize extreme parameter weights:

$$\frac{\lambda}{2}||\mathbf{w}||^2 = \frac{\lambda}{2}\sum_{j=1}^{n} w_j^2$$

$$J(\mathbf{w}) = \sum_{i=1}^{n}\left[-y^i log(\phi(z^i)) - (1-y^i) log((1-\phi(z^i)))\right] + \frac{\lambda}{2}||\mathbf{w}||^2$$

Bigger λ increases the importance of regularization.

Note, for regularization it is important to standardize the features!

L2 regularization in Keras

```
from keras.regularizers import l2

model.add(Dense(64, input_dim=64, W_regularizer=l2(0.01)))
```

Dropout

Overfitting

- Humans tend to believe what comes frequently in their lives or daily routine
  - a recurrent advertisement make us believe it
  - in return, we are less open to any new information
- Similarly, machines tend to overfit what they have seen during training
  - result in less robust model to any unseen scenario

Dropout

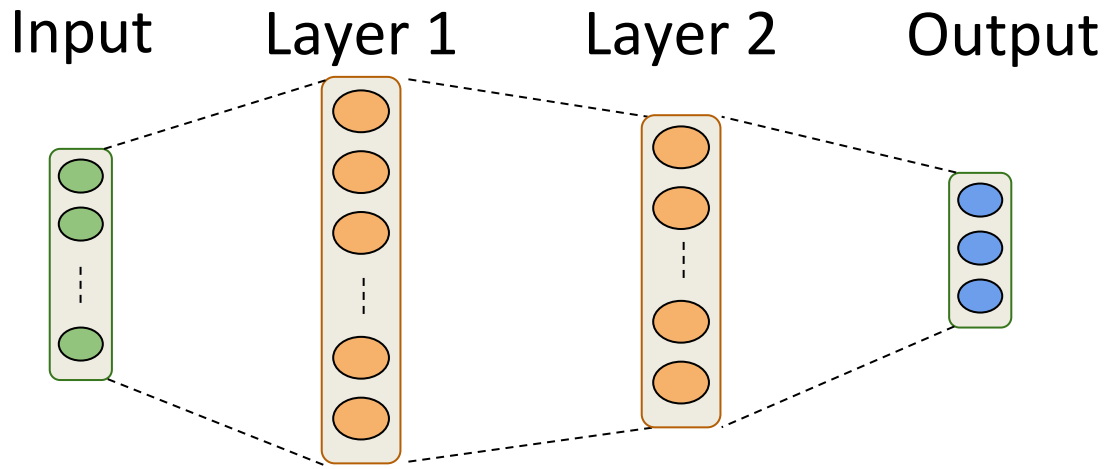
- Neural network models tend to overfit the training data
  - performs poorly on unseen data
- How can we expose our network to diverse scenarios?
- Can we make the training of the model more robust and improve generalization?

Dropout

"Randomly drop neurons from the network during training"

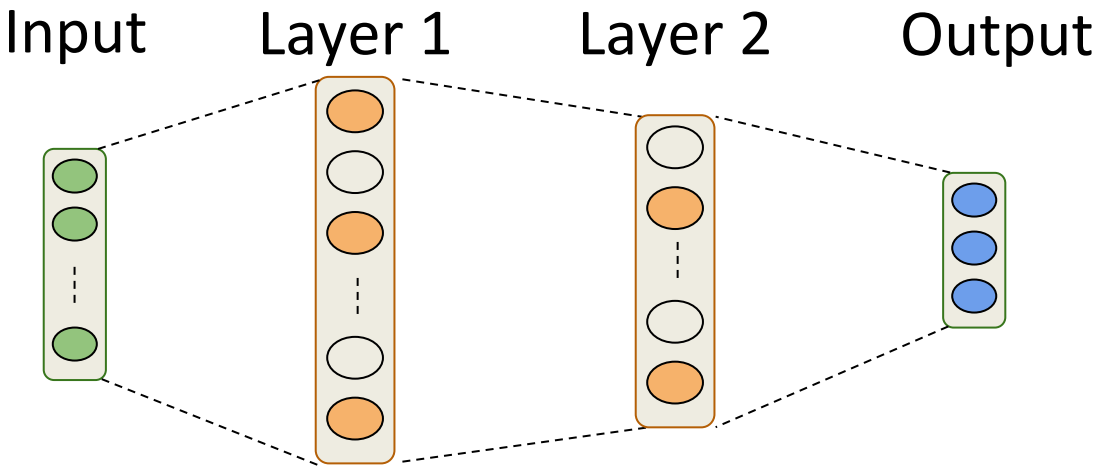
- **Intuition:** give network a wider class of scenarios to tackle
- By dropping a few neurons, we are essentially forcing the model to learn in scenarios when some information is missing
- Some other neuron(s) have to step up to handle this situation

Dropout



- Let’s say we want to randomly drop a few neurons from every layer

Dropout



- In other words, some information in the network is missing
- Now model has to learn to reduce loss with fewer neurons
- This improves generalization of the model

Dropout

- Algorithmically, say we want to apply dropout of p=0.5 on the complete network

- At train time, for every layer in every iteration:
  - For every neuron
    - predict a random number between 0 and 1
    - if number is less than 0.5, drop that neuron and its connections

- At test time, **always use the complete network**
  - compensate for missing activations during training by reducing all activations by the factor p

Dropout

- Frequently used while training models
  - specially when training data is small

Dropout

Implementing in Keras

```
model.add(Dense(100, ...))
model.add(Dropout(0.5))
...
```

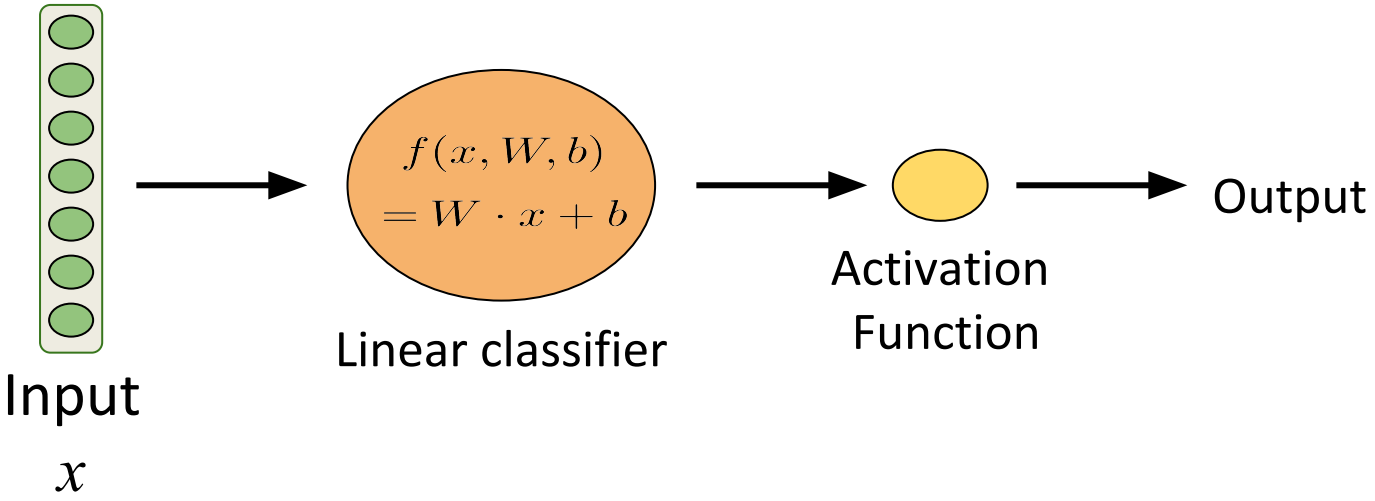
Multi-class classification

Learning Rate and Optimization

Activation Functions

Neuron

A Neuron can be thought of as *a linear classifier* plus an *activation function*



Activation Functions

- Intuitively, a neuron looks at a particular feature of the data

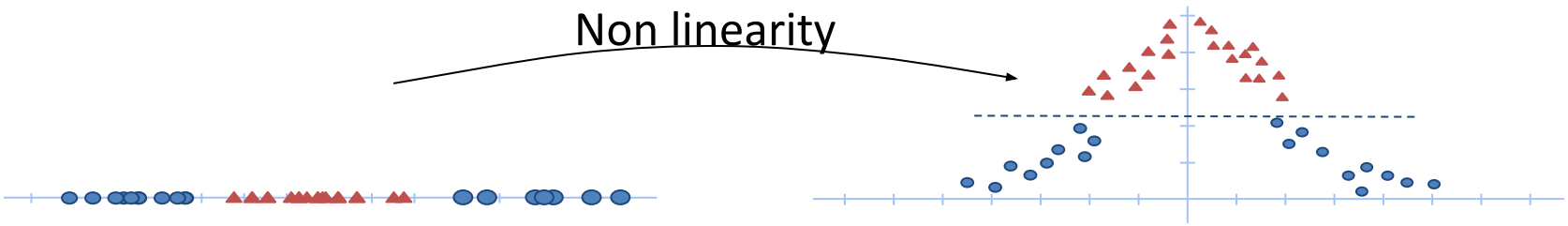
Activation Functions

- Intuitively, a neuron looks at a particular feature of the data
- The activation after the linear classifier gives us an idea of how much the neuron “supports” the feature

As an example, the output of a neuron will be high if the feature it supports is contained in the input
(like “low speed” in the current “car”)

Activation Functions

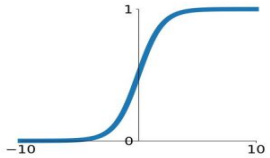
- Intuitively, a neuron looks at a particular feature of the data
- The activation after the linear classifier gives us an idea of how much the neuron “supports” the feature
- Activations also helps us map linear spaces into non-linear spaces



Activation Functions

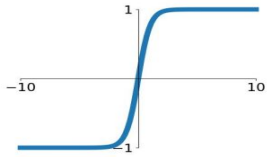
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



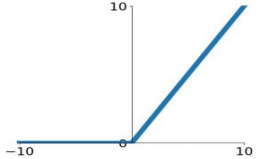
tanh

$$\tanh(x)$$



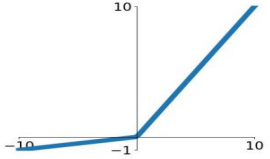
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

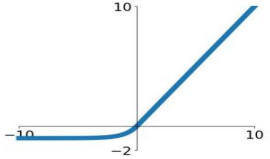


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Multi-class classification

Multi-class Classification - From Previous week