Deep Learning Basics

GO DATA DRIVEN

Goal

- Become familiar with common components of a neural network.
- Understand the effects that different hyperparamter choices can have on the model.

Deep Learning Basics

GO DATA DRIVEN

Program

Architecture:

- Fully-connected layers
- Activation function
- Bias
- Batch normalization layer
- Dropout layer

Hyperparameters:

- loss function
- gradient descent type
- weight updates
- weight initialisation

Architecture

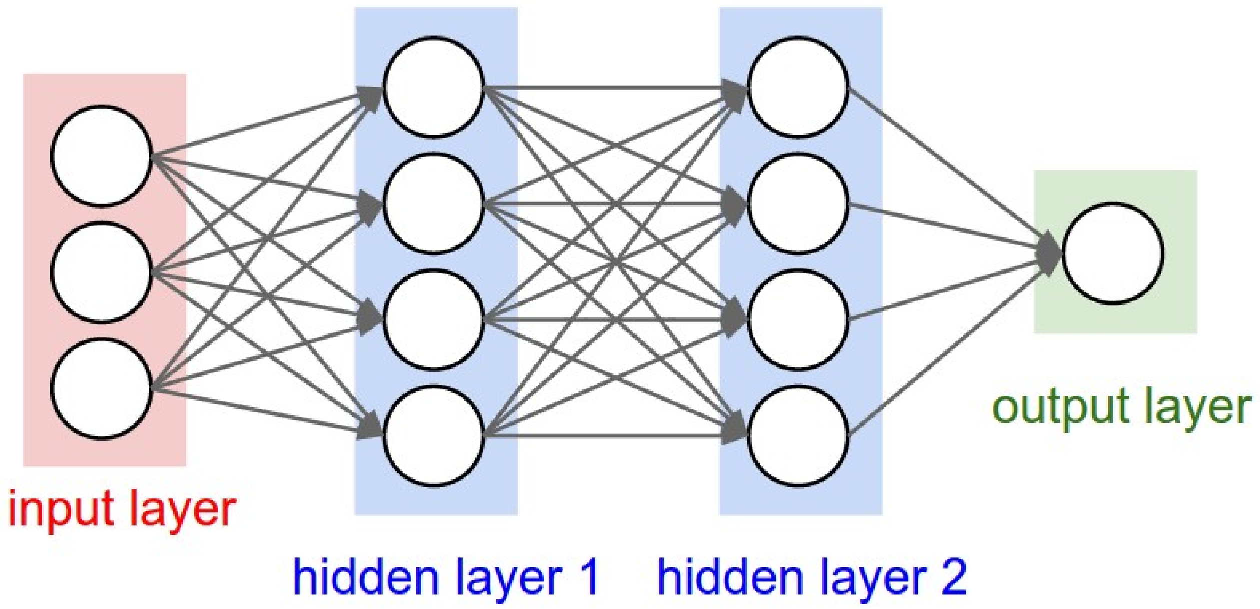
We will discuss the following neural network components

- Fully-connected layers
- Activation function
- Bias
- Batch normalization layer
- Dropout layer

Fully-Connected layer

All inputs of one layer connected to every activation unit of the next layer.

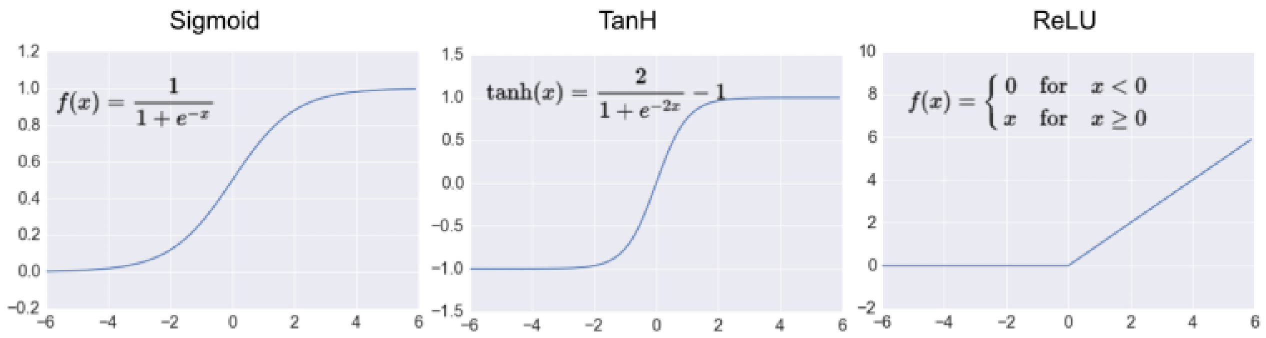
Also known as *Linear* or *Dense* layer



Activation

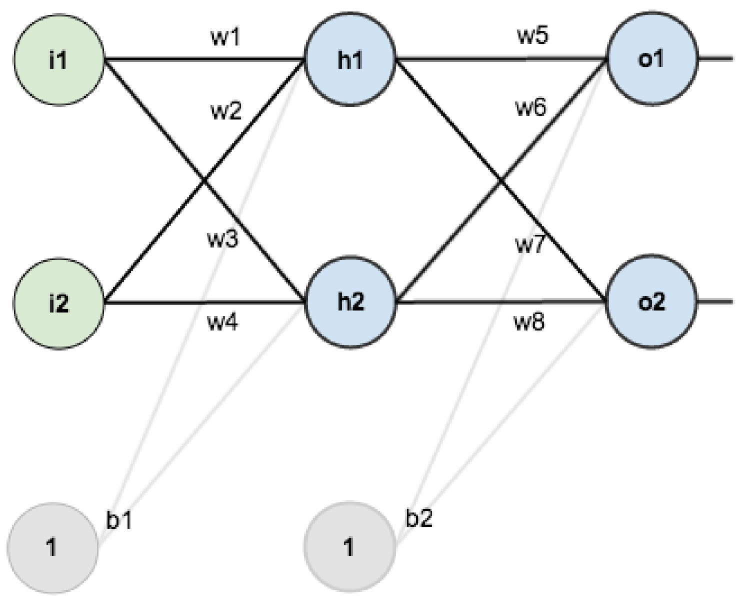
Introduces non-linearity into the network.

No trainable parameters.



Bias

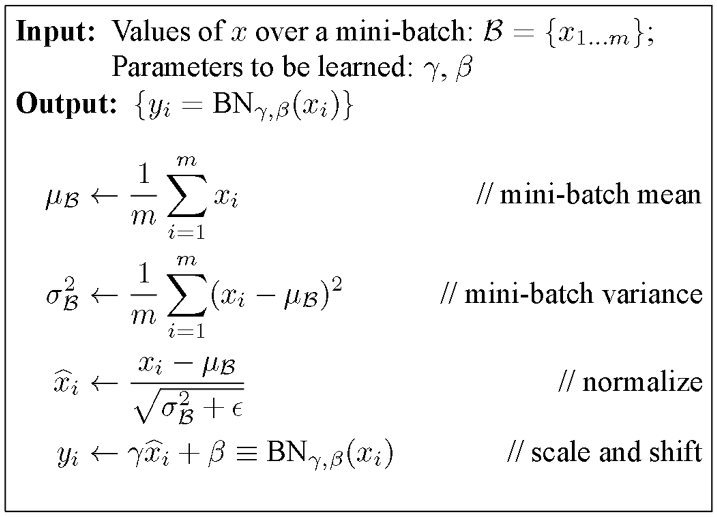
An additional paramter that allows you to shift the input to the activation function to the left or right (which may be critical for successful learning).



Batch Normalization

Batch normalization (Loffe et al.)

- Normalize the layer inputs with batch normalization.
- This helps to ensure all layers activated in near optimal “regime” of the activation functions.
- Since the gradients’ dependency on the scale of the weights is reduced, it allows us to use higher learning rates,
- which means training is accelerated, as less iterations are required to converge to a given loss value.



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1 \ldots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Batch Norm (notes)

Batch Norm learns 4 parameters

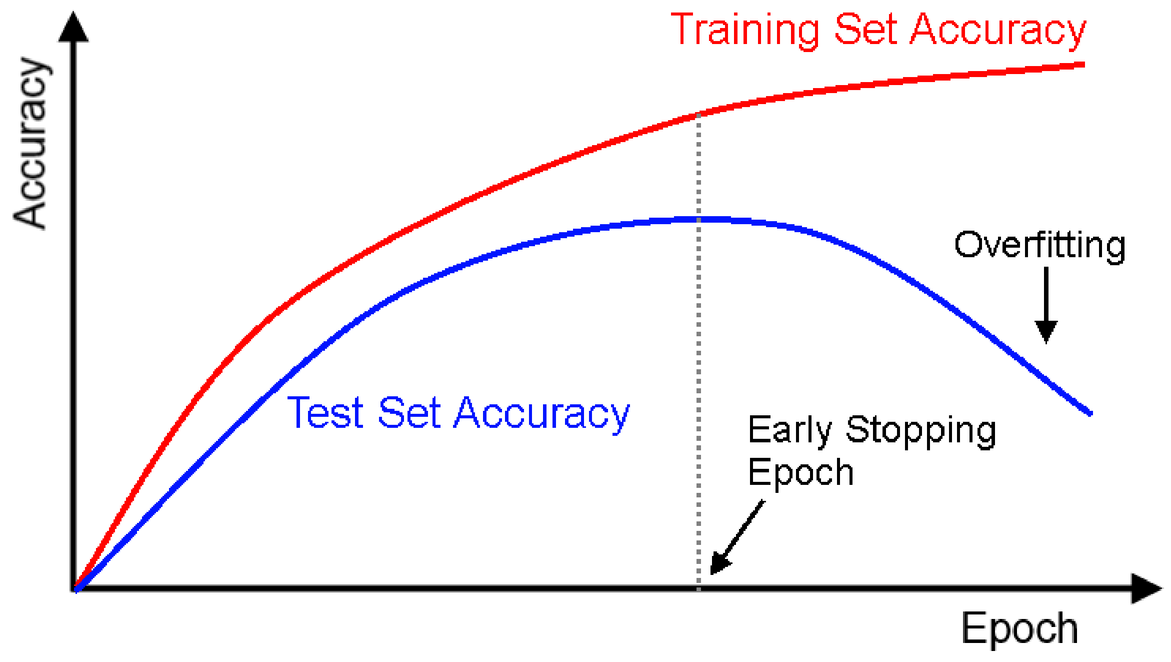
- β
- γ
- running mean μ (for inference stage)
- running variance σ^2 (for inference stage)

Normalization (further reading)

- Weight normalization (Salimans et al.)
- Layer normalization (Ba et al.)

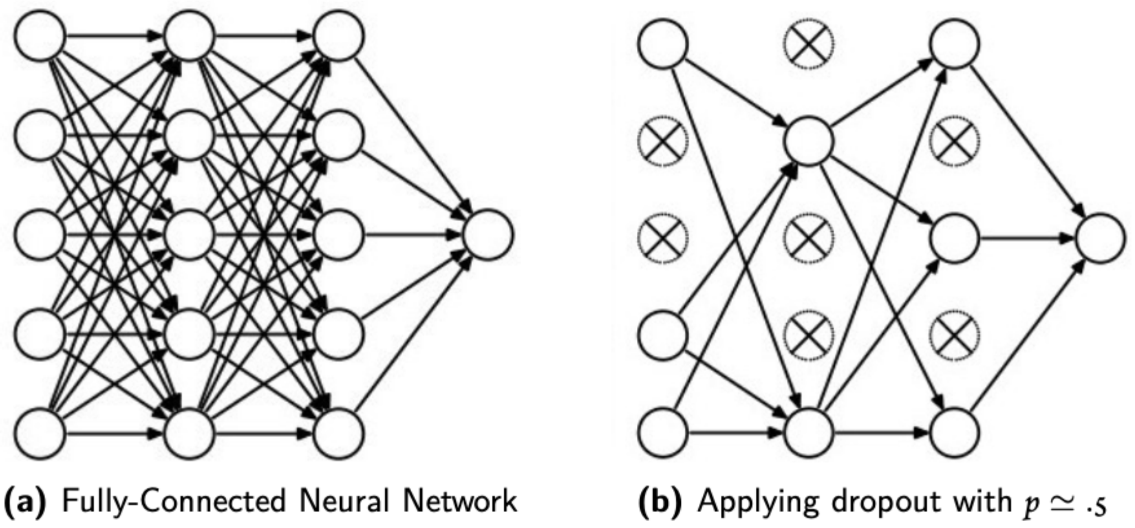
Regularization

The great flexibility of neural networks makes them very powerful, however this comes at the price of easily overfitting of the data.



Dropout

- "Drop" neurons in the network with probability p (every mini-batch/epoch)
- No trainable paramters


(a) Fully-Connected Neural Network
(b) Applying dropout with $p \simeq .5$

Dropout

- Computing the gradient is done with respect to the error, but also with respect to what all other units are doing. Therfore certain neurons may fix the mistakes of other neurons.
- Dropout prevents over-reliance on a subset of the neurons in a layer
- every neuron becomes more robust

Hyperparameters

We shall discuss the follow hyperparamter choices,

- loss function
- gradient descent type
- weight updates
- weight initialisation

Loss function

Classification:

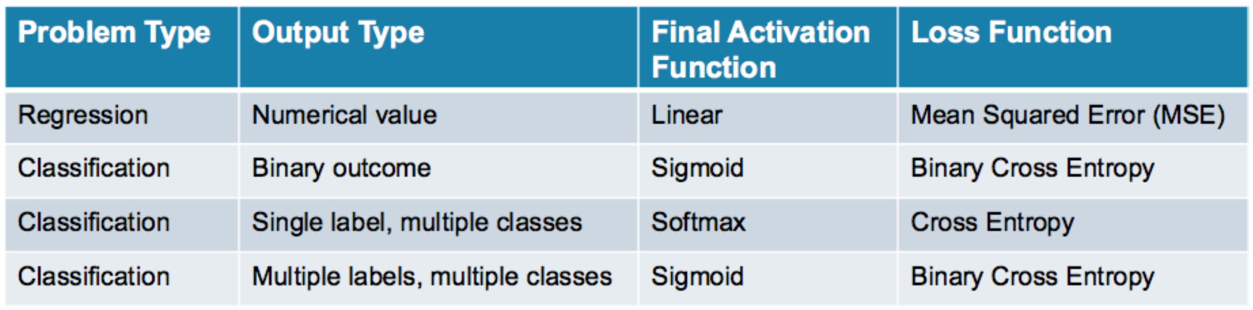
- Binary cross entropy: $L_{binary} = \frac{1}{N}\Sigma_{i=1}^{N}[y_i log(\hat{y}_i) + (1-y_i)log(1-\hat{y}_i)]$
- Categorical cross entropy: $L_{categorical} = \frac{1}{N}\Sigma_{i=1}^{N}\Sigma_{j=i}^{c}[y_{ij}log(\hat{y}_{ij})]$

Regression

- Root mean squared error (RMSE): $L_{RMSE} = \sqrt{\frac{1}{N}\Sigma_{i=1}^{n}(y-\hat{y})^2}$

Loss function

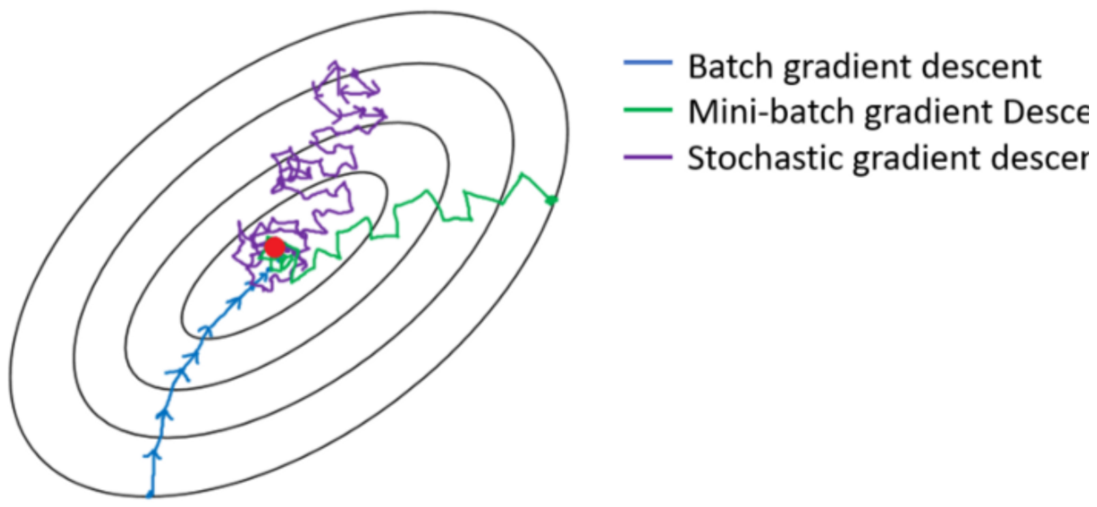
- **Multi-class classification**
  - **softmax** output layer with **categorical** cross-entropy and **one-hot** targets.
- **Binary or multi-label classification**
  - **sigmoid** output layer with **binary** cross-entropy and **binary** vector targets.
- **Regression**
  - **linear** output layer with **RMSE**
  - Not performing? Try **discretizing output** through binning. Otherwise, go for a different learning algorithm.



Problem Type	Output Type	Final Activation Function	Loss Function
Regression	Numerical value	Linear	Mean Squared Error (MSE)
Classification	Binary outcome	Sigmoid	Binary Cross Entropy
Classification	Single label, multiple classes	Softmax	Cross Entropy
Classification	Multiple labels, multiple classes	Sigmoid	Binary Cross Entropy

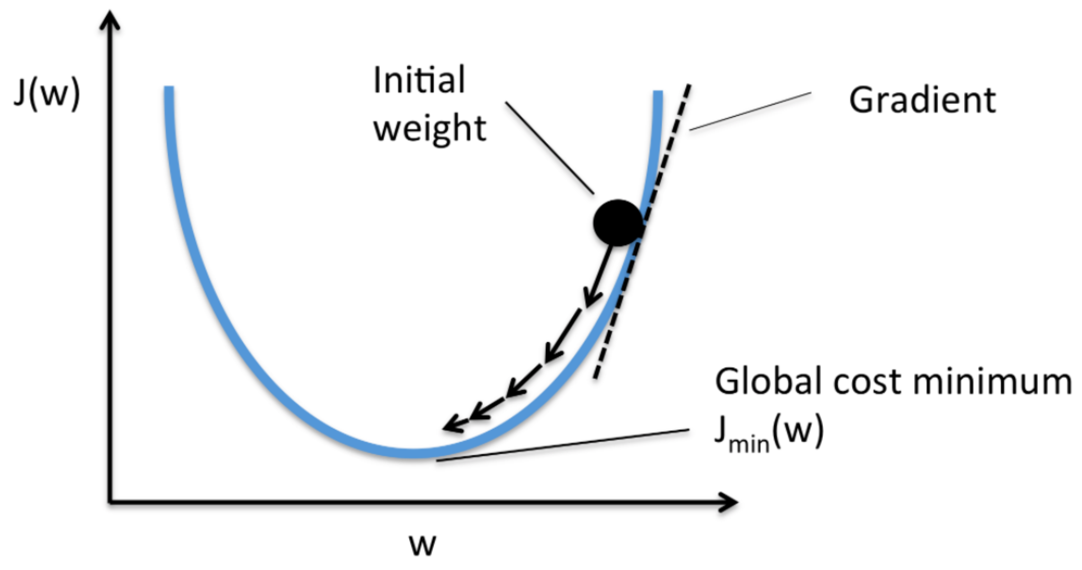
Gradient descent

- Stochastic gradient descent - feed a single datapoint in at each pass.
- Batch gradient descent - feed in the whole batch of data at each pass.
- Mini-batch gradient descent - feed in a group of data at each pass.



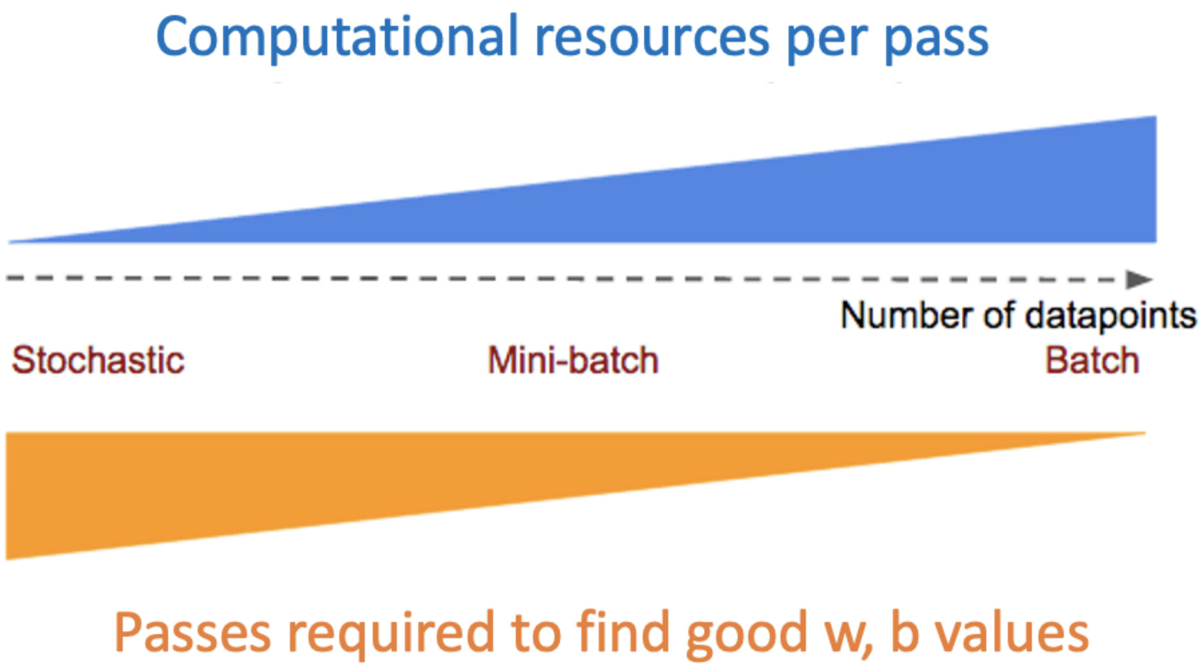
Gradient descent

We update our model using gradient descent.



$$w' = w - \eta \frac{\partial J(w)}{\partial w}$$

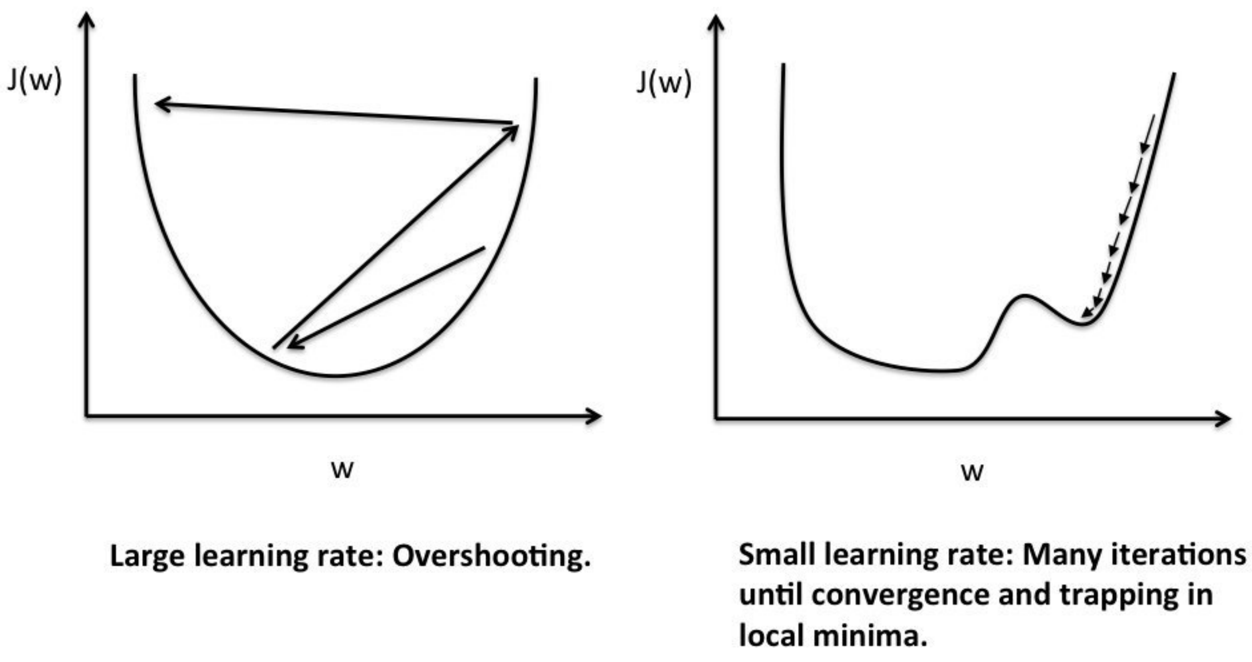
Gradient descent


Computational resources per pass
Number of datapoints
Stochastic
Mini-batch
Batch
Passes required to find good w, b values

- **Forward pass**: a data sample is passed forward through the network to determine a prediction
- **Backward pass**: recursively compute the error backwards from the last layer following the chain-rule and update the weights w.r.t. the known target output.
- **Epoch**: training the neural network with all the training data for one cycle.

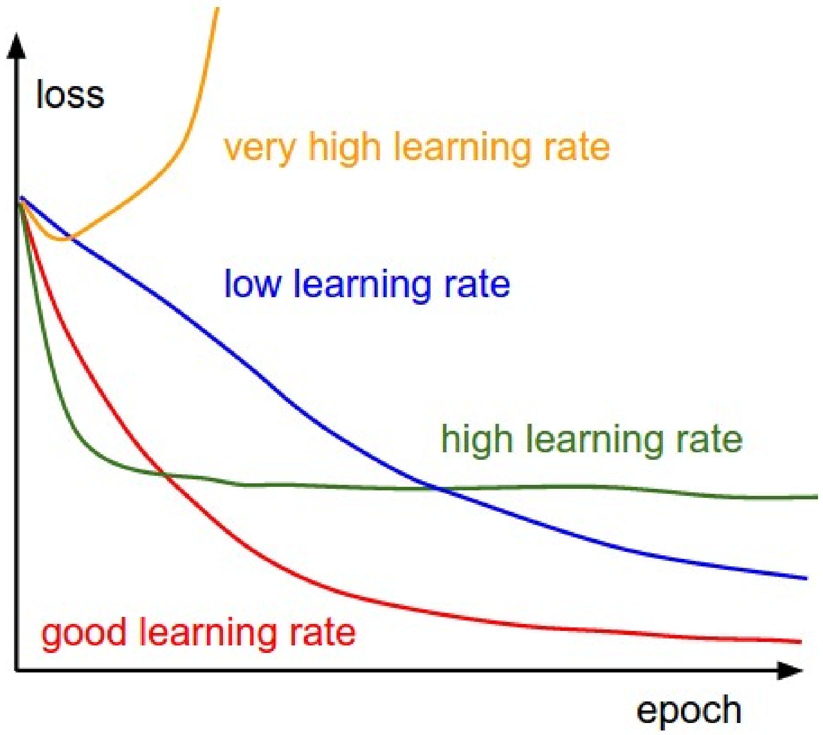
Weight updates

Learning rate: small value η typically between 1.0 and 10^-6


J(w)
w
Large learning rate: Overshooting.
J(w)
w
Small learning rate: Many iterations until convergence and trapping in local minima.

Weight updates

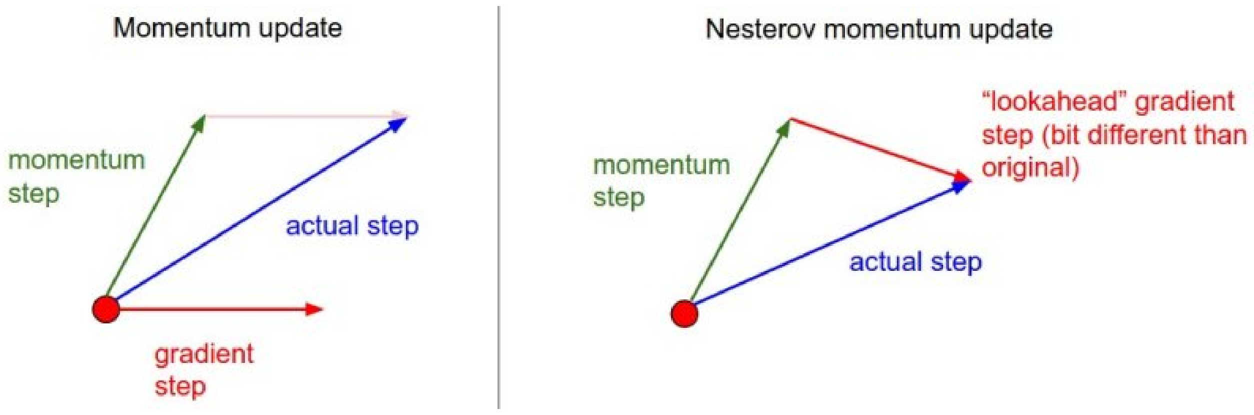
Learning rate: small value η typically between 1.0 and 10^-6



Weight updates

Momentum: take into account the gradient estimation of the previous batches

SGD with momentum, Nesterov momentum



Momentum (further reading)

The main difference is in classical momentum you first correct your velocity and then make a big step according to that velocity (and then repeat), but in Nesterov momentum you first making a step into velocity direction and then make a correction to a velocity vector based on new location (then repeat).

i.e. without momentum:

```
vW(t+1) = - scaling * gradient_F( W(t) )
```

```
W(t+1) = W(t) + vW(t+1)
```

Classical momentum:

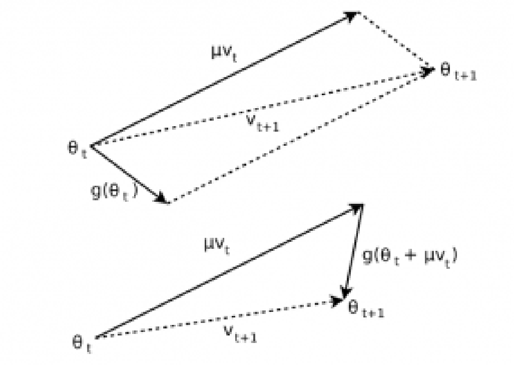
```
vW(t+1) = momentum*Vw(t) - scaling * gradient_F( W(t) )
```

```
W(t+1) = W(t) + vW(t+1)
```

While Nesterov momentum is this:

```
vW(t+1) = momentum*Vw(t) - scaling .* gradient_F( W(t) + momentum*vW(t) )
```

```
W(t+1) = W(t) + vW(t+1)
```



source.) source

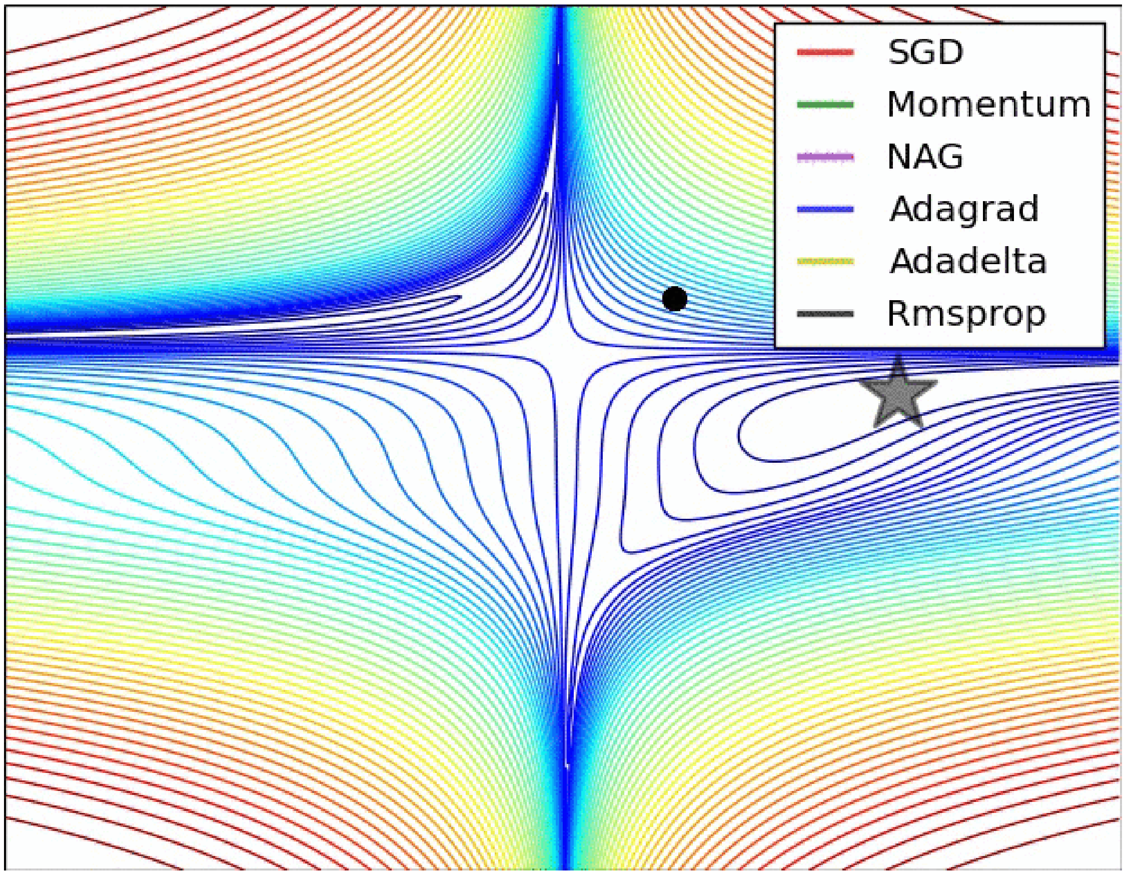
Weight updates

Adaptive learning rate: adapt the learning rate based on the gradient history (removing the dependency on hyperparamter choice).

AdaGrad, AdaDelta, RMSprop

Momentum & adaptive learning rate: *Adam, Nadam*

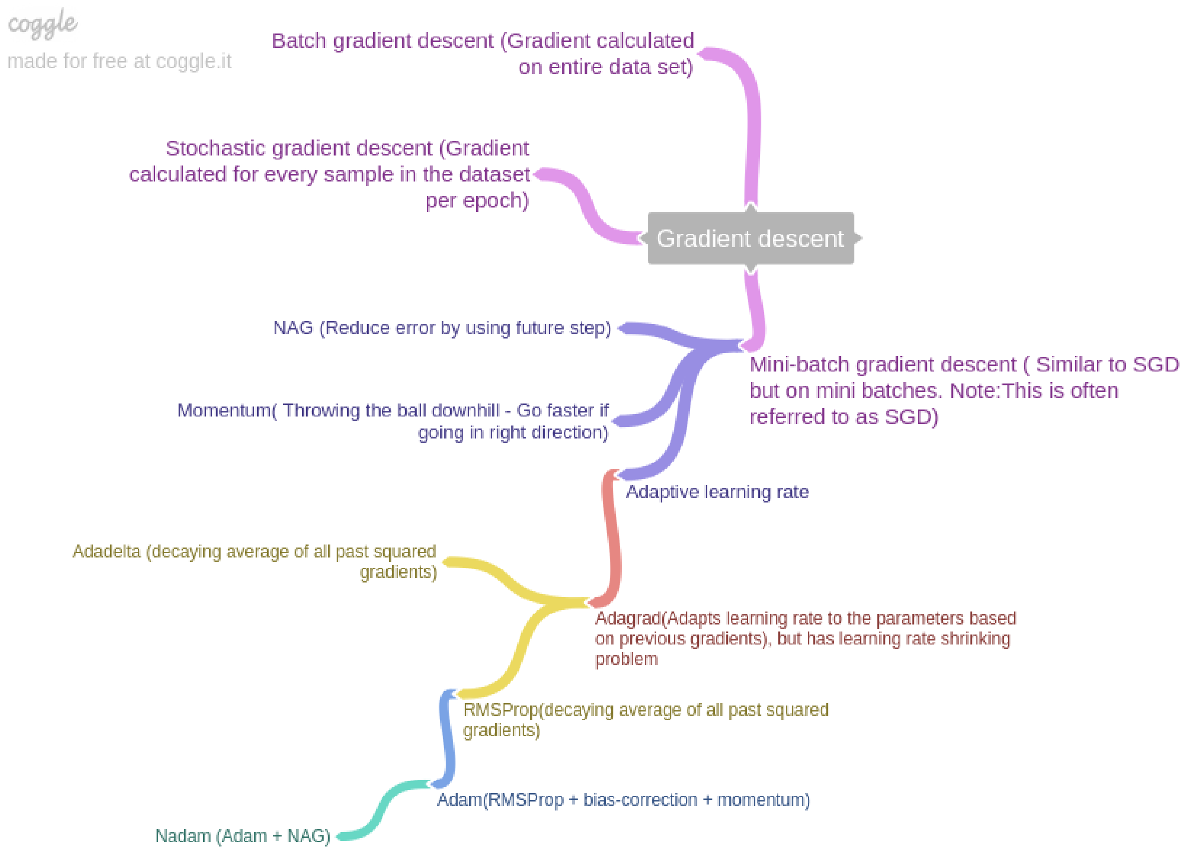
More on this later!



Optimizers (further reading)

UvA notes on optimizers

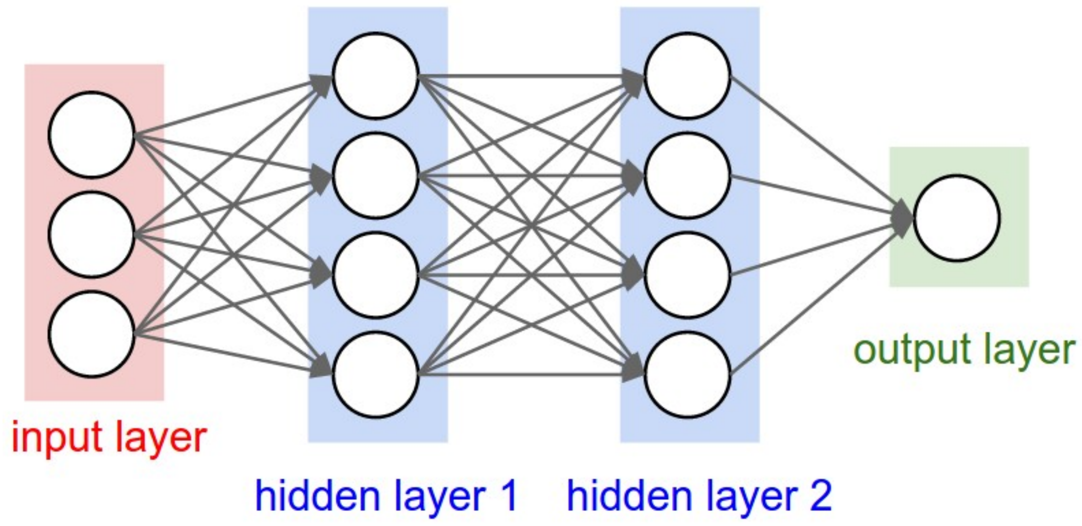
Weight updates


coggle
made for free at coggle.it
Batch gradient descent (Gradient calculated on entire data set)
Stochastic gradient descent (Gradient calculated for every sample in the dataset per epoch)
Gradient descent
NAG (Reduce error by using future step)
Momentum(Throwing the ball downhill - Go faster if going in right direction)
Mini-batch gradient descent (Similar to SGD but on mini batches. Note:This is often referred to as SGD)
Adaptive learning rate
Adadelta (decaying average of all past squared gradients)
Adagrad(Adapts learning rate to the parameters based on previous gradients), but has learning rate shrinking problem
RMSProp(decaying average of all past squared gradients)
Adam(RMSProp + bias-correction + momentum)
Nadam (Adam + NAG)

Weight initialization

There are a few contradictory requirements:

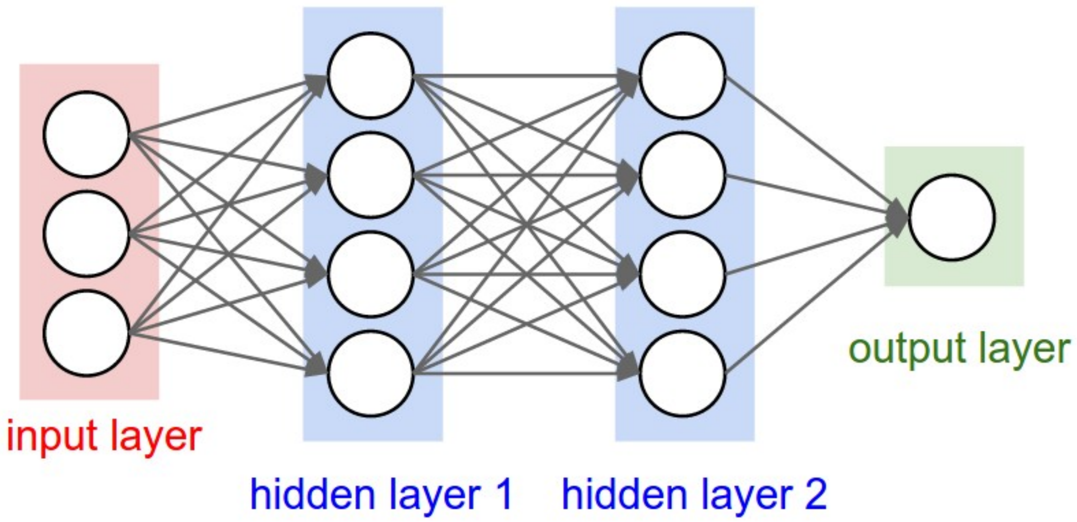
- Weights need to be small enough magnitude $\rightarrow$ Otherwise output values explode
- Weights need to be large enough magnitude $\rightarrow$ Otherwise signal too weak to propagate



Weight initialization

Naive approaches: All zero

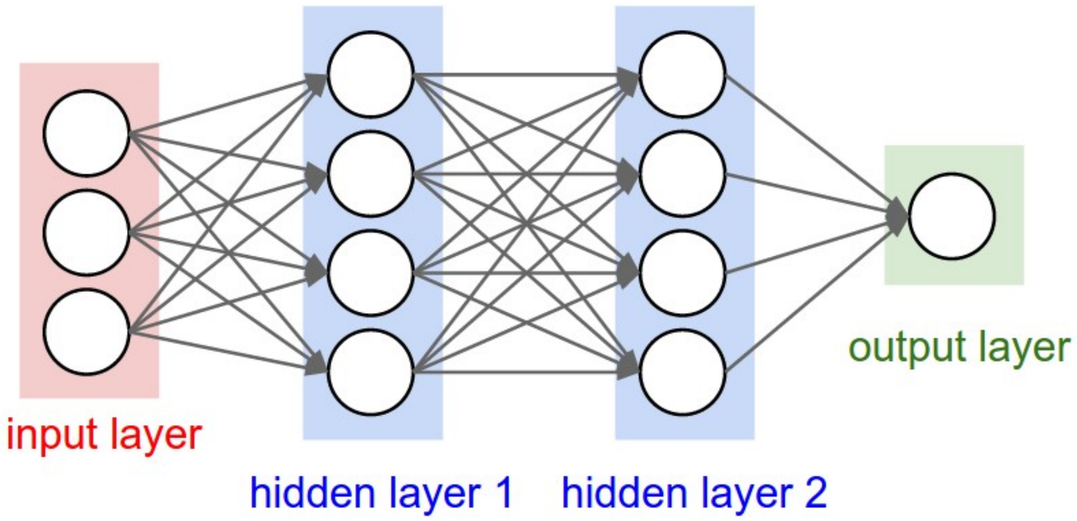
Every hidden unit will get zero signal. No matter what the input was, the output would be the same!



Weight initialization

Naive approaches: All constant (e.g. all 1.0)

- Input to each neuron in a layer will be the same,
- therefore the update each neuron in a layer receives will be the same,
- this will prevent different neurons in a layer from learning different things.



Weight initialiation

Solution: Break symmetry with a random initializaiton.

- Xavier or Glorot init: $w \sim \sqrt{\frac{2}{n_{in}+n_{out}}} \cdot N(0,1)$ (Glorot et al.)
- He init: $w \sim \sqrt{\frac{2}{n_{in}}} \cdot N(0,1)$ (He et al.)

where $n{in}𝑎𝑛𝑑n{out}$ represent the number of in and outgoing connections

Recap Hyperparameters

- **Gradient descent**: dependent on data, usually mini-batch
- **Error function**: dependent on the type of problem; influences final activation
- **Weight updates**: dependent on data, usually Adam is a good optimizer choice
- **Weight initialisation**: He or Xavier

Hyperparameters Questions

- Which is computationally more expensive: SGD, batch gradient descent or mini-batch gradient descent?

batch gradient descent

- Why is cross-entropy preferred over e.g. classification error (/accuracy)?

Accuracy is not a continuously differentiable function of the weights!

- Which optimizer combines both an adaptive learning rate with momentum?

- Why would you not initialize the weights to 0?

- In what cases would you use He/Xavier initialization over random initialization?

Summary

In this notebook we covered,

Architecture:

- Fully-connected layers
- Activation function
- Bias
- Batch normalization layer
- Dropout layer

Hyperparameters:

- loss function
- gradient descent type
- weight updates
- weight initialisation

Neural Networks in Keras

Let's put our knowledge to practice.

Keras basics