CSC311 Tutorial #5
Neural Networks

Fall 2019
Ehsan Mehralian*

University of Toronto

*Based on the lectures given by Professor Sanja Fidler, Andrew Ng and the prev. tutorials by Yujia Li and Boris Ivanovic.

Outline

- Neural Networks Intro.
- Backpropagation
- Momentom
- Preventing Overfitting
- Questions

Neural Networks

High-Level Overview

- A **Neural Network** is (generally) comprised of:
  - **Neurons** which pass input values through functions and output the result
  - **Weights** which carry values between neurons
- We group neurons into **layers**. There are 3 main types of layers:
  - **Input Layer**
  - **Hidden Layer(s)**
  - **Output Layer**

High-Level Overview

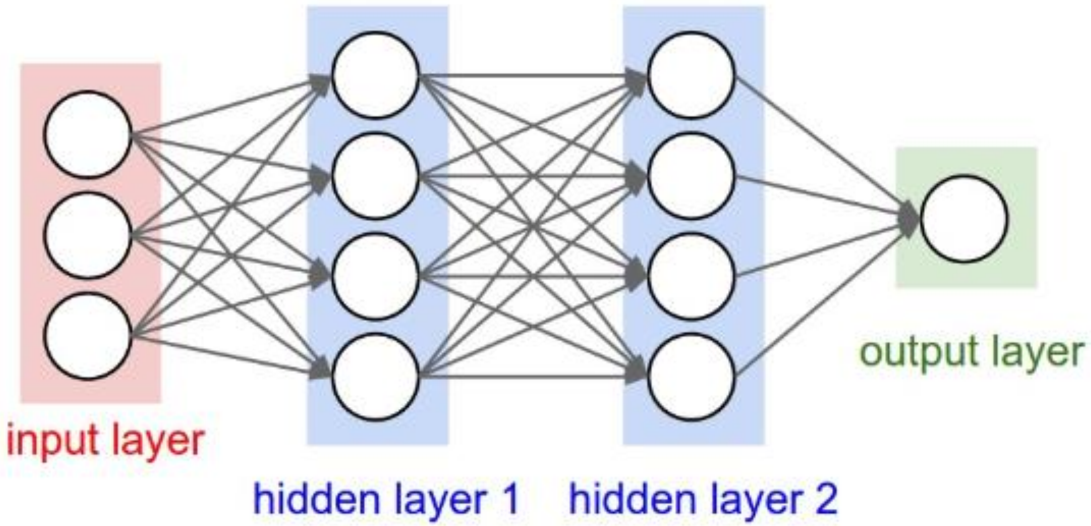


Figure: A 3-layer neural net with 3 input units, 4 hidden units in the first and second hidden layer and 1 output unit

- Naming conventions; a N-layer neural network:
  - $N-1$ layers of hidden units
  - One output layer

[http://cs231n.github.io/neural-networks-1/]

Neuron Breakdown

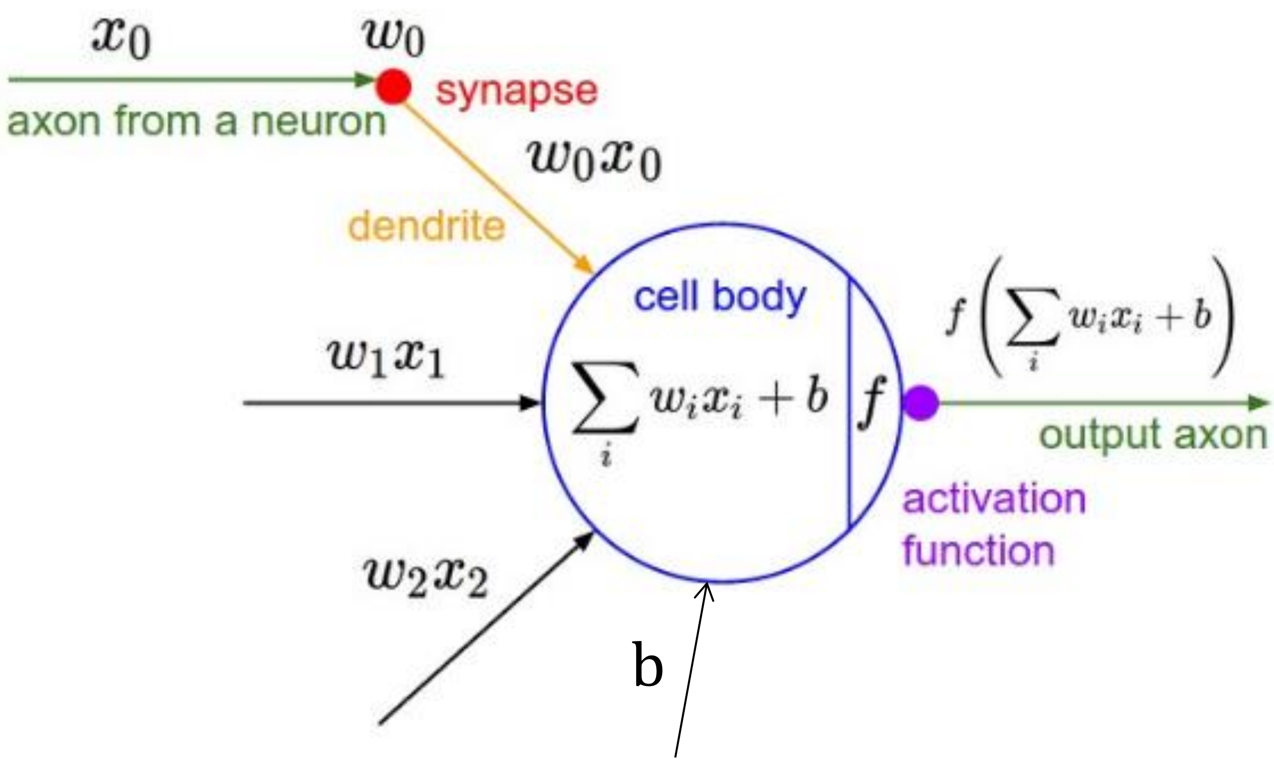


Figure: A mathematical model of the neuron in a neural network

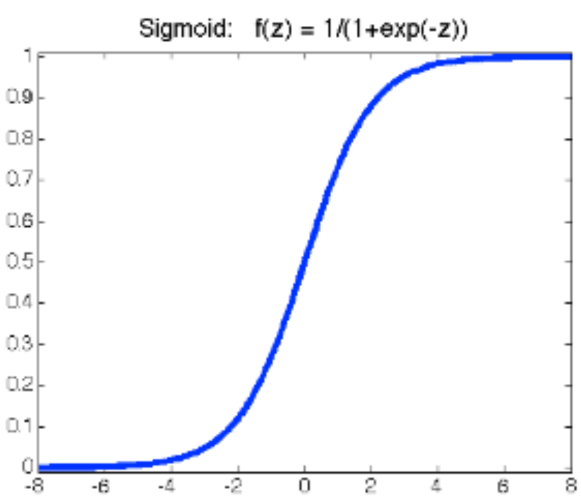
[Pic credit: http://cs231n.github.io/neural-networks-1/]

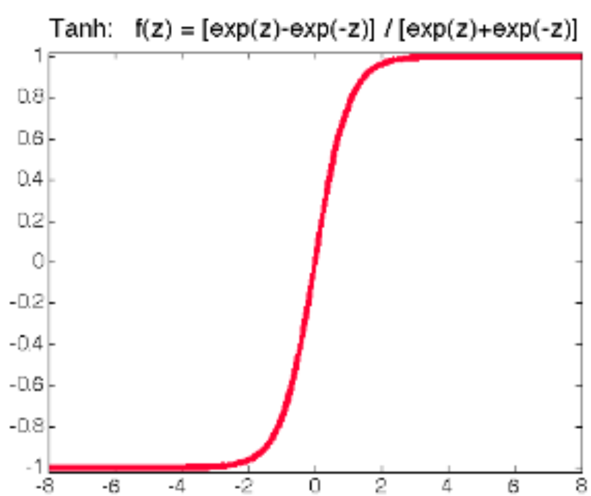
Activation Functions

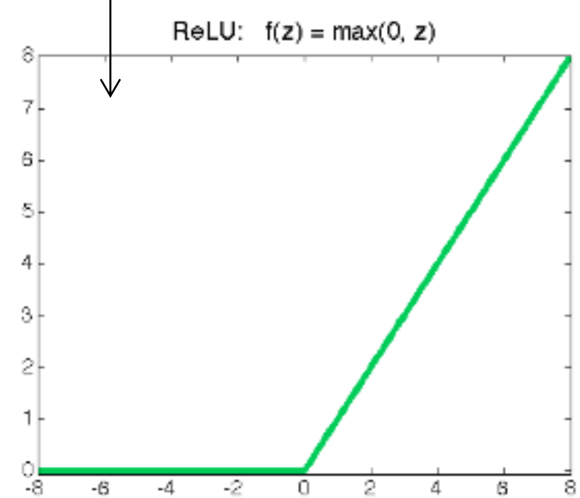
Most commonly used activation functions:

- Sigmoid: $\sigma(z) = \frac{1}{1+\exp(-z)}$
- Tanh: $\tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$
- ReLU (Rectified Linear Unit): $\mathrm{ReLU}(z) = \max(0, z)$

Most popular recently for deep learning





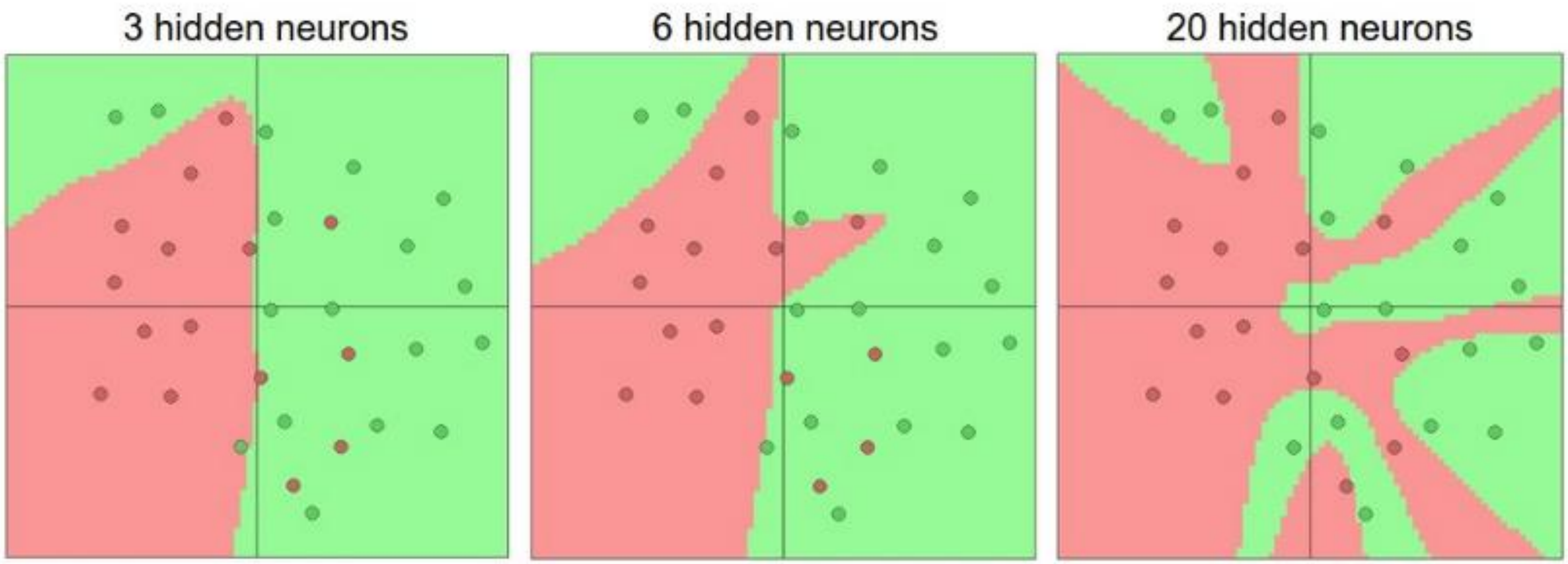


Representation Power

With nonlinear activation functions

- Neural network with at **least one hidden layer** is a universal approximator (can represent any function).

  Proof in: Approximation by Superpositions of Sigmoidal Function, Cybenko, paper



- The capacity of the network increases with more hidden units and more hidden layers

What does this mean?

- Neural Networks are **POWERFUL**, it's exactly why with recent computing power there was a renewed interest in them.

BUT

- *"With great power comes great overfitting."*
  - *Boris Ivanovic, 2016*

- Last slide, "20 hidden neurons" is an example.

How to mitigate this?

- **Stay Tuned**!

- First, how do we even use or train neural networks?

Training Neural Networks (Key Idea)

- Find weights:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{n=1}^{N} \operatorname{loss}(\mathbf{o}^{(n)}, \mathbf{t}^{(n)})$$

  where $\mathbf{o} = f(\mathbf{x}; \mathbf{w})$ is the output of a neural network

- Define a loss function, eg:
  - Squared loss: $\sum_k \frac{1}{2}(o_k^{(n)} - t_k^{(n)})^2$ **(Regression)**
  - Cross-entropy loss: $-\sum_k t_k^{(n)} \log o_k^{(n)}$ **(Classification)**

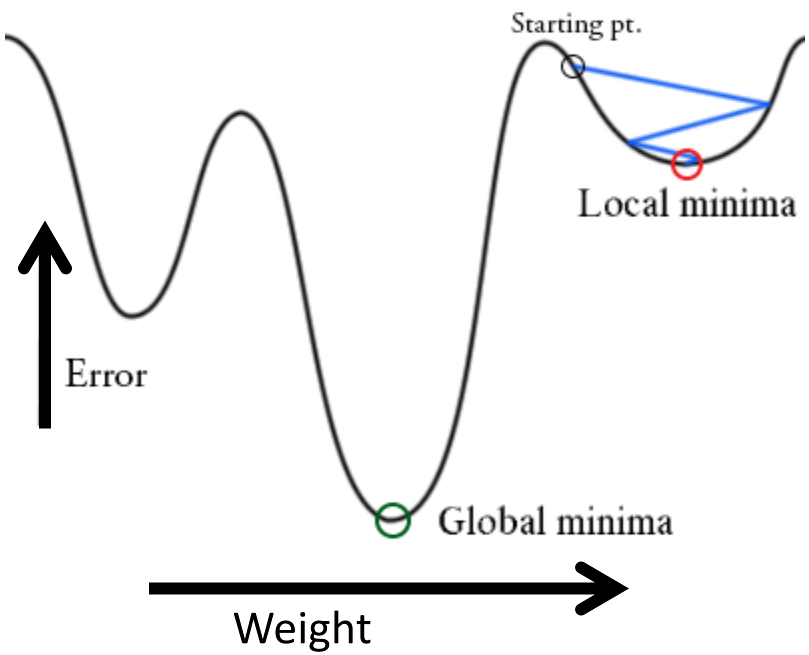
- Gradient descent:

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \frac{\partial E}{\partial \mathbf{w}^t}$$

  where η is the learning rate (and E is error/loss)

Training Compared to Other Models

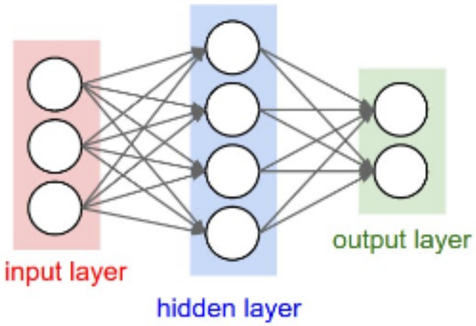
- Training Neural Networks is a **NON-CONVEX OPTIMIZATION PROBLEM**.
- This means we can run into many local optima during training.



Training Neural Networks (Implementation)

- We need to first perform a **forward pass**
- Then, we update weights with a **backward pass**

Forward Pass (AKA "Inference")



- Output of the network can be written as:

$$h_j(\mathbf{x}) = f(v_{j0} + \sum_{i=1}^{D} x_i v_{ji})$$

$$o_k(\mathbf{x}) = g(w_{k0} + \sum_{j=1}^{J} h_j(\mathbf{x}) w_{kj})$$

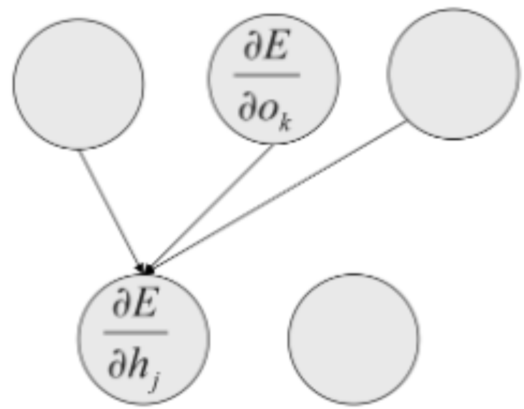
(j indexing hidden units, k indexing the output units, D number of inputs)

- Activation functions f, g: sigmoid/logistic, tanh, or rectified linear (ReLU)

$$\sigma(z) = \frac{1}{1 + \exp(-z)}, \quad \tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}, \quad \mathrm{ReLU}(z) = \max(0, z)$$

Backward Pass (AKA “Backprop.”)

- Compute error derivatives in each hidden layer from error derivatives in layer above. [assign blame for error at k to each unit j according to its influence on k (depends on w_{kj})]



- Use error derivatives w.r.t. activities to get error derivatives w.r.t. the weights.

Learning Weights during Backprop

- Do exactly what we've been doing!
- Take the derivative of the error/cost/loss function w.r.t. the weights and minimize via gradient descent!

Gradient descent:

$$\mathbf{w}^{t+1} = \mathbf{w}^{t} - \eta \frac{\partial E}{\partial \mathbf{w}^{t}}$$

where η is the learning rate (and E is error/loss)

Useful Derivatives

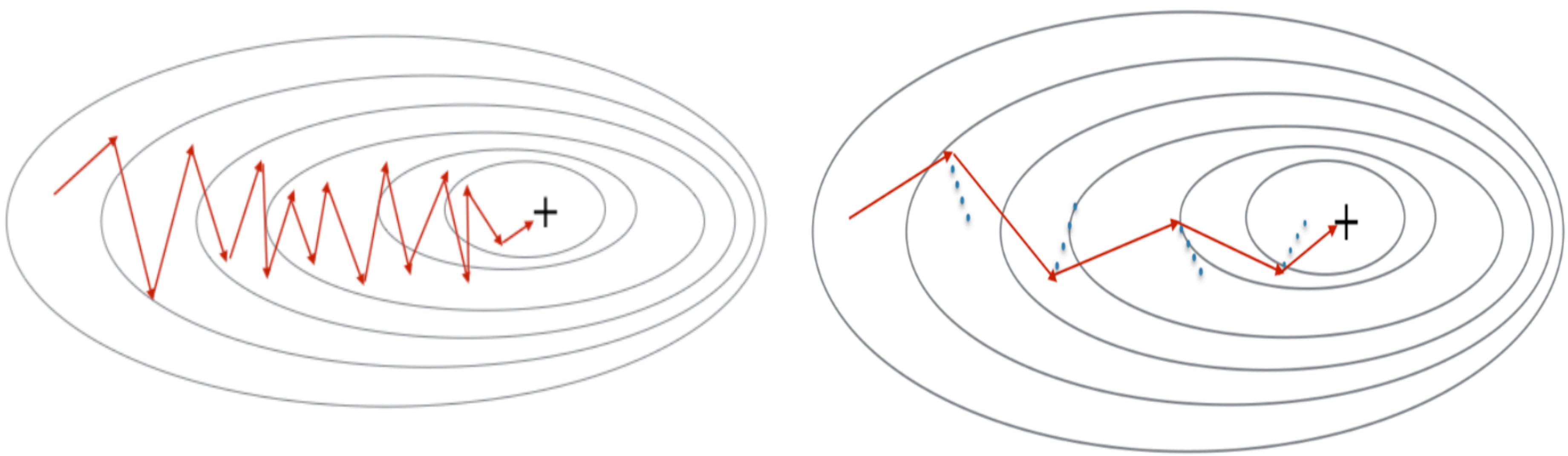
name	**function**	**derivative**
Sigmoid	$\sigma(z) = \frac{1}{1+\exp(-z)}$	$\sigma(z) \cdot (1 - \sigma(z))$
Tanh	$\tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$	$1/\cosh^2(z)$
ReLU	$\mathrm{ReLU}(z) = \max(0, z)$	$\begin{cases} 1, & \text{if } z > 0 \\ 0, & \text{if } z \leq 0 \end{cases}$

Gradient Descent With Momentum

Basic Idea

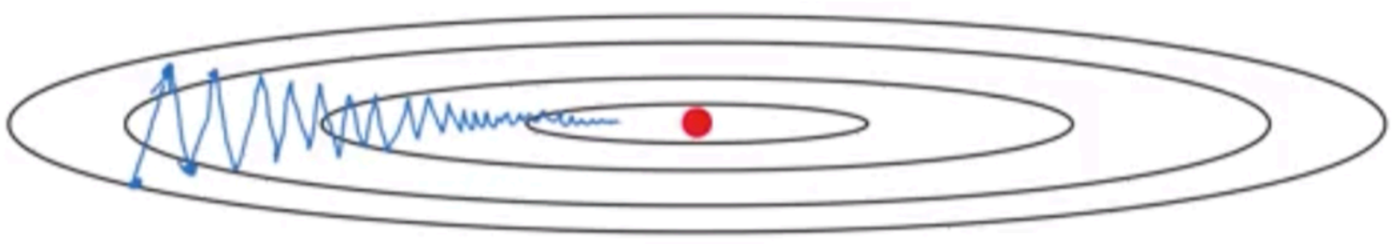
- Compute an exponentially weighted average of your gradients, and then use that gradient to update your weights

- Almost always works faster than the standard gradient descent



Gradient Decent

- Gradient descents takes a lot of steps. Slowly oscillate toward the minimum
- Oscillation slows down gradient descent and prevents you from using a much larger learning rate
- What if:
    - On the vertical axis a bit slower learning
    - On the horizontal axis a bit faster learning



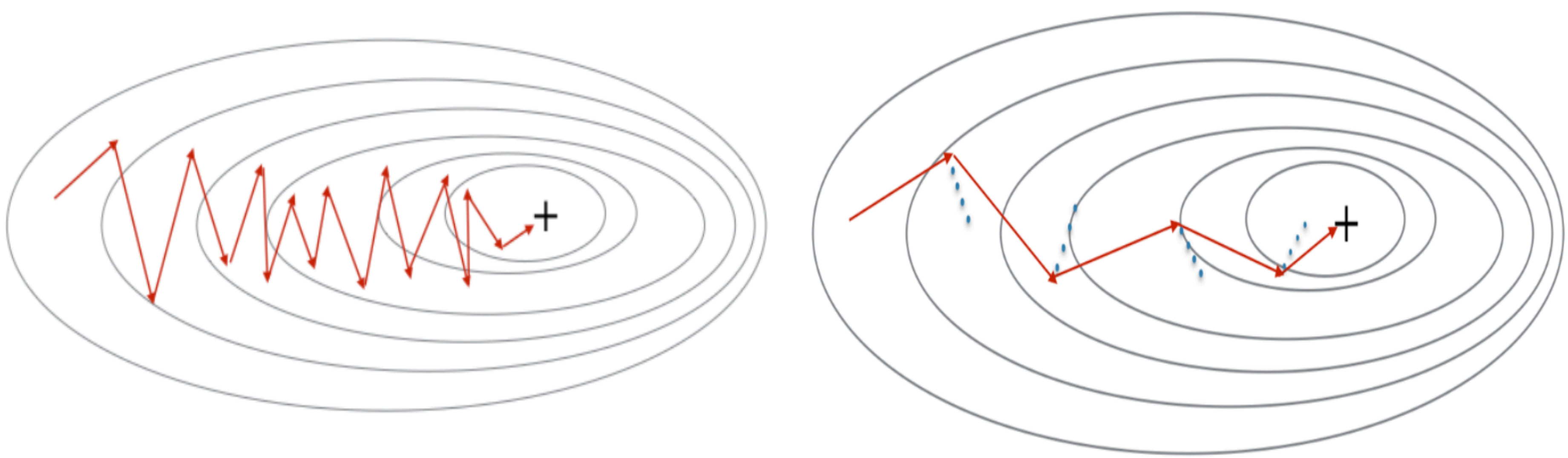
Implementation details

Momentum	Gradient Decent
On iteration t:	On iteration t:
compute $\frac{\partial E}{\partial w^t}$	compute $\frac{\partial E}{\partial w^t}$
$V_{w^t} = \beta V_{w^t} + (1-\beta)\frac{\partial E}{\partial w^t}$	$W^{t+1} = W^t - \eta\frac{\partial E}{\partial w^t}$
$W^{t+1} = W^t - \eta V_{W^t}$	

- Extra hyper parameter β, most common value for Beta is 0.9 (average last ten iteration's gradients)

Momentum

- Smooth out the steps of gradient descent:
    - Vertical direction: average out positive and negative numbers, so the average will be close to zero
    - Horizontal direction: all the derivatives are pointing to the right of the horizontal direction, the average will still be pretty big

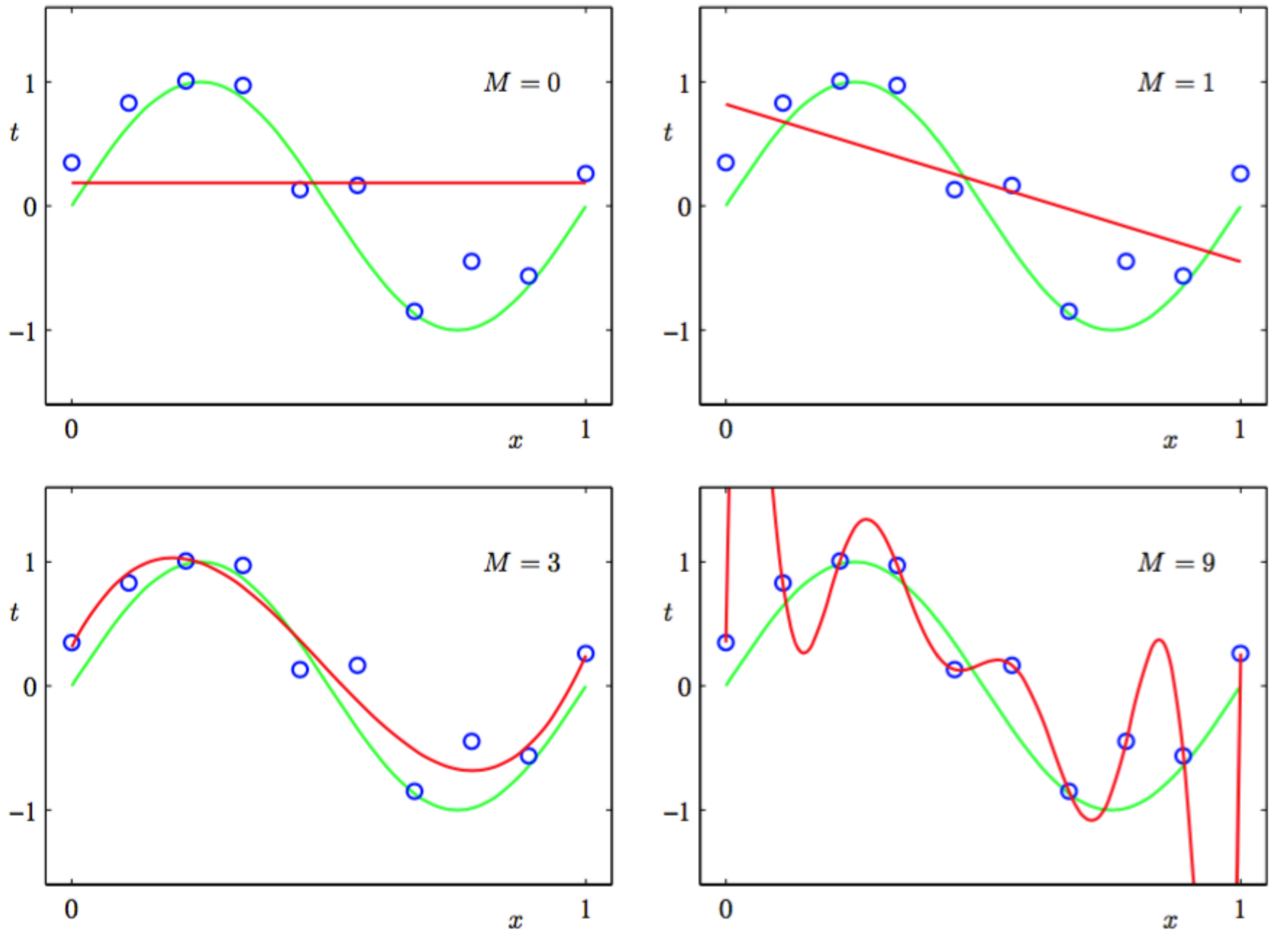


Overfitting

- The training data contains information about the regularities in the mapping from input to output. But it also contains noise
  - The target values may be unreliable.
  - There is sampling error. There will be accidental regularities just because of the particular training cases that were chosen

- When we fit the model, it cannot tell which regularities are real and which are caused by sampling error.
  - So it fits both kinds of regularity.
  - If the model is very flexible it can model the sampling error really well. This is a disaster.

Overfitting



Picture credit: Chris Bishop. Pattern Recognition and Machine Learning. Ch.1.1.

Preventing Overfitting

Standard ways to limit the capacity of a neural net:

- Limit the number of hidden units.
- Limit the size of the weights.
- Stop the learning before it has time to overfit.

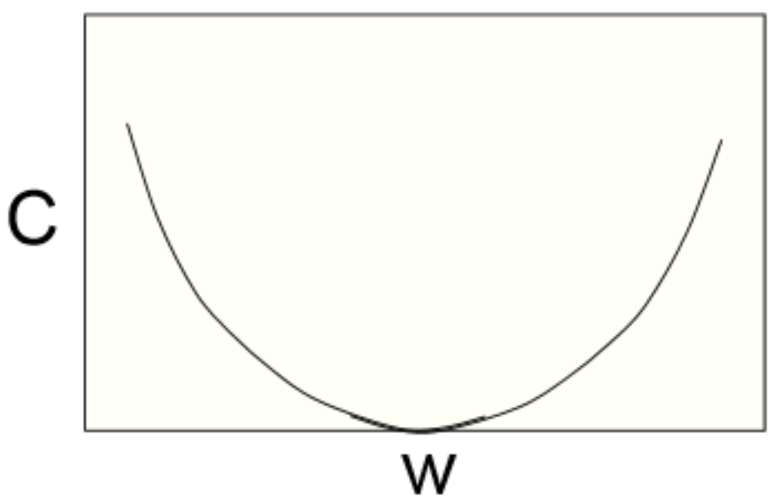
Limiting the Size of the Weights

Weight-decay involves adding an extra term to the cost function that penalizes the squared weights.

- Keeps weights small unless they have big error derivatives.

$$C = E + \frac{\lambda}{2}\sum_i w_i^2$$

$$\frac{\partial C}{\partial w_i} = \frac{\partial E}{\partial w_i} + \lambda w_i$$

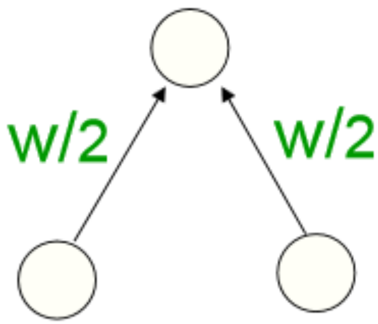


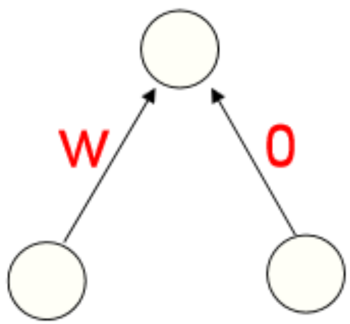
$$when \quad \frac{\partial C}{\partial w_i} = 0, \quad w_i = -\frac{1}{\lambda}\frac{\partial E}{\partial w_i}$$

The Effects of Weight-Decay

- It prevents the network from using weights that it does not need
  - This can often improve generalization a lot.
  - It helps to stop it from fitting the sampling error.
  - It makes a smoother model in which the output changes more slowly as the input changes.

- But, if the network has two very similar inputs it prefers to put half the weight on each rather than all the weight on one → other form of weight decay?





Early Stopping

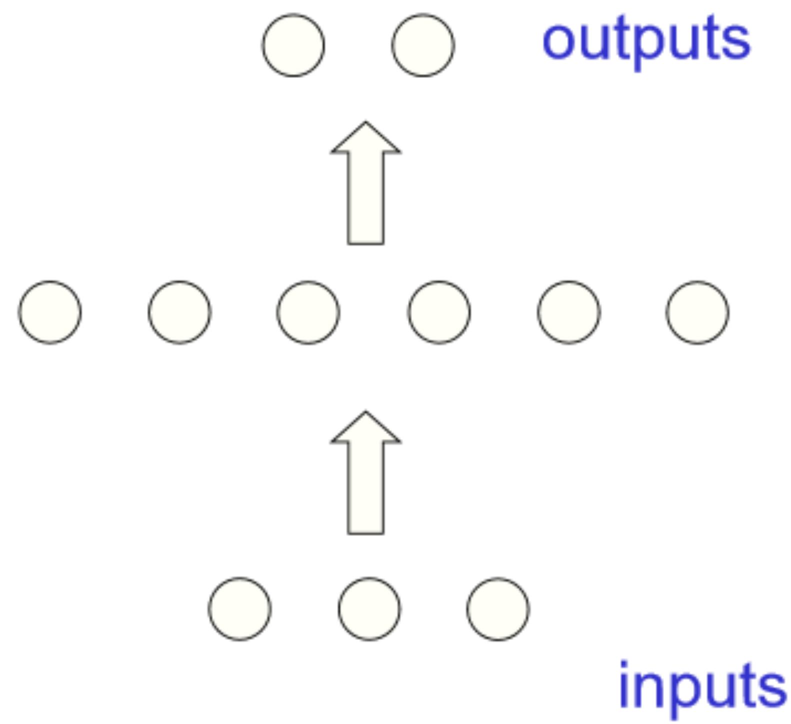
- If we have lots of data and a big model, its very expensive to keep re-training it with different amounts of weight decay

- It is much cheaper to start with very small weights and let them grow until the performance on the validation set starts getting worse

- The capacity of the model is limited because the weights have not had time to grow big.

Why Early Stopping Works

- When the weights are very small, every hidden unit is in its linear range.
  - So a net with a large layer of hidden units is linear.
  - It has no more capacity than a linear net in which the inputs are directly connected to the outputs!
- As the weights grow, the hidden units start using their non-linear ranges so the capacity grows.



Questions