Assignment Lecture 2

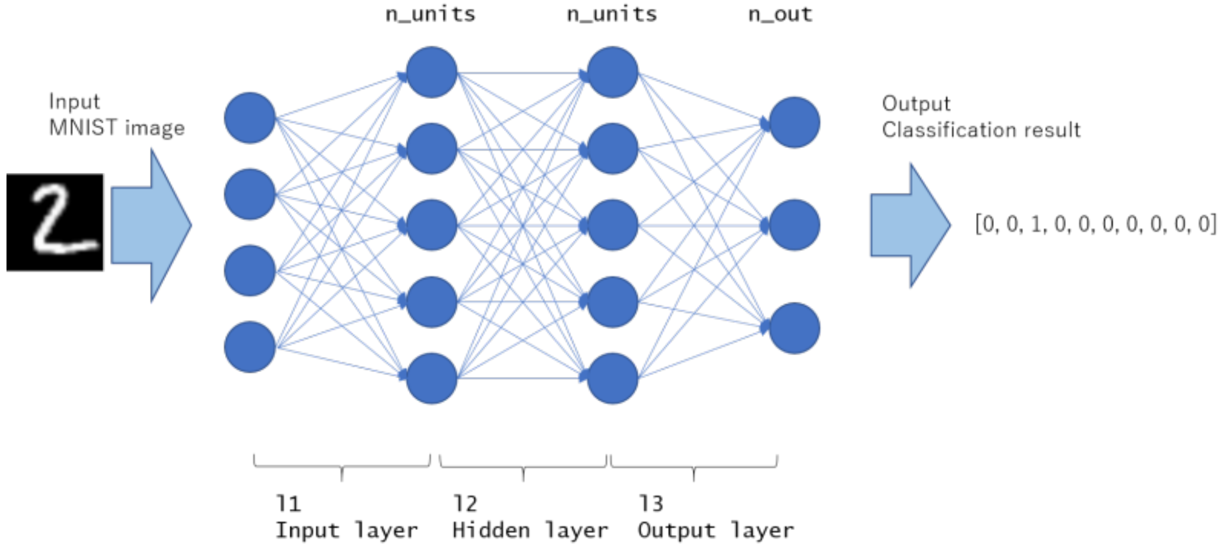
Multi-layer Neural Networks

Håkon Hukkelås | hakon.hukkelas@ntnu.no

Assignment 1 questions?

Assignment 2

- Multi-layer neural networks



Task 1a: Backpropagation (0.75 points)

By using the definition of δ_j, show that Equation 2 can be written as;

$$w_{ji} := w_{ji} - \alpha \delta_j x_i, \tag{3}$$

and show that $\delta_j = f'(z_j) \sum_k w_{kj} \delta_k$.

Hint 1: A good starting point is to try rewriting $\alpha \frac{\partial C}{\partial w_{ji}}$ using the chain rule.

Hint 2: From the previous assignment, we know that $\delta_k = \frac{\partial C}{\partial z_k} = -(y_k - \hat{y}_k)$.

Task 1b: Vectorize computation (0.25 points)

The computation is much faster when you update all w_{ji} and w_{kj} at the same time, using matrix multiplications rather than for-loops. Show the update rule for the weight matrix from the hidden layer to output layer and for the weight matrix from input layer to hidden layer, using matrix/vector notation.

We expect you to clearly define the shape of each vector/matrix in your calculation.

Hint: If you're stuck on this task, take a look in Chapter 2 in Nielsen's book.

[1]Note that we only count the layers with actual learnable parameters (hidden layer and output layer).

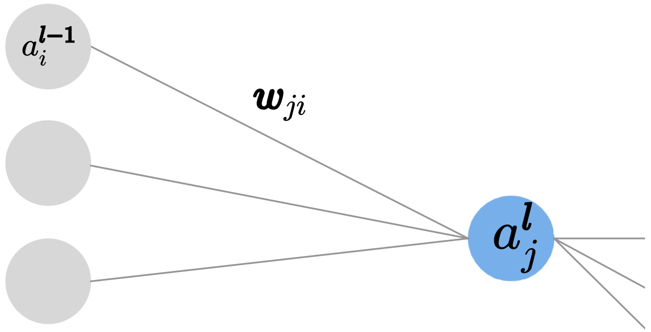
Assignment 2 - Task 1

- Chain rule, chain rule, chain rule… Again!

- Some good resources:
  - 3blue1brown videos:
    - https://www.youtube.com/watch?v=Ilg3gGewQ5U
    - https://www.youtube.com/watch?v=tIeHLnjs5U8&
  - Nielsen book, Chapter 2

Backpropagation - Delta Rule

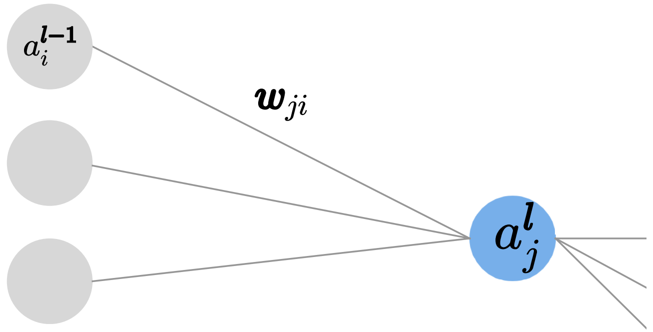
- Previous assignment (1-layer network);
  - $\frac{\partial C}{\partial w_{ji}^l} = x_i \frac{\partial C}{\partial z_j^l}$



- Notation:
  - δ_j^l : The error for node j in layer l
  - w_{ji}^l : weight from neuron i to neuron j
  - $z_i = w_i^T a_i^{l-1}$
  - a_i^{l-1} : activation for neuron i in layer $l-1$
  - C: Cost function / objective function

Backpropagation - Delta Rule

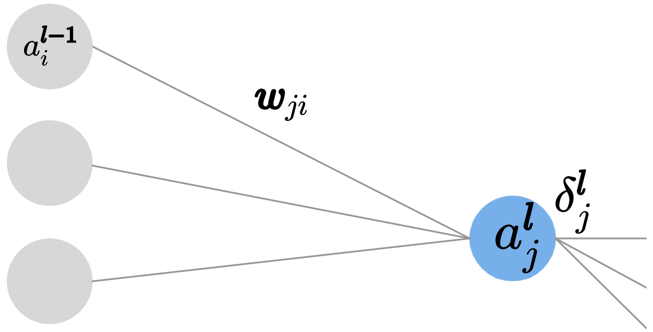
- Previous assignment (1-layer network);
  - $\frac{\partial C}{\partial w_{ji}^{l}} = x_i \frac{\partial C}{\partial z_j^{l}} = a_i^{l-1} \frac{\partial C}{\partial z_j^{l}}$, where $l = 1$



- Notation:
  - δ_j^l : The error for node j in layer l
  - w_{ji}^l : weight from neuron i to neuron j
  - $z_i = w_i^T a_i^{l-1}$
  - a_i^{l-1} : activation for neuron i in layer $l-1$
  - C: Cost function / objective function

Backpropagation - Delta Rule

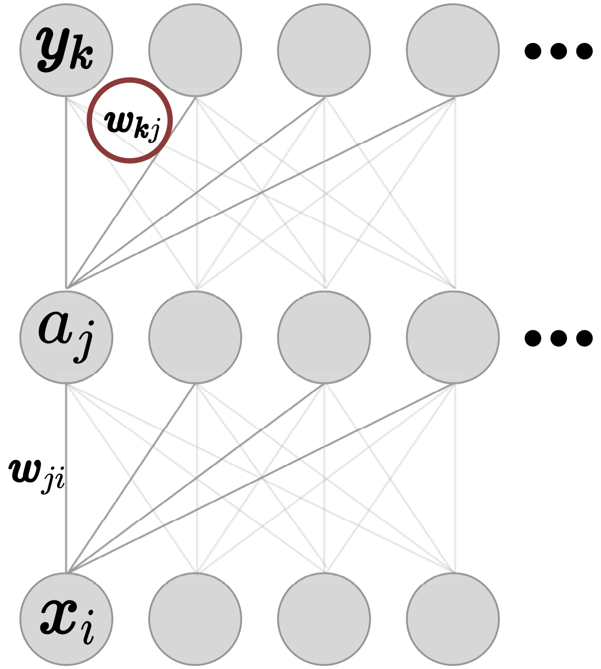
- Previous assignment (1-layer network);
  - $\frac{\partial C}{\partial w_{ji}^l} = x_i \frac{\partial C}{\partial z_j^l} = a_i^{l-1} \frac{\partial C}{\partial z_j^l}$, where $l = 1$
- Written with delta rule;
  - $\frac{\partial C}{\partial w_{ji}^l} = a_i^{l-1} \delta_j^l$, where $\delta_j^l = \frac{\partial C}{\partial z_j^l}$
- Notation:
  - δ_j^l : The error for node j in layer l
  - w_{ji}^l : weight from neuron i to neuron j
  - $z_i = w_i^T a_i^{l-1}$
  - a_i^{l-1} : activation for neuron i in layer $l-1$
  - C: Cost function / objective function



Gradient: Hidden -> Output Layer

- Use the same as assignment 1.

$$\frac{dC(w)}{dw_{kj}} = -a_j(y_k - \hat{y}_k)$$



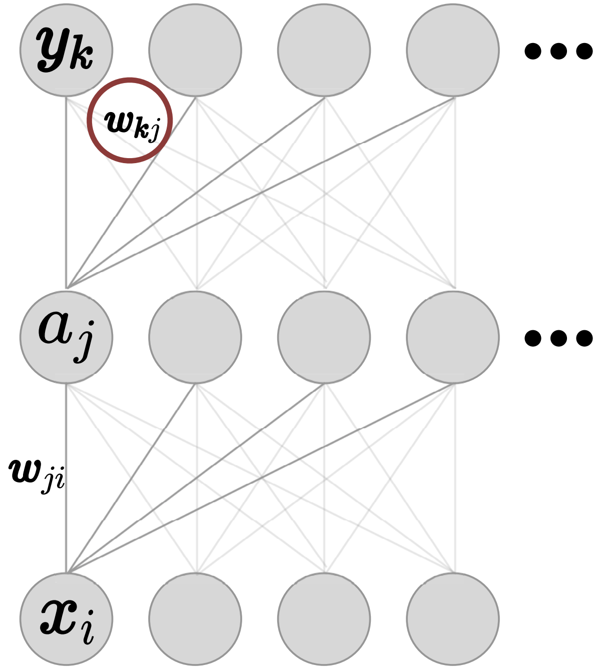
Gradient: Hidden -> Output Layer

- Use the same as assignment 1.

$$\frac{dC(w)}{dw_{kj}} = -a_j(y_k - \hat{y}_k)$$

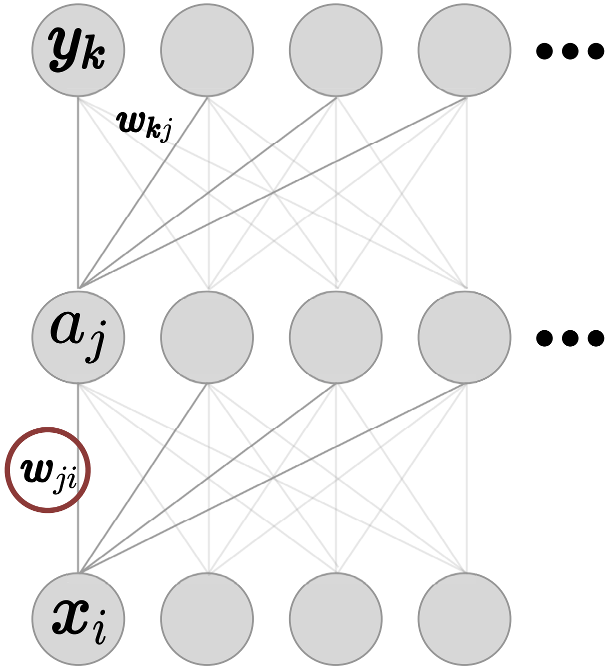
- Or with the delta rule:

$$\frac{dC(w)}{dw_{kj}} = a_j\delta_k, \quad \text{where } \delta_k = -(y_k - \hat{y}_k)$$



Gradient: Input -> Hidden

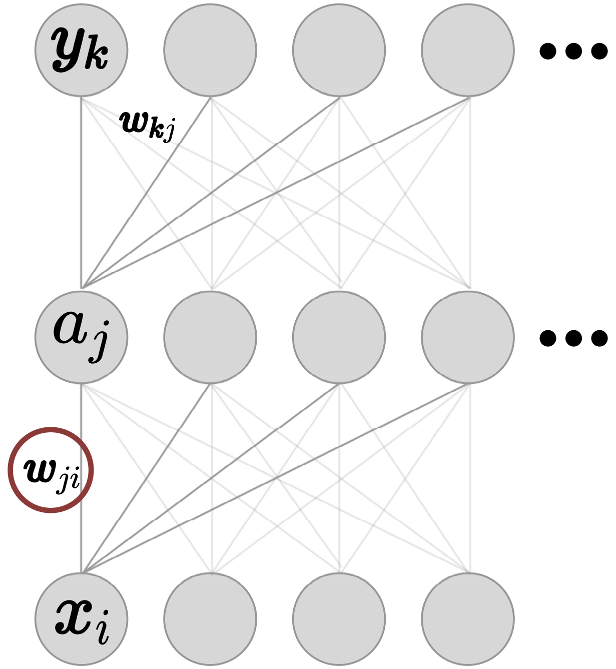
- Know that $\frac{\partial C}{\partial w_{ji}} = \delta_j x_i$



Gradient: Input -> Hidden

- Know that $\frac{\partial C}{\partial w_{ji}} = \delta_j x_i$

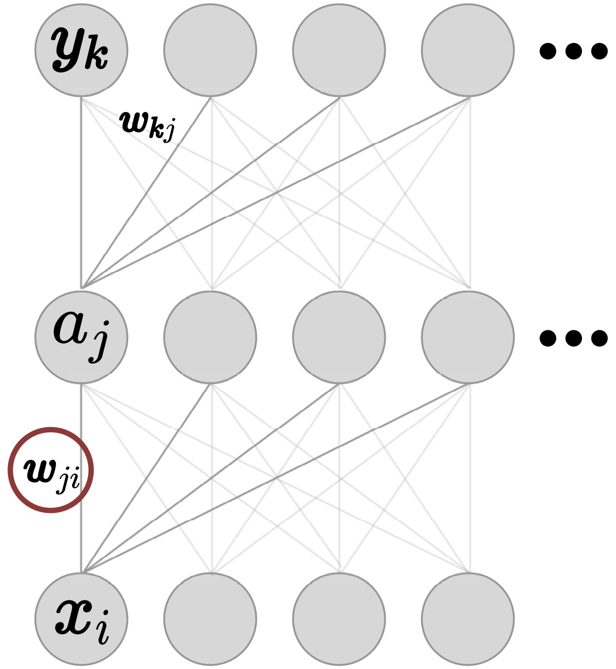
- $\delta_j = \frac{\partial C}{\partial z_j} =$



Gradient: Input -> Hidden

- Know that $\frac{\partial C}{\partial w_{ji}} = \delta_j x_i$

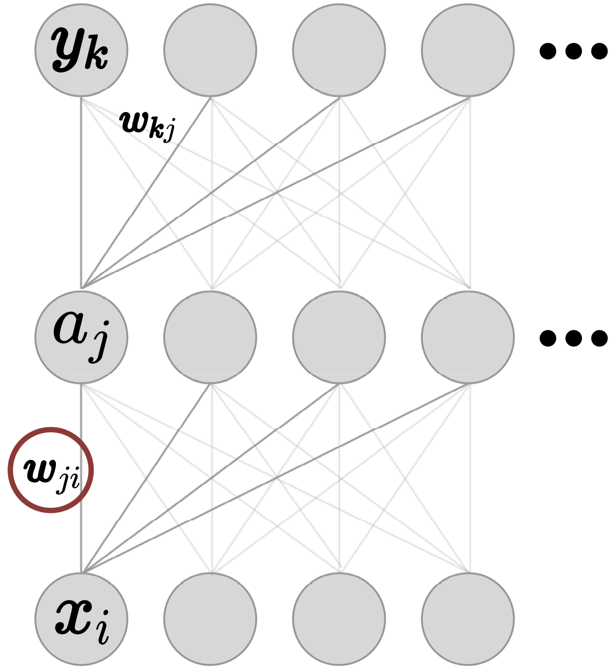
- $\delta_j = \frac{\partial C}{\partial z_j} = \quad \frac{\partial C}{\partial z_k}\frac{\partial z_k}{\partial a_j}\frac{\partial a_j}{\partial z_j}$



Gradient: Input -> Hidden

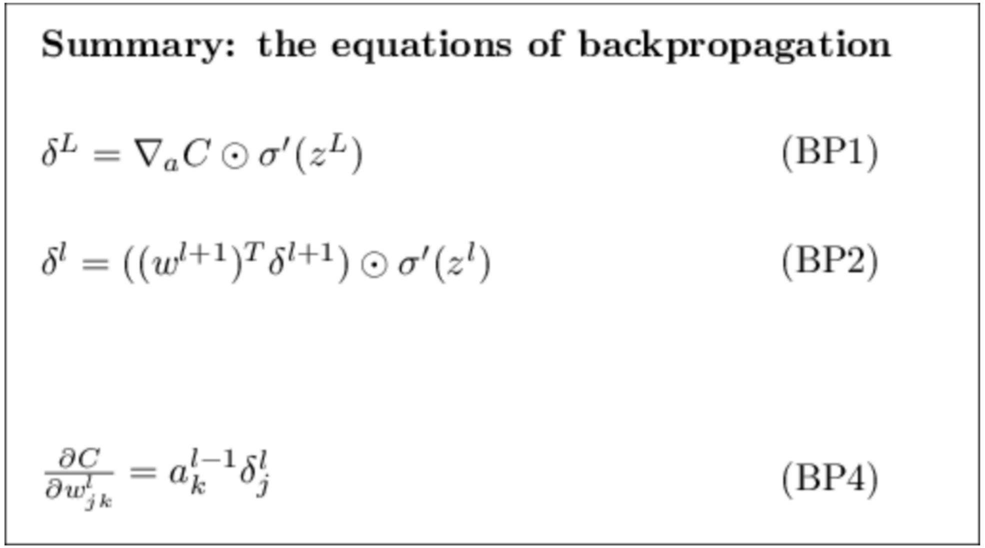
- Know that $\frac{\partial C}{\partial w_{ji}} = \delta_j x_i$

- $\delta_j = \frac{\partial C}{\partial z_j} = \sum_k \frac{\partial C}{\partial z_k} \frac{\partial z_k}{\partial a_j} \frac{\partial a_j}{\partial z_j}$



Task 1.2 - Vectorizing the delta-rules

- Hint: Solution given in Nielsen book, Chapter 2



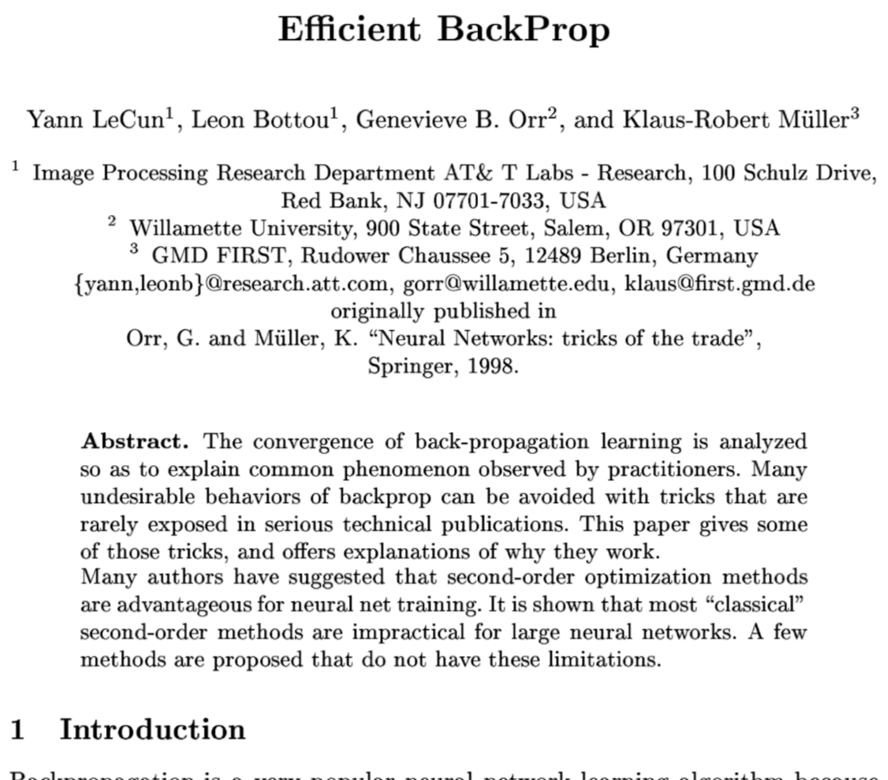
Summary: the equations of backpropagation

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \qquad \text{(BP1)}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \qquad \text{(BP2)}$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \qquad \text{(BP4)}$$

Task 3: Tricks of the trade



Efficient BackProp

Yann LeCun[1], Leon Bottou[1], Genevieve B. Orr[2], and Klaus-Robert Müller[3]

[1] Image Processing Research Department AT& T Labs - Research, 100 Schulz Drive, Red Bank, NJ 07701-7033, USA
[2] Willamette University, 900 State Street, Salem, OR 97301, USA
[3] GMD FIRST, Rudower Chaussee 5, 12489 Berlin, Germany
{yann,leonb}@research.att.com, gorr@willamette.edu, klaus@first.gmd.de

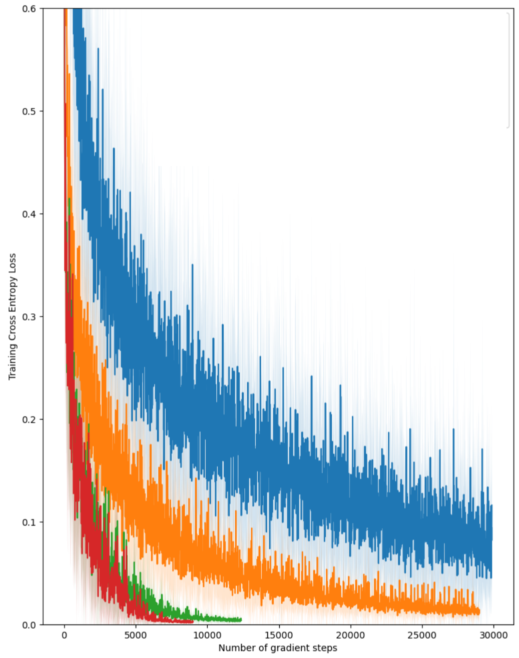
originally published in
Orr, G. and Müller, K. "Neural Networks: tricks of the trade",
Springer, 1998.

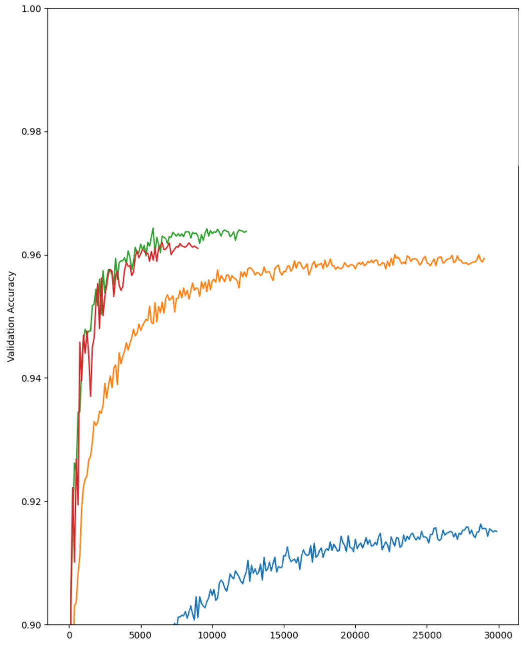
Abstract. The convergence of back-propagation learning is analyzed so as to explain common phenomenon observed by practitioners. Many undesirable behaviors of backprop can be avoided with tricks that are rarely exposed in serious technical publications. This paper gives some of those tricks, and offers explanations of why they work.
Many authors have suggested that second-order optimization methods are advantageous for neural net training. It is shown that most "classical" second-order methods are impractical for large neural networks. A few methods are proposed that do not have these limitations.

1 Introduction

Backpropagation is a very popular neural network learning algorithm because

Task 3: Tricks of the trade





Trick of the trade #1: Data shuffling

- Networks learns the fastest from the most unexpected sample
- Easy: Choose the sample that is the most unfamiliar to the network

Trick of the trade #1: Data shuffling

- Networks learns the fastest from the most unexpected sample
- Easy: Choose the sample that is the most unfamiliar to the network

- This is not trivial… Instead, shuffle training set after each epoch!

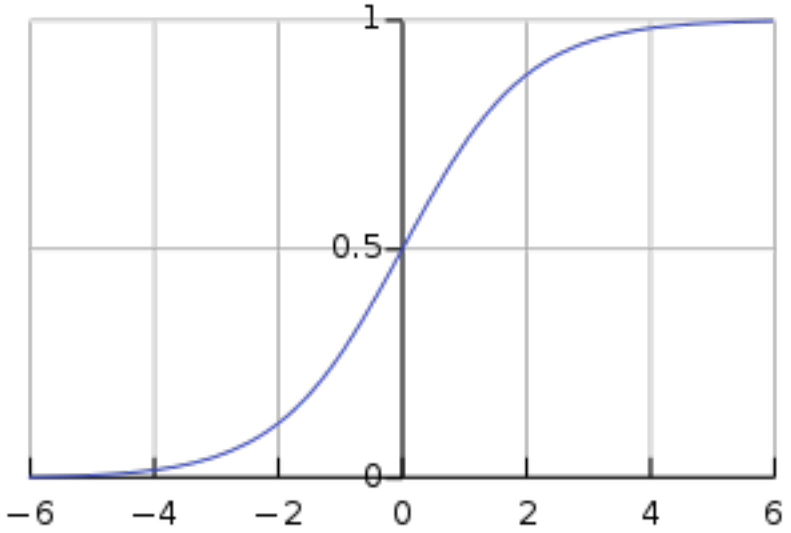
Normalized inputs converges faster

If all inputs are positive, the update of the weights will be of the same sign ($sign(\delta)$).

This will

Trick of the trade #2: Improved Sigmoid

- Normalizing inputs => Faster convergence
- Neurons converge faster if the mean input is zero-centered (Details in section 4.3 of paper)



- Sigmoid is not zero-centered

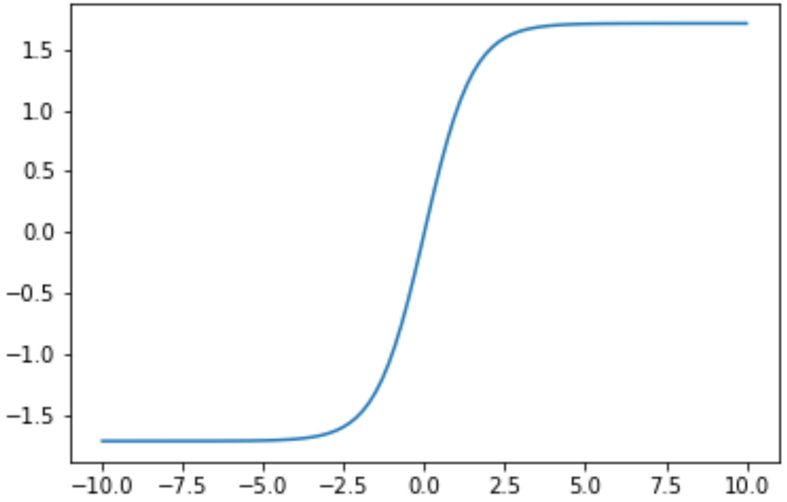
Trick of the trade #2: Improved Sigmoid

- Tanh is zero-centered
- Use:

$$f(x) = 1.7159\tanh(\frac{2}{3}x)$$

- NB: Need to derive gradient again!

$$f'(x) = 1.7159\frac{2}{3\cosh^2(\frac{2x}{3})}$$



Trick of the trade #3: Weight Initialization

- Weights too small => small gradients => slow training
- Weights too large => sigmoid saturated => small gradients => slow training

$$z_j = \sum w_{ji} x_i$$

Trick of the trade #3: Weight Initialization

- Weights too small => small gradients => slow training
- Weights too large => sigmoid saturated => small gradients => slow training

$$z_j = \sum w_{ji} x_i$$

- Initialize with a mean of 0 and standard deviation:

$$\sigma = \frac{1}{\sqrt{\text{fan-in}}}$$

Trick of the trade #3: Weight Initialization

- Weights too small => small gradients => slow training
- Weights too large => sigmoid saturated => small gradients => slow training

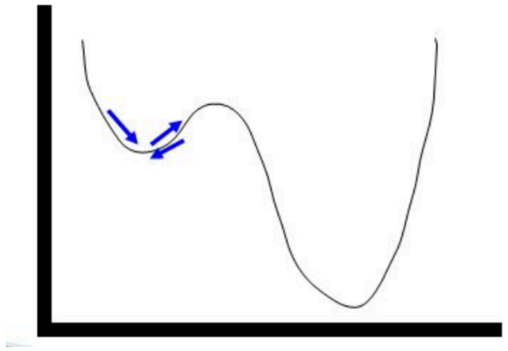
$$z_j = \sum w_{ji} x_i$$

- Initialize with a mean of 0 and standard deviation:

$$\sigma = \frac{1}{\sqrt{\text{fan-in}}}$$ fan in=number of input units

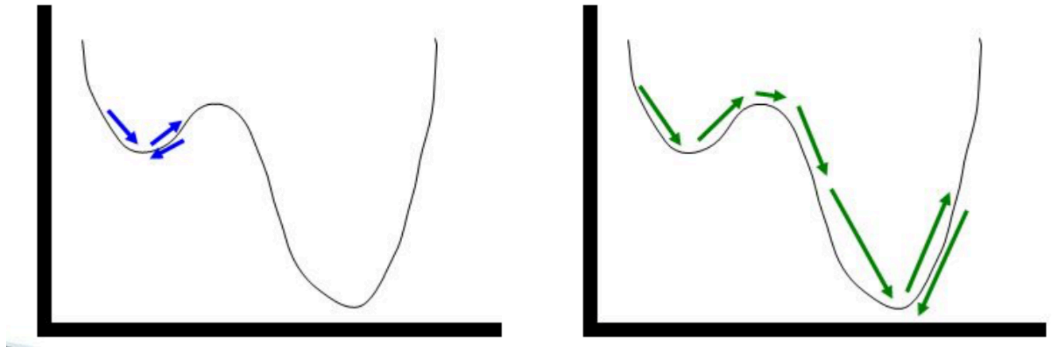
Trick of the trade #4: Momentum

- Keep “speed” from previous gradient descent step



Trick of the trade #4: Momentum

- Keep “speed” from previous gradient descent step



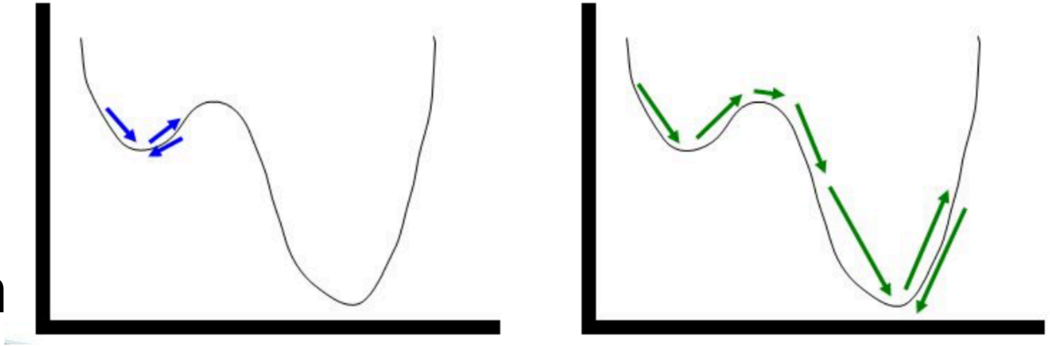
Trick of the trade #4: Momentum

- Keep “speed” from previous gradient descent step

$$\Delta w(t+1) = \frac{\partial C}{\partial w} + \mu \Delta w(t)$$

$$w := w - \alpha \Delta w(t)$$

- Faster convergence
- Can dodge local minim



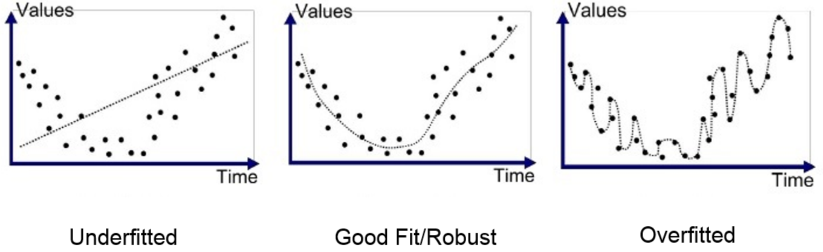
Task 4: Network topology

- How does network architecture change the performance?

Task 4: Network topology

- How does network architecture change the performance.
- Change number of hidden units

- More complex model => more overfitting?
- Less complex model => underfitting?

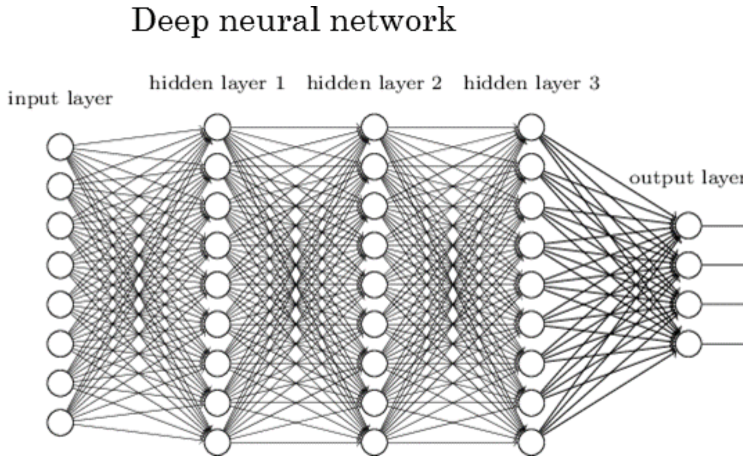


Side-note: What is network complexity?

- It is **NOT** the computational complexity

- More layers -> More non-linear functions (activation functions) -> Higher complexity
- More neurons -> Higher complexity

- Measured in #parameters
- #parameters = #weights + #bias



Questions?