

Lecture 13: Deep Learning

Andreas Wichert

Department of Computer Science and Engineering

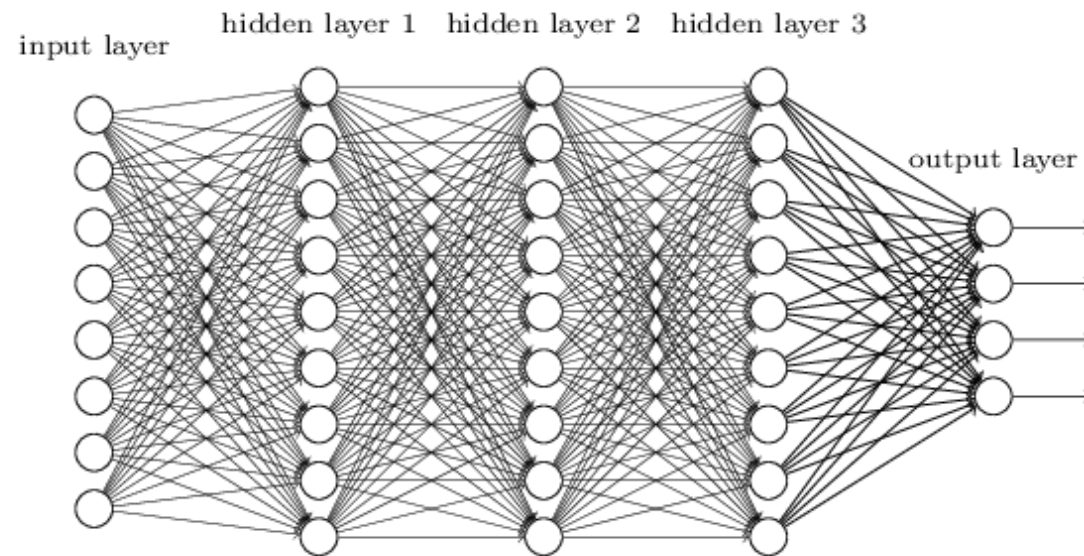
Técnico Lisboa

Deep Learning and Backpropagation

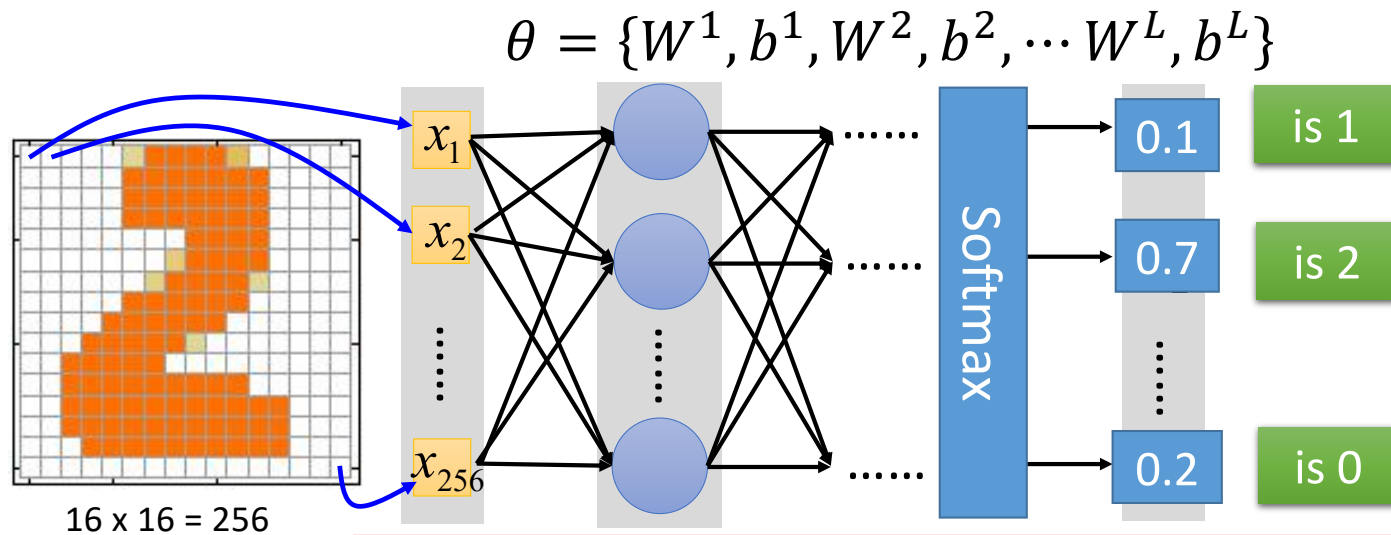
- According to the universality theorem, a neural network with a single hidden layer is capable of approximating any continuous function
- However, attempting to build a network with only one layer to approximate complex functions often requires a very large number of nodes

- Why does deep learning not have a local minimum?
- Is this true?


- It is assumed that an artificial neural network with several hidden layers is less likely to be stuck in a local minima and it is easier to find the right parameters as demonstrated by empirical experiments




How to set network parameters



Set the network parameters θ such that

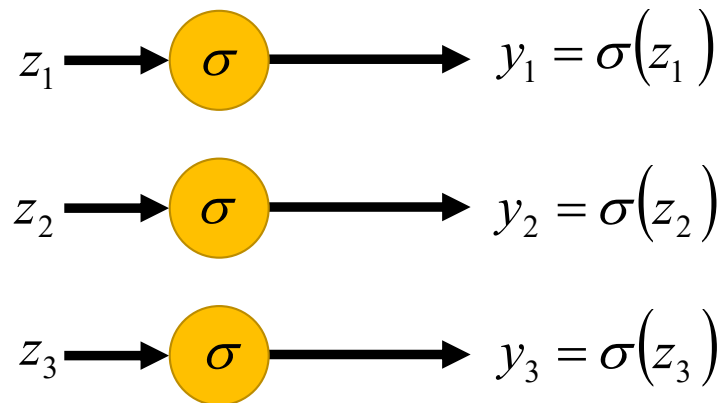
Input:  \rightarrow y_1 has the maximum value

Input:  \rightarrow y_2 has the maximum value

Softmax

- Softmax layer as the output layer

Ordinary Layer

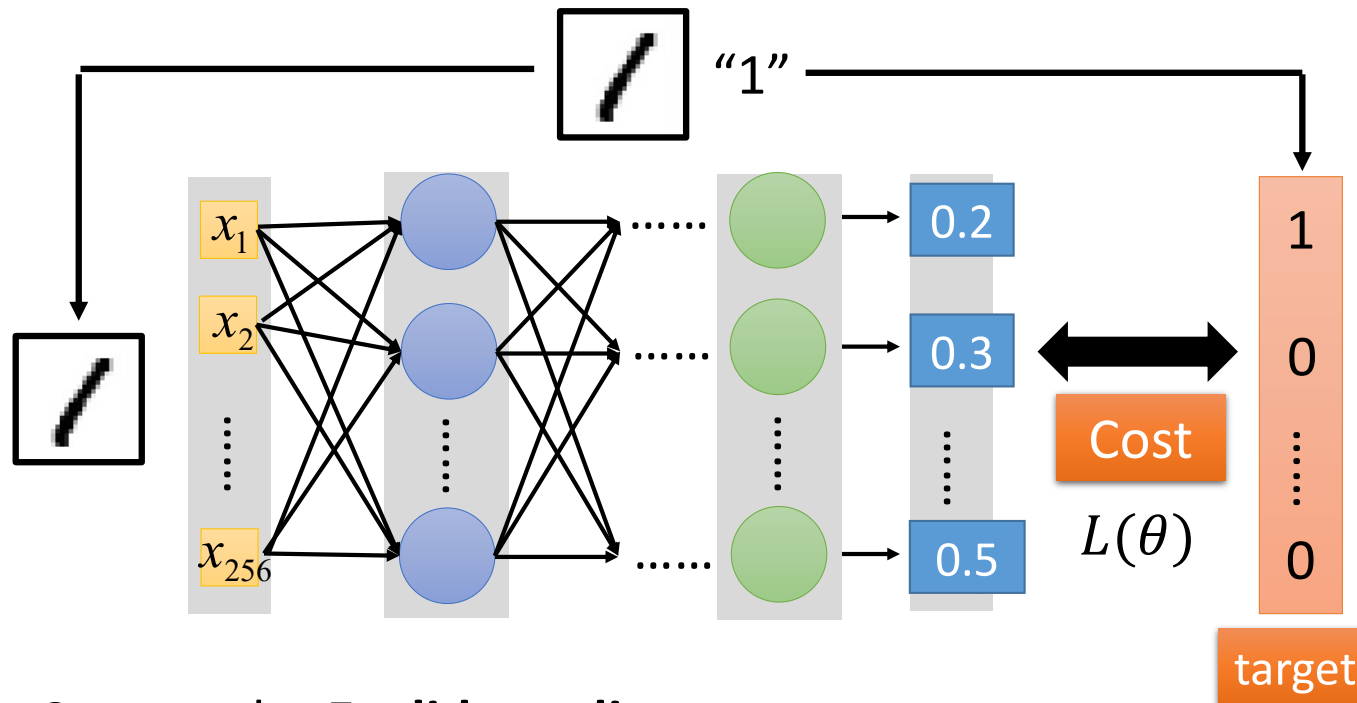


In general, the output of network can be any value.

May not be easy to interpret

Cost

Given a set of network parameters θ ,
each example has a cost value.



Cost can be **Euclidean distance** or **cross entropy** of the network output and target

Mini-Batch

- A gradient is usually determined over the whole training data set.
- This is called the batch gradient descent
- The model updates parameters after processing the whole training data (one epoch)
- In deep learning the training data set can be too big to fit the computer memory and the gradient cannot be computed efficiently

Mini-Batch

- In stochastic gradient descent one updates model parameters after processing every instance, however the model updates are noisy and for big training data sets not computationally efficient
- Therefore, mini batch gradient descent is introduced as a trade-off, learning is preformed in small groups.
 - For example, if the training data has 50000 instances, and the size of a mini batch is set to 50, then there will be 1000 mini batches.
 - They are as well called mini batch stochastic methods or stochastic methods

Cross Entropy

$$H(p, q) = - \sum_{x \in X} p(x) \log q(x) = -\mathbb{E}_{x \sim p} \log q(x)$$

- It is not a distance, because it is not symmetric
- When computing many of these quantities, it is common to encounter ex-pressions of the form $0 \cdot \log 0$
- By convention, in the context of information theory, we treat these expressions as $x \cdot \log x = 0$.

$$L(\mathbf{x}, \mathbf{y}, \theta) = - \sum_{k=1}^N (y_k \log p(c_k | \mathbf{x}))$$

$$J(\theta) = - \sum_{k=1}^N (y_k \log p(c_k | \mathbf{x})) = - \mathbb{E}_{x, y \sim p_{data}} \log p(c_k | \mathbf{x})$$

Learning consists of minimizing the loss function, we rewrite it for convention with $\mathbf{w} = \theta$ as

$$E(\mathbf{w}) = - \sum_{k=1}^N y_k \log o_k$$

in which $y_{kt} \in \{0, 1\}$ and o_k are probabilities.

Cross-entropy vs. Quadratic loss

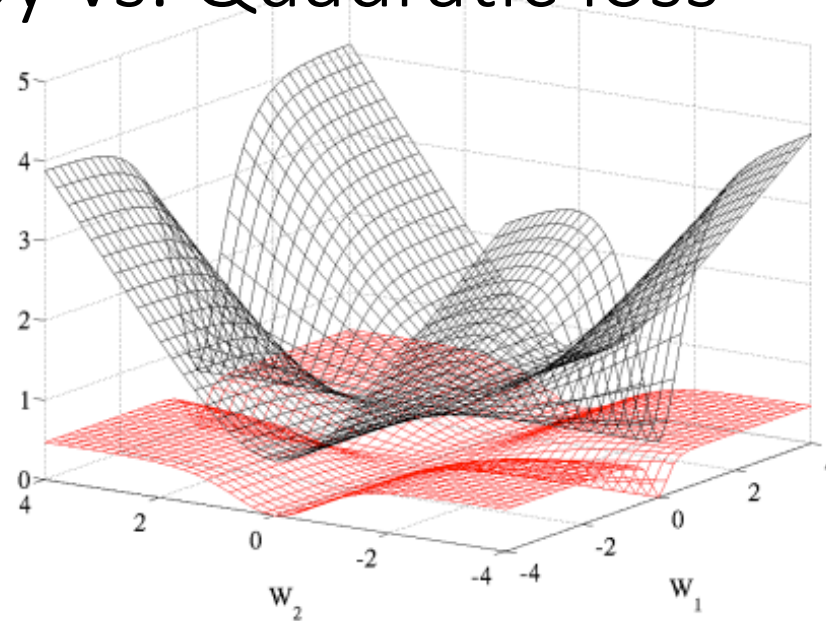


Figure 5: *Cross entropy (black, surface on top) and quadratic (red, bottom surface) cost as a function of two weights (one at each layer) of a network with two layers, W_1 respectively on the first layer and W_2 on the second, output layer.*

Figure from Glorot & Benthio (2010)

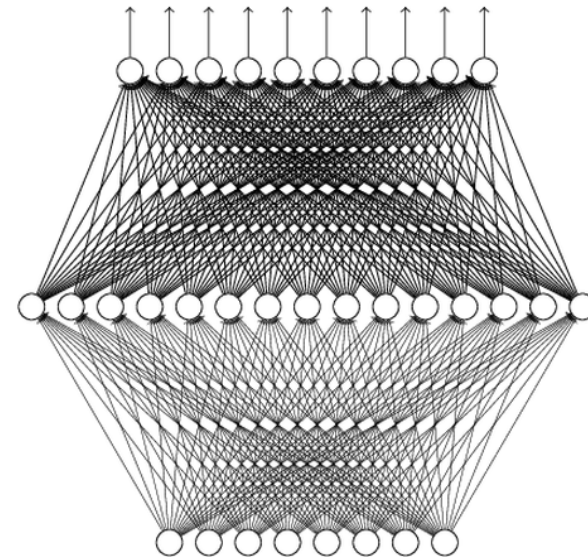
Universality Theorem

Any continuous function f

$$f : \mathbb{R}^N \rightarrow \mathbb{R}^M$$

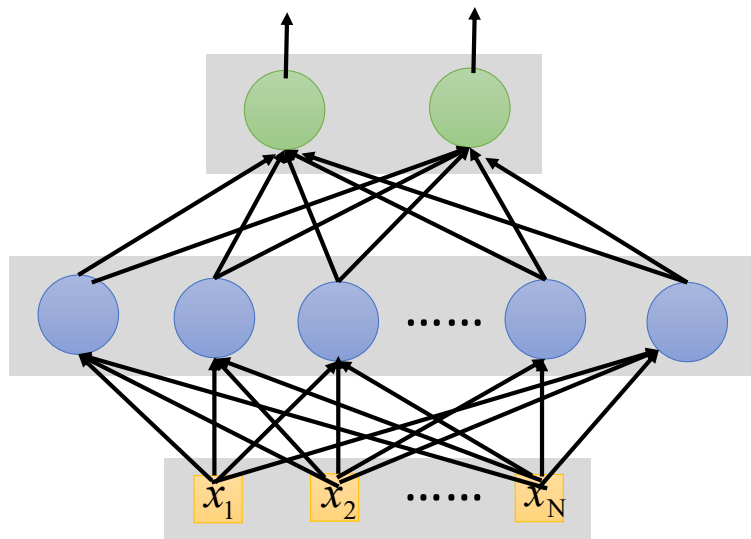
Can be realized by a network
with one hidden layer

(given **enough** hidden neurons)

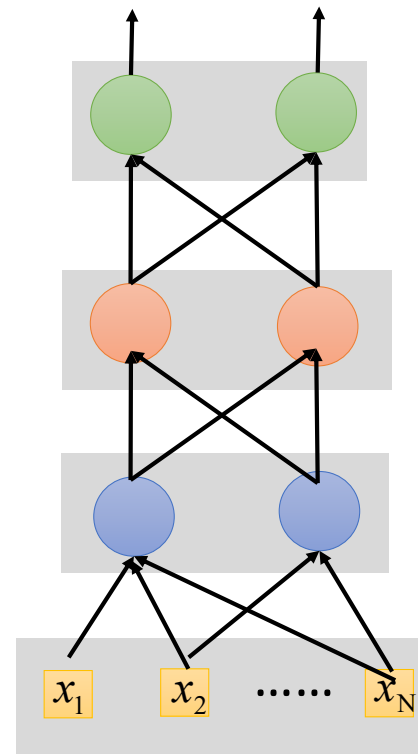


Why “Deep” neural network not “Fat” neural network?

Fat + Short v.s. Thin + Tall

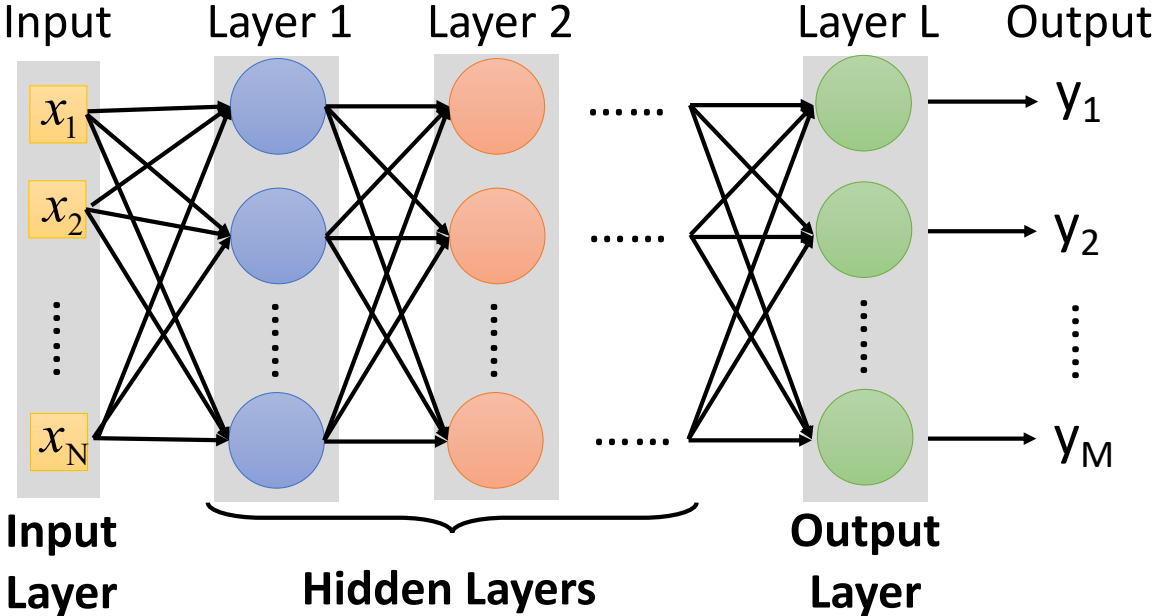


Shallow



Deep

Deep Means Many Hidden Layers



Why: Hierarchical Organization

- The idea of hierarchical structures is based on the decomposition of a hierarchy into simpler parts.
- Hierarchy offers a more efficient way of representing information. Deep learning enables high-level abstractions in data by architectures composed of multiple nonlinear transformations.
- It offers a natural progression from low-level structures to high-level structure, as demonstrated by natural complexity

Why: Boolean Functions

- Any Boolean function can be represented by a truth table.
- For D variables there are 2^D rows in the table.
 - A table can be described by a disjunctive normal form (DNF).
 - A disjunctive normal form can be described as an OR of ANDs.
- Each AND operation can be implemented by a perceptron.
 - All the AND perceptrons are ordered in one hidden layer.
 - Their output is fed into one perceptron that implements an OR operation. This kind of representation can lead to an exponential explosion of the AND operations.
- One could try to represent the formula by a circuit of bigger depth but lesser complexity.

Can represent Big Training Sets

- Deep networks may be trained on big training sets since they have many free parameters.
- “Flat” networks cannot do it, since for big training lead to the the curse of dimensionality.

Why: Curse of dimensionality

- In a “fat” neural networks there are many hidden neurons, its many outputs represent a high dimensional vector that is classified by the output neurons.
- The high dimensionality of the vector influences negatively the classification of the output neurons during learning and generalization.
 - It corresponds to the curse of dimensionality.
- Deep neural networks can avoid the curse of dimensionality problem by constraining the number of hidden neurons.

Why: Local Minima

- It is assumed that an artificial neural network with several hidden layers is less likely to be stuck in a local minima as demonstrated by empirical experiments.
- It was commonly thought that simple gradient descent would get trapped in poor local minima.
 - In practice, poor local minima are rarely a problem with large networks but a big problem with small networks.
- Most local minima are equivalent in large networks and close to global minimum Chomoranksa et. al (2015).
 - However this is not true for small networks where bad local minima are present Swirszcz et. al. (2016).
- For large networks, the loss function may have a large number of saddle points where the gradient is zero Dauphin et. al (2015).

Why: Efficient Model Selection

- To overcome overfitting one has to use a model that has the right capacity.
- However this task is difficult and costly since it involves the search through many different architectures and parameters.
- Many experiments with different number of neurons and hidden layers have to be done.
- Instead one chooses a deep over-parameterized neural network.
 - The search for the model of the right capacity is done by a search for the correct regularization value.
 - This kind of search is easier to implement

Why: Efficient Model Selection

- This kind of search is easier to implement since for example the l_2 or l_1 regularization are described by **one** variable α

$$\Delta w_j = -\eta \left(\frac{\partial E}{\partial w_j} \right) - \alpha \cdot w_j$$

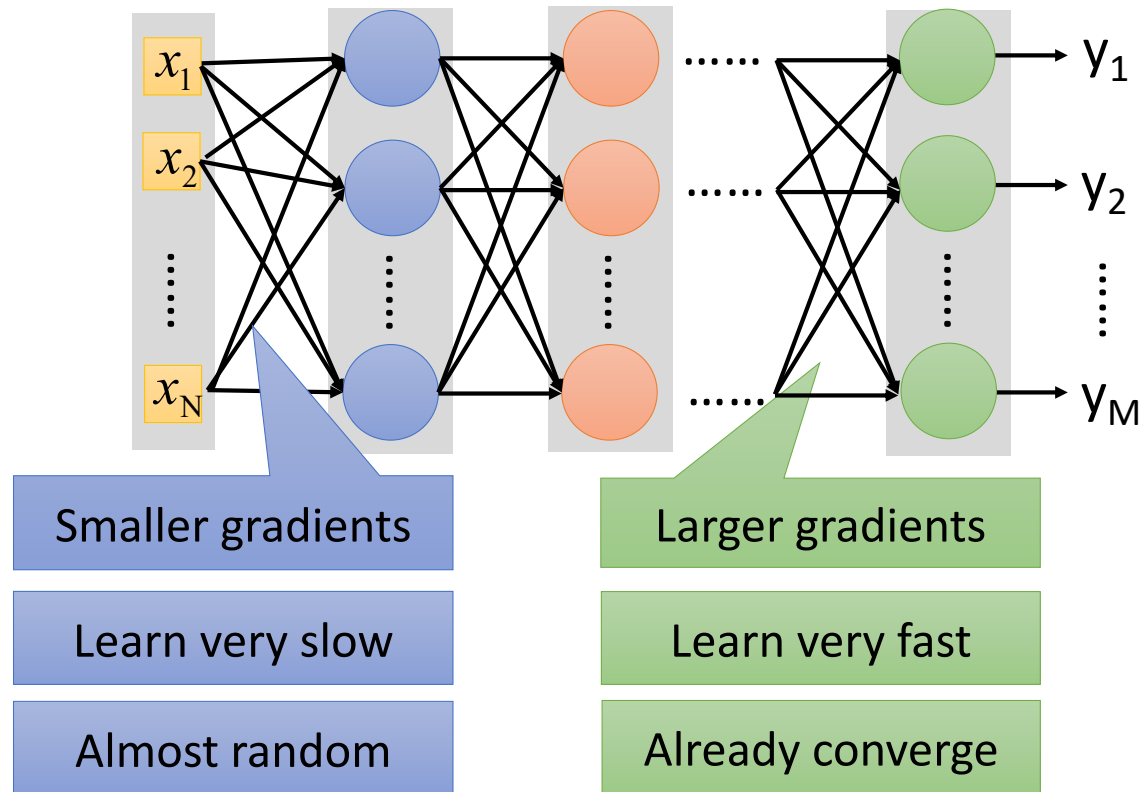
$$\Delta w_j = -\eta \left(\frac{\partial E}{\partial w_j} \right) - \alpha \cdot \text{sign}(w_j).$$

- The search for the correct model complexity can be done efficiently, by empirical experiments for searching for just one correct α value.
- This leads to a model with the optimal predictive capability is the one that leads to the best balance between bias and variance.

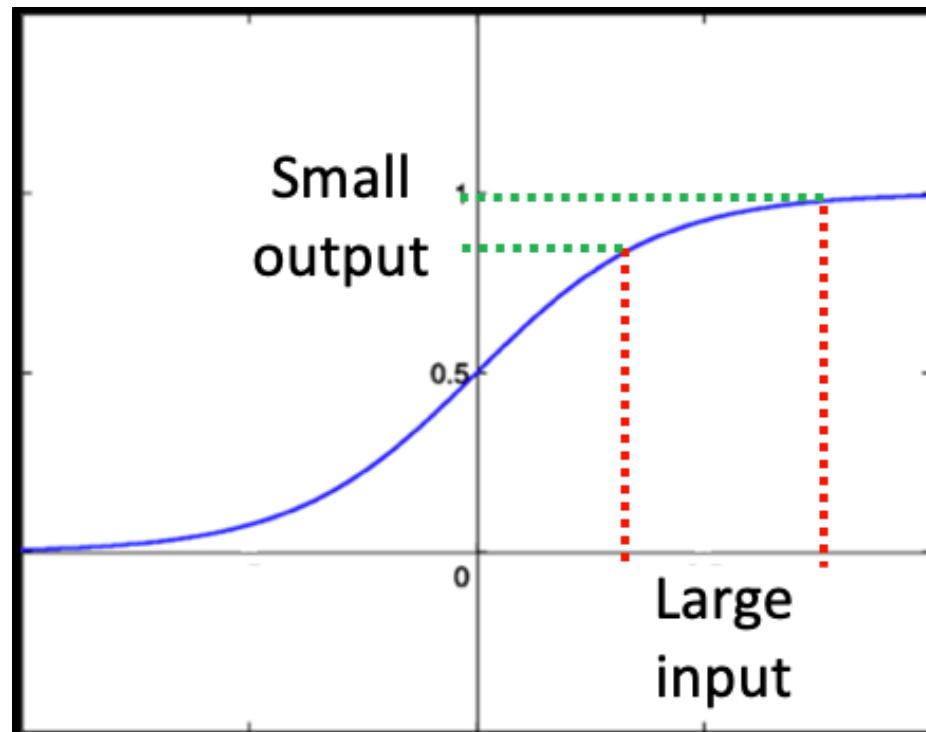
Why Not?

- It seems as well that the deep learning revolution results mainly from brute force, it is not based on new mathematical models and appears to be biologically unlikely.
- Deep neural networks require a very large labeled sample training set that can become a bottleneck, since in many special application it is difficult to generate big labeled sample training sets.
- Often these huge sets have to be manually labeled by some experts which results in high costs.

Vanishing Gradient Problem



Vanishing Gradient Problem, sigmoid

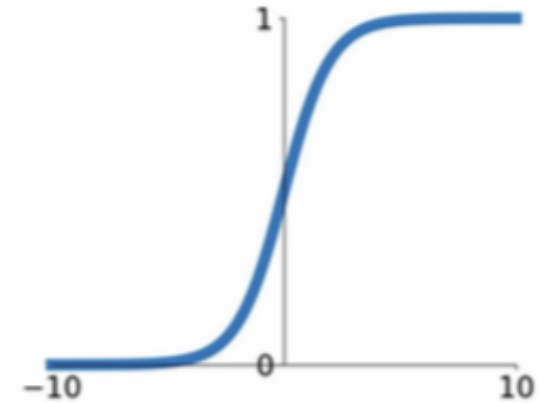


Sigmoid

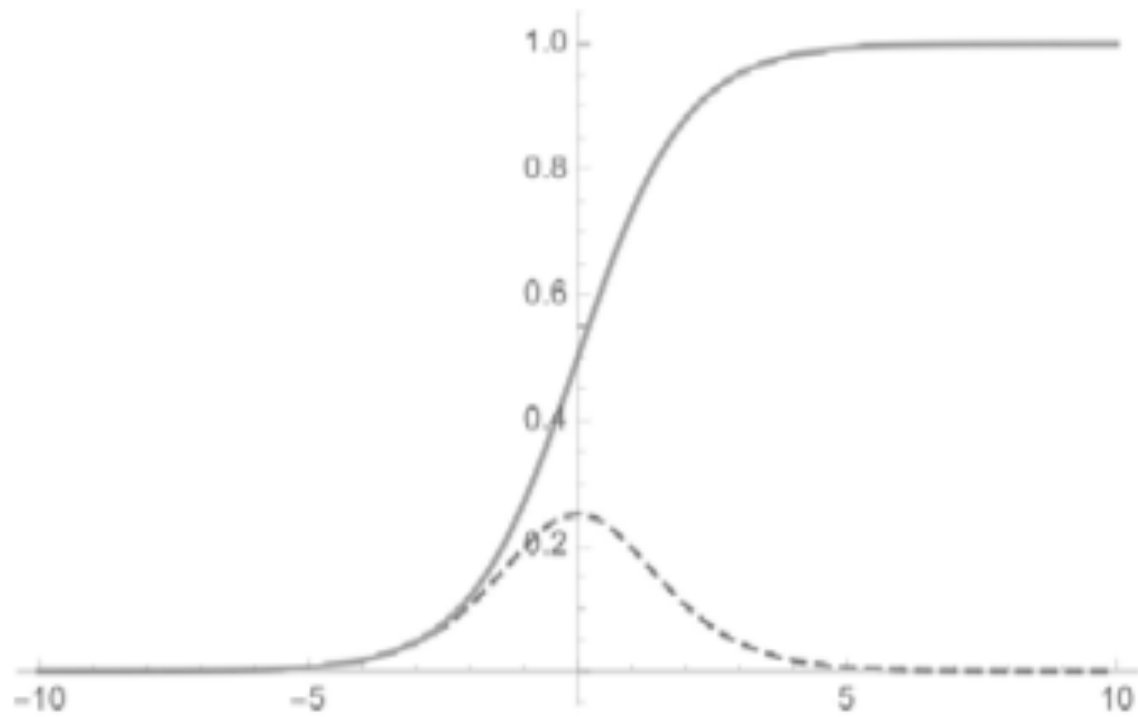
$$f(x) = \sigma(x) = \frac{1}{1 + e^{(-\alpha \cdot x)}}$$

$$f'(x) = \sigma'(x) = \alpha \cdot \sigma(x) \cdot (1 - \sigma(x))$$

- Squashes numbers to range [0,1]
- Historically popular
- Have nice interpretation as a saturating “firing rate” of a neuron
- 3 problems:
 - Saturated neurons “kill” the gradients
 - Sigmoid outputs are not zero-centered
 - $\exp()$ is a bit compute expensive



Sigmoid

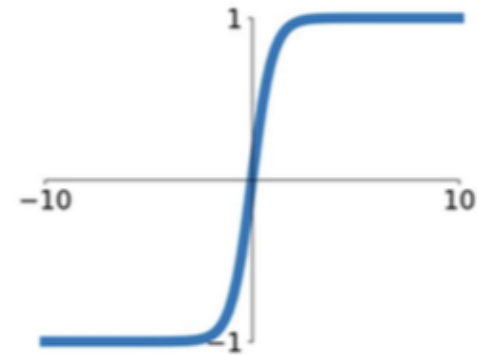


- The sigmoid function and the derivative indicated by dotted line.

$$f(x) = \tanh(\alpha \cdot x)$$

$$f'(x) = \alpha \cdot (1 - f(x)^2)$$

- Squashes numbers to range [-1,1]
- - zero centered (nice)
- - kills gradients when saturated 😞

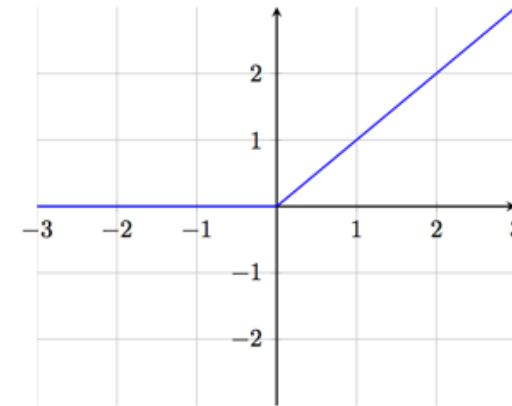


tanh(x)

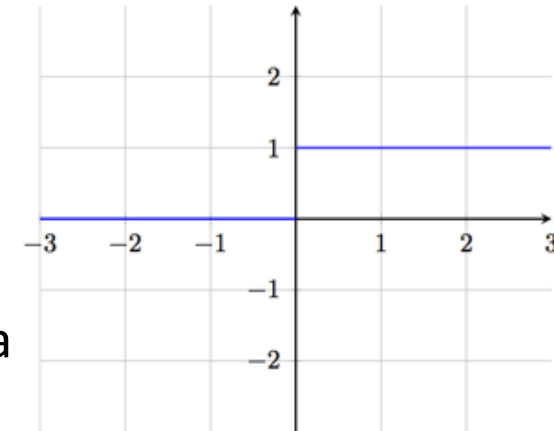
Rectified Linear Unit (ReLU)

- $f(x) = \max(0, x)$
 - Function defined as the positive part of its argument
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- More biologically plausible

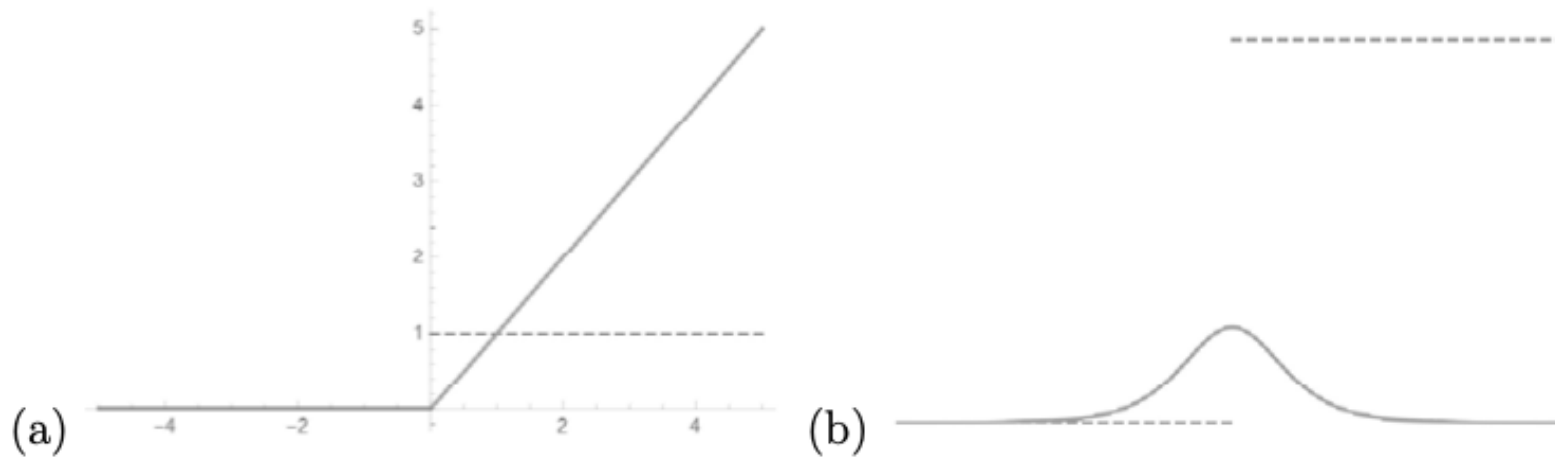
- But: Not zero-centered output ☹️
- Non-differentiable at zero; however it is differentiable a value of 0 or 1



ReLU function

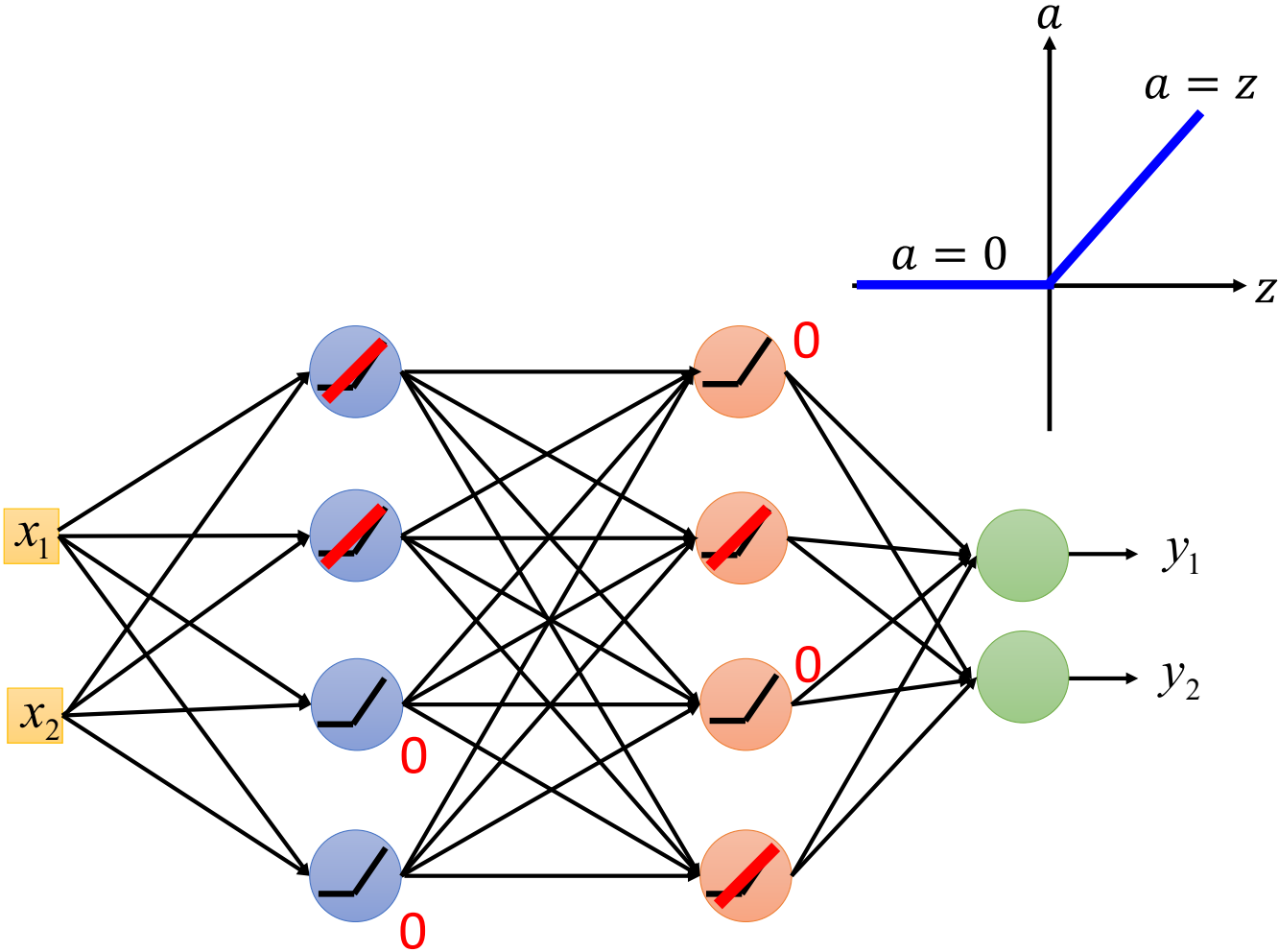


Derivative of ReLU function



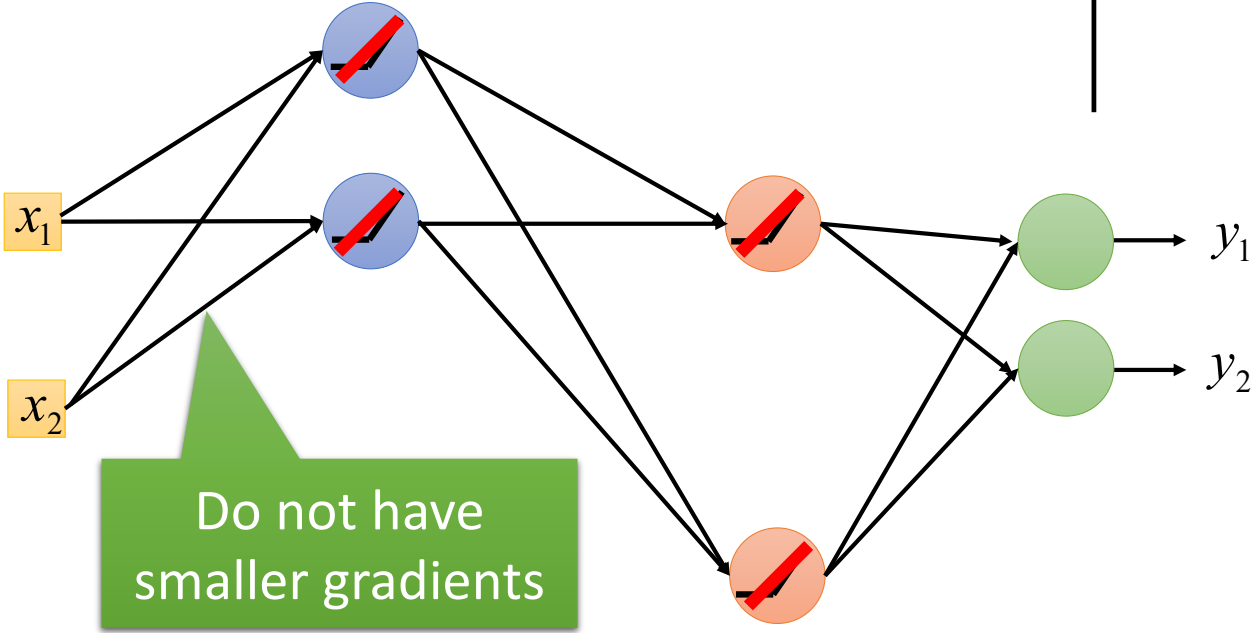
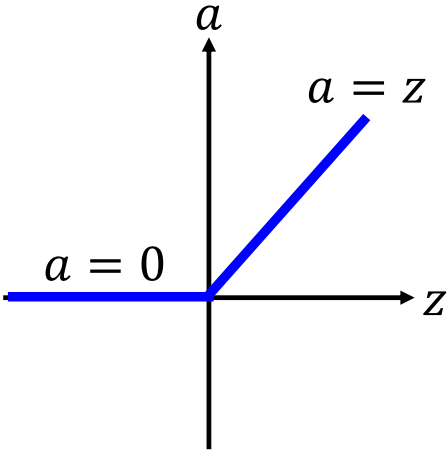
- (a) Rectifier activation function (ReLU), the derivative is indicated by the dotted line. (b) Comparing the the derivative of the sigmoid activation function and the rectifier activation function indicated by a dotted line.

ReLU

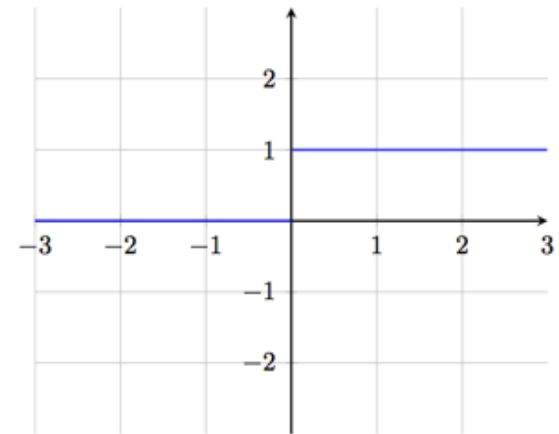


ReLU

A Thinner linear network



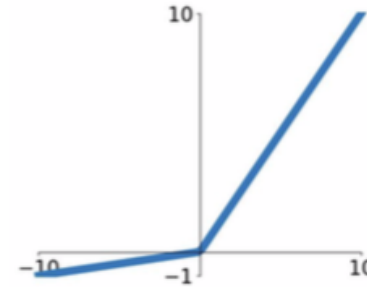
- Not zero-centered output
- An annoyance (?):
- What is the gradient when $x < 0$?
- ReLU zero will never activate again (it dies)
 - Is this good or bad?



Derivative of ReLU function

Leaky ReLU

$$f(x) = \max(0.01x, x)$$



- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- Will not die

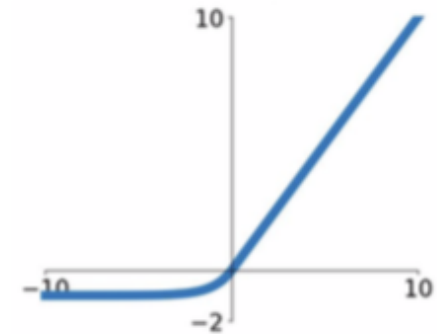
Parametric ReLUs

$$f(x) = \max(\alpha x, x)$$

- Parametric ReLUs (PReLU) take this idea further by making the coefficient of leakage α into a parameter that is learned along with the other neural network parameters

Exponential Linear Units (ELU)

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha (\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

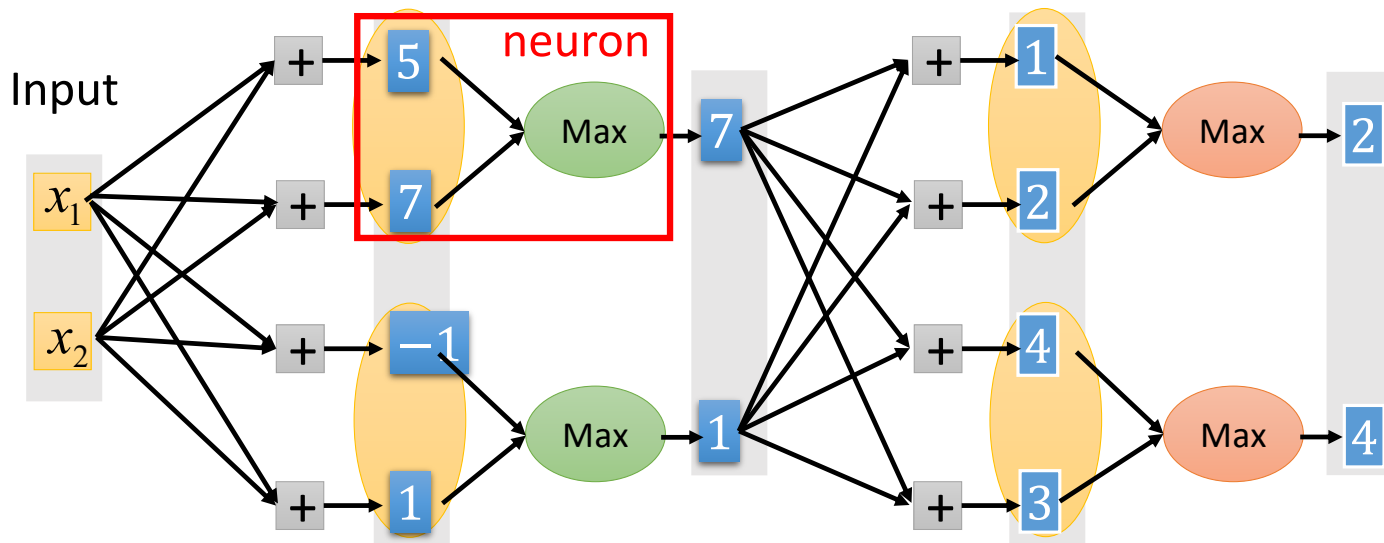


- All benefits of ReLU
- - Closer to zero mean outputs
- - Negative saturation regime
- compared with Leaky ReLU adds some robustness to noise
- Computation requires $\exp()$ ☹️

Maxout

ReLU is a special cases of Maxout

- Learnable activation function [Ian J. Goodfellow, ICML'13]



You can have more than 2 elements in a group.

Maxout “Neuron”

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

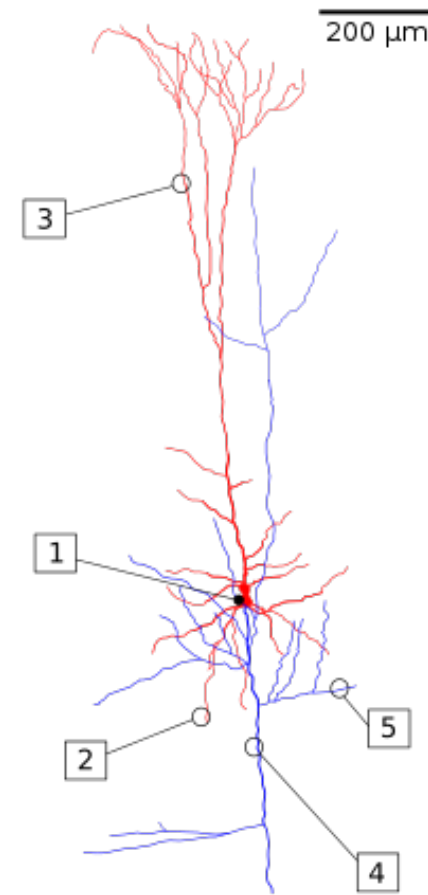
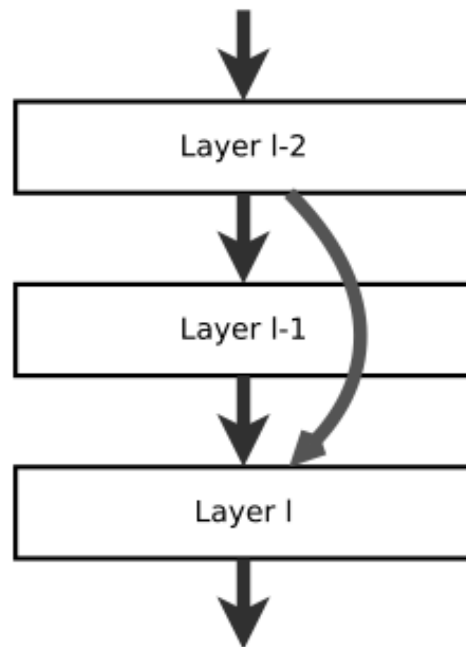
- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

- Problem: doubles the number of parameters/neuron :(

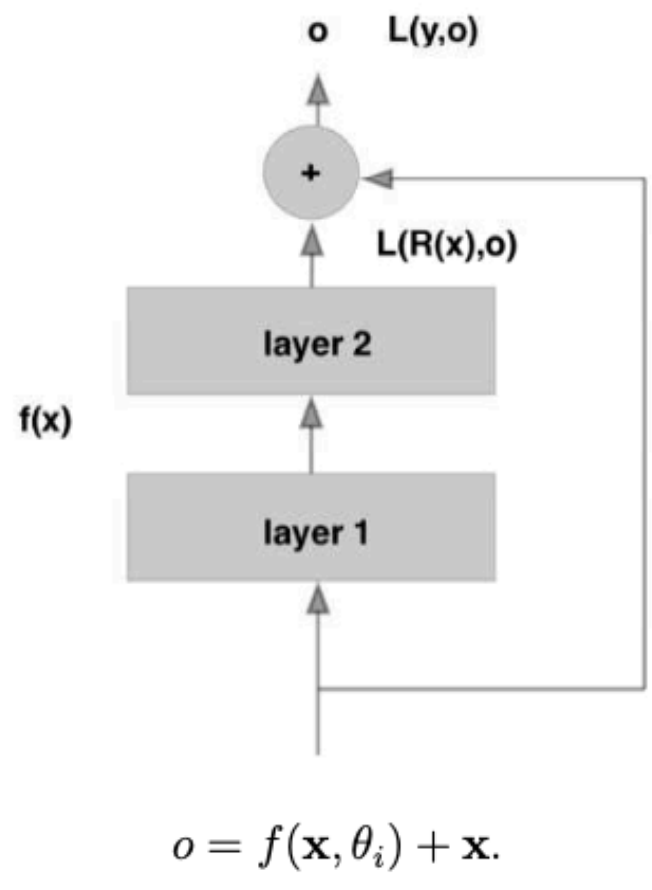
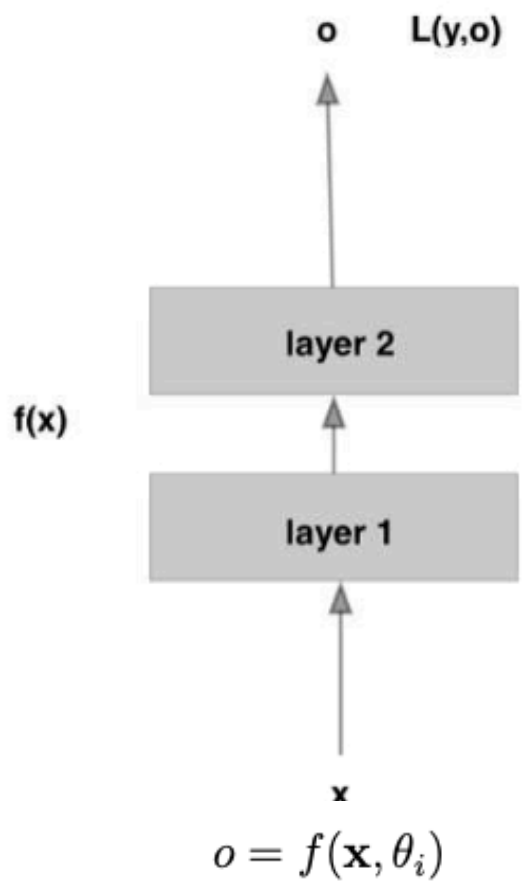
- Use ReLU.
- Be careful with your learning rates
- Try out Leaky ReLU / Maxout / ELU

- Try out tanh but don't expect much
- Don't use sigmoid

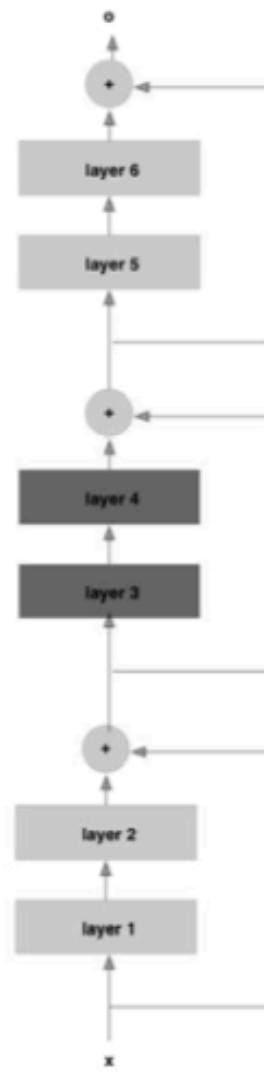
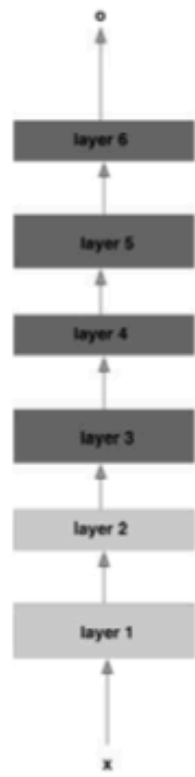
Residual neural network



Pyramid Cells



building block
 $o_\theta = f(\mathbf{x}, \theta)$



- Residual neural networks do this by utilizing *skip connections*, or *short-cuts* to jump over some layers.
- One motivation for skipping over layers is to avoid the problem of vanishing gradients, by reusing activations from a previous layer until the adjacent layer learns its weights.
- During training, the weights adapt to mute the upstream layer, and amplify the previously-skipped layer.
 - In the simplest case, only the weights for the adjacent layer's connection are adapted, with no explicit weights for the upstream layer.

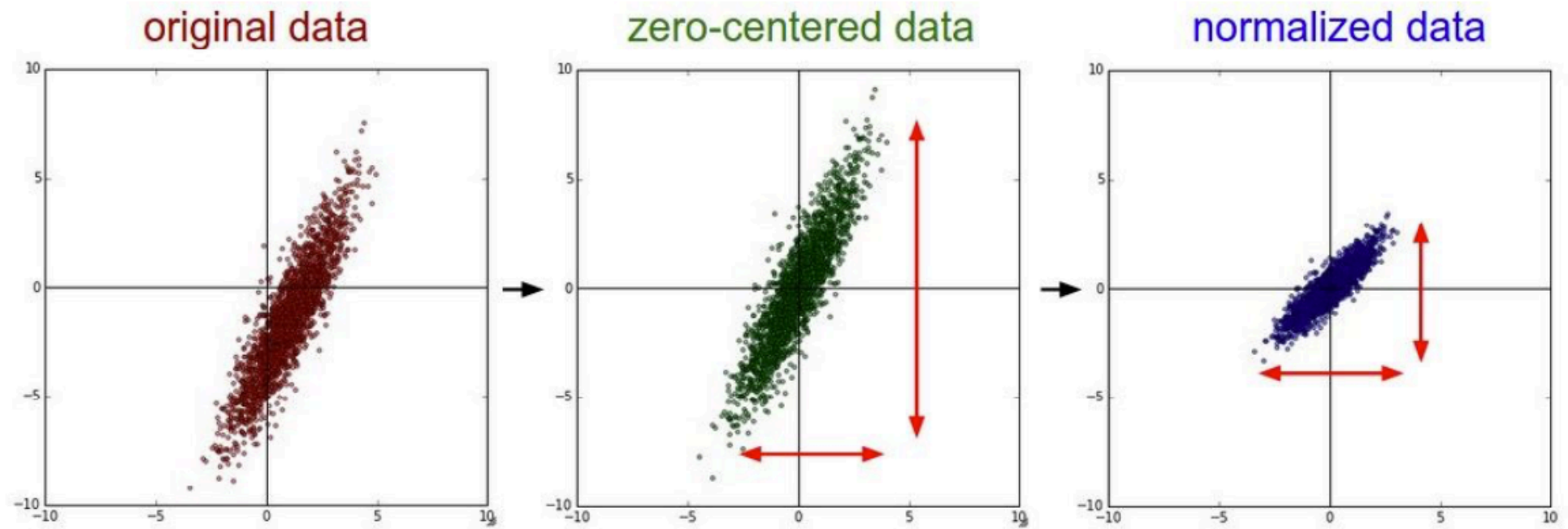
Weight Initialization

- **Small random numbers**
(gaussian with zero mean and $1e-2$ standard deviation)
- Works for small networks, but problems with deeper networks.
- Deeper networks:
 - All activations can become zero!
 - Almost all neurons completely saturated, either -1 and 1.
 - Gradients will be all zero.

Xavier initialization

- Weights from a Gaussian distribution with
 - zero mean
 - variance of $1/N$
 - N specifies the number of input neurons.
 - when using the ReLU nonlinearity it breaks 😞
- Better [He et al., 2015]:
 - Weights from a Gaussian distribution with
 - zero mean
 - variance of $2/N$
 - Does not break with ReLU

Preprocess the data

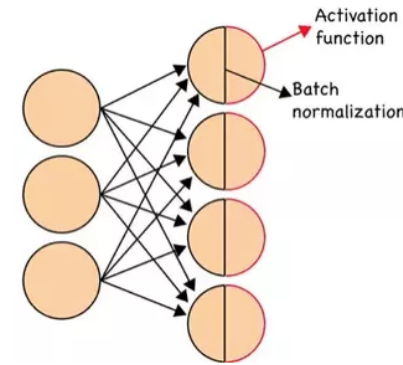


Batch Normalization

- We normalize all training data so that it resembles a normal distribution (that means, zero mean and a unitary variance)
- In the intermediate layers the distribution of the activations is constantly changing during training
 - This slows down the training process because each layer must learn to adapt themselves to a new distribution in every training step.
 - Batch normalization is a method we can use to normalize the inputs of each layer, in order to fight the internal covariate shift problem

Batch Normalization

- During training time, a **batch normalization layer** does the following:
 - Calculate the mean and variance of the layers input
 - Normalize the layer inputs using the previously calculated batch statistics
 - Scale and shift in order to obtain the output of the layer
 - γ and β are **learned during training** along with the original parameters of the network.



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots x_m\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

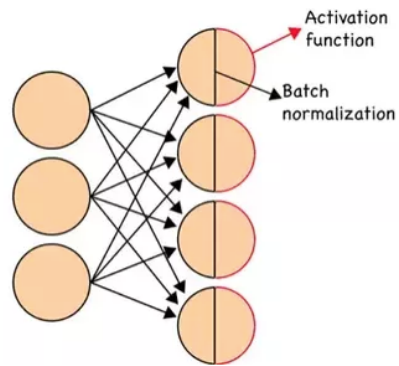
$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{ mini-batch mean}$$

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

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

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

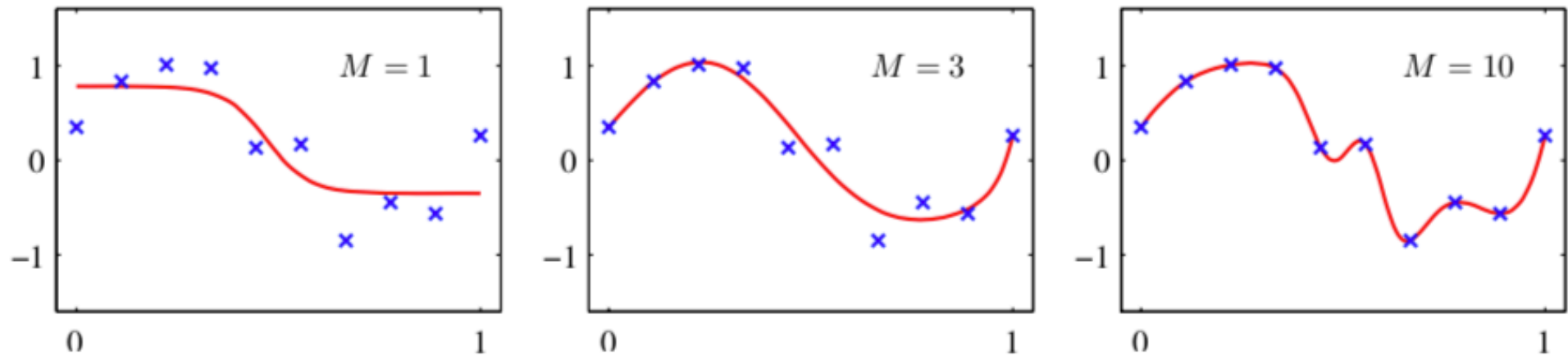
Test



- During test (or inference) time, the mean and the variance are fixed. They are estimated using the previously calculated means and variances of each training batch.

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



- Examples of two-layer networks trained on 10 data points drawn from the sinusoidal data set. The graphs show the result of fitting networks having $M = 1$, 3 and 10 hidden units, respectively, by minimizing a sum-of-squares error function

Preventing overfitting

- Get more data!
 - Always the best
- Use a model that has the right capacity:
 - enough to model the true regularities
 - not enough to also model the spurious regularities (assuming they are weaker)
- Early stopping
 - Start with small weights and stop the learning before it overfits
- Weight-decay: Penalize large weights using penalties or constraints on their squared values (L2 penalty) or absolute values (L1 penalty).
- Use Dropout that drops randomly some weights
- Noise: Add noise to the weights or the activities

Using a Validation Set

- Divide the total dataset into three subsets:
- Training data is used for learning the parameters of the model.
- Validation data is not used of learning but is used for deciding what type of model and what amount of regularization works best.
- Test data is used to get a final, unbiased estimate of how well the network works. We expect this estimate to be worse than on the validation data

l_2 Regularization

- The standard L_2 weight penalty involves adding an extra term to the cost function that penalizes the squared weights
- This keeps the weights small unless they have big error derivatives.
- It prevents the network from using weights that it does not need.
 - This can often improve generalization a lot because it helps to stop the network from fitting the sampling error.
 - It makes a smoother model in which the output changes more slowly as the input changes.

l_2 Regularization

$$\tilde{E}(\mathbf{w}) = \frac{1}{2} \cdot \sum_{k=1}^N (y_k - o_k)^2 + \frac{\lambda}{2} \|\mathbf{w}\|_2^2, \text{ or } \tilde{E}(\mathbf{w}) = - \sum_{k=1}^N y_k \log o_k + \frac{\lambda}{2} \|\mathbf{w}\|_2^2$$

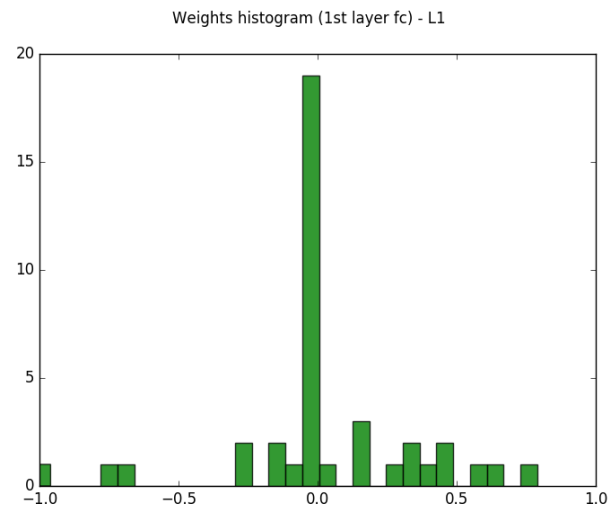
$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2} \|\mathbf{w}\|_2^2$$

$$\frac{\partial \tilde{E}}{\partial w_j} = \frac{\partial E}{\partial w_j} + \lambda \cdot w_j.$$

$$\Delta w_j = -\eta \left(\frac{\partial E}{\partial w_j} + \lambda \cdot w_j \right) = -\eta \left(\frac{\partial E}{\partial w_j} \right) - \eta \lambda \cdot w_j = -\eta \left(\frac{\partial E}{\partial w_j} \right) - \alpha \cdot w_j$$

l_1 Regularization

- Sometimes it works better to penalize the absolute values of the weights.
 - This makes some weights equal to zero which helps interpretation.



l_1 Regularization

$$\tilde{E}(\mathbf{w}) = \frac{1}{2} \cdot \sum_{k=1}^N (y_k - o_k)^2 + \lambda \cdot \|\mathbf{w}\|_1, \text{ or } \tilde{E}(\mathbf{w}) = - \sum_{k=1}^N y_k \log o_k + \lambda \cdot \|\mathbf{w}\|_1$$

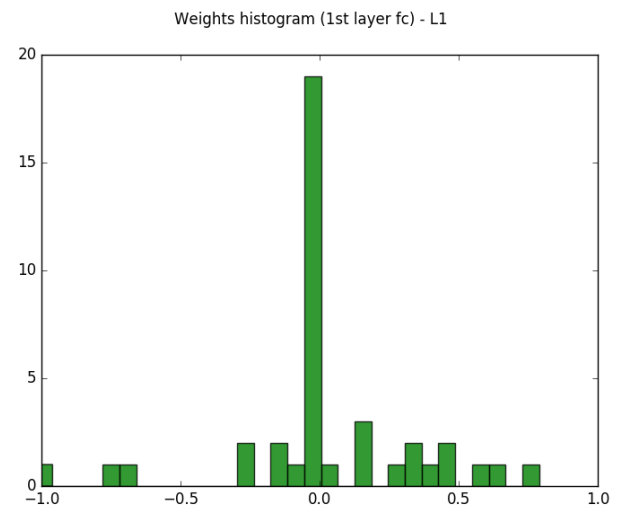
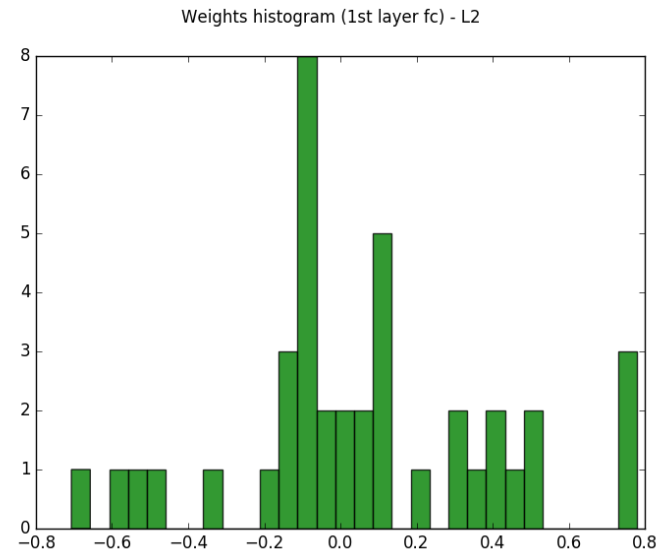
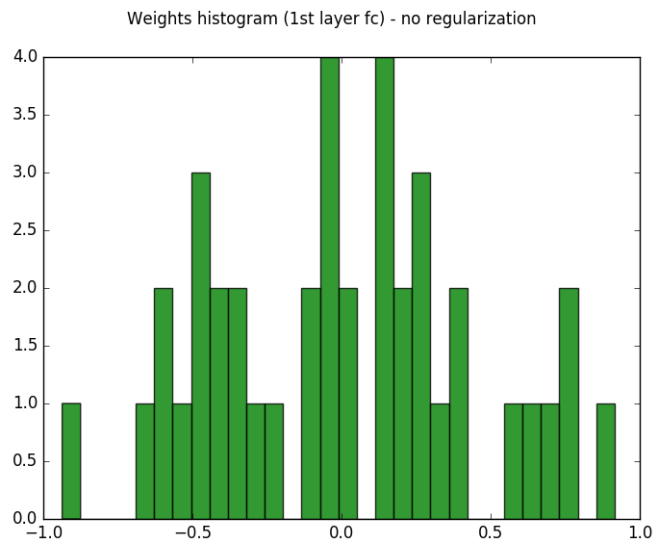
$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \lambda \|\mathbf{w}\|_1$$

$$\frac{\partial \tilde{E}}{\partial w_j} = \frac{\partial E}{\partial w_j} + \lambda \cdot \text{sign}(w_j).$$

$$\Delta w_j = -\eta \left(\frac{\partial E}{\partial w_j} + \lambda \cdot \text{sign}(w_j) \right) = -\eta \left(\frac{\partial E}{\partial w_j} \right) - \eta \lambda \cdot \text{sign}(w_j)$$

$$\Delta w_j = -\eta \left(\frac{\partial E}{\partial w_j} \right) - \alpha \cdot \text{sign}(w_j)$$

l_2 versus l_1 Regularization



l_p Regularization

$$\tilde{E}(\mathbf{w}) = \frac{1}{2} \cdot \sum_{k=1}^N (y_k - o_k)^2 + \frac{\lambda}{2} \|\mathbf{w}\|_p^p, \quad \text{or} \quad \tilde{E}(\mathbf{w}) = - \sum_{k=1}^N y_k \log o_k + \frac{\lambda}{2} \|\mathbf{w}\|_p^p$$

$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2} \|\mathbf{w}\|_p^p$$

$$\frac{\partial \tilde{E}}{\partial w_j} = \frac{\partial E}{\partial w_j} + \lambda \cdot p \cdot w_j^{p-1}.$$

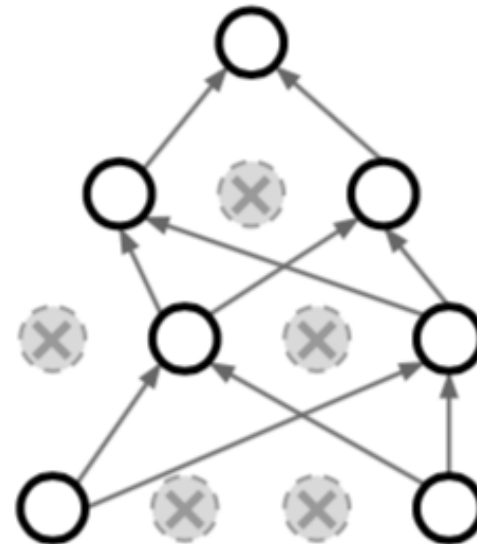
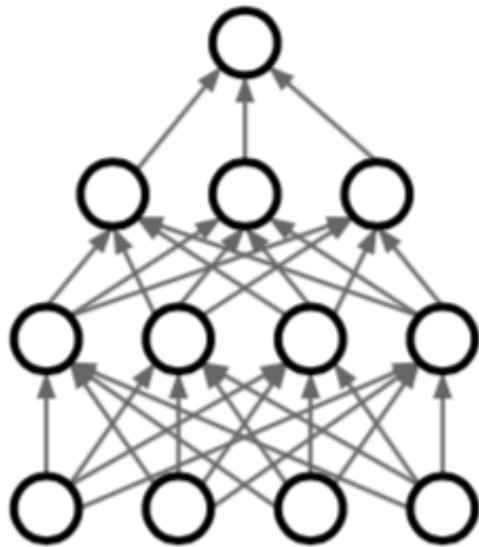
$$\Delta w_j = -\eta \left(\frac{\partial E}{\partial w_j} + \lambda \cdot w_j^{p-1} \right) = -\eta \left(\frac{\partial E}{\partial w_j} \right) - \eta \cdot p \lambda \cdot w_j^{p-1} = -\eta \left(\frac{\partial E}{\partial w_j} \right) - \alpha \cdot w_j^{p-1}$$

Regularization: Dropout

- Dropout is a stochastic regularization method.
- In each forward pass, randomly a set some neurons is set to zero (sleep) for one pass.
- The probability of dropping the set out is described by a hyper-parameter; usually 0.5 is commonly used.
 - For input nodes, this should be low, because information is directly lost when input nodes are ignored.

Regularization: Dropout

- In each forward pass, randomly set some neurons to zero (for one pass only)
 - Probability of dropping is a hyperparameter; 0.5 is common



Regularization: Dropout

- Only the reduced network is trained on the data in that stage
- The removed nodes are then reinserted into the network with their original weights.
 - For input nodes, this should be low, because information is directly lost when input nodes are ignored.
- By avoiding training all nodes on all training data, dropout decreases overfitting.
 - It also significantly improves training speed.
 - It reduces node interactions, leading them to learn more robust features
- It is a stochastic process, since each time a different set of neurons is dropped out and not allowed to learn

Faster Optimizers: Momentum

- One can in some cases speed up training by using a faster gradient descent as for example using the momentum α with τ indicating the time step of the algorithm

$$\Delta w_{ti}(\tau + 1) = -\eta \frac{\partial E}{\partial w_{ti}} + \alpha \cdot \Delta w_{ti}(\tau)$$

$$w_{ti}^{new} = w_{ti}^{old} + \Delta w_{ti}(\tau + 1).$$

- It prohibits fast changes of the direction of the gradient. The momentum parameter α is chosen between 0 and 1, usually 0.9 is a good value.

Nestrov Momentum

- The idea of Nesterov momentum optimization is to measure the gradient of the cost function not at the local position but slightly ahead in the direction of the momentum $w_{ti} + \Delta w_{ti}(\tau)$

$$\Delta w_{ti}(\tau + 1) = -\eta \frac{\partial E(w_{ti} + \Delta w_{ti}(\tau))}{\partial w_{ti}} + \alpha \cdot \Delta w_{ti}(\tau),$$

$$w_{ti}^{new} = w_{ti}^{old} + \Delta w_{ti}(\tau + 1).$$

AdaGrad

- The AdaGrad algorithm scales down the gradient vector along the steepest dimensions and uses a different learning rate for every parameter w_j at every time step
 - It scales them inversely proportional to the square root of the sum of all squared values of the gradient representing the scale variable $s_{ti}(\tau)$
 - At the begin of learning with $\tau = 0$ the scale variable is initialized to zero $s_{ti}(0) = 0$.

$$s_{ti}(\tau + 1) = s_{ti}(\tau) + \frac{\partial E}{\partial w_{ti}} \cdot \frac{\partial E}{\partial w_{ti}}$$

and we perform the scaled gradient descent

$$w_{ti}^{new} = w_{ti}^{old} - \eta \cdot \frac{\frac{\partial E}{\partial w_{ti}}}{\sqrt{s_{ti}(\tau + 1) + \epsilon}}$$

ϵ is a smoothing term, a very tiny number to avoid division by zero

RMSProp

- The AdaGrad algorithm can slow down a bit too fast and may end up never converging to a minimum.
- RMSProp changes the gradient into an exponential weighted moving average.
 - It discards history from extreme past so that it can converge rapidly, it accumulating only the gradients from the most recent iterations

$$s_{ti}(\tau + 1) = \alpha \cdot s_{ti}(\tau) + (1 - \alpha) \cdot \frac{\partial E}{\partial w_{ti}} \cdot \frac{\partial E}{\partial w_{ti}}$$

$$w_{ti}^{new} = w_{ti}^{old} - \eta \cdot \frac{\frac{\partial E}{\partial w_{ti}}}{\sqrt{s_{ti}(\tau + 1) + \epsilon}}$$

α being the decay rate with a typical value of 0.9.

Adam

- Adam which stands for adaptive moment estimation, combines the ideas of momentum

$$\Delta w_{ti}(\tau + 1) = -(1 - \alpha_1) \frac{\partial E}{\partial w_{ti}} + \alpha_1 \cdot \Delta w_{ti}(\tau)$$

- and RMSProp

$$s_{ti}(\tau + 1) = \alpha_2 \cdot s_{ti}(\tau) + (1 - \alpha_2) \cdot \frac{\partial E}{\partial w_{ti}} \cdot \frac{\partial E}{\partial w_{ti}}$$

Adam

$$\Delta w_{ti}(\tau + 1) = \frac{\Delta w_{ti}(\tau + 1)}{1 - \alpha_1^{\tau+1}}$$

and

$$s_{ti}(\tau + 1) = \frac{s_{ti}(\tau + 1)}{1 - \alpha_2^{\tau+1}}.$$

Then we apply the update

$$w_{ti}^{new} = w_{ti}^{old} + \eta \cdot \frac{\Delta w_{ti}(\tau + 1)}{\sqrt{s_{ti}(\tau + 1) + \epsilon}}$$

Notation...

- In deep learning community one often uses a vector notation and indicates the free parameter as θ since beside the weights there are as well other parameters
- The notation for Adam would be for momentum

$$\mathbf{m} = \alpha_1 \cdot \mathbf{m} - (1 - \alpha_1) \nabla_{\theta} L(\theta)$$

- and RMSProp

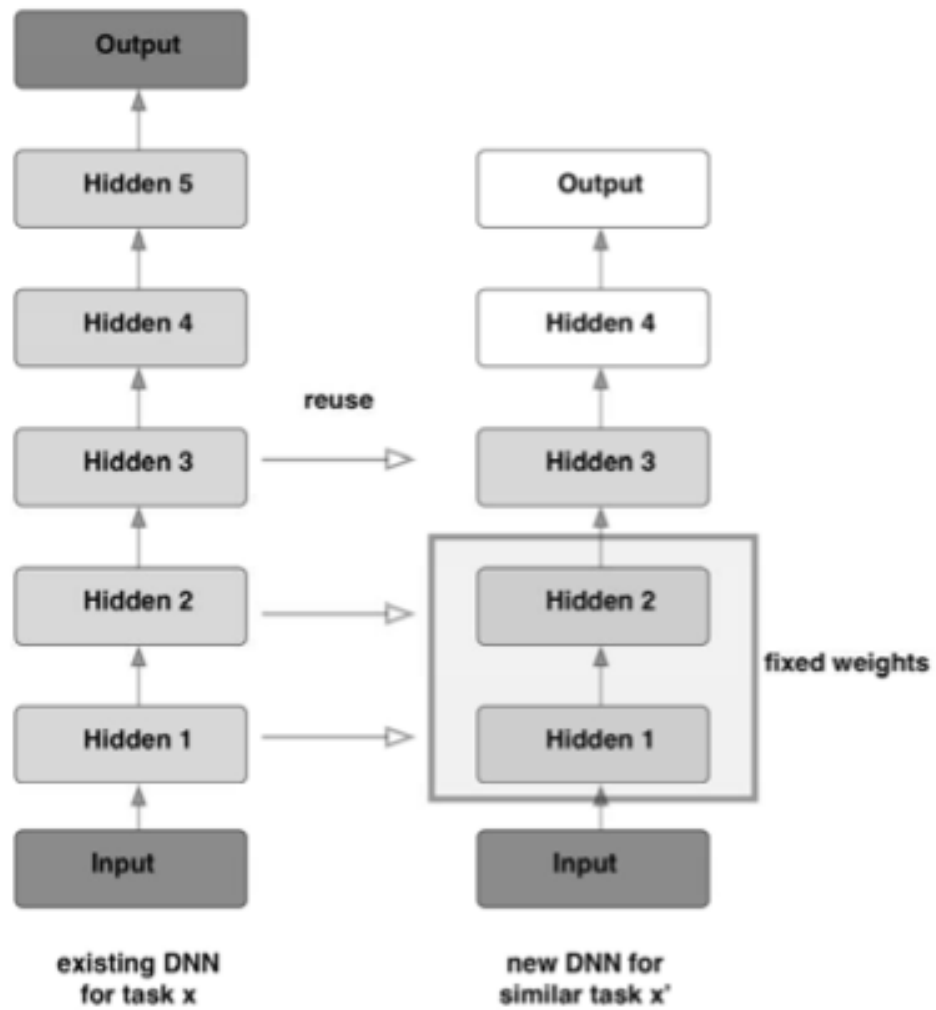
$$\mathbf{s} = \alpha_2 \cdot \mathbf{s} + (1 - \alpha_2) \nabla_{\theta} L(\theta) \odot \nabla_{\theta} L(\theta)$$

$$\mathbf{m} = \frac{\mathbf{m}}{1 - \alpha_1^t}$$

\odot represents the element-wise multiplication

Transfer Learning

- You need a lot of a data if you want to train
- Transfer learning and domain adaptation refer to the situation where what has been learned in one setting (i.e., distribution P_1) is exploited to improve generalization in another setting (say distribution P_2).
- We assume that many of the factors that explain the variations in P_1 are relevant to the variations that need to be captured for learning P_2 .



Local Minima (what we know)

- It was commonly thought that simple gradient descent would get trapped in poor local minima
- In practice, poor local minima are rarely a problem with large networks.
- Regardless of the initial conditions, the system nearly always reaches solutions of very similar quality.
 - Recent theoretical and empirical results strongly suggest that local minima are not a serious issue in general.
 - Instead, the landscape is packed with a combinatorially large number of saddle points where the gradient is zero

Conclusion

- Deep learning is: a **black box** but it is also a **black art**.
- Many approaches and hyperparameters:
 - activation functions,
 - learning rate,
 - momentum?
- Often these need tweaking, and you need to know what they do to change them intelligently.

Next

- Convolutional Neural Networks

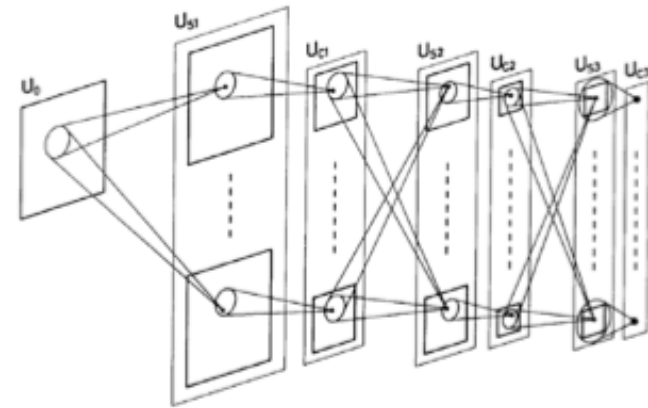
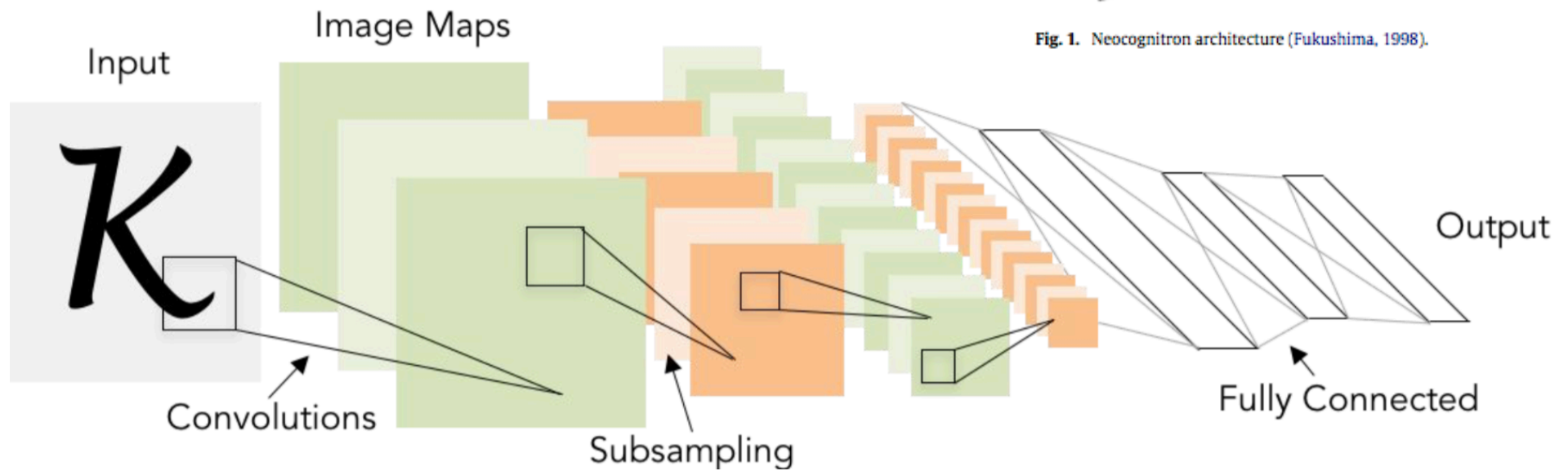
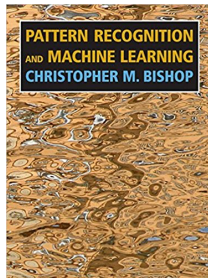


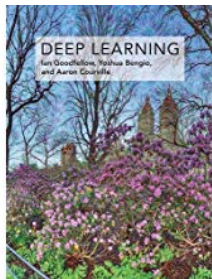
Fig. 1. Neocognitron architecture (Fukushima, 1998).



Literature

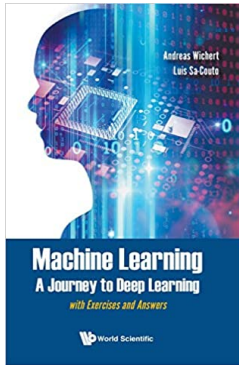


- Christopher M. Bishop, Pattern Recognition and Machine Learning (Information Science and Statistics), Springer 2006
 - Chapter 5



- Deep Learning, I. Goodfellow, Y. Bengio, A. Courville MIT Press 2016
 - Chapter 6, 7, 8

Literature



- Machine Learning - A Journey to Deep Learning, A. Wichert, Luis Sa-Couto, World Scientific, 2021
 - Chapter 12