Cognitive Algorithms
Lecture 6

Multilayer Perceptrons

Klaus-Robert Müller, Johannes Niediek,
Augustin Krause, Joanina Oltersdorff, Ken Schreiber

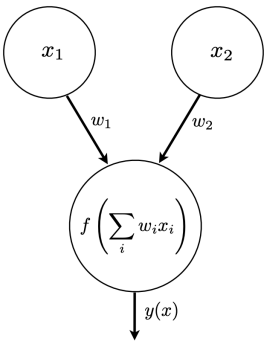
Technische Universität Berlin
Machine Learning Group

Last time: unsupervised learning

Learning without labels

- Dimensionality reduction
  - PCA
  - Non-negative matrix factorization
- Clustering
  - K-Means

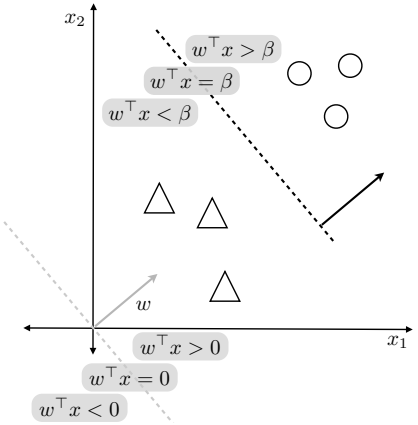
Single(-layer) perceptron



- Input nodes x_i receive information
- Inputs are multiplied with weighting factors w_i and summed up
- The sum is mapped through a non-linear function $f(\cdot)$

$$\text{e.g. } f(\boldsymbol{x}) = \begin{cases} +1 & \text{if } \boldsymbol{x} > 0 \\ -1 & \text{if } \boldsymbol{x} \leq 0 \end{cases}$$

Linear classification and the perceptron



$$\boldsymbol{w}^\top \boldsymbol{x} - \beta = \begin{cases} > 0 & \text{if } \boldsymbol{x} \text{ belongs to } \bigcirc \\ < 0 & \text{if } \boldsymbol{x} \text{ belongs to } \triangle \end{cases}$$

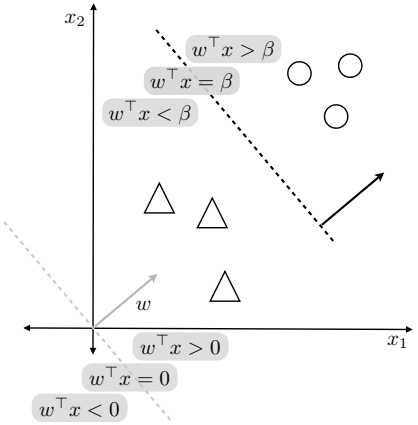
The *offset* β can be included in $\boldsymbol{w}$

$$\tilde{\boldsymbol{x}} \leftarrow \begin{bmatrix} 1 \\ \boldsymbol{x} \end{bmatrix} \qquad \tilde{\boldsymbol{w}} \leftarrow \begin{bmatrix} -\beta \\ \boldsymbol{w} \end{bmatrix}$$

such that

$$\tilde{\boldsymbol{w}}^\top \tilde{\boldsymbol{x}} = \boldsymbol{w}^\top \boldsymbol{x} - \beta.$$

Linear classification and the perceptron



How to find a good $\boldsymbol{w}$?

1. We need an **error function** that tells us how good $\boldsymbol{w}$ is.
2. Then we choose $\boldsymbol{w}$ such that the error function is minimized.

The perceptron error function

Perceptron error

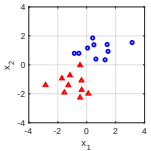
The perceptron error $\mathcal{E}_\mathcal{P}$ is a function of the weights $\boldsymbol{w}$

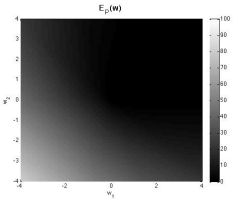
$$\mathcal{E}_\mathcal{P}(\boldsymbol{w}) := -\sum_{m\in\mathcal{M}} \boldsymbol{w}^\top \boldsymbol{x}_m y_m, \tag{1}$$

where $\mathcal{M}$ denotes the index set of all *misclassified* data $\boldsymbol{x}_m$.

Then find $\operatorname{argmin}_{\boldsymbol{w}} \mathcal{E}_\mathcal{P}(\boldsymbol{w})$.

Data $\boldsymbol{x} \in \mathbb{R}^2$





The perceptron learning algorithm

How to find a good $\boldsymbol{w}$? The perceptron error $\mathcal{E}_{\mathcal{P}}(\boldsymbol{w}) = -\sum_{m \in \mathcal{M}} \boldsymbol{w}^\top \boldsymbol{x}_m y_m$ can be minimized *iteratively* using **stochastic gradient descent** [Bottou, 2010; Robbins and Monro, 1951]

1. Initialize $\boldsymbol{w}^{\text{old}}$ (randomly, $1/n$, ...)
2. While there are misclassified data points
   - Pick a random misclassified data point $\boldsymbol{x}_m$
   - Descent in direction of the gradient at single data point $\boldsymbol{x}_m$

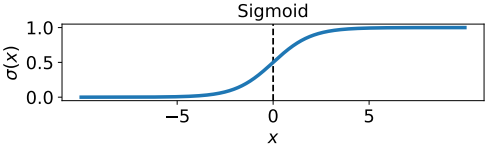
$$\begin{aligned} \mathcal{E}_m(\boldsymbol{w}) &= -\boldsymbol{w}^\top \boldsymbol{x}_m y_m \\ \nabla \mathcal{E}_m(\boldsymbol{w}) &= -\boldsymbol{x}_m y_m \\ \boldsymbol{w}^{\text{new}} &\leftarrow \boldsymbol{w}^{\text{old}} - \eta \nabla \mathcal{E}_m(\boldsymbol{w}^{\text{old}}) = \boldsymbol{w}^{\text{old}} + \eta \boldsymbol{x}_m y_m \end{aligned}$$

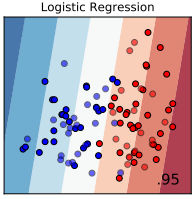
Logistic regression

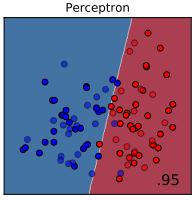
Now let's model the probability that $\boldsymbol{x}$ belongs to one of the classes.
Replace $f(\cdot)$ with the *logistic* (also called "sigmoid") function

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

- Interpret $\sigma(x)$ as probability that x in class $+1$
- Train with gradient descent



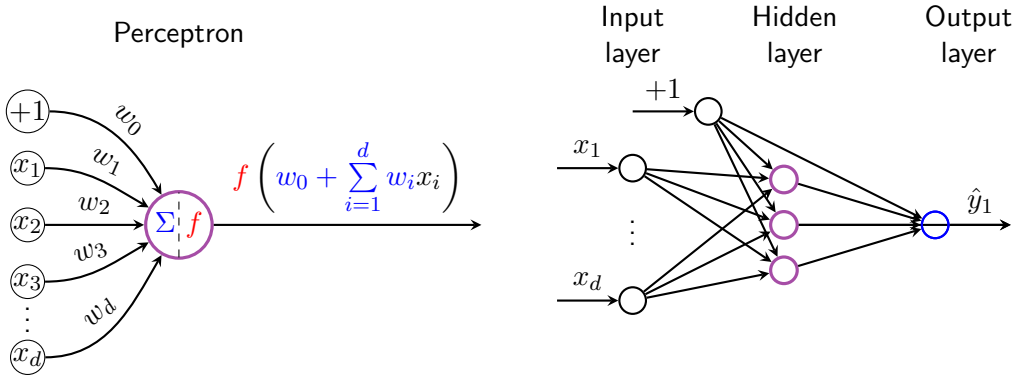




This is called **logistic regression**.

What is a multi-layer perceptron

And what is it capable of?



Adding more perceptrons in parallel yields
a multi-layer perceptron (MLP) with a single hidden layer

$$\hat{y}_1(x) = \sum_j w_j^o f(\boldsymbol{w}_j^{h\top} \boldsymbol{x} + w_{j,0}^h) + w_0^o$$

Multi-layer perceptron

Output can be

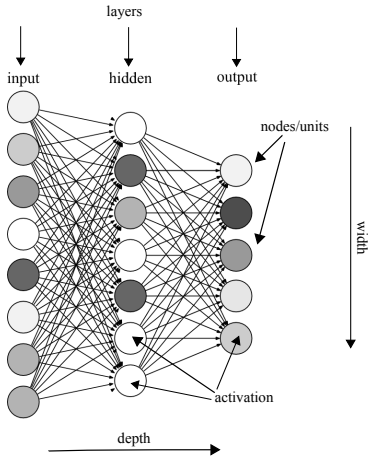
- Class or regression (via different activation functions)
- Multidimensional (via the number of output nodes)

Let's see what that does:

https://playground.tensorflow.org/

(Homework) How does the network perform if we:

- add more layers (deeper net)
- add more nodes to layers (wider net)
- change the activation ($f(\cdot)$)



Universal approximation theorem

One can show that a single hidden layer **MLP** is a universal function approximator, meaning it **can model any** suitably smooth **function, given enough hidden units**, to any desired level of accuracy [Hornik, 1991].

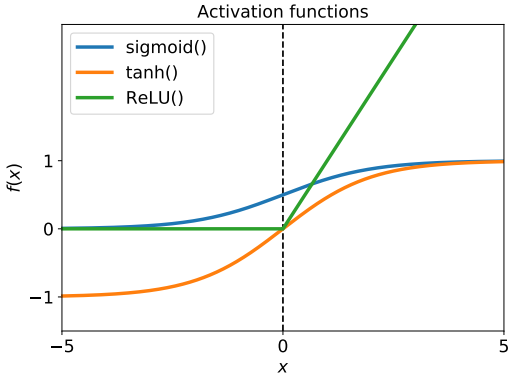
We only need a non-polynomial activation function [Leshno et al., 1993].

Activation Functions

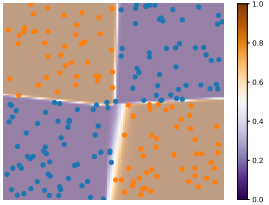
We can use different activation functions. In order for the MLP to be powerful we need **non-linear functions**.

Popular activation functions are:

- logistic function (= sigmoid) $\sigma(x) := (1 + \exp(-x))^{-1}$
- $\tanh(x) \qquad (= 2 \cdot \sigma(x) - 1)$
- **Re**ctified **L**inear **U**nit $f(x) = \max(0, x)$

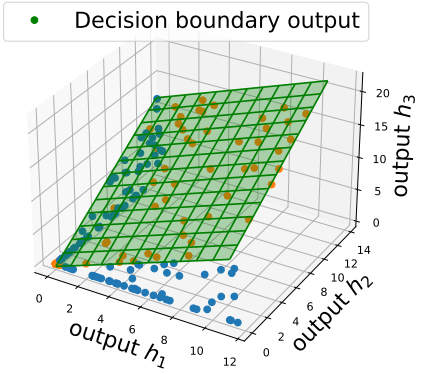


Single hidden layer MLP for XOR



Input layer | ReLU activation | Sigmoid activation

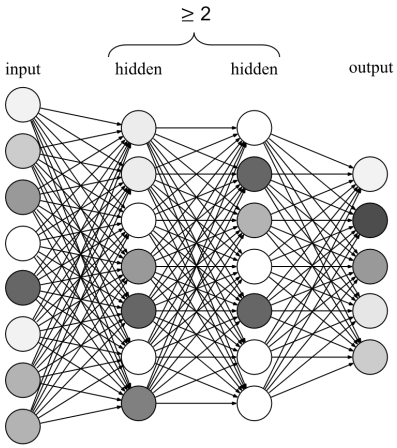
Representation of data points at hidden layer:



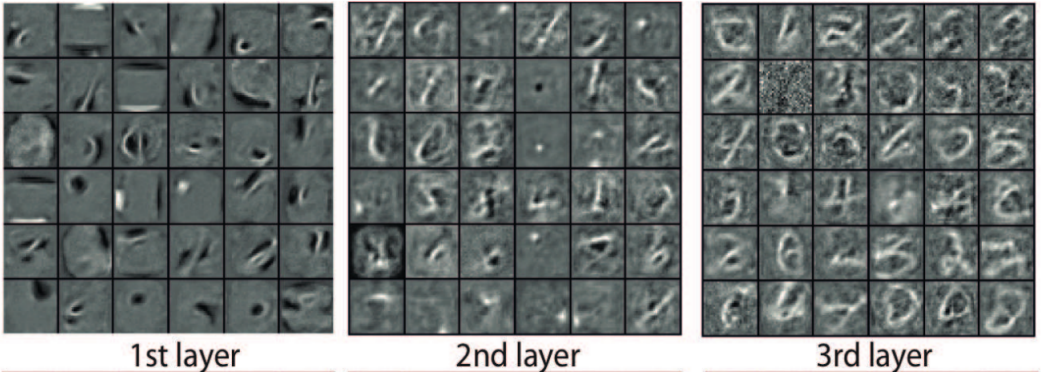
$\rightarrow$ hidden layer learns representation in feature space

Deep Learning

We use the name *Deep Neural Networks (DNN)* when there are 2 or more hidden layers

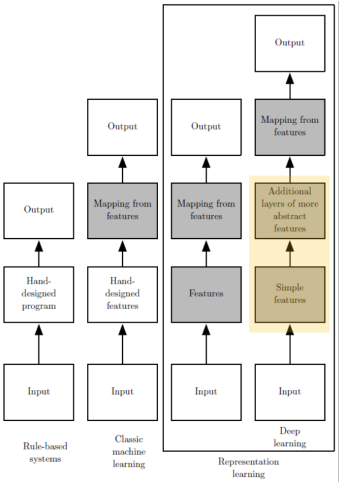


What do the hidden layers learn?



Erhan et al. [2010] (the pictures show inputs that maximize the activation of a given unit)

Deep learning

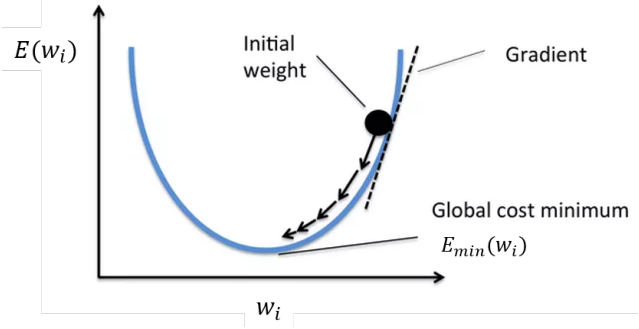


Automatic feature extraction

How do we train the weights of a neural network?

Gradient Descent

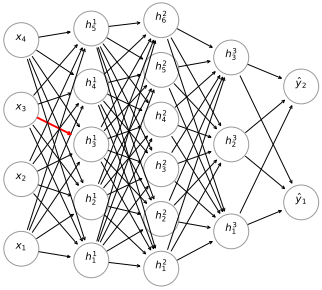
Learning by gradient descent in the space of parameters



$$w_{ij} \leftarrow w_{ij} - \eta \frac{\partial E}{\partial w_{ij}}$$

where E is some error function and η is the learning rate.

How to get gradient for every weight

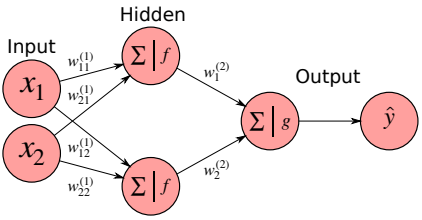


we use the
Chain Rule:

$$\frac{\partial(f \circ g)(x)}{\partial x} = \frac{\partial f(g(x))}{\partial g(x)} \frac{\partial g(x)}{\partial x}$$

$(f \circ g)(x)$ is equivalent to writing $f(g(x))$

Applying the chain rule



$$f(x) = \sigma(x) = \frac{\exp(x)}{1 + \exp(x)}$$

$$g(x) = x$$

$$E = \frac{1}{2}(y - \hat{y})^2$$

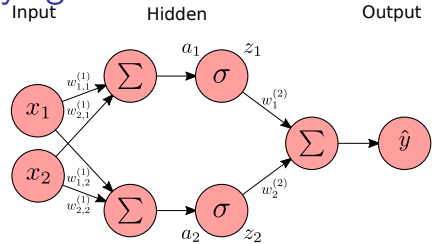
The MLP calculates:

$$\hat{y} = g\left(\sum_i w_i^{(2)} f(\sum_j w_{j,i}^{(1)} x_j)\right)$$

in matrix notation:

$$= g\left(\boldsymbol{w}^{(2)^\top} f(W^{(1)^\top} X)\right)$$

Applying the chain rule



Warning: a_i and z_i are used in the opposite way in many books

$$f(x) = \sigma(x) = \frac{\exp(x)}{1+\exp(x)}$$

$$\sigma(x)' = \sigma(x)(1-\sigma(x))$$

$$E = \frac{1}{2}(\hat{y}-y)^2$$

$$\hat{y} = w_1^{(2)} \underbrace{\sigma(\underbrace{w_{11}^{(1)}x_1 + w_{21}^{(1)}x_2}_{a_1})}_{z_1} + w_2^{(2)} \underbrace{\sigma\underbrace{(w_{12}^{(1)}x_1 + w_{22}^{(1)}x_2)}_{a_2}}_{z_2}$$

$$\frac{\partial E}{\partial w_{11}^{(1)}} = \frac{\partial E}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial w_{11}^{(1)}} = \frac{\partial E}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z_1}\frac{\partial z_1}{\partial w_{11}^{(1)}} = \frac{\partial E}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z_1}\frac{\partial z_1}{\partial a_1}\frac{\partial a_1}{\partial w_{11}^{(1)}}$$

Applying the chain rule

Remember:

$$\sigma(x)' = \sigma(x)(1-\sigma(x)), \; E = \frac{1}{2}(\hat{y}-y)^2$$

$$\hat{y} = w_1^{(2)}\sigma(\underbrace{w_{11}^{(1)}x_1 + w_{21}^{(1)}x_2}_{a_1}) + w_2^{(2)}\underbrace{\sigma(w_{12}^{(1)}x_1 + w_{22}^{(1)}x_2)}_{z_2}$$

$$\frac{\partial E}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z_1}\frac{\partial z_1}{\partial a_1}\frac{\partial a_1}{\partial w_{11}^{(1)}}$$

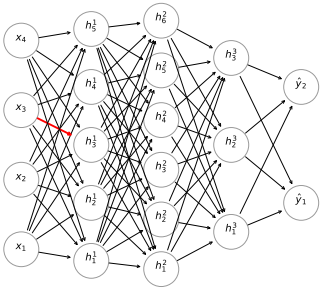
$$\frac{\partial E}{\partial \hat{y}} = (\hat{y}-y)$$

$$\frac{\partial \hat{y}}{\partial z_1} = w_1^{(2)}$$

$$\frac{\partial z_1}{\partial a_1} = \sigma(a_1)(1-\sigma(a_1))$$

$$\frac{\partial a_1}{\partial w_{11}^{(1)}} = x_1$$

$$\Rightarrow \frac{\partial E}{\partial w_{11}^{(1)}} = (\hat{y}-y)\cdot w_1^{(2)}\cdot\sigma(a_1)(1-\sigma(a_1))\cdot x_1$$



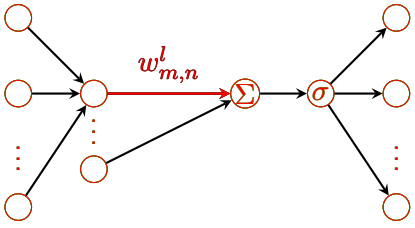
- If we applied the chain rule like above to every weight separately, it would very **slow**
- We can use **backpropagation** to efficiently compute the gradients with respect to all weights at once

Backpropagation

Efficiently using the chain rule

1. Take datapoints X
2. Predict label $\hat{\boldsymbol{y}}$ (forward)
   Save activation value a for every node
3. Backpropagate error $E(\hat{\boldsymbol{y}}, \boldsymbol{y})$
   Calculate $\frac{\partial E}{\partial z^L}$ $(= \frac{\partial E}{\partial \hat{\boldsymbol{y}}})$
   go backwards to calculate gradients
   $$\frac{\partial E}{\partial w^l_{m,n}} = \frac{\partial E}{\partial a^{l+1}_n} \frac{\partial a^{l+1}_n}{\partial w^l_{m,n}}$$
4. Update weights with $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \frac{\partial E}{\partial \boldsymbol{w}}$

Save $a^{l+1}_n = \sum_i w^{l+1}_{i,n} z^l_i$



$$\frac{\partial E}{\partial z^l_n} = \sum_p \frac{\partial E}{\partial a^{l+1}_p} \frac{\partial a^{l+1}_p}{\partial z^l_n}$$

$$\frac{\partial E}{\partial a^l_n} = \frac{\partial E}{\partial z^l_n} \frac{\partial z^l_n}{\partial a^l_n}$$

Key ideas of backpropagation: the chain rule

- For a function $f : \mathbb{R}^n \to \mathbb{R}^m$, denote by $(Df)|_x$ the Jacobi matrix of f at x:

$$((Df)|_x)_{ij} = \frac{\partial f_i}{\partial x_j}(x).$$

- The gradient is just the special case $m = 1$, with $f : \mathbb{R}^n \to \mathbb{R}$ and

$$((\nabla f)|_x)_j = ((Df)|_x)_j = \frac{\partial f}{\partial x_j}(x).$$

- With $f : \mathbb{R}^k \to \mathbb{R}$ and $g : \mathbb{R}^n \to \mathbb{R}^k$, we can write the *chain rule* as

$$\nabla(f \circ g)|_x = D(f \circ g)|_x = (Df)|_{g(x)} \cdot (Dg)|_x = (\nabla f)|_{g(x)} \cdot (Dg)|_x.$$

Always think of the gradient as a one-row vector!

Key ideas of backpropagation: order of multiplication

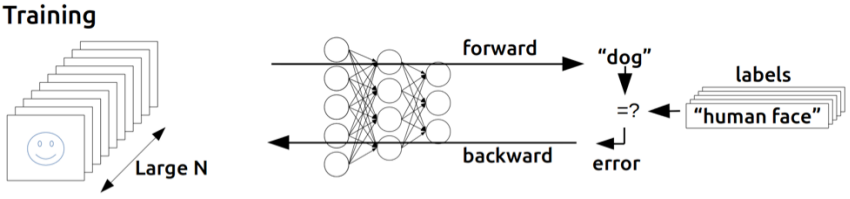
- For matrices A, B and vectors x, y of compatible sizes, we have

$$(AB)x = A(Bx) \text{ and } (y^\top A)B = y^\top(AB).$$

  But for many sizes of A, B, x, y, calculating $A(Bx)$ and $(y^\top A)B$ is computationally much cheaper than $(AB)x$ and $y^\top(AB)$.
- In the MLP setting, we usually have $E \circ \sigma \circ L^k \circ \cdots \circ \sigma \circ L^1$, where E is the loss, σ is the activation, and L^j are linear operations.
- For weight updating, we need $\nabla E \cdot (D\sigma) \cdot (DL^k) \cdot \cdots \cdot (D\sigma) \cdot (DL^1)$.
- Backpropagation means: first calculate $t_1 := \nabla E \cdot (D\sigma)$, then $t_2 := t_1 \cdot (DL^k)$, etc. This is fast!

Summary: Backpropagation

In a neural network with hidden layers there are many more parameters as in the Perceptron. So an efficient way to compute the gradients is needed.



https://devblogs.nvidia.com

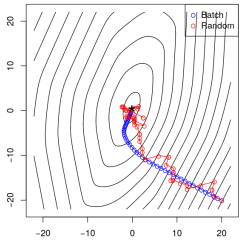
Repeat until convergence/stopping criteria:

1. Forward pass: Apply the current weights to your data
2. Backward pass: Error backpropagation for every individual w_{ij}
3. Update weights: $w_{ij} \leftarrow w_{ij} - \eta \partial E / \partial w_{ij}$

Gradient Descent (GD) vs. Stochastic Gradient Descent (SGD)

Updates the p parameters of the MLP using the entire data set are computationally expensive

- full-batch mode (GD): use all n training samples before updating the weights ($\mathcal{O}(pn)$)
- mini-batch mode (SGD): use a subset of the training data with m samples ($\mathcal{O}(pm)$)

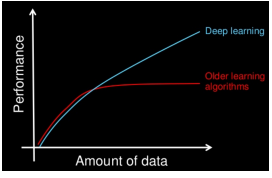


fromTibshirani's lecture

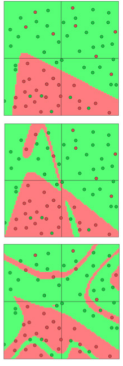
Algorithms for adaptive learning rate η can yield speed-ups (e.g. Adam, many more exist)

Practical considerations

- MLP is a standard feed-forward artificial neural network (ANN)
- Need a lot of data to work
- Best for high-dimensional datasets
- MLPs tend to overfit easily
  - Techniques to avoid overfitting: dropout, regularization, early stopping, ...



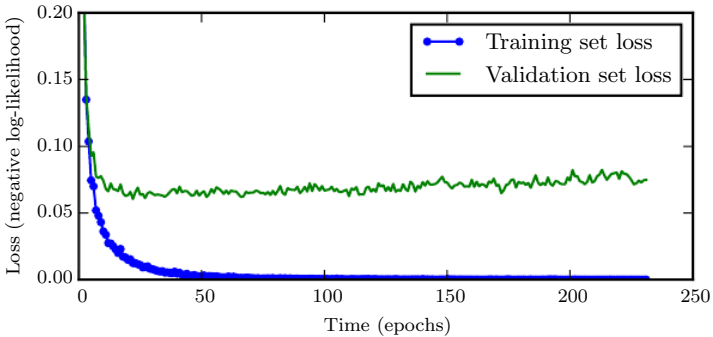
from Andrew Ng



Output of MLP with 1,2 and 4 hidden layers

from https://www.kdnuggets.com

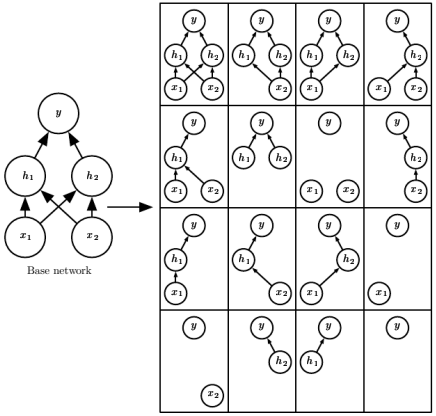
How to prevent overfitting: Early Stopping



"The algorithm terminates when no parameters have improved over the best recorded validation error for some pre-specified number of iterations" [Goodfellow et al., 2016]

How to prevent overfitting: Dropout

- Randomly set weights to 0 temporarily during training
- Makes the network more robust



[Goodfellow et al., 2016]

When training a neural network from scratch there are many parameters you need to choose!
This is mostly done by **trial and error**.[1]

- Which regularization
- Which architecture
- Which activation function
- Which learning rate
- ...

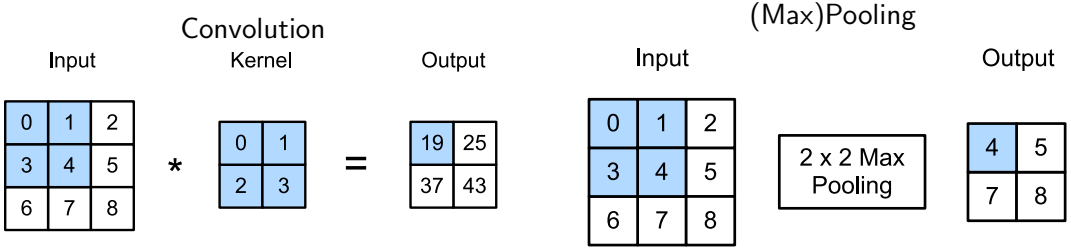
You can also use grid search for some parameters.

[1] There are many efforts to go about it more systematically, but no really established method yet.

Let's see some more advanced examples of neural networks

Convolutional neural networks (CNN)

Convolutional neural networks (CNNs) are widely used in computer vision, for example in object recognition. They usually consist of various convolutional and pooling layers:



[Zhang et al., 2019]

Other architectures: convolutional neural networks

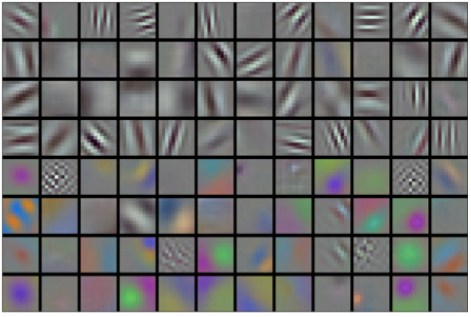


Fig. 9.1.1: Image filters learned by the first layer of AlexNet

Zhang et al. [2019]

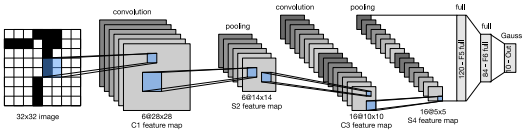
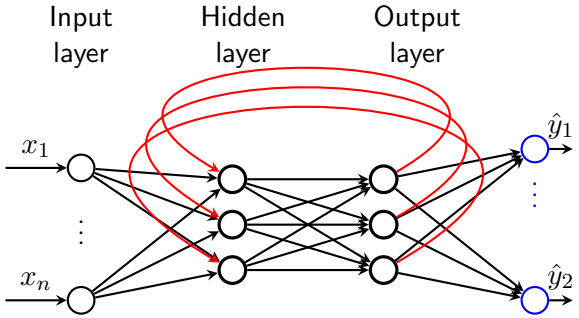


Fig. 8.6.1: Data flow in LeNet 5. The input is a handwritten digit, the output a probabilitiy over 10 possible outcomes.

Zhang et al. [2019]

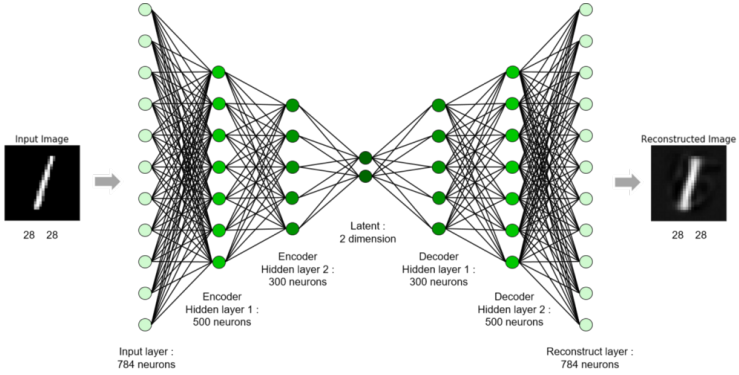
Recurrent Neural Network

- Used for time-series data
- Can use recurrence for representing states



Auto-encoders

- Used for unsupervised learning
- Finds low-dimensional representation of data



Summary I

- Multilayer perceptrons/neural networks are very powerful
- More complicated to train than linear models
  - Non-convex problem
  - Use stochastic gradient descent (SGD)
- Many choices need to be made
- Training can be costly
  - This is because we have many parameters and big datasets
- The field of neural networks/deep learning is developing and changing very quickly
- Trial and error (no complete theory of neural networks exists)!

Further sources

Open source books and lectures for further reading (and watching):

- Deep Learning by Goodfellow, Bengio & Courville (2016), free online book
- Dive into Deep Learning by Zhang, Lipton, Li & Smola (2019)
  with many coding examples
- 3Blue1Brown videos on MLPs and Backpropagation Different Notation!
- Video Lectures by Geoff Hinton

References

L. Bottou. Large-scale machine learning with stochastic gradient descent. In Y. Lechevallier and G. Saporta, editors, *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT'2010)*, pages 177–187, Paris, France, 2010. Springer.

D. Erhan, A. Courville, and Y. Bengio. Understanding representations learned in deep architectures. *Department dInformatique et Recherche Operationnelle, University of Montreal, QC, Canada, Tech. Rep*, 1355:1, 2010.

I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*, volume 1. MIT press Cambridge, 2016.

K. Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991.

M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, 6(6):861–867, 1993.

H. Robbins and S. Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, 22(3):400—407, 1951.

A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola. *Dive into Deep Learning*. 2019. `http://www.d2l.ai`.