Machine Learning for Networks: Neural Networks

Andrea Araldo

September 21, 2023

ML methods

	ML task	Linear Regression	Logistic Regression	Tree-based learning	Neural Networks	k-Neirest Neighbors
Supervised	Regression	x			x	
	Classification		x	x	x	
Unsupervised	Clustering				x	x
	Dimensionality reduction				x	
	Anomaly detection			x	x	x
	Recommender Systems				x	

- Structure of NNs
- Training (backpropagation)
- Design choices and hyper-paramters

Section 1

Introduction

Our heritage

BULLETIN OF
MATHEMATICAL BIOPHYSICS
VOLUME 5, 1943

A LOGICAL CALCULUS OF THE
IDEAS IMMANENT IN NERVOUS ACTIVITY

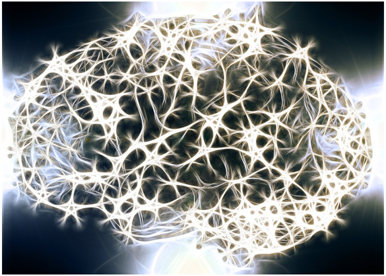
WARREN S. MCCULLOCH AND WALTER PITTS

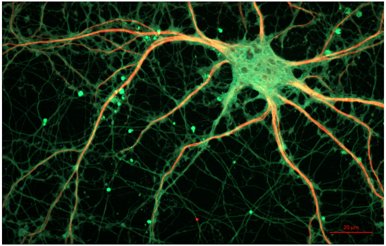
FROM THE UNIVERSITY OF ILLINOIS, COLLEGE OF MEDICINE,
DEPARTMENT OF PSYCHIATRY AT THE ILLINOIS NEUROPSYCHIATRIC INSTITUTE,
AND THE UNIVERSITY OF CHICAGO

18K citations!

Because of the "all-or-none" character of nervous activity, neural events and the relations among them can be treated by means of propositional logic. It is found that the behavior of every net can be described in these terms, with the addition of more complicated logical means for nets containing circles; and that for any logical expression satisfying certain conditions, one can find a net behaving in the fashion it describes. It is shown that many particular choices among possible neurophysiological assumptions are equivalent, in the sense that for every net behaving under one assumption, there exists another net which behaves under the other and gives the same results, although perhaps not in the same time. Various applications of the calculus are discussed.

Neural Network - Human brain





- https://pixabay.com/en/neurons-brain-cells-brain-structure-1739997/

- *Walter Pitts*: logician

- *Warren McCulloc*: neurophysiologist

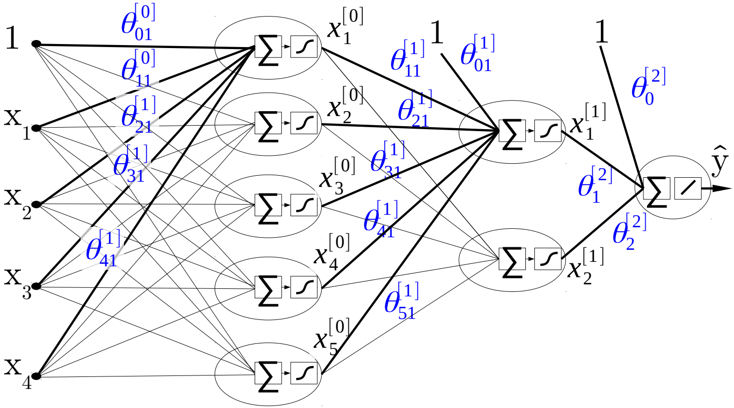
The life of a genius

Walter Pitts:

- When he was 12, he criticized Principia Mathematica from Bertrand Russel.
- Russel invited him to Cambridge University and Pitts refused.

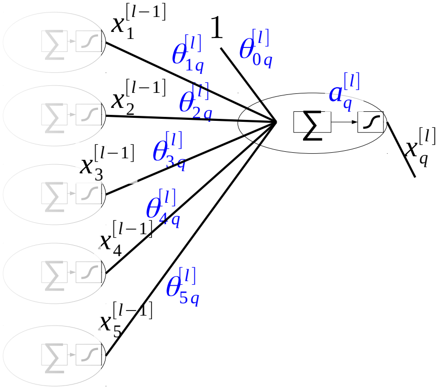
source: Wikipedia

Neural Network - Multi-Layer Perceptron (MLP)



Neural Network - Single neuron

Let us look at the q-th neuron in the l-layer. Do you recognize it?



- Output from the previous layer:

$$\mathbf{x}^{[l-1]} = (1, x_1^{[l-1]}, x_2^{[l-1]}, \dots)$$

- Weights:

$$\theta_q^{[l]} = (\theta_{0q}^{[l]}, \theta_{1q}^{[l]}, \dots)$$

- Weighted input

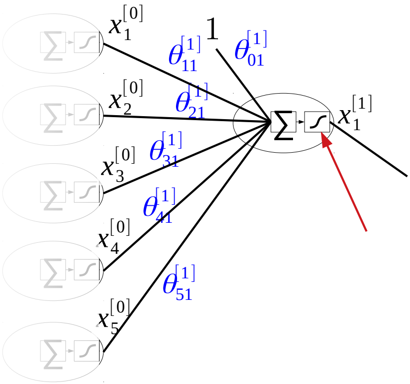
$$a_q^{[l]} = {\theta_q^{[l]}}^T \cdot \mathbf{x}^{[l-1]}$$

- Activation function $\sigma(\cdot)$
- Output:

$$x_q^{[l]} = \sigma(a_q^{[l]})$$

This can be fed to further neurons.

Activation Functions



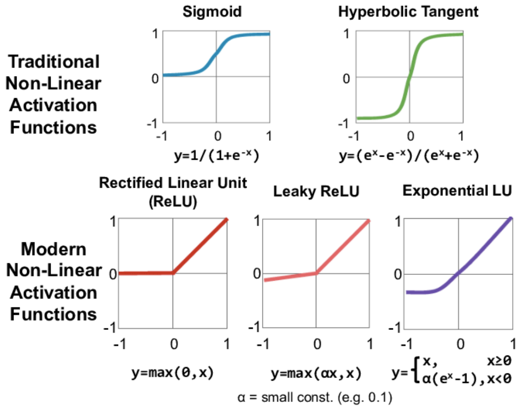
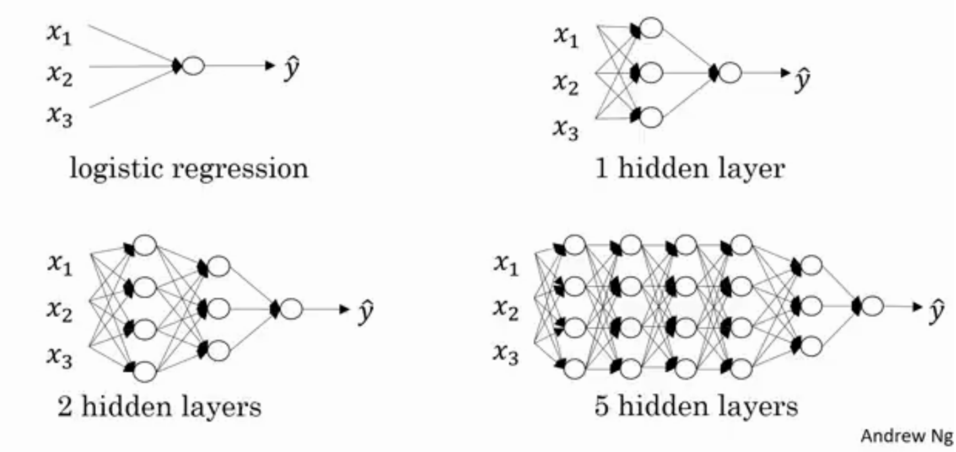


Figure from [SCYE17].

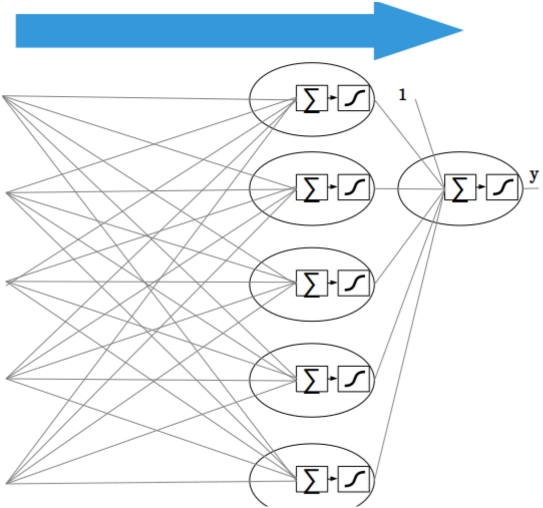
Depth of a NN



Source: Andrew Ng, Deep Neural Networks (see also Fig.10-7 of [Ger19])

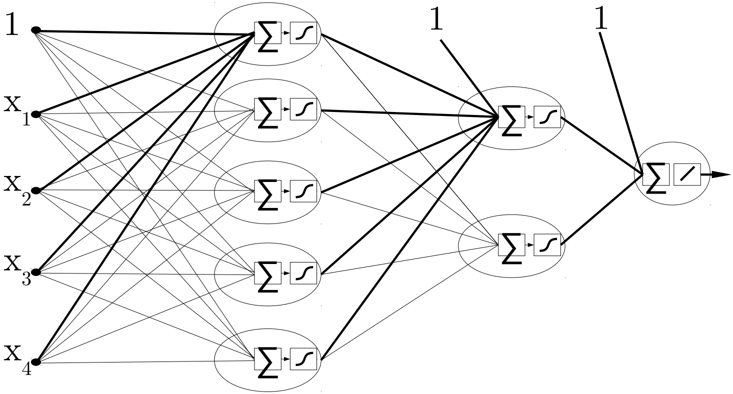
Deep NN: NN with many hidden layers.

Prediction with neural networks

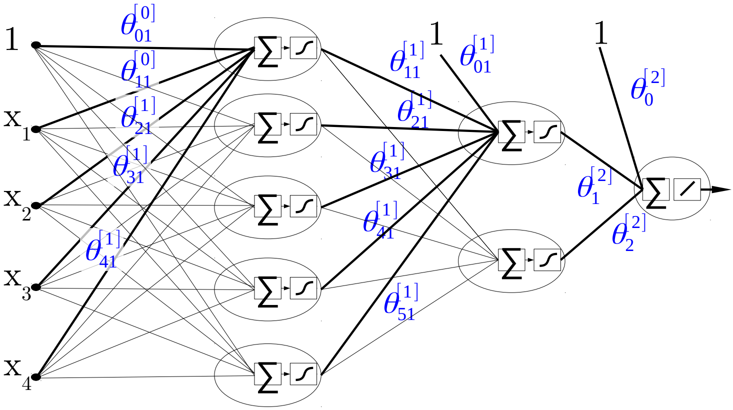


Information is processed from left to right (forward propagation)

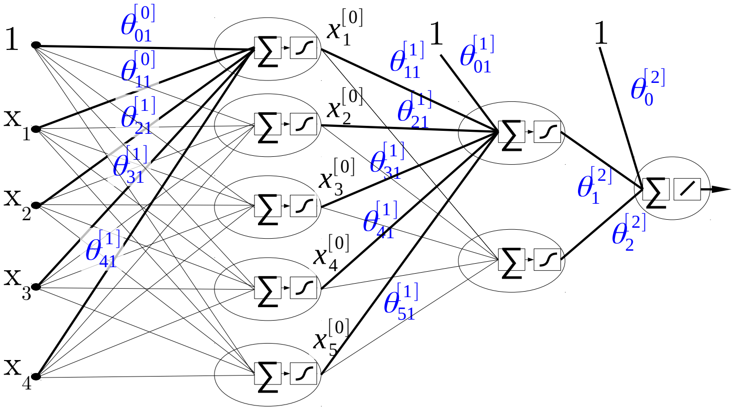
NN for regression



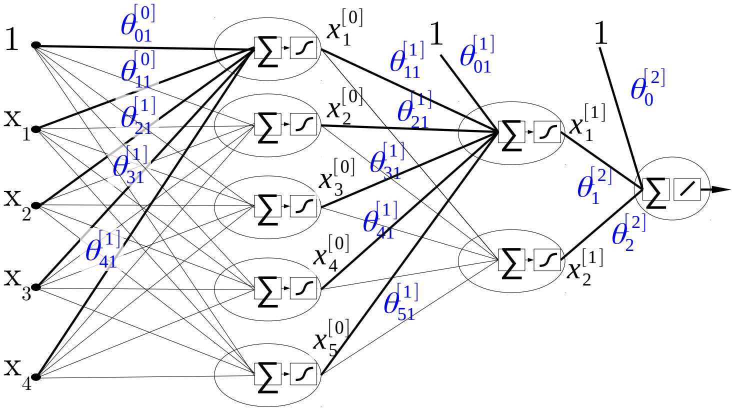
NN for regression



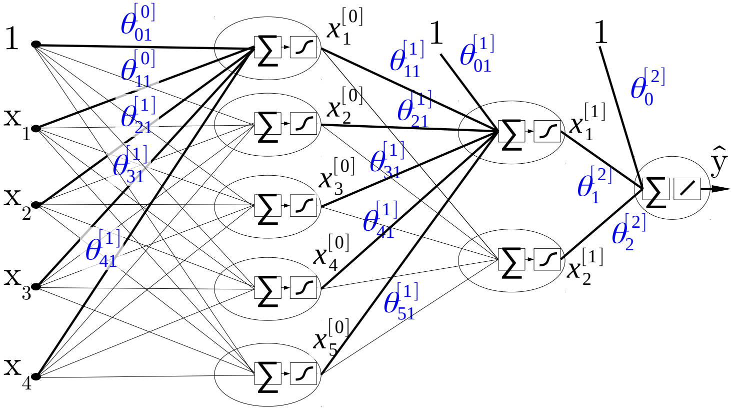
NN for regression



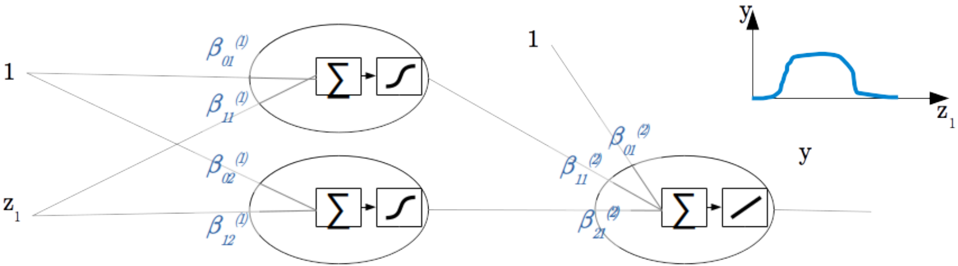
NN for regression



NN for regression



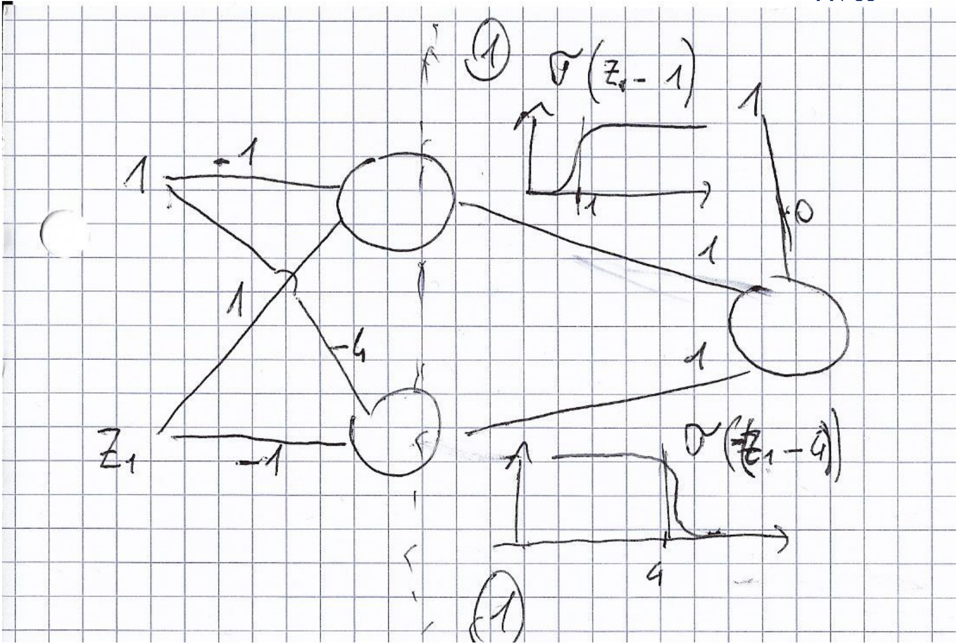
$$\hat{y} = h_\theta(\mathbf{x})$$



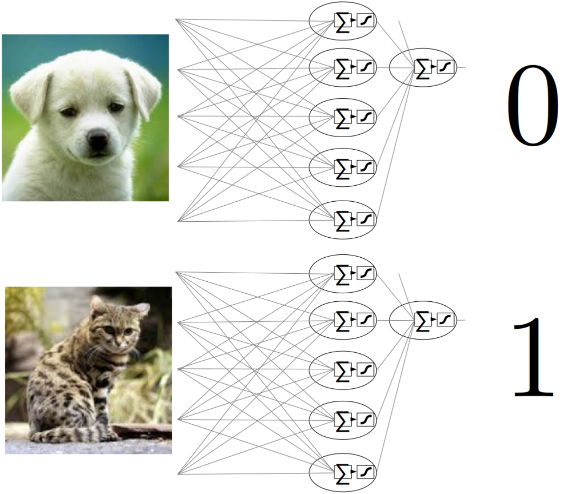
A single hidden layer neural network is a *universal approximator*: any continuous function can be approximated to arbitrary accuracy, provided that there are enough neurons.

Let's write the weights to approximate the function above . . . .

Solution



Binary Classification: Desiderata

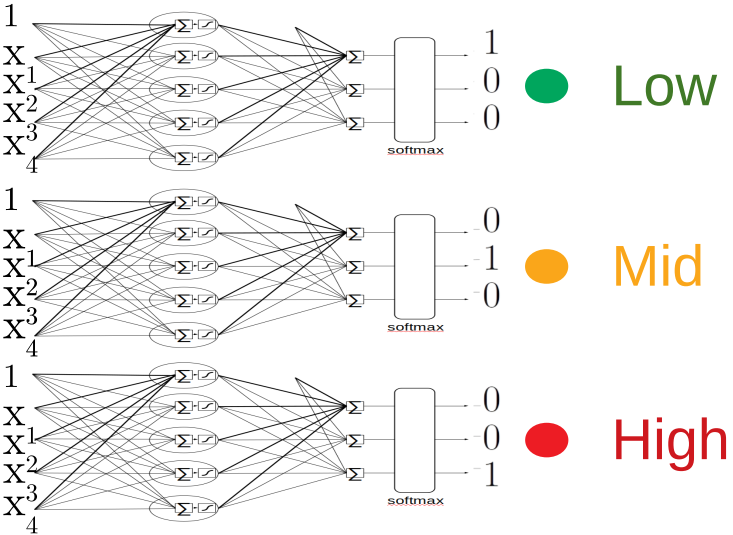


Source: Google

Multiclass Classification: Desiderata

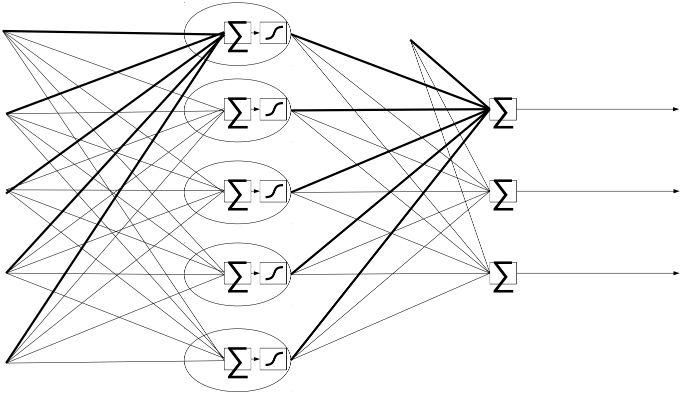
Ex.: TELSTRA **Telstra Kaggle Competition** [Tai17]:

- Features: description of events (location, resource involved, type of event)
- Predict: the severity of fault

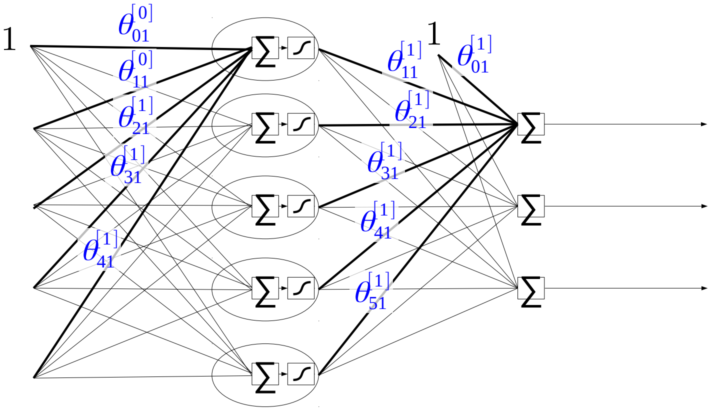


A class is *coded* in a string with one 1.

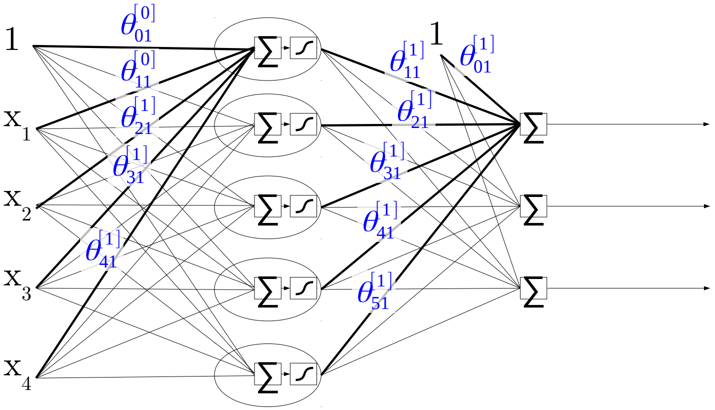
NN for classification



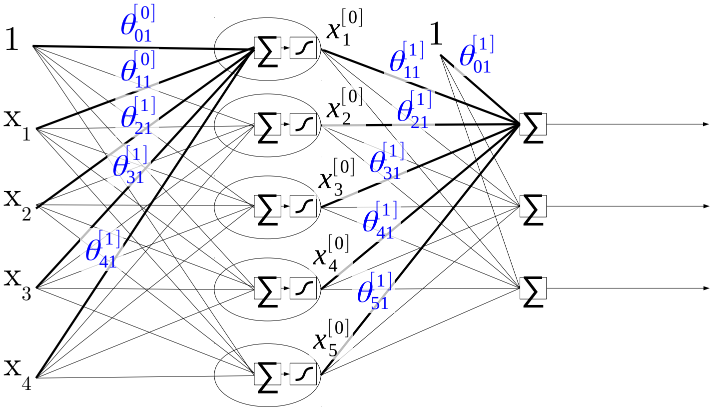
NN for classification



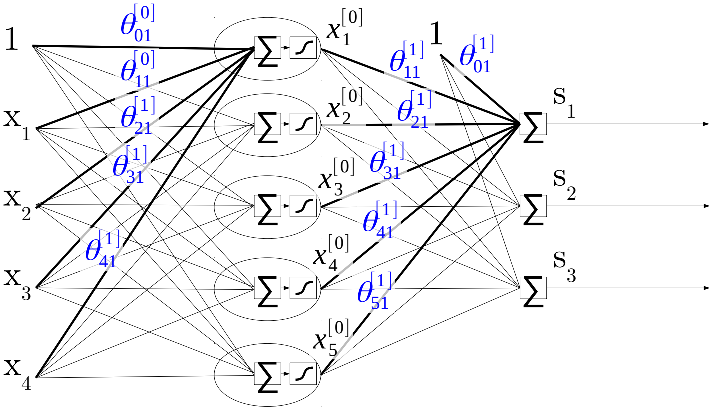
NN for classification



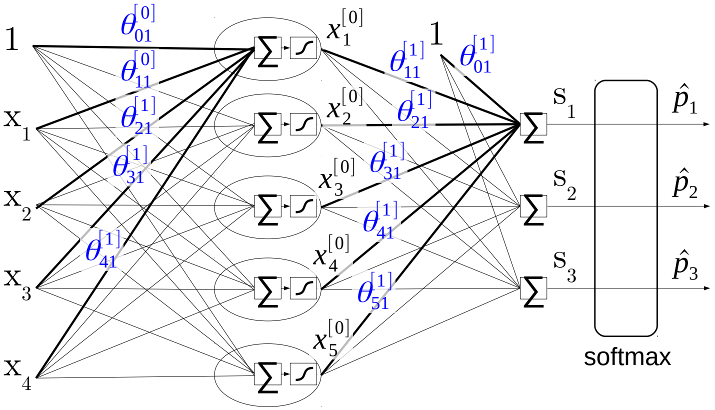
NN for classification



NN for classification



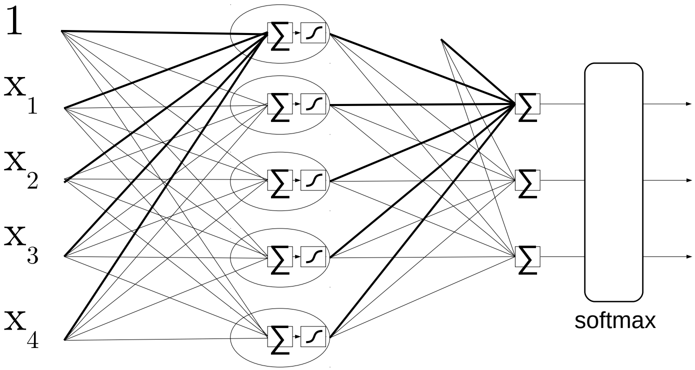
NN for classification



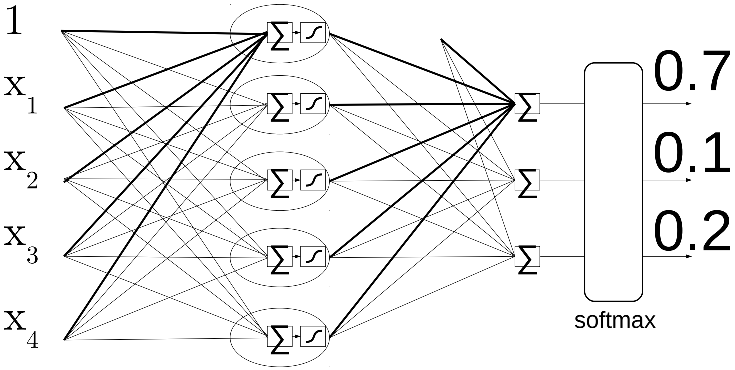
$$\hat{p}_k^{(i)} = \text{softmax}(s_k(\mathbf{x})) = \frac{\exp s_k(\mathbf{x})}{\sum_{z=1}^{K} \exp s_z}$$

$$k^* = \arg\max_k \hat{p}_k^{(i)}$$

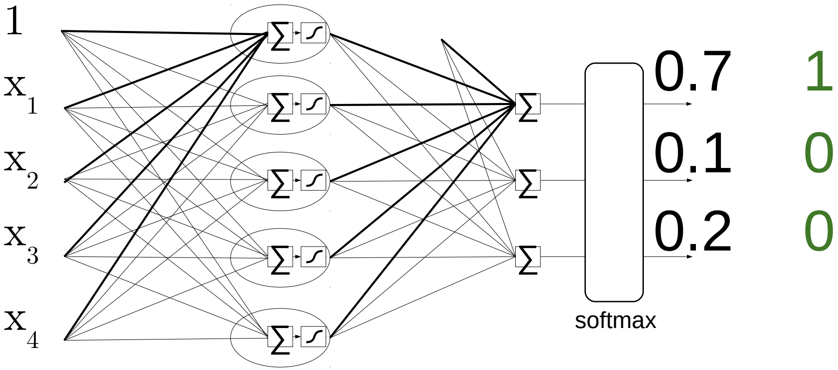
Multiclass Classification: Prediction



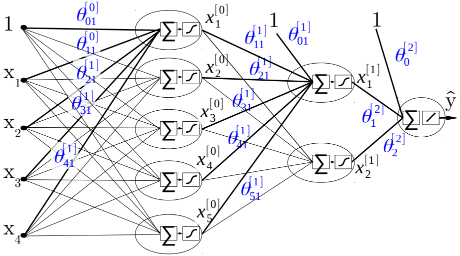
Multiclass Classification: Prediction



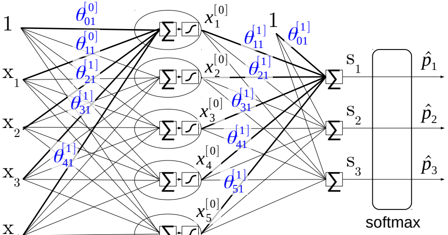
Multiclass Classification: Prediction



Activation functions in the output layer



Regression: no activation function



Classification: softmax

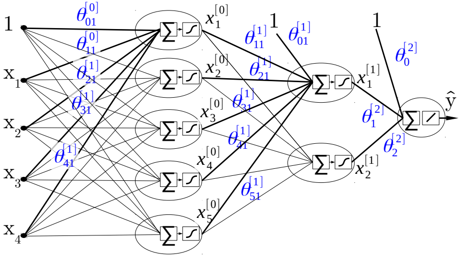
Section 2

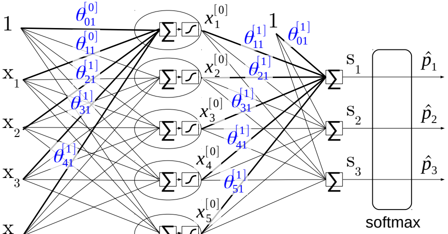
Training (backpropagation)

Training

- A NN is completely specified by the weight matrix θ
- Training: Given a training set of $(\mathbf{X}_{\text{train}}, \mathbf{y}_{\text{train}})$, find the "best" matrix

$$\theta^* \triangleq \arg\min_{\theta} J(\theta, \mathbf{X}_{\text{train}}, \mathbf{y}_{\text{train}}) = \arg\min_{\theta} \sum_i J(\theta, \mathbf{x}^{(i)}, y^{(i)})$$





Regression:

$$J(\theta, \mathbf{x}^{(i)}, y^{(i)}) = \text{MSE} = (y^{(i)} - \hat{y}^{(i)})^2$$

Classification:

$$J(\theta, \mathbf{x}^{(i)}, y^{(i)}) = \text{cross-entropy} = -\ln \hat{p}_{y^{(i)}}^{(i)}$$

Neural Network - Loss minimization

- Objective: $\min_{\theta} J(\theta, \mathbf{X}, \mathbf{y})$
  - where $(\mathbf{X}, \mathbf{y})$ is the training dataset.
- Initialize θ randomly
  - ... but wisely (see pagg 333-4 of [Ge19] for initialization techniques)
- Gradient Descent: at each iteration

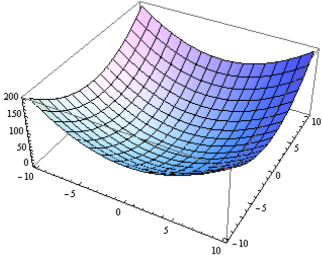
$$\theta := \theta - \eta \cdot \nabla_{\theta} J(\theta, \mathbf{X}, \mathbf{y})$$

- η: learning rate
- Gradient: $\nabla J(\theta, \mathbf{X}, \mathbf{y}) \triangleq \left(\frac{\partial}{\partial {\theta_{qv}}^{[l]}} J(\theta, \mathbf{X}, \mathbf{y}) \right)_{qvl}$
  - ${\theta_{qv}}^{[l]}$: weight of layer l connecting neuron q to neuron v.

Non-convexity

Logistic Regression:
$J(\theta, \mathbf{X}, \mathbf{y})$ derivable and convex (unique minimum)

$\Longrightarrow$ Convergence to minimum guaranteed.



By JackB09 [Public domain], via Wikimedia Commons

Neural Network:
$J(\theta, \mathbf{X}, \mathbf{y})$ derivable but not convex (local minima)

$\Longrightarrow$ Gradient descent may be trapped in local minima

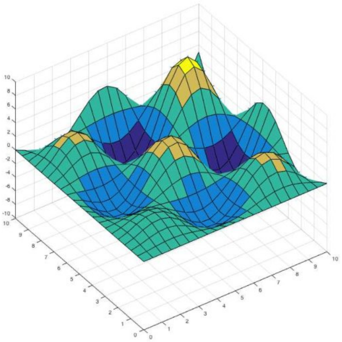
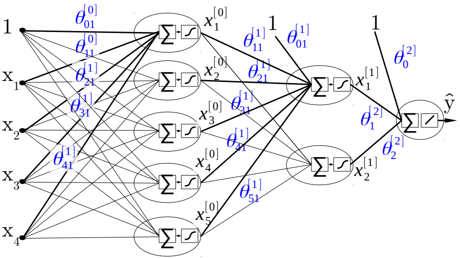
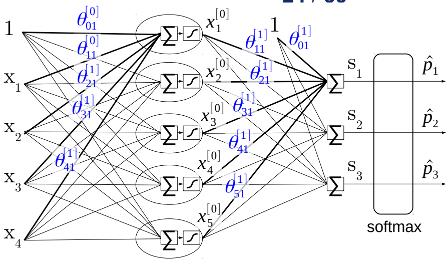


Figure from Bauso, Dario & Gao, Jian & Tembine, Hamidou. (2017). Distributionally Robust Games: f-Divergence and Learning.

Derivatives in the last layer L





Regression:

$$J(\theta, \mathbf{x}^{(i)}, y^{(i)}) = (y^{(i)} - \hat{y}^{(i)})^2$$

$$= \left(\underbrace{y^{(i)} - \overbrace{\sum_q \theta_q{}^{[L]} \cdot x_q^{(i)}{}^{[L-1]}}^{\triangleq \text{Weighted input } a^{[L]}}}_{\varepsilon^{(i)}} \right)^2$$

$$\Longrightarrow \frac{\partial}{\partial \theta_q{}^{[L]}} J(\theta, \mathbf{x}^{(i)}, y^{(i)}) = -2 \cdot \varepsilon^{(i)} \cdot x_q^{(i)}{}^{[L-1]}$$

*****

Classification:

$$J(\theta, \mathbf{x}^{(i)}, y^{(i)}) = -\ln \hat{p}_{k(i)}^{(i)}$$

$$= -\ln \frac{\exp\left(\theta_{k(i)}^{[L]}{}^T \cdot \mathbf{x}^{(i)}{}^{[L-1]}\right)}{\sum_{z=1}^K \exp\left(\theta_z^{[L]}{}^T \cdot \mathbf{x}^{(i)}{}^{[L-1]}\right)}$$

$k(i)$: true class of sample i.

$\theta_z^{[L]}$: vector of weights of the z-th exit.

We can compute $\frac{\partial}{\partial \theta_{qz}^{[L]}} J(\theta, \mathbf{x}^{(i)}, y^{(i)})$

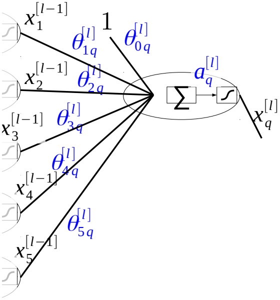
Backpropagation (i.e. $\nabla_\theta J()$ computation)

Let us compute $\frac{\partial J}{\partial \theta_{vq}^{[l]}}(\theta, \mathbf{x}, y)$, $\forall$ training sample $(\mathbf{x}, y)$:

- Consider the q-th neuron of intermediary layer l.
- *Weighted input* (to the neuron):

$$a_q^{[l]} = \theta_q^{[l]T} \cdot \mathbf{x}^{[l-1]} = \sum_z \theta_{zq}^{[l]} \cdot x_z^{[l-1]}$$

- $\frac{\partial J}{\partial \theta_{vq}^{[l]}} = \underbrace{\frac{\partial J}{\partial a_q^{[l]}}}_{\triangleq \text{ error } \delta_q^{[l]}} \cdot \underbrace{\frac{\partial a_q^{[l]}}{\partial \theta_{vq}^{[l]}}}_{x_v^{[l]}} = \delta_q^{[l]} \cdot x_v^{[l-1]}$
- Multivariable chain rule of derivation:
$\delta_q^{[l]} \triangleq \frac{\partial J}{\partial a_q^{[l]}} = \sum_z \frac{\partial J}{\partial a_z^{[l+1]}} \cdot \frac{\partial a_z^{[l+1]}}{\partial a_q^{[l]}} = \sum_z \delta_z^{[l+1]} \cdot \frac{\partial a_z^{[l+1]}}{\partial a_q^{[l]}}$
- Recall that
$a_z^{[l+1]} = \sum_{z'} \theta_{z'z}^{[l+1]} \cdot x_{z'}^{[l]} = \sum_{z'} \theta_{z'z}^{[l+1]} \cdot \sigma(a_{z'}^{[l]})$
- $\Longrightarrow \frac{\partial a_z^{[l+1]}}{\partial a_q^{[l]}} = \theta_{qz}^{[l+1]} \cdot \sigma'(a_q^{[l]})$
- $\Longrightarrow \delta_q^{[l]} = \sigma'(a_q^{[l]}) \cdot \sum_z \delta_z^{[l+1]} \cdot \theta_{qz}^{[l+1]}$

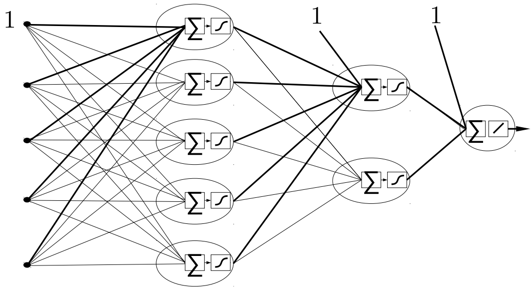


The errors $\delta_z^{[l+1]}$ **propagate back** to layer l.

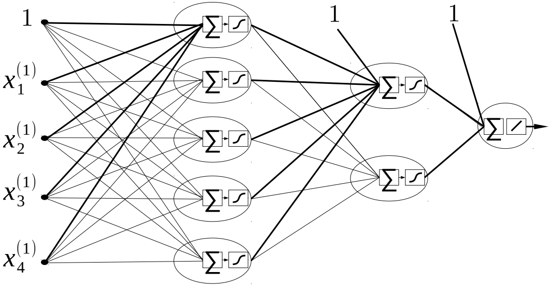
In NN for regression, in the last layer L there is only one neuron and

$$\delta^{[L]} = 1$$

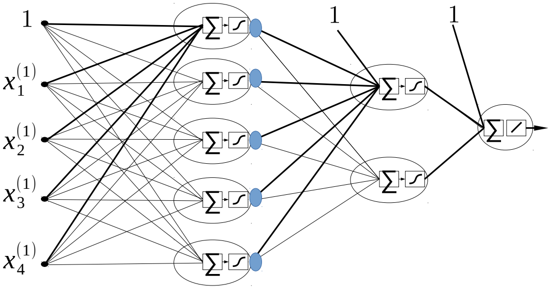
Forward and Backward Propagation



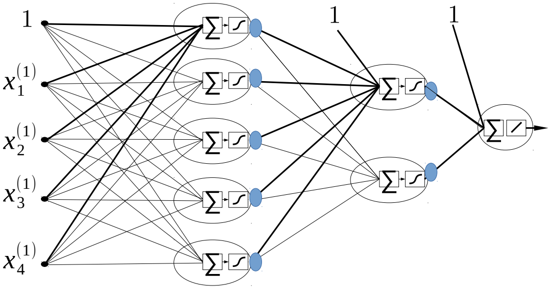
Forward and Backward Propagation



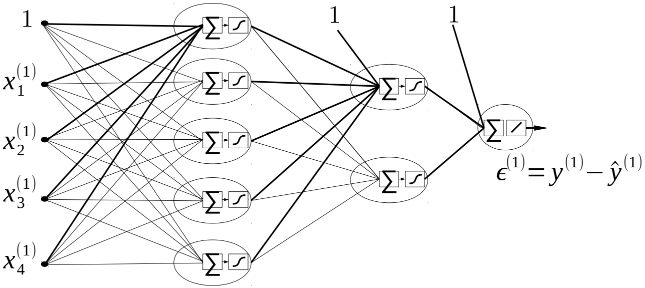
Forward and Backward Propagation



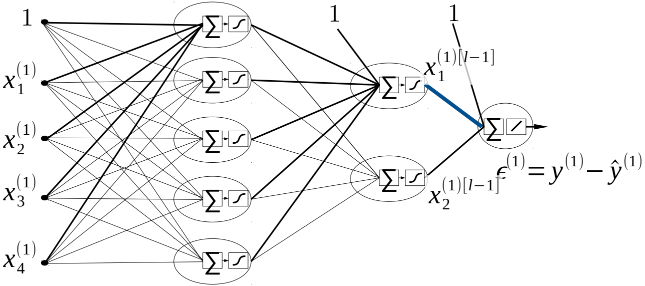
Forward and Backward Propagation



Forward and Backward Propagation



Forward and Backward Propagation

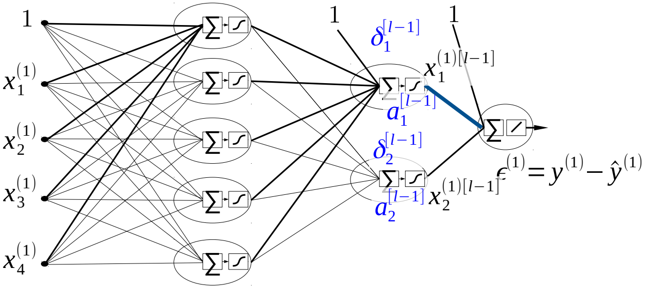


- Compute

$$\frac{\partial}{\partial \theta_q^{[L]}} J(\theta, \mathbf{x}^{(1)}, y^{(1)}) = \varepsilon^{(1)} \cdot x_q^{(1)^{[L-1]}}$$

for all $\theta_q^{[L]}$ in the last layer

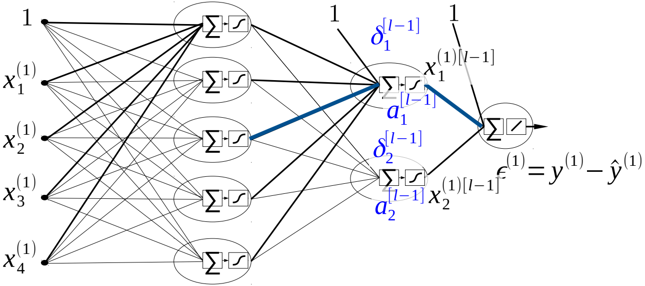
Forward and Backward Propagation



- Compute

$$\delta_q^{[l-1]} = \sigma'(a^{[l-1]}) \cdot \sum_z \delta_z^{[l]} \cdot \theta_{qz}^{[l]}$$

Forward and Backward Propagation



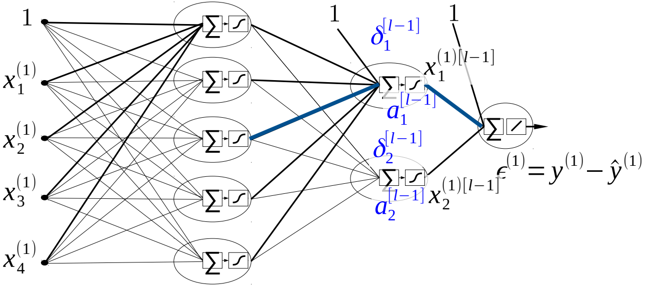
- Compute

$$\delta_q^{[l-1]} = \sigma'(a^{[l-1]}) \cdot \sum_z \delta_z^{[l]} \cdot \theta_{qz}^{[l]}$$

- and

$$\frac{\partial}{\partial \theta_{vq}^{[l-1]}} J = \delta_q^{[l-1]} \cdot {x_v^{(1)}}^{[l-1]}$$

Forward and Backward Propagation

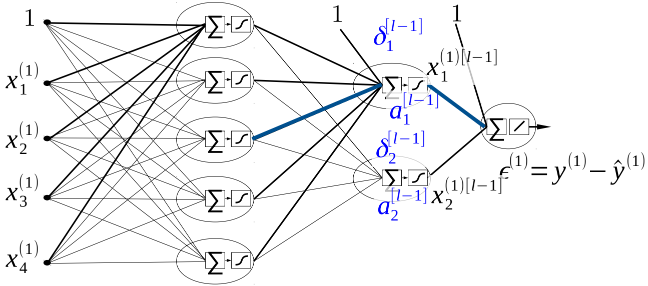


- Do the same for all the weights, backward, to compute

$$\frac{\partial}{\partial \theta_{vq}^{[l-1]}} J = \delta_q^{[l-1]} \cdot x_v^{(1)\,[l-1]}$$

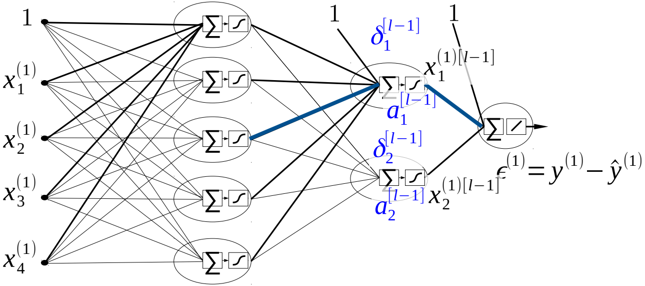
for all weights $\theta_{vq}^{[l]}$

Forward and Backward Propagation



- We thus obtain $\nabla J(\theta, \mathbf{x}^{(i)}, y^{(i)})$

Forward and Backward Propagation



- Do the same for all the samples
- Finally

$$\nabla J(\theta, \mathbf{X}, \mathbf{y}) = \frac{1}{M} \sum_{i=1}^{M} \nabla J(\theta, \mathbf{x}^{(i)}, y^{(i)})$$

And update

$$\theta := \theta - \eta \cdot \nabla J(\theta, \mathbf{X}, \mathbf{y})$$

Training strategies

1. **Full** Gradient Descent
   - Predict $\hat{y}^{(i)}$ for all $\mathbf{x}^{(i)}$ in $\mathscr{D}^{\text{train}}$
   - $\nabla J(\theta, \mathbf{X}, \mathbf{y}) = \frac{1}{M} \sum_{i=1}^{M} \nabla J(\theta, \mathbf{x}^{(i)}, y^{(i)})$
   - Update weights $\theta := \theta - \eta \cdot \nabla J(\theta, \mathbf{X}, \mathbf{y})$
2. **Stochastic** Gradient Descent
(update parameters at each sample)
   - For each sample $\mathbf{x}^{(i)}$
     - Predict $\hat{y}^{(i)}$
     - Compute $J(\theta, \mathbf{x}^{(i)}, y^{(i)})$
     - Assume $\nabla J(\theta, \mathbf{x}^{(i)}, y^{(i)}) \simeq \nabla J(\theta, \mathbf{X}, \mathbf{y})$
     - Update weights $\theta := \theta - \eta \cdot \nabla J(\theta, \mathbf{x}^{(i)}, y^{(i)})$
3. **Batch** Gradient Descent
   - Divide $\mathscr{D}^{\text{train}}$ in batches
   - Update the parameters after predicting each batch.

- **Epoch**: Sequence of predictions on the entire $\mathscr{D}^{\text{train}}$
- How many parameter updates per-epoch (using the 3 strategies)?
- Usually **many epochs** are needed

Multi-Layer Perceptron Implementation

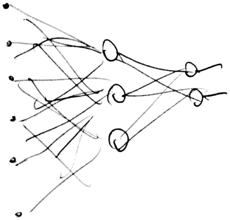
On the right: weights θ (potentiometers adjust via motors)

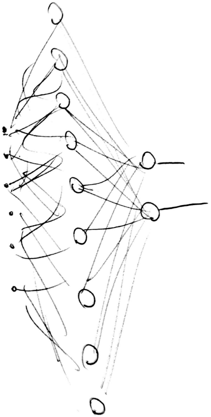
$$\theta := \theta - \eta \cdot \nabla J(\theta, \mathbf{x}^{(i)}, y^{(i)})$$

Figure from [Bis06]

Section 3

Design of NNs



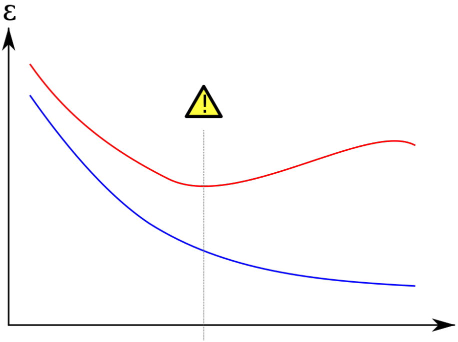


Compression vs. Augmentation.

Some authors use the same number of neurons per layer - pag. 324 of [Ger19]

Model complexity and Overfitting

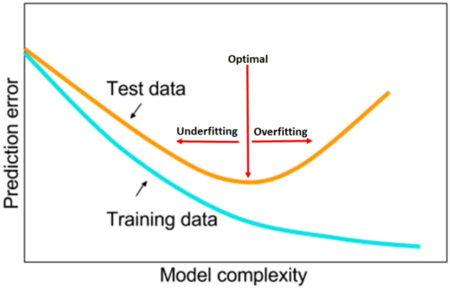
Overfitting



From User:Gringer, Wikipedia

Solution:

- Use smaller architectures
- Regularize
- Early Stopping:
  stop training when the test error does not improve for some consecutive epochs



From [Smi18]

Regularization

- NN with many parameters are **too flexible**: they can approximate weird functions
- To avoid **overfitting** the training data, we must reduce their flexibility
- **Regularization**
- The loss function to minimize during training is
  - For regression

$$J(\theta, \mathbf{X}, \mathbf{y}) = \underbrace{\frac{1}{M}\sum_{i=1}^{M}(y^{(i)} - \hat{y}^{(i)})^2}_{\text{Error term}} + \underbrace{\alpha||\theta||^2}_{\text{Regularization term}}$$

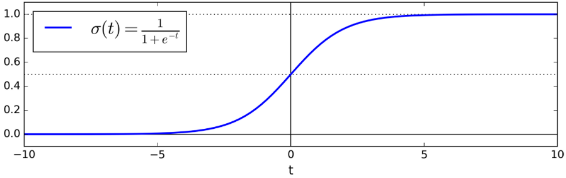
  - For classification

$$J(\theta, \mathbf{X}, \mathbf{y}) = \underbrace{-\frac{1}{M}\sum_{i=1}^{M}\ln \hat{p}_{y^{(i)}}^{(i)}}_{\text{Error term}} + \underbrace{\alpha||\theta||^2}_{\text{Regularization term}}$$

    where $y^{(i)}$ is the true class of sample i

Scaling

- Activation functions like sigmoids are intended to get values in a small range, otherwise they *saturate*.



Ex. If we enter to the neuron $8, 10$ the output is practically the same.

- Scaling is also needed because we regularize NNs
- $\Longrightarrow$ Always scale the dataset (StandardScaler)

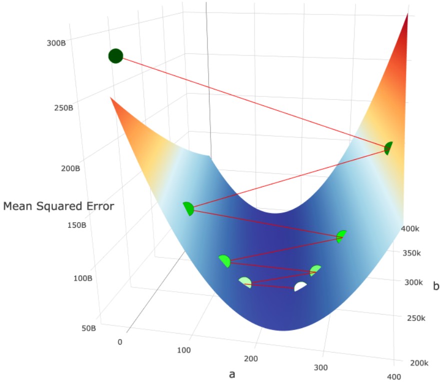
The art of Designing NN

- Hyper-parameters:
  - Architecture
    - Layers? (start with few, increase if needed)
    - Neurons per layer?
  - Learning rate η: too high: noise; too low: slow to converge.
  - How many epochs?
  - Regularization weight α
  - Weight initialization.
  - Batch size.
  - Activation Functions.
- Strategies for tuning
  - Grid search (time consuming)
  - Random search, Bayesian Optimization, Design Space Exploration (time consuming) - see pagg.320-323 of [Ger19]
  - Trial and error, experience (people with less money need to be smarter)

Fixed learning rate

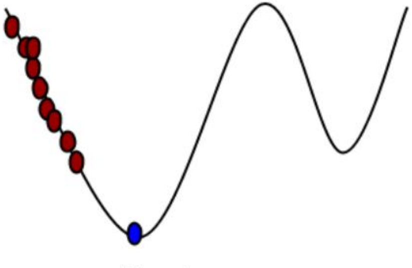
$$\theta := \theta - \eta \cdot \nabla J(\theta, \mathbf{X}, \mathbf{y})$$

η too large

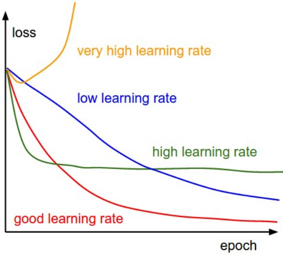


From S. Harrington blog

η too small

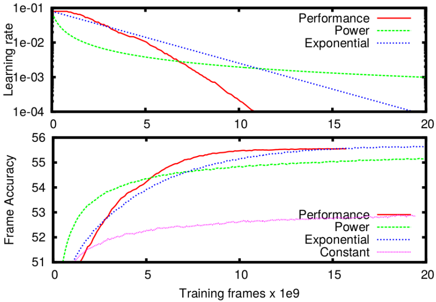


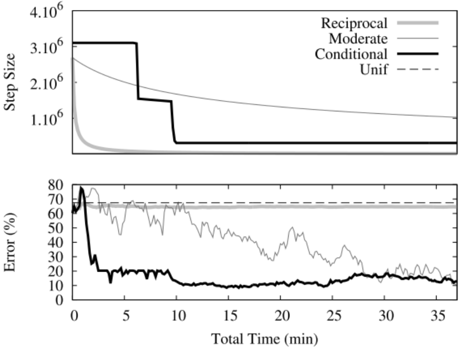
From ProgrammerSought blog.



From CS231 class at Stanford

Start with large learning rates and reduce them after parameter updates





From [Sen13] and [ADR18]

See pagg.359-364 of [Ger19] to know more.

Faster optimizers

Gradient descent is

$$\theta := \theta - \eta \cdot \nabla_{\theta} J(\theta, \mathbf{X}, \mathbf{y})$$

Other optimizers use a different parameter update equation, using gradient in a smarter way.

Most popular: Adaptive Moment Estimation (Adam)

Animated comparison of optimizers.

See pagg. 351-359 of [Ger19] to know more.

Batch size

"Friends, don't let friends use mini-batches larger than 32 "
Yann LeCun tweet, 2018

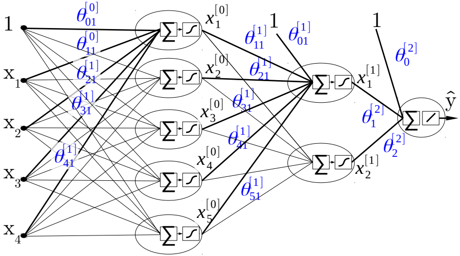
From Wikipedia
Yann LeCun (Facebook, New York University, ACM Turing Award)

Large minibatches

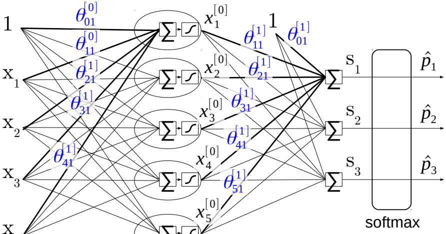
- Allow to use GPU parallelization
- Risk of instability in loss minimization

Activation functions

In the last layer



Regression: no activation function



Classification: softmax

In the hidden layer: The most popular is **relu**

Sigmoid: old school, don't use it

The derivative of the sigmoid is almost zero far from zero

⇒ Vanishing gradient (p 325 of [Ger19])

Updates by gradient descent are too small

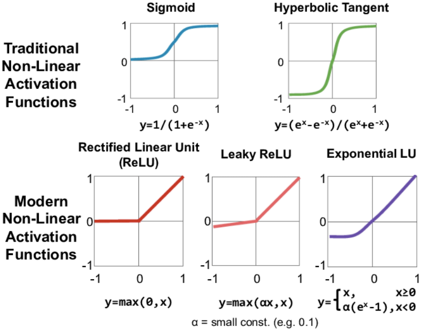


Figure from [SCYE17].

Section 4

Complex architectures

Other architectures

Multi-layer perceptron Other architectures

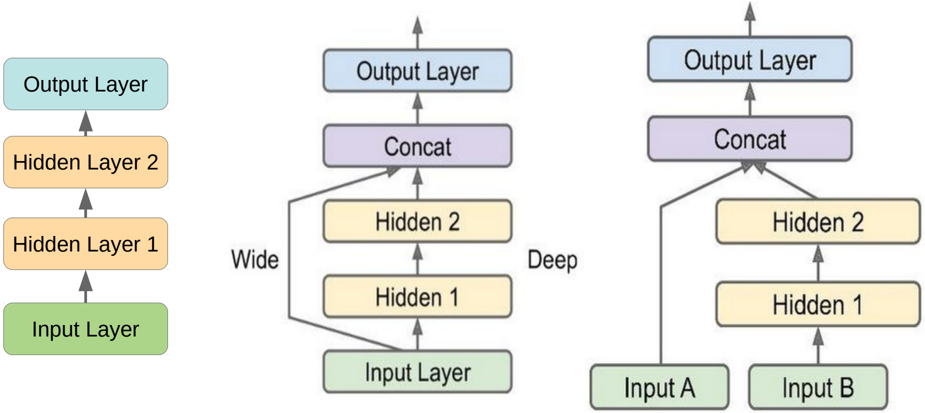


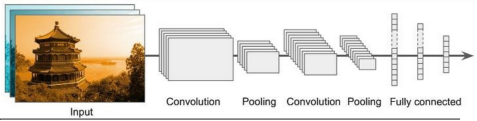
Figure from [Ger19].

- Limit of Multi-Layer Perceptron: all input data are “deformed” by hidden layers.
- Other architectures are able to *bypass* some hidden layer
- Feel free to experiment with them in your project (pagg.308-313 of [Ger19]).

Notable deep Neural Networks

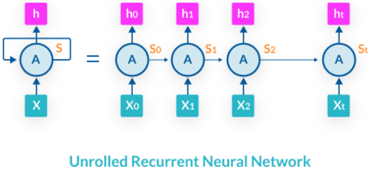
Convolutional NN:

- Image processing



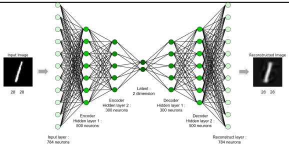
Recurrent NN:

- Time series, language modeling



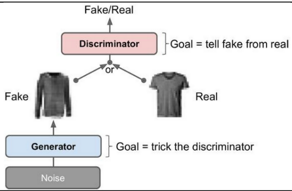
Autoencoder (AE):

- Dimensionality reduction and anomaly detection



Generative Adversarial Networks (GAN):

- To generate images or sounds



Figures from [Ger19], missinglink.ai, medium.com, [Ger19]

How to choose the right Neural Network?

No standard procedure ⇒ Need for intuition, experience and, more importantly, trial and test ⇒ Neural Networks are an art!

However, some rough guidelines are:

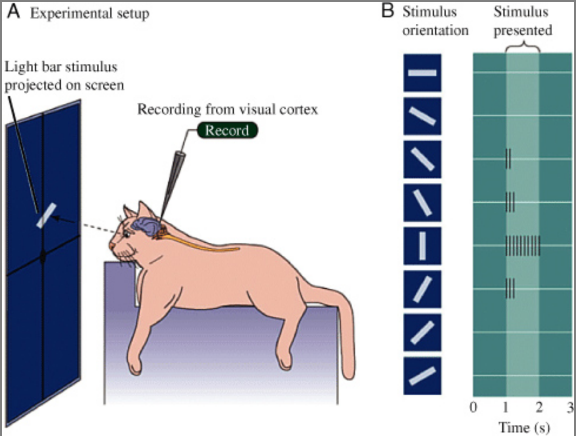
- Image in input ⇒ Convolutional Neural Networks (slide 42)
- Time series in input ⇒ Recurrent Neural Networks (slide 42)
- The output layer depends on the task (regression or classification - slide 19)
- Size: start with a small neural network (few layers, few neurons per layer) and check the test result. Improve this result via Early Stopping and Regularization (slide 31). The result will be your reference baseline. Then, try with bigger architectures and compare the test error with the reference baseline (slide 42)
- Activation function and optimizers: use the latest findings from research (e.g., relu as activation function and Adam as optimizer - slide 9)
- If you have a lot of servers and a lot of time: automatically train several neural networks and get the best after some days / weeks! (grid search, randomized search)

Note that guidelines are continuously broken/replaced, as deep learning progresses!

Let's code ...

Go to notebook 04.neural-networks.ipynb

A celebrity among cats: Hubel and Wiesel (Nobel prize ’81) cat (Harvard)

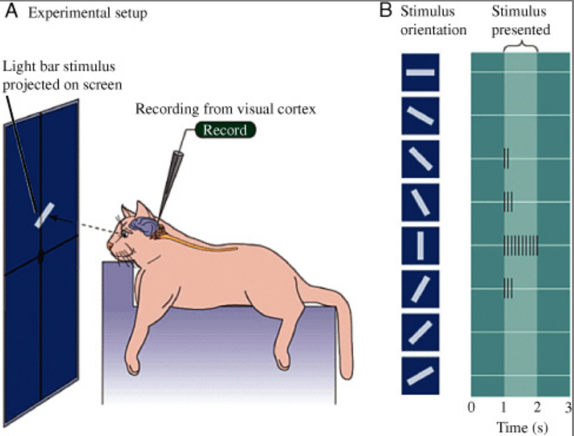


Source: Purves, Brains: How They Seem to Work

Source: youtube

Convolutional NN

A celebrity among cats: Hubel and Wiesel (Nobel prize ’81) cat (Harvard)

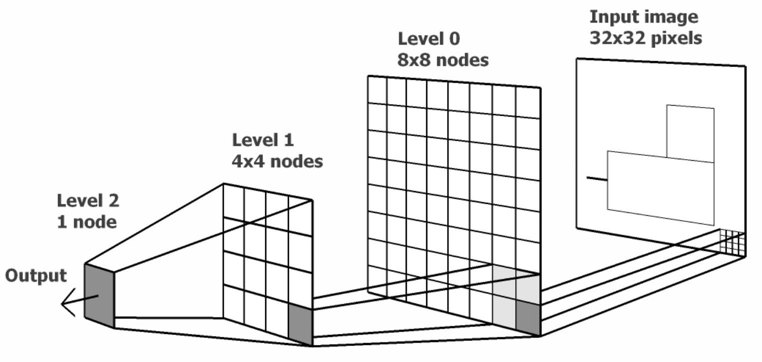


Source: Purves, Brains: How They Seem to Work

Source: youtube

Convolutional NN

- Instead of representing neurons stacked in columns, for image recognition it is easier to imagine them organized in matrices
- No difference in terms of mathematics



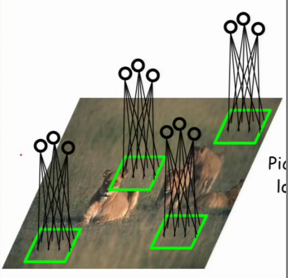
Source: Saulius Garalevicius 2010

Convolutional NN

Main idea

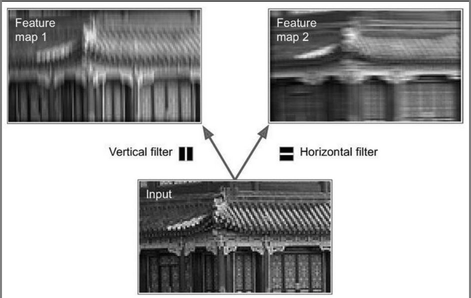
- In “classic” NN, we let it learn “wild” by
  - drawing all weights
  - let the weights take any value
- Can we learn from the cat?
  - Add structure to the architecture of the NN
  - Add constraints to the values of the weights
  - Do this by taking inspiration from the way vision works in living beings

1st Hidden Layer: Feature Maps



Source: Nando de Freitas, Lectures Machine Learning, University of British Columbia

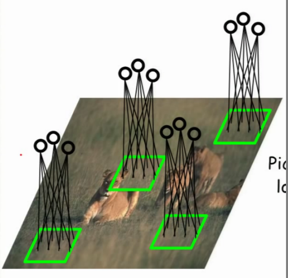
- Each neuron in the 1st layer is only connected to a *patch* (e.g., 5px X 5px) of pixels.
- Several neurons (3 in the example) are attached to the same patch, each looking for a different *feature*. Output ~ 0 or ~ 1.
- *Feature map*: Set of neurons looking for the same feature



From [Ger19]

- A neuron "implements" a vertical filter when its weights are 1 in the center line and 0 elsewhere.

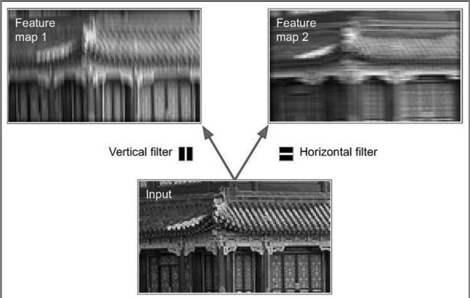
1st Hidden Layer: Feature Maps



Source: Nando de Freitas, Lectures Machine Learning, University of British Columbia

Who decides the filters?

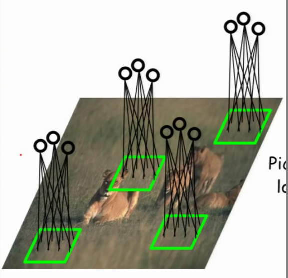
- Each neuron in the 1st layer is only connected to a *patch* (e.g., 5px X 5px) of pixels.
- Several neurons (3 in the example) are attached to the same patch, each looking for a different *feature*. Output ~ 0 or ~ 1.
- *Feature map*: Set of neurons looking for the same feature



From [Ger19]

- A neuron "implements" a vertical filter when its weights are 1 in the center line and 0 elsewhere.

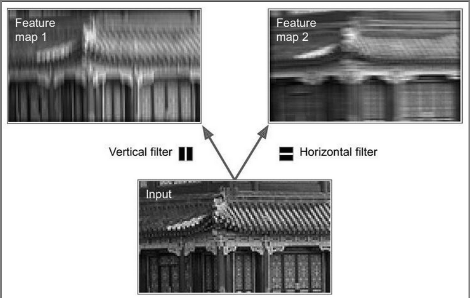
1st Hidden Layer: Feature Maps



Source: Nando de Freitas, Lectures Machine Learning, University of British Columbia

Who decides the filters?
Gradient descent

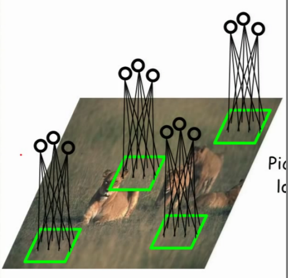
- Each neuron in the 1st layer is only connected to a *patch* (e.g., 5px X 5px) of pixels.
- Several neurons (3 in the example) are attached to the same patch, each looking for a different *feature*. Output ~ 0 or ~ 1.
- *Feature map*: Set of neurons looking for the same feature



From [Ger19]

- A neuron "implements" a vertical filter when its weights are 1 in the center line and 0 elsewhere.

1st Hidden Layer: Feature Maps

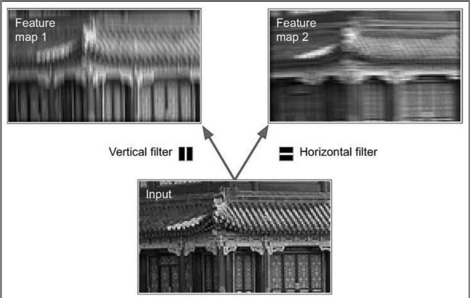


Source: Nando de Freitas, Lectures Machine Learning, University of British Columbia

Who decides the filters?
Gradient descent

How to force the neurons of a feat.map to look for the same feature?

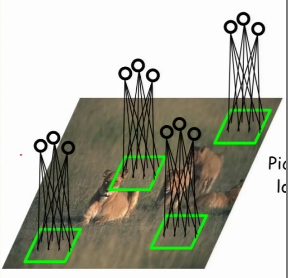
- Each neuron in the 1st layer is only connected to a *patch* (e.g., 5px X 5px) of pixels.
- Several neurons (3 in the example) are attached to the same patch, each looking for a different *feature*. Output ~ 0 or ~ 1.
- *Feature map*: Set of neurons looking for the same feature



From [Ger19]

- A neuron "implements" a vertical filter when its weights are 1 in the center line and 0 elsewhere.

1st Hidden Layer: Feature Maps

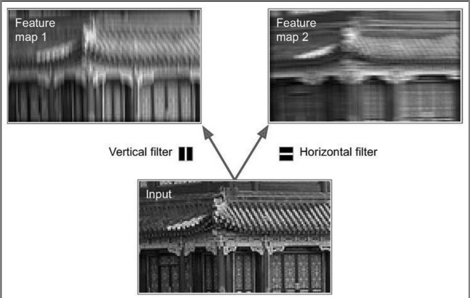


Source: Nando de Freitas, Lectures Machine Learning, University of British Columbia

Who decides the filters?
Gradient descent

How to force the neurons of a feat.map to look for the same feature?
Shared weights

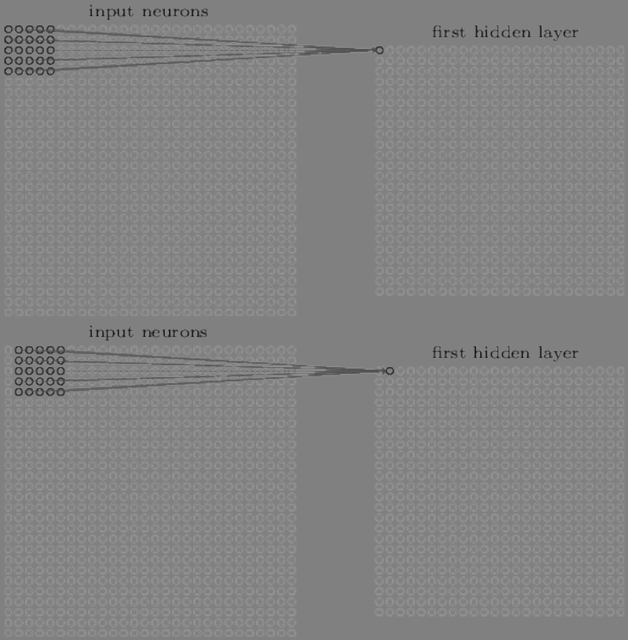
- Each neuron in the 1st layer is only connected to a *patch* (e.g., 5px X 5px) of pixels.
- Several neurons (3 in the example) are attached to the same patch, each looking for a different *feature*. Output ~ 0 or ~ 1.
- *Feature map*: Set of neurons looking for the same feature



From [Ger19]

- A neuron "implements" a vertical filter when its weights are 1 in the center line and 0 elsewhere.

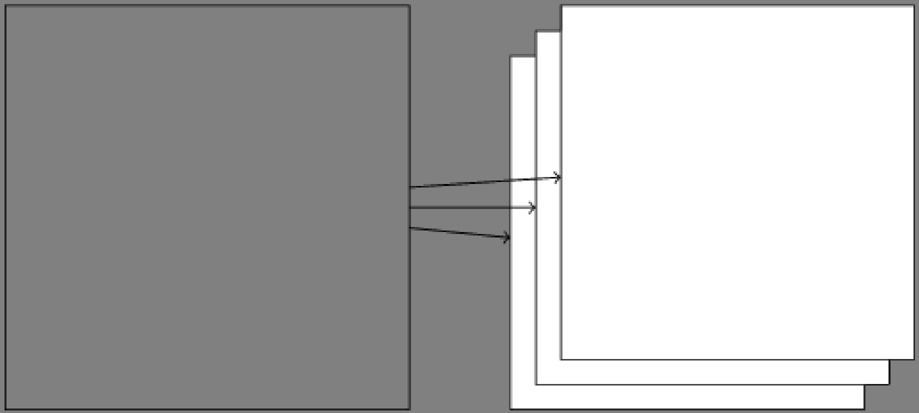
1st Hidden Layer: Feature Map



- Patches “seen” by neurons on the same feature map often overlaps
- Hyperparameter: *stride length* (by how much we slide the patch.)
- Sliding patch

Source: M. Nielsen - Neural Networks and Deep Learning

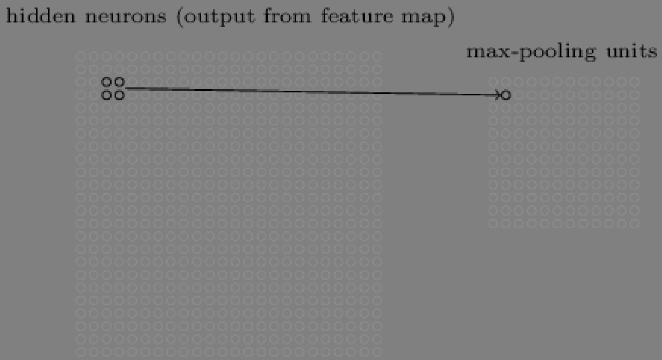
1st Hidden Layer: Feature Map



- We can visually organize the 1st layer as a set of *feature maps*

Source: M. Nielsen - Neural Networks and Deep Learning

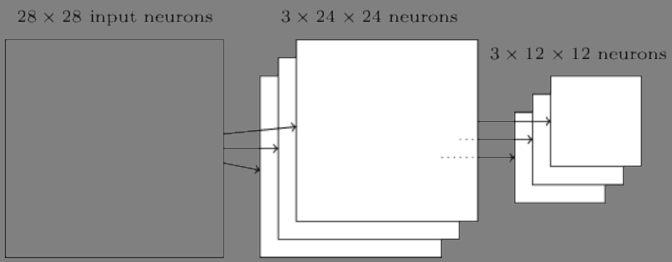
2nd Hidden Layer: Pooling



- Max (or other function - hyperparameter!) of the output of a patch in a feature map
- Meaning: is the feature present in a region of the image?
- No weights to learn here

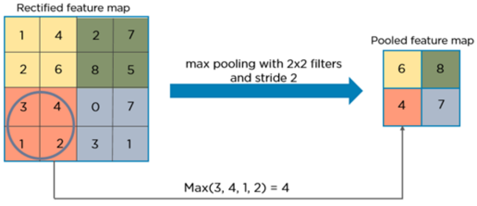
Source: M. Nielsen - Neural Networks and Deep Learning

2nd Hidden Layer: Pooling



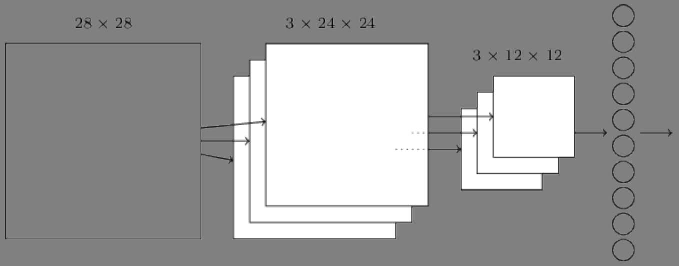
Source: M. Nielsen - Neural Networks and Deep Learning. Pooling layer is the one on the right.

- One pool per feature map
- Similar to convolutional layer, but no weights
  - Just take the average or the max of the patch
- Goal: summarizing features



From video on Simplilearn.

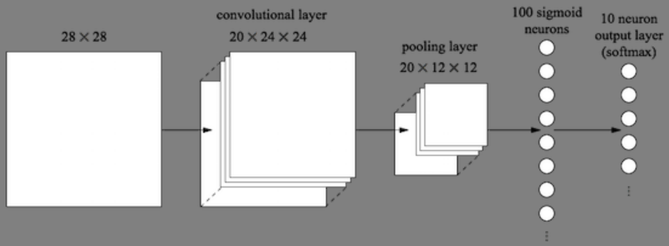
Output Layer



- One category per output neuron
- Ex: bus, car, truck, etc.

Source: M. Nielsen - Neural Networks and Deep Learning

Many possible architectures for convolutional NN



- Ex: Add another hidden layer to summarize information further, before classification

Source: M. Nielsen - Neural Networks and Deep Learning

Recurrent NN

- In all the NN seen so far, all neurons take input that depends on the current sample forward-propagated
- In Recurrent NN, samples are submitted in sequences
- Some neurons is connected to previous samples
- What is this model aimed for?
- Language processing, Speach recognition
- Zaremba (NUY) and Sutskever (Google), “’Learning to execute”
  - Their NN takes the words, one by one, of a (very simple) python scripts
  - It learns to predict the output!

- In all the NN seen so far, all neurons take input that depends on the current sample forward-propagated
- In Recurrent NN, samples are submitted in sequences
- Some neurons is connected to previous samples
- What is this model aimed for?
- Language processing, Speach recognition
- Zaremba (NUY) and Sutskever (Google), “’Learning to execute”
  - Their NN takes the words, one by one, of a (very simple) python scripts
  - It learns to predict the output!

Autoencoders (AE)

- Generative models
- Training: Find the weights submitting many images of cats
- Use: Give it a random input and get a synthetic cat image
- “Few” neurons in the hidden layer. Why?
- Serious use: drug discovery, music generation

Recap

In this lesson

- Structure of NNs
- Training (backpropagation)
- Design choices and hyper-paramters

In next lesson

- Random Forests

References I

[ADR18] Andrea Araldo, György Dán, and Dario Rossi, *Caching Encrypted Content via Stochastic Cache Partitioning*, IEEE/ACM Transactions on Networking **26** (2018), no. 1, 548–561.

[Bis06] Christopher M. Bishop, *Pattern Recognition and Machine Learning*, 2006.

[Ger19] Aurelien Geron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, O'Reilly, 2019.

[SCYE17] Vivienne Sze, Yu Hsin Chen, Tien Ju Yang, and Joel S. Emer, *Efficient Processing of Deep Neural Networks: A Tutorial and Survey*, Proceedings of the IEEE **105** (2017), no. 12, 2295–2329.

[Sen13] Andrew Senior, *An Empirical Study of the Learning Rates in Deep Neural Networks for Speech Recognition*, IEEE International Conference on Acoustics, Speech and Signal Processing, 2013.

References II

[Smi18] Leslie N. Smith, *A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay*, Tech. report, US Naval Research Laboratory, 2018.

[Tai17] D. S. Taimanov, *Network disruption prediction based on neural networks*, 2017.

To know more

- Convolutional Neural Networks for Computer Vision (Ch.11 of [Ger19])
- Christopher M. Bishop. Pattern Recognition and Machine Learning, Springer - Sections 5.1, 5.2, 5.3
- http://neuralnetworksanddeeplearning.com