Introduction to Neural Networks

E. Scornet

September 2020

1. Introduction

2. Neural Network architecture
   - Neurons
   - A historical model/algorithm - the perceptron
   - Going beyond perceptron - multilayer neural networks
   - Neural network training

3. Hyperparameters
   - Activation functions
   - Output units
   - Loss functions
   - Weight initialization

4. Regularization
   - Penalization
   - Dropout
   - Batch normalization
   - Early stopping

5. All in all

Outline

Supervised Learning

Supervised Learning Framework

- Input measurement $\mathbf{X} \in \mathcal{X}$
- Output measurement $Y \in \mathcal{Y}$.
- $(\mathbf{X}, Y) \sim \mathbb{P}$ with $\mathbb{P}$ unknown.
- Training data : $\mathcal{D}_n = \{(\mathbf{X}_1, Y_1), \ldots, (\mathbf{X}_n, Y_n)\}$ (i.i.d. $\sim \mathbb{P}$)

- Often
  - $\mathbf{X} \in \mathbb{R}^d$ and $Y \in \{-1, 1\}$ (classification)
  - or $\mathbf{X} \in \mathbb{R}^d$ and $Y \in \mathbb{R}$ (regression).
- A predictor is a function in $\mathcal{F} = \{f : \mathcal{X} \to \mathcal{Y} \text{ meas.}\}$

Goal

- Construct a good predictor $\widehat{f}$ from the training data.

- Need to specify the meaning of good.
- Classification and regression are almost the same problem!

Loss and Probabilistic Framework

Loss function for a generic predictor

- Loss function : $\ell(Y, f(\mathbf{X}))$ measures the goodness of the prediction of Y by $f(\mathbf{X})$
- Examples:
  - Prediction loss: $\ell(Y, f(\mathbf{X})) = \mathbf{1}_{Y \neq f(\mathbf{X})}$
  - Quadratic loss: $\ell(Y, f(\mathbf{X})) = |Y - f(\mathbf{X})|^2$

Risk function

- Risk measured as the average loss for a new couple:

$$\mathcal{R}(f) = \mathbb{E}_{(X,Y)\sim\mathbb{P}}\left[\ell(Y, f(\mathbf{X}))\right]$$

- Examples:
  - Prediction loss: $\mathbb{E}\left[\ell(Y, f(\mathbf{X}))\right] = \mathbb{P}\left\{Y \neq f(\mathbf{X})\right\}$
  - Quadratic loss: $\mathbb{E}\left[\ell(Y, f(\mathbf{X}))\right] = \mathbb{E}\left[|Y - f(\mathbf{X})|^2\right]$

- Beware: As $\widehat{f}$ depends on $\mathcal{D}_n$, $\mathcal{R}(\widehat{f})$ is a random variable!

Supervised Learning

Experience, Task and Performance measure

- Training data : $\mathcal{D} = \{(\mathbf{X}_1, Y_1), \ldots, (\mathbf{X}_n, Y_n)\}$ (i.i.d. $\sim \mathbb{P}$)
- Predictor: $f : \mathcal{X} \to \mathcal{Y}$ measurable
- Cost/Loss function : $\ell(Y, f(\mathbf{X}))$ measure how well $f(\mathbf{X})$ "predicts" Y
- Risk:

$$\mathcal{R}(f) = \mathbb{E}\left[\ell(Y, f(\mathbf{X}))\right] = \mathbb{E}_X\left[\mathbb{E}_{Y|\mathbf{X}}\left[\ell(Y, f(\mathbf{X}))\right]\right]$$

- Often $\ell(Y, f(\mathbf{X})) = |f(\mathbf{X}) - Y|^2$ or $\ell(Y, f(\mathbf{X})) = \mathbf{1}_{Y \neq f(\mathbf{X})}$

Goal

- Learn a rule to construct a predictor $\widehat{f} \in \mathcal{F}$ from the training data $\mathcal{D}_n$ s.t. the risk $\mathcal{R}(\widehat{f})$ is small on average or with high probability with respect to $\mathcal{D}_n$.

Best Solution

- The best solution f^* (which is independent of $\mathcal{D}_n$) is

$$f^* = \arg\min_{f \in \mathcal{F}} R(f) = \arg\min_{f \in \mathcal{F}} \mathbb{E}\left[\ell(Y, f(\mathbf{X}))\right]$$

Bayes Predictor (explicit solution)

- In binary classification with $0-1$ loss:

$$f^*(\mathbf{X}) = \begin{cases} +1 & \text{if } \mathbb{P}\{Y=+1|\mathbf{X}\} \geq \mathbb{P}\{Y=-1|\mathbf{X}\} \\ & \Leftrightarrow \mathbb{P}\{Y=+1|\mathbf{X}\} \geq 1/2 \\ -1 & \text{otherwise} \end{cases}$$

- In regression with the quadratic loss

$$f^*(\mathbf{X}) = \mathbb{E}\left[Y|\mathbf{X}\right]$$

Issue: Solution requires to know $\mathbb{E}\left[Y|\mathbf{X}\right]$ for all values of $\mathbf{X}$!

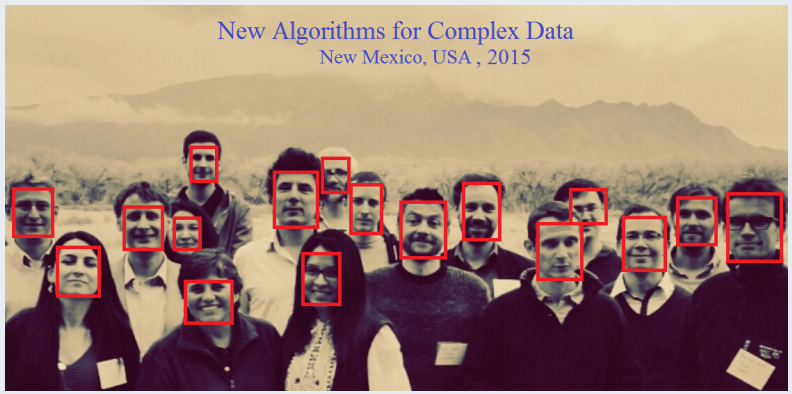
Examples

Spam detection (Text classification)

- Data: email collection
- Input: email
- Output : Spam or No Spam

Examples

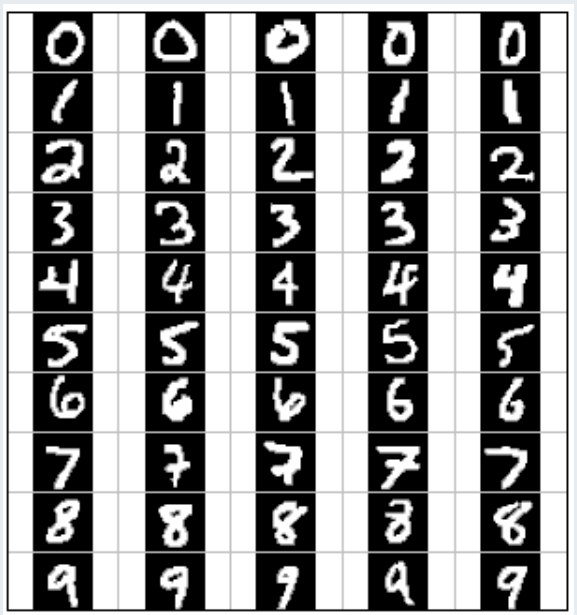
Face Detection



- Data: Annotated database of images
- Input : Sub window in the image
- Output : Presence or no of a face...

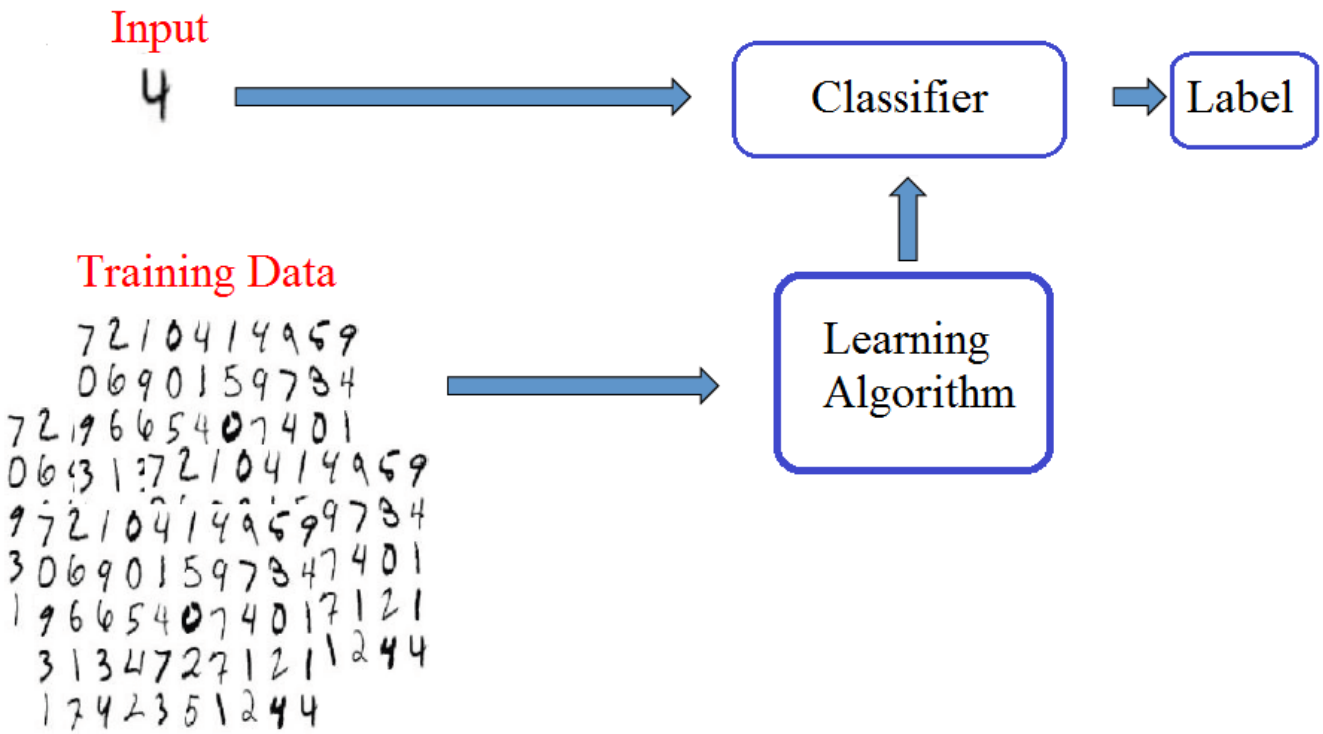
Examples

Number Recognition



- Data: Annotated database of images (each image is represented by a vector of $28 \times 28 = 784$ pixel intensities)
- Input: Image
- Output: Corresponding number

Machine Learning



A definition by Tom Mitchell (http://www.cs.cmu.edu/~tom/)

A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

Unsupervised Learning

Experience, Task and Performance measure

- Training data : $\mathcal{D} = \{\mathbf{X}_1, \ldots, \mathbf{X}_n\}$ (i.i.d. $\sim \mathbf{P}$)
- Task: ???
- Performance measure: ???

- No obvious task definition!

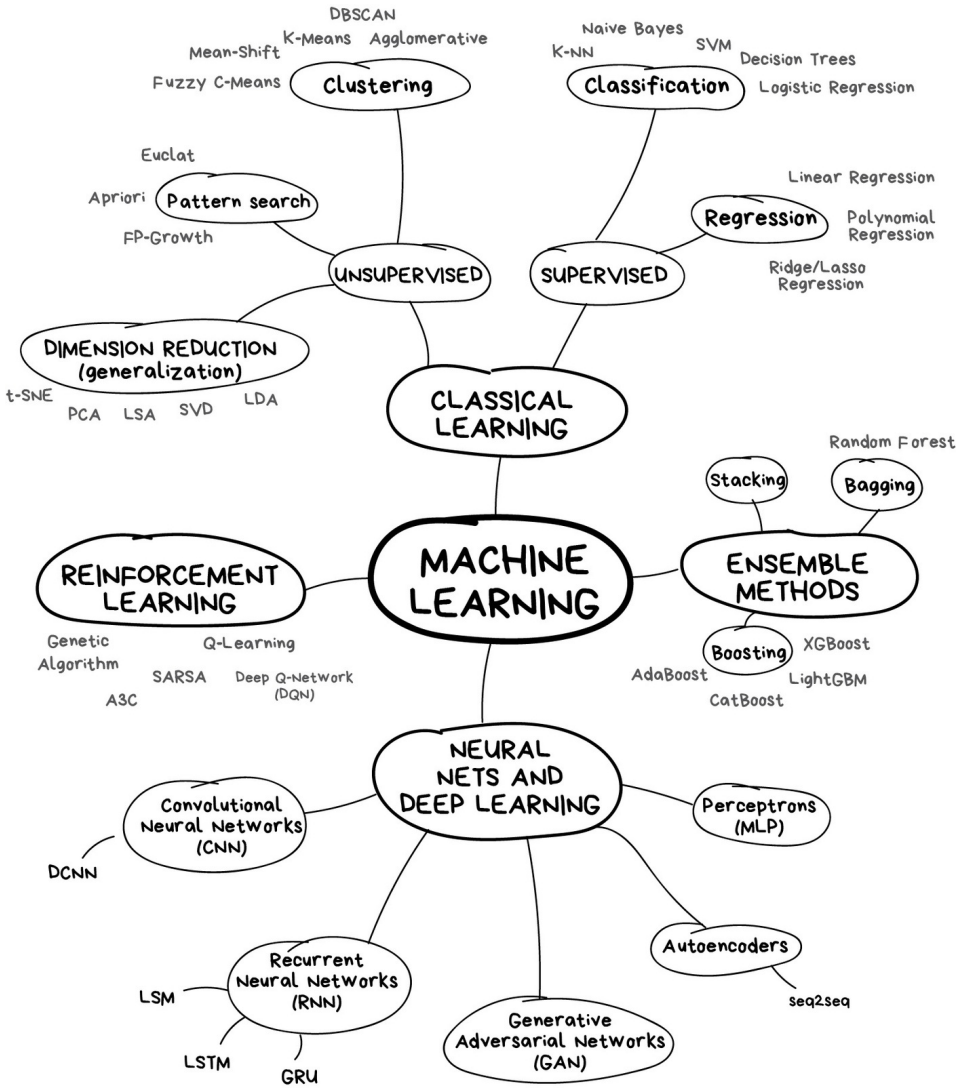
Examples of tasks

- **Clustering (or unsupervised classification):** construct a grouping of the data in homogeneous classes.
- **Dimension reduction:** construct a map of the data in a low dimensional space without distorting it too much.

Marketing

- **Data:** Base of customer data containing their properties and past buying records
- **Goal:** Use the customers *similarities* to find groups.
- **Two directions:**
  - **Clustering:** propose an explicit *grouping* of the customers
  - **Visualization:** propose a representation of the customers so that the groups are *visibles*

Machine Learning



Outline

Outline

Outline

What is a neuron?

This is a real neuron!

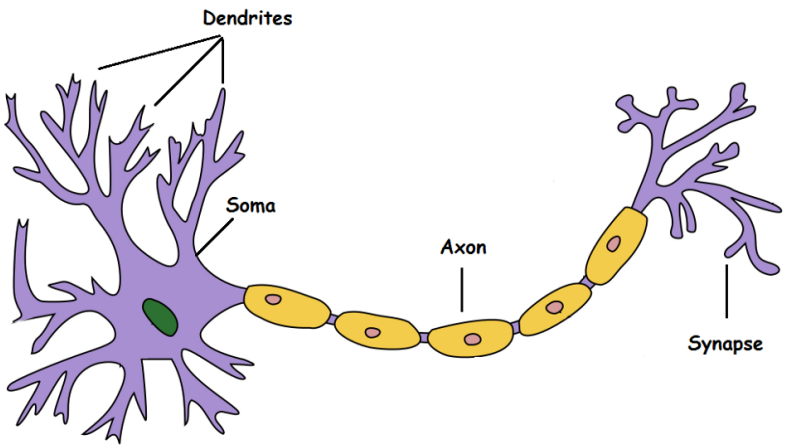


Figure: Real Neuron - diagram

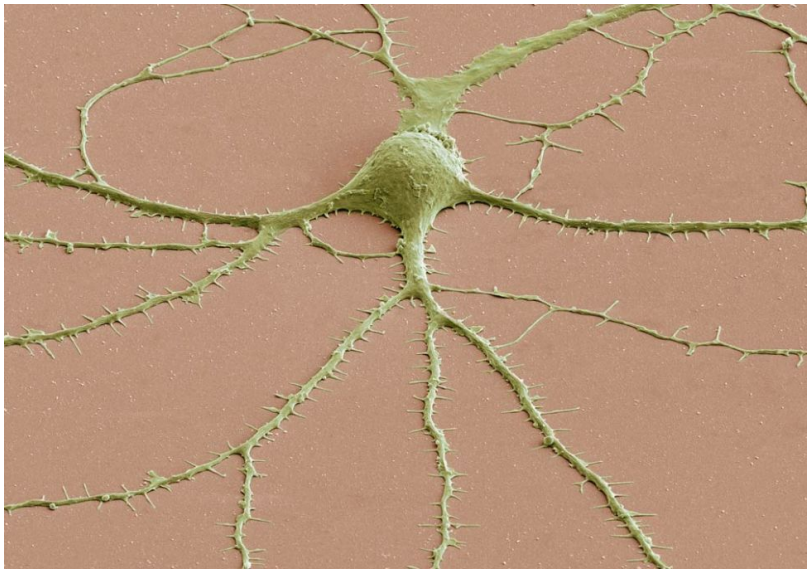


Figure: Real Neuron

The idea of neural networks began unsurprisingly as a model of how neurons in the brain function, termed 'connectionism' and used connected circuits to simulate intelligent behaviour.

McCulloch and Pitts neuron - 1943

In 1943, portrayed with a simple electrical circuit by neurophysiologist Warren McCulloch and mathematician Walter Pitts.

A McCulloch-Pitts neuron takes binary inputs, computes a weighted sum and returns 0 if the result is below threshold and 1 otherwise.

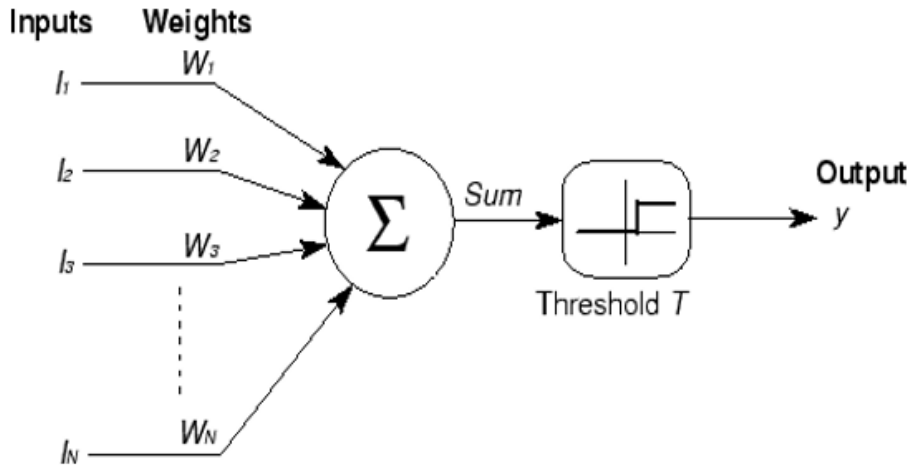


Figure: ["A logical calculus of the ideas immanent in nervous activity", McCulloch and Pitts 1943]

McCulloch and Pitts neuron - 1943

In 1943, portrayed with a simple electrical circuit by neurophysiologist Warren McCulloch and mathematician Walter Pitts.

A McCulloch-Pitts neuron takes binary inputs, computes a weighted sum and returns 0 if the result is below threshold and 1 otherwise.

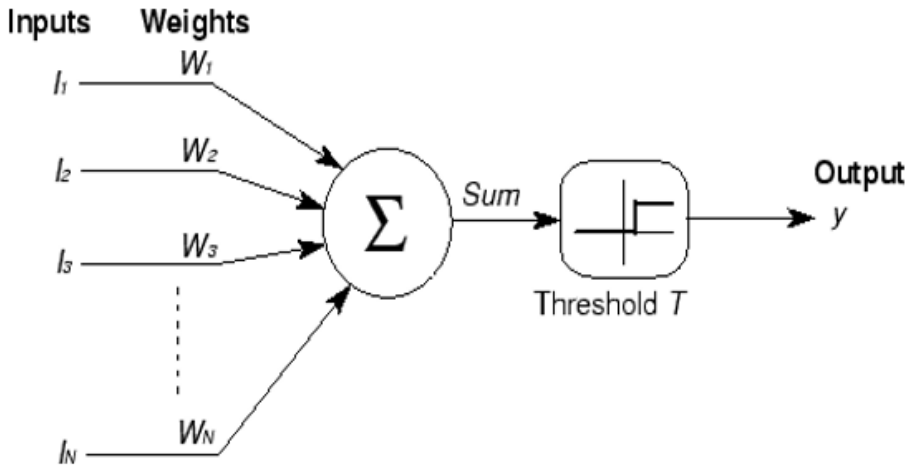


Figure: ["A logical calculus of the ideas immanent in nervous activity", McCulloch and Pitts 1943]

Donald Hebb took the idea further by proposing that neural pathways strengthen over each successive use, especially between neurons that tend to fire at the same time.

[*The organization of behavior: a neuropsychological theory*, Hebb 1949]

Outline

Perceptron - 1958

In the late 50s, Frank Rosenblatt, a psychologist at Cornell, worked on on decision systems present in the eye of a fly, which determine its flee response.

In 1958, he proposed the idea of a Perceptron, calling it Mark I Perceptron. It was a system with a simple input-output relationship, modelled on a McCulloch-Pitts neuron.

["Perceptron simulation experiments", Rosenblatt 1960]

Perceptron diagram

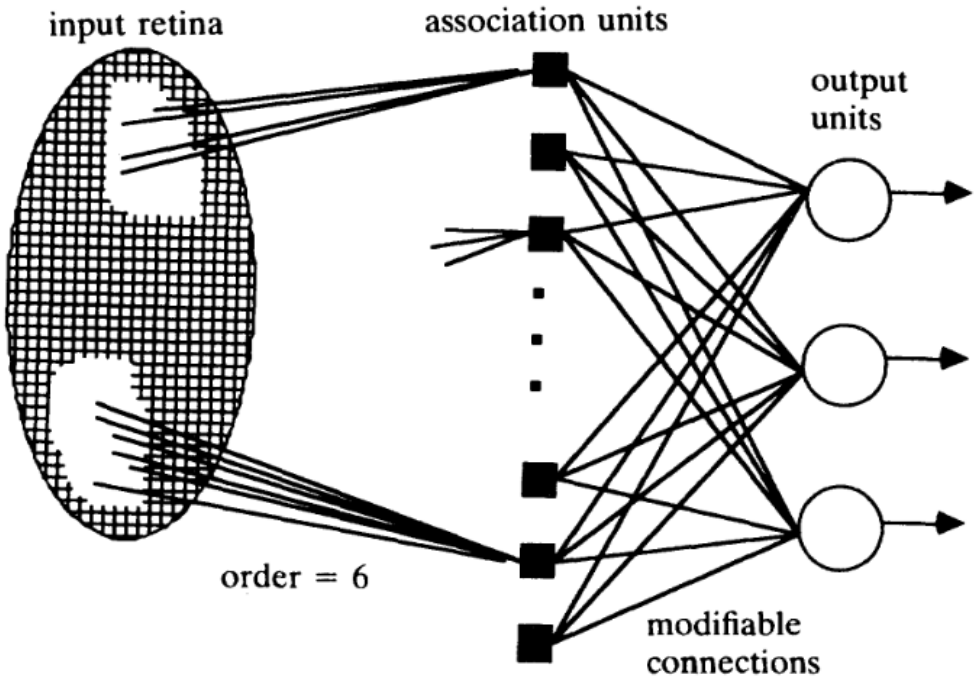
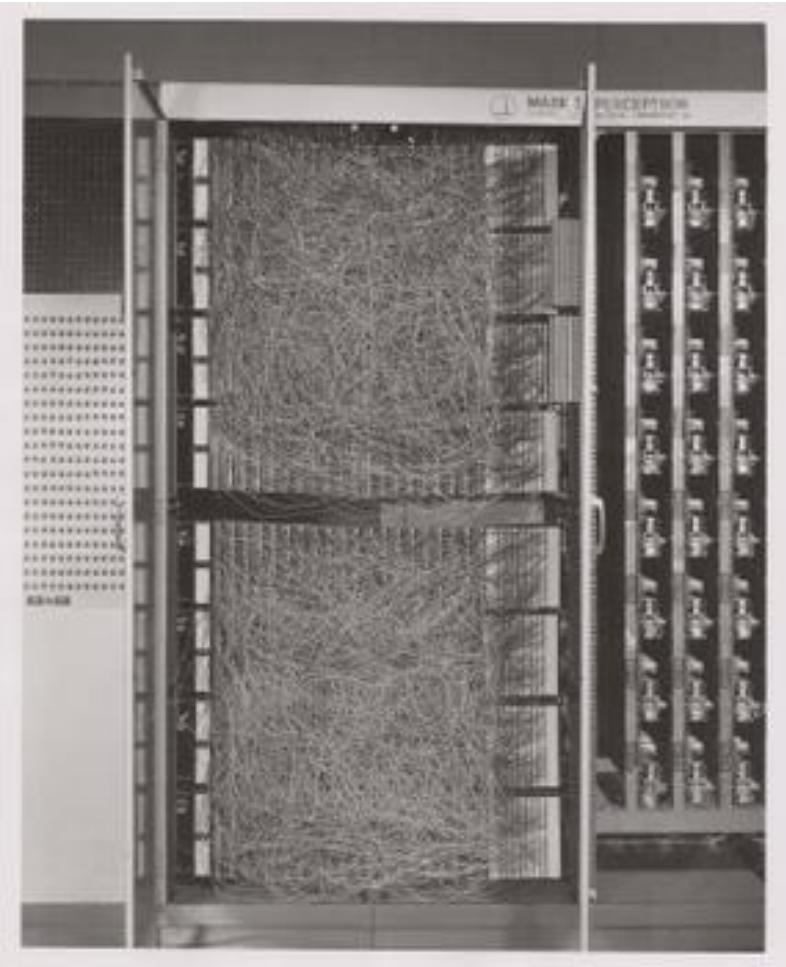


Figure: True representation of perceptron

The connections between the input and the first hidden layer cannot be optimized!

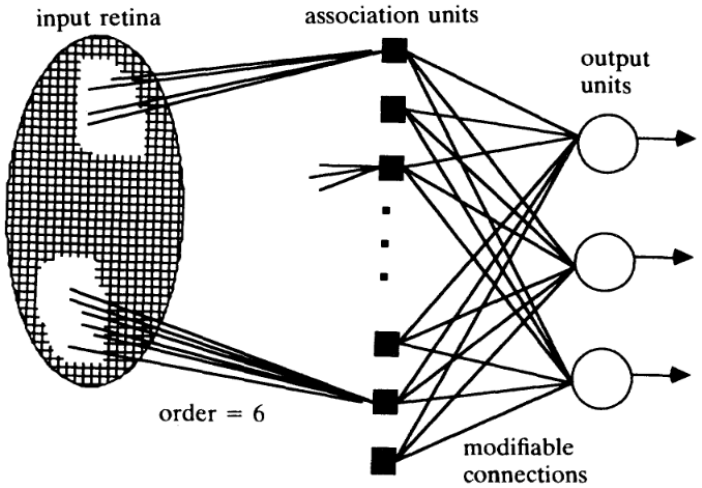
Perceptron Machine

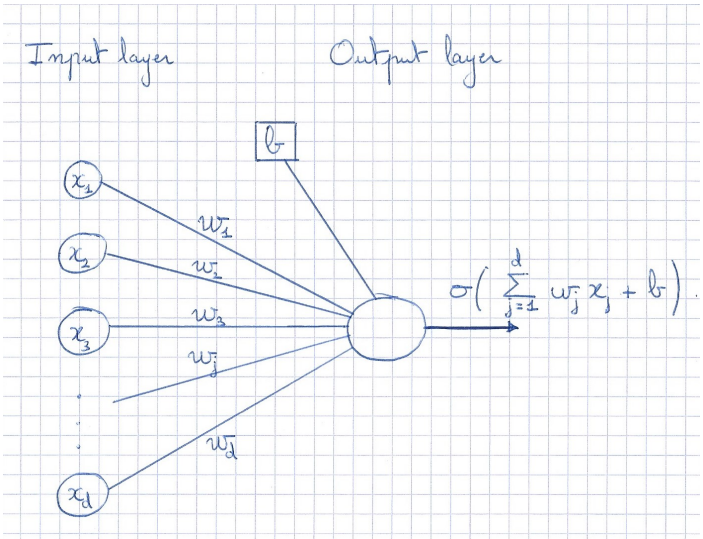


First implementation: Mark I Perceptron (1958).

- The machine was connected to a camera (20x20 photocells, 400-pixel image)
- Patchboard: allowed experimentation with different combinations of input features
- Potentiometers: that implement the adaptive weights

Parameters of the perceptron





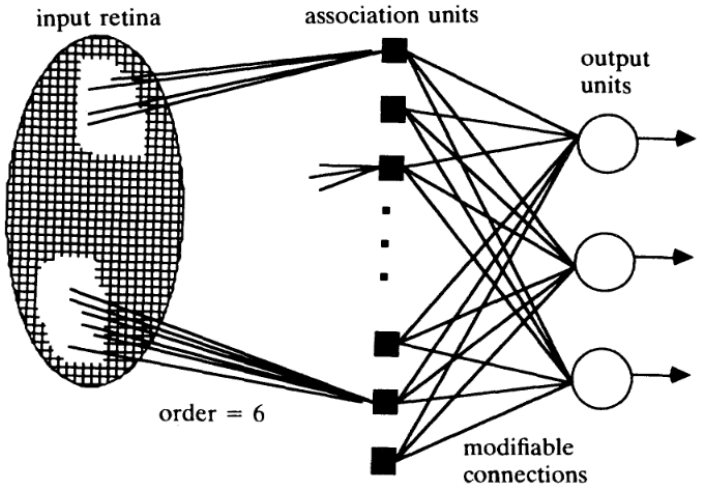
Activation function: $\sigma(z) = \mathbb{1}_{z>0}$

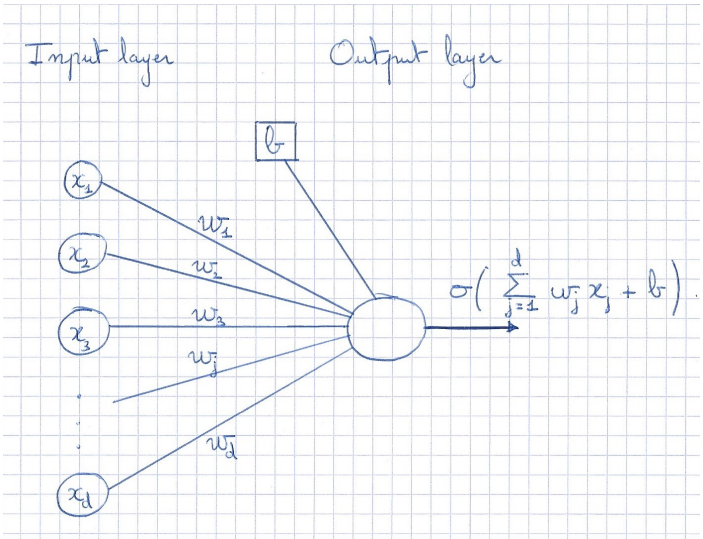
Parameters:

- Weights $\mathbf{w} = (w_1, \ldots, w_d) \in \mathbb{R}^d$
- Bias: b

The output is given by $f_{(\mathbf{w},b)}(\mathbf{x})$.

Parameters of the perceptron





Activation function: $\sigma(z) = \mathbb{1}_{z>0}$

Parameters:

- Weights $\mathbf{w} = (w_1, \ldots, w_d) \in \mathbb{R}^d$
- Bias: b

The output is given by $f_{(\mathbf{w},b)}(\mathbf{x})$.

How do we estimate $(\mathbf{w}, b)$?

The Perceptron Algorithm

We have $\mathcal{D}_n = \{(\mathbf{x}_i, y_i), i = 1, \ldots, n\}$, with $y_i \in \{-1, 1\}$.

To ease notations, we put $\tilde{\mathbf{w}} = (w_1, \ldots, w_d, b)$ and $\tilde{\mathbf{x}}_i = (\mathbf{x}_i, 1)$

Perceptron Algorithm - first (iterative) learning algorithm

- Start with $\tilde{\mathbf{w}} = 0$.
- Repeat over all samples:
  - if $y_i < \tilde{\mathbf{w}}, \tilde{\mathbf{x}}_i > \leq 0$ modify $\tilde{\mathbf{w}}$ into $\tilde{\mathbf{w}} + y_i \tilde{\mathbf{x}}_i$,
  - otherwise do not modify $\tilde{\mathbf{w}}$.

Exercise

- What is the rational behind this procedure?
- Is this procedure related to a gradient descent method?

Gradient descent:

1. Start with $\tilde{\mathbf{w}} = \tilde{\mathbf{w}}_0$
2. Update $\tilde{\mathbf{w}} \leftarrow \tilde{\mathbf{w}} - \eta \nabla \mathcal{L}(\tilde{\mathbf{w}})$, where $\mathcal{L}$ is the loss to be minimized.
3. Stop when $\tilde{\mathbf{w}}$ does not vary too much.

Solution

Perceptron algorithm can be seen as a stochastic gradient descent.

Perceptron Algorithm

- Repeat over all samples:
  - if $y_i < \tilde{\mathbf{w}}, \tilde{\mathbf{x}}_i > \leq 0$ modify $\tilde{\mathbf{w}}$ into $\tilde{\mathbf{w}} + y_i \tilde{\mathbf{x}}_i$,
  - otherwise do not modify $\tilde{\mathbf{w}}$.

A sample is misclassified if $y_i < \tilde{\mathbf{w}}, \tilde{\mathbf{x}}_i > \leq 0$. Thus we want to minimize the loss

$$\mathcal{L}(\tilde{\mathbf{w}}) = -\sum_{i \in \mathcal{M}_{\tilde{\mathbf{w}}}} y_i < \tilde{\mathbf{w}}, \tilde{\mathbf{x}}_i >,$$

where $\mathcal{M}_{\tilde{\mathbf{w}}}$ is the set of indices misclassified by the hyperplane $\tilde{\mathbf{w}}$.

Stochastic Gradient Descent:

1. Select randomly $i \in \mathcal{M}_{\tilde{\mathbf{w}}}$
2. Update $\tilde{\mathbf{w}} \leftarrow \tilde{\mathbf{w}} - \eta \nabla \mathcal{L}_i(\tilde{\mathbf{w}}) = \tilde{\mathbf{w}} - \eta y_i \tilde{\mathbf{x}}_i$

Perceptron algorithm: $\eta = 1$.

Exercise

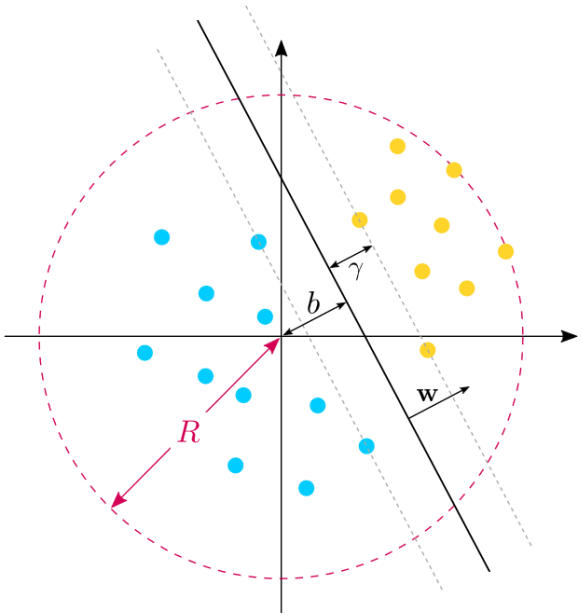


Figure: From `http://image.diku.dk/kstensbo/notes/perceptron.pdf`

Let $R = \max_i \|\mathbf{x}_i\|$. Let $\tilde{\mathbf{w}}^\star$ be the optimal hyperplane of margin

$$\gamma = \min_i y_i \langle \tilde{\mathbf{w}}^\star, \tilde{\mathbf{x}}_i \rangle,$$

with

$$\|\tilde{\mathbf{w}}^\star\| = 1.$$

Theorem (Block 1962; Novikoff 1963)

Assume that the training set $\mathcal{D}_n = \{(\boldsymbol{x}_1, y_i), \ldots, (\boldsymbol{x}_n, y_n)\}$ *is linearly separable* ($\gamma > 0$). *Start with* $\tilde{\boldsymbol{w}}_0 = 0$. *Then the number of updates* k *of the perceptron algorithm is bounded by*

$$k + 1 \leq \frac{1 + R^2}{\gamma^2}.$$

Exercise: Prove it!

Solution

According to Cauchy-Schwarz inequality,

$$\langle \tilde{\mathbf{w}}^\star, \tilde{\mathbf{w}}_{k+1} \rangle \leq \|\tilde{\mathbf{w}}_{k+1}\|_2 \, .$$

since $\|\tilde{\mathbf{w}}^\star\|_2 = 1$ with equality if and only if $\tilde{\mathbf{w}}_{k+1} \propto \tilde{\mathbf{w}}^\star$. By construction,

$$\begin{aligned}
\langle \tilde{\mathbf{w}}^\star, \tilde{\mathbf{w}}_{k+1} \rangle &= \langle \tilde{\mathbf{w}}^\star, \tilde{\mathbf{w}}_k \rangle + y_i \langle \tilde{\mathbf{w}}^\star, \tilde{\mathbf{x}}_i \rangle \\
&\geq \langle \tilde{\mathbf{w}}^\star, \tilde{\mathbf{w}}_k \rangle + \gamma \\
&\geq \langle \tilde{\mathbf{w}}^\star, \tilde{\mathbf{w}}_0 \rangle + (k+1)\gamma \\
&\geq (k+1)\gamma,
\end{aligned}$$

since $\tilde{\mathbf{w}}_0 = 0$. We have

$$\begin{aligned}
\|\tilde{\mathbf{w}}_{k+1}\|^2 &= \|\tilde{\mathbf{w}}_k\|^2 + \|\tilde{\mathbf{x}}_i\|^2 + 2\langle \tilde{\mathbf{w}}_k, y_i \tilde{\mathbf{x}}_i \rangle \\
&\leq \|\tilde{\mathbf{w}}_k\|^2 + R^2 + 1 \\
&\leq (k+1)(1+R^2),
\end{aligned}$$

since $\|\tilde{\mathbf{x}}_i\|^2 \leq R^2 + 1$ and $\langle \tilde{\mathbf{w}}_k, y_i \tilde{\mathbf{x}}_i \rangle \leq 0$ since the point $(\tilde{\mathbf{x}}_i, y_i)$ is misclassified. Finally,

$$\begin{aligned}
&(k+1)\gamma \leq \sqrt{(k+1)(1+R^2)} \\
\Leftrightarrow &k+1 \leq \frac{1+R^2}{\gamma^2}.
\end{aligned}$$

Perceptron - Summary and drawbacks

Perceptron algorithm

- We have a data set $\mathcal{D}_n = \{(\mathbf{x}_i, y_i), i = 1, \ldots, n\}$
- We use the perceptron algorithm to learn the weight vector $\mathbf{w}$ and the bias b.
- We predict using $f_{(\mathbf{w},b)}(\mathbf{x}) = \mathbb{1}_{\langle \mathbf{w},\mathbf{x} \rangle + b > 0}$.

Limitations

Perceptron - Summary and drawbacks

Perceptron algorithm

- We have a data set $\mathcal{D}_n = \{(\mathbf{x}_i, y_i), i = 1, \ldots, n\}$
- We use the perceptron algorithm to learn the weight vector $\mathbf{w}$ and the bias b.
- We predict using $f_{(\mathbf{w},b)}(\mathbf{x}) = \mathbb{1}_{\langle \mathbf{w},\mathbf{x} \rangle + b > 0}$.

Limitations

- The decision frontier is linear! Too simple model.
- The perceptron algorithm does not converge if data are not linearly separable: in this case, the algorithm must not be used.
- In practice, we do not know if data are linearly separable... Perceptron should never be used!

Perceptron will make machine intelligent... or not!

The Perceptron project led by Rosenblatt was funded by the US Office of Naval Research.

The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence. Later perceptrons will be able to recognize people and call out their names and instantly translate speech in one language to speech and writing in another language, it was predicted.

Press conference, 7 July 1958, New York Times.

For an extensive study of the perceptron, [*Principles of neurodynamics. perceptrons and the theory of brain mechanisms*, Rosenblatt 1961]

AI winter

In 1969, Minsky and Papert exhibit the fact that it was difficult for perceptron to

- detect parity (number of activated pixels)
- detect connectedness (are the pixels connected?)
- represent simple non linear function like XOR

There is no reason to suppose that any of [the virtue of perceptrons] carry over to the many-layered version. Nevertheless, we consider it to be an important research problem to elucidate (or reject) our intuitive judgement that the extension is sterile. Perhaps some powerful convergence theorem will be discovered, or some profound reason for the failure to produce an interesting "learning theorem" for the multilayered machine will be found.

["Perceptrons.", Minsky and Papert 1969]

This book is the starting point of the period known as "AI winter", a significant decline in funding of neural network research.

Controversy between Rosenblatt, Minsky, Papert:

["A sociological study of the official history of the perceptrons controversy", Olazaran 1996]

AI Winter: the XOR function

Exercise. Consider two binary variables $x_1 \in \{0,1\}$ and $x_2 \in \{0,1\}$. The logical function AND applied to x_1, x_2 is defined as

$$\begin{aligned} \text{AND} : &\{0,1\}^2 \to \{0,1\} \\ &(x_1, x_2) \mapsto \begin{cases} 1 & \text{if } x_1 = x_2 = 1 \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

1) Find a perceptron (i.e., weights and bias) that implements the AND function.

The logical function XOR applied to x_1, x_2 is defined as

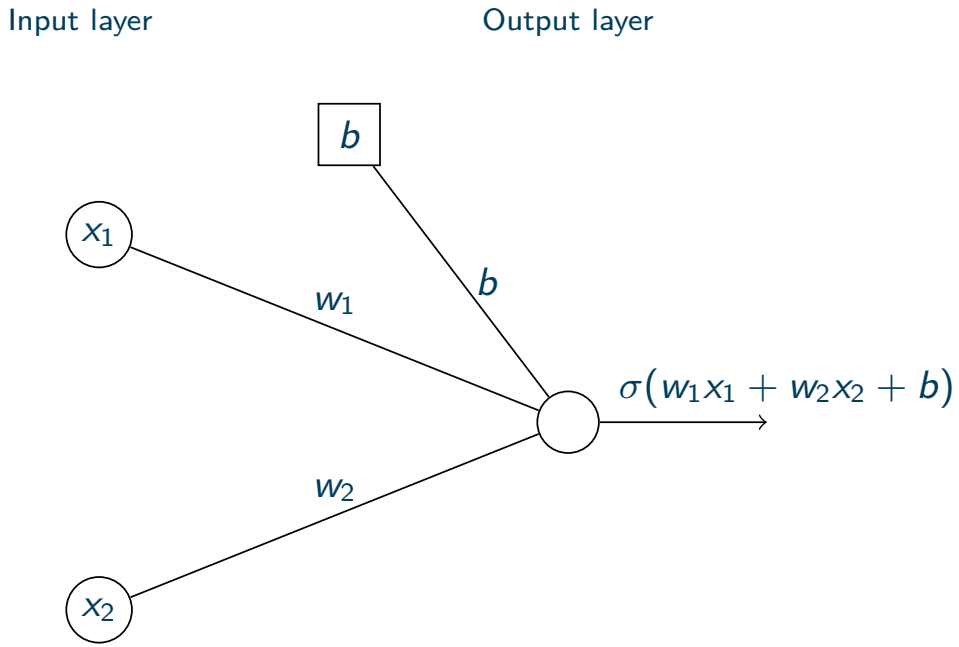
$$\begin{aligned} \text{XOR} : &\{0,1\}^2 \to \{0,1\} \\ &(x_1, x_2) \mapsto \begin{cases} 0 & \text{if } x_1 = x_2 = 0 \text{ or } x_1 = x_2 = 1 \\ 1 & \text{otherwise} \end{cases} \end{aligned}$$

2) Prove that no perceptron can implement the XOR function.

3) Find a neural network with one hidden layer that implements the XOR function.

Solution

1. The following perceptron with $w_1 = w_2 = 1$ and $b = -1.5$ implements the AND function:



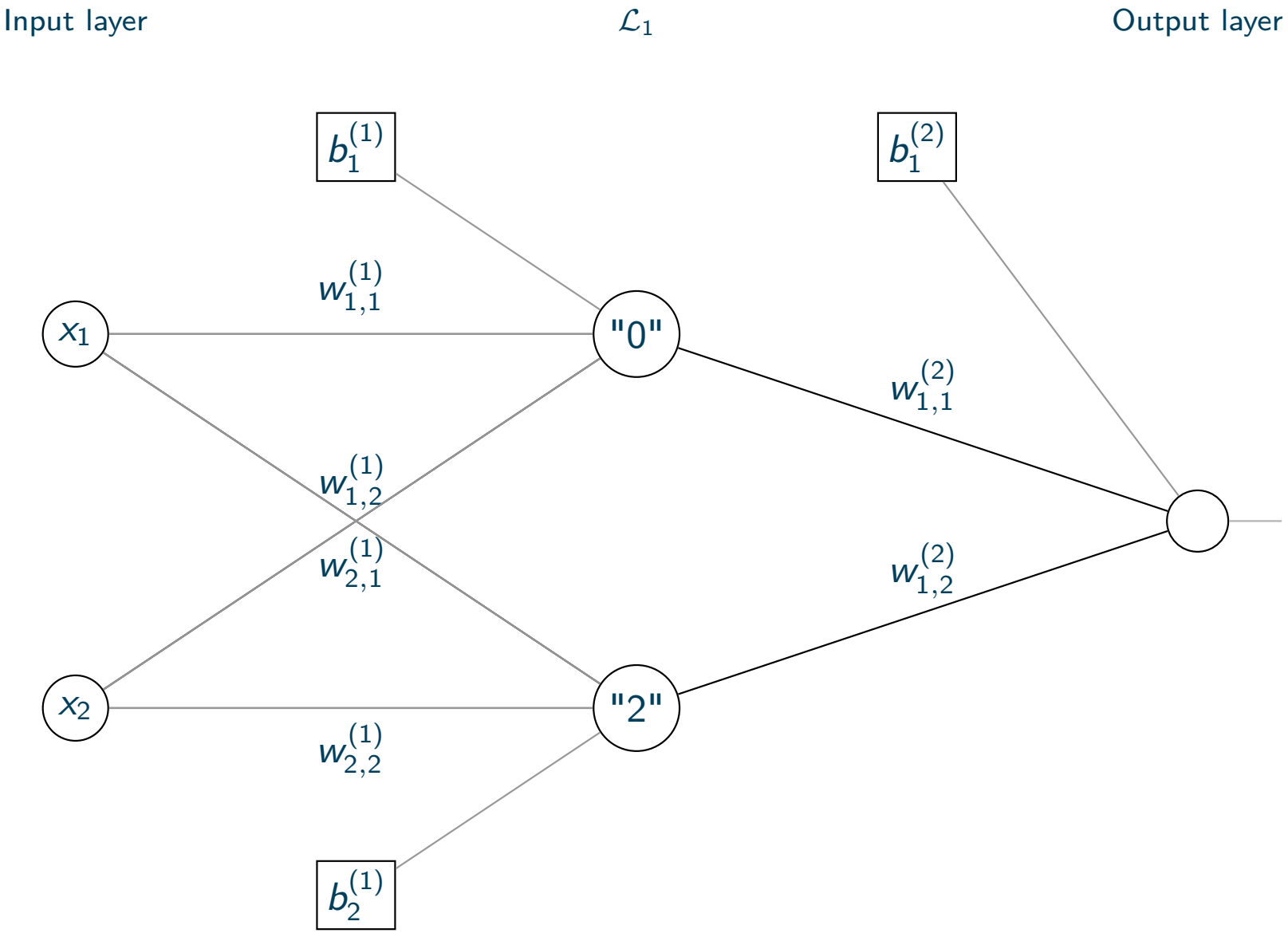
2. The perceptron algorithm builds a hyperplane and predicts accordingly, depending on which side of the hyperplane the observation falls into. Unfortunately the XOR function cannot be represented with a hyperplane in the original space $\{0, 1\}^2$.

Solution

3. The following network implements the XOR function with

$$\begin{cases} w_{1,1}^{(1)} = -1 \\ w_{1,2}^{(1)} = -1 \\ b_1^{(1)} = 0.5 \end{cases} \quad \begin{cases} w_{2,1}^{(1)} = 1 \\ w_{2,2}^{(1)} = 1 \\ b_2^{(1)} = -1.5 \end{cases} \quad \begin{cases} w_{1,1}^{(2)} = -1 \\ w_{1,2}^{(2)} = -1 \\ b_1^{(2)} = 0.5 \end{cases}$$

Solution


Input layer
$\mathcal{L}_1$
Output layer
$b_1^{(1)}$
$b_1^{(2)}$
$w_{1,1}^{(1)}$
x_1
"0"
$w_{1,1}^{(2)}$
$w_{1,2}^{(1)}$
$w_{2,1}^{(1)}$
$w_{1,2}^{(2)}$
x_2
"2"
$w_{2,2}^{(1)}$
$b_2^{(1)}$

Outline

Moving forward - ADALINE, MADALINE

In 1959 at Stanford, Bernard Widrow and Marcian Hoff developed AdaLinE (ADAptive LINear Elements) and **MAdaLinE** (Multiple AdaLinE) the latter being the first network successfully applied to a real world problem.

[*Adaptive switching circuits*, Bernard Widrow and Hoff 1960]

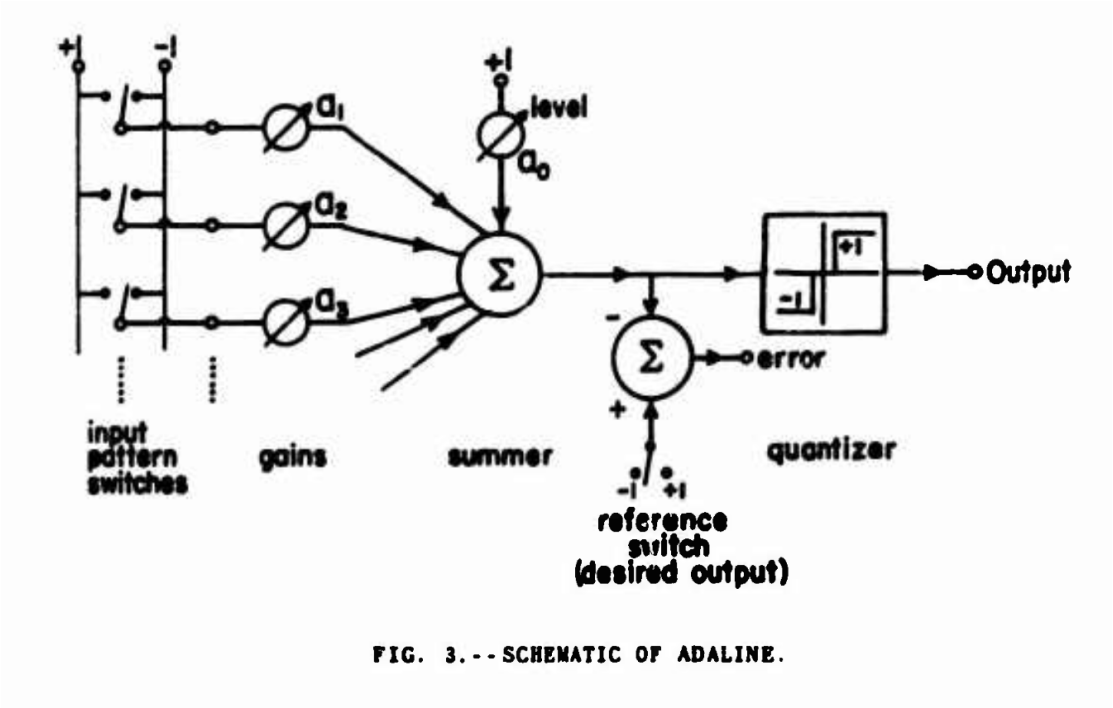


FIG. 3.--SCHEMATIC OF ADALINE.

- Loss: square difference between a weighted sum of inputs and the output
- Optimization procedure: trivial gradient descent

MADALINE

Many Adalines: network with one hidden layer composed of many Adaline units.

["Madaline Rule II: a training algorithm for neural networks", Winter and Widrow 1988]

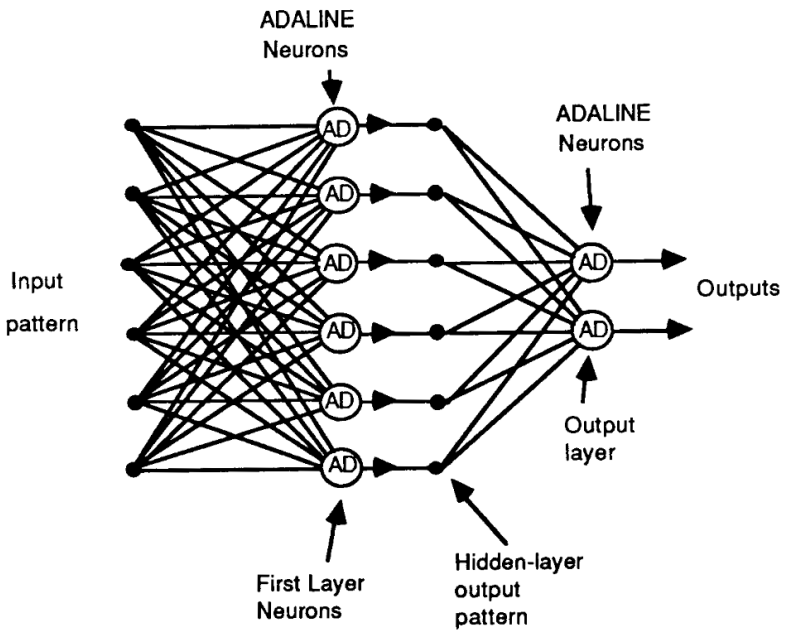


Figure 1: Layered feed-forward ADALINE network.

Applications:

- Speech and pattern recognition

  ["Real-Time Adaptive Speech-Recognition System", Talbert et al. 1963]

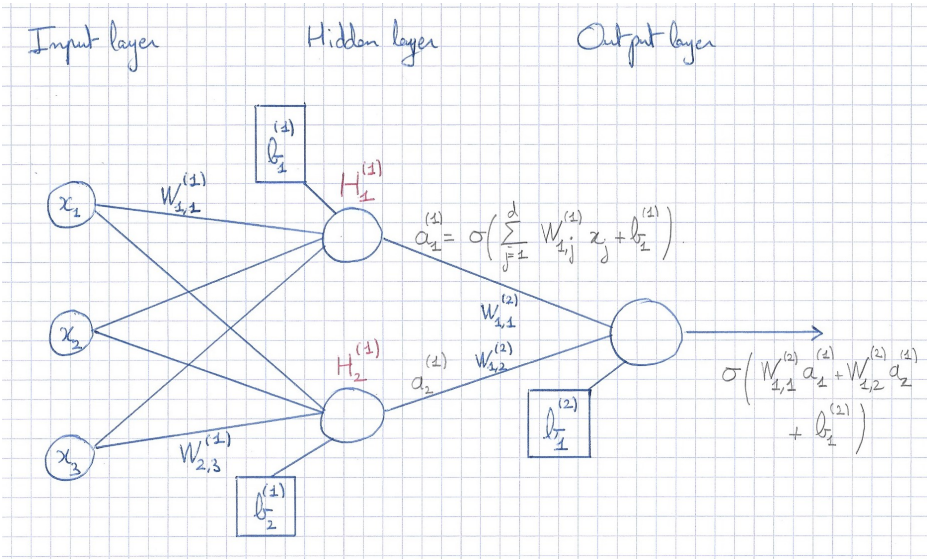
- Weather forecasting

  ["Application of the adaline system to weather forecasting", Hu 1964]

- Adaptive filtering and adaptive signal processing

  ["Adaptive signal processing", Bernard and Samuel 1985]

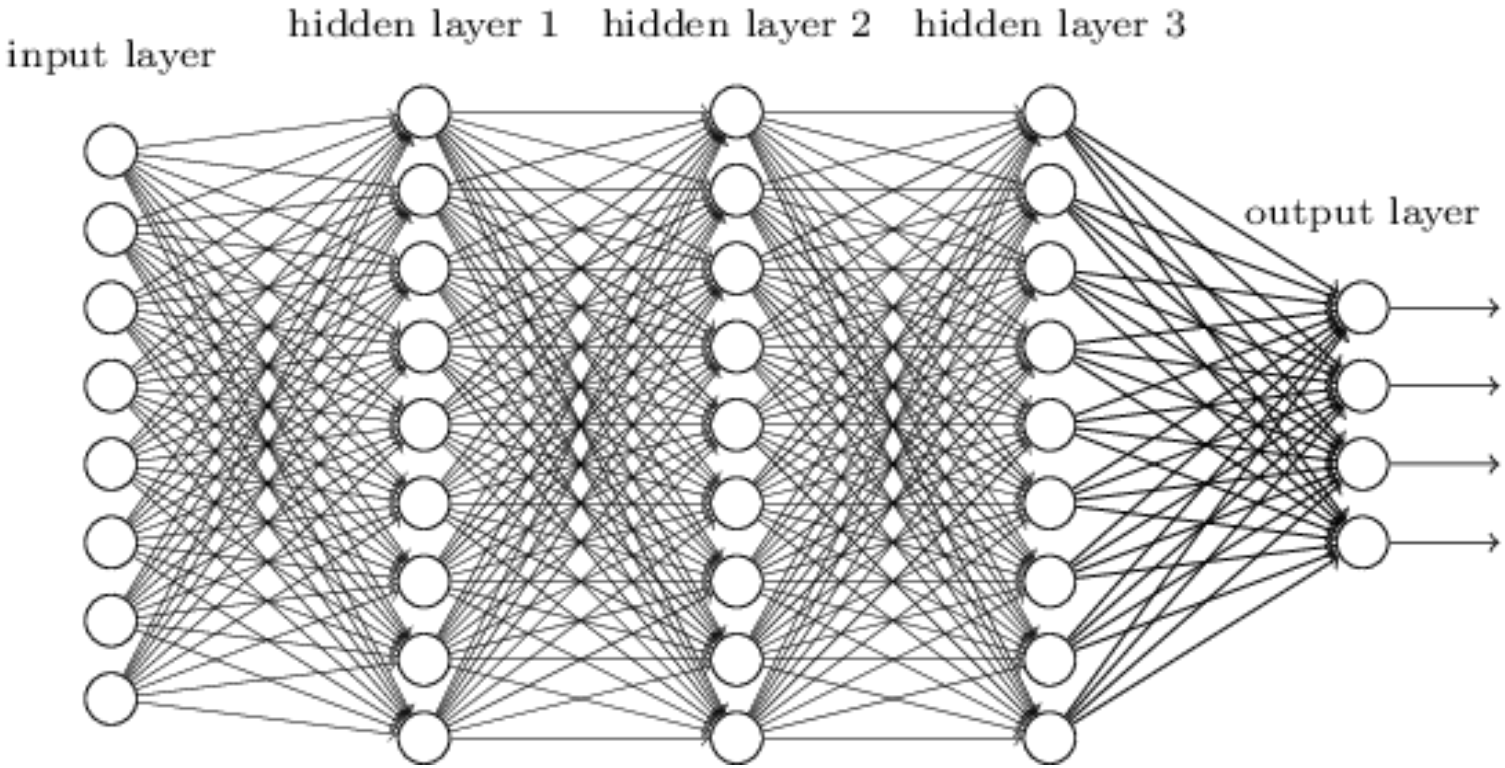
Neural network with one hidden layer



Generic notations:

- $W_{i,j}^{(\ell)}$: weights between the j neuron in the $\ell - 1$ layer and the i neuron of the ℓ layer.
- $b_j^{(\ell)}$: bias of the j neuron of the ℓ layer.
- $a_j^{(\ell)}$: output of the j neuron of the ℓ layer.
- $z_j^{(\ell)}$: input of the j neuron of the ℓ layer, such that $a_j^{(\ell)} = \sigma(z_j^{(\ell)})$.

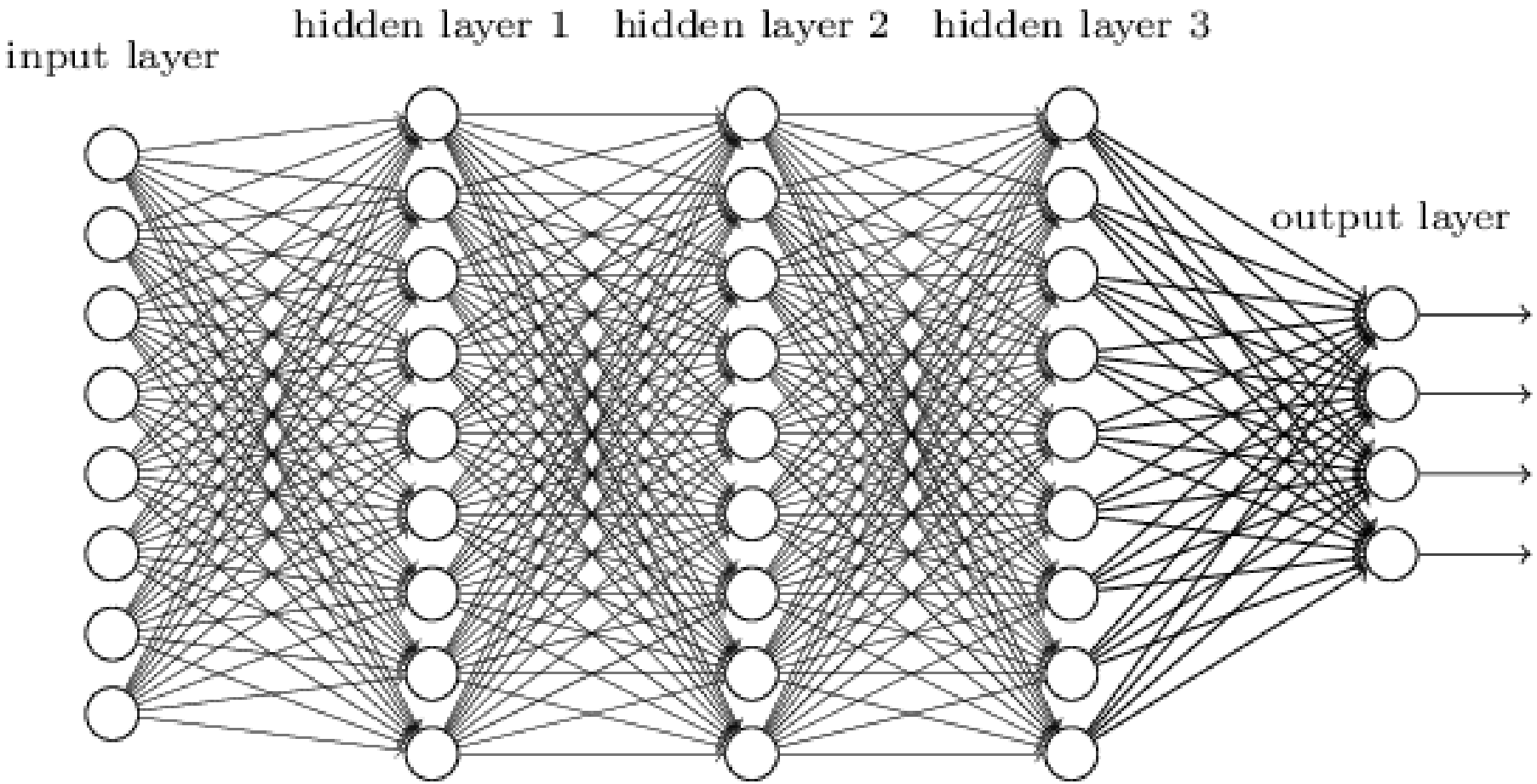
How to find weights and bias?



Perceptron algorithm does not work anymore!

Outline

How to find weights and bias?



Gradient Descent Algorithm

- The prediction of the network is given by $f_\theta(\mathbf{x})$.
- Empirical risk minimization:

$$\underset{\theta}{\text{argmin}} \frac{1}{n} \sum_{i=1}^{n} \ell(Y_i, f_\theta(\mathbf{X}_i)) \equiv \underset{\theta}{\text{argmin}} \frac{1}{n} \sum_{i=1}^{n} \ell_i(\theta)$$

Stochastic Gradient descent rule:

While $|\theta_t - \theta_{t-1}| \geq \varepsilon$ do

- Sample $I_t \subset \{1, \ldots, n\}$
-

$$\theta_{t+1} = \theta_t - \eta \left(\frac{1}{|I_t|} \sum_{i \in I_t} \nabla_\theta \ell_i(\theta_t) \right)$$

Gradient Descent Algorithm

- The prediction of the network is given by $f_\theta(\mathbf{x})$.
- Empirical risk minimization:

$$\underset{\theta}{\text{argmin}} \frac{1}{n} \sum_{i=1}^{n} \ell(Y_i, f_\theta(\mathbf{X}_i)) \equiv \underset{\theta}{\text{argmin}} \frac{1}{n} \sum_{i=1}^{n} \ell_i(\theta)$$

Stochastic Gradient descent rule:

While $|\theta_t - \theta_{t-1}| \geq \varepsilon$ do

- Sample $I_t \subset \{1, \ldots, n\}$
-

$$\theta_{t+1} = \theta_t - \eta \left(\frac{1}{|I_t|} \sum_{i \in I_t} \nabla_\theta \ell_i(\theta_t) \right)$$

How to compute $\nabla_\theta \ell_i$ efficiently?

Backprop Algorithm

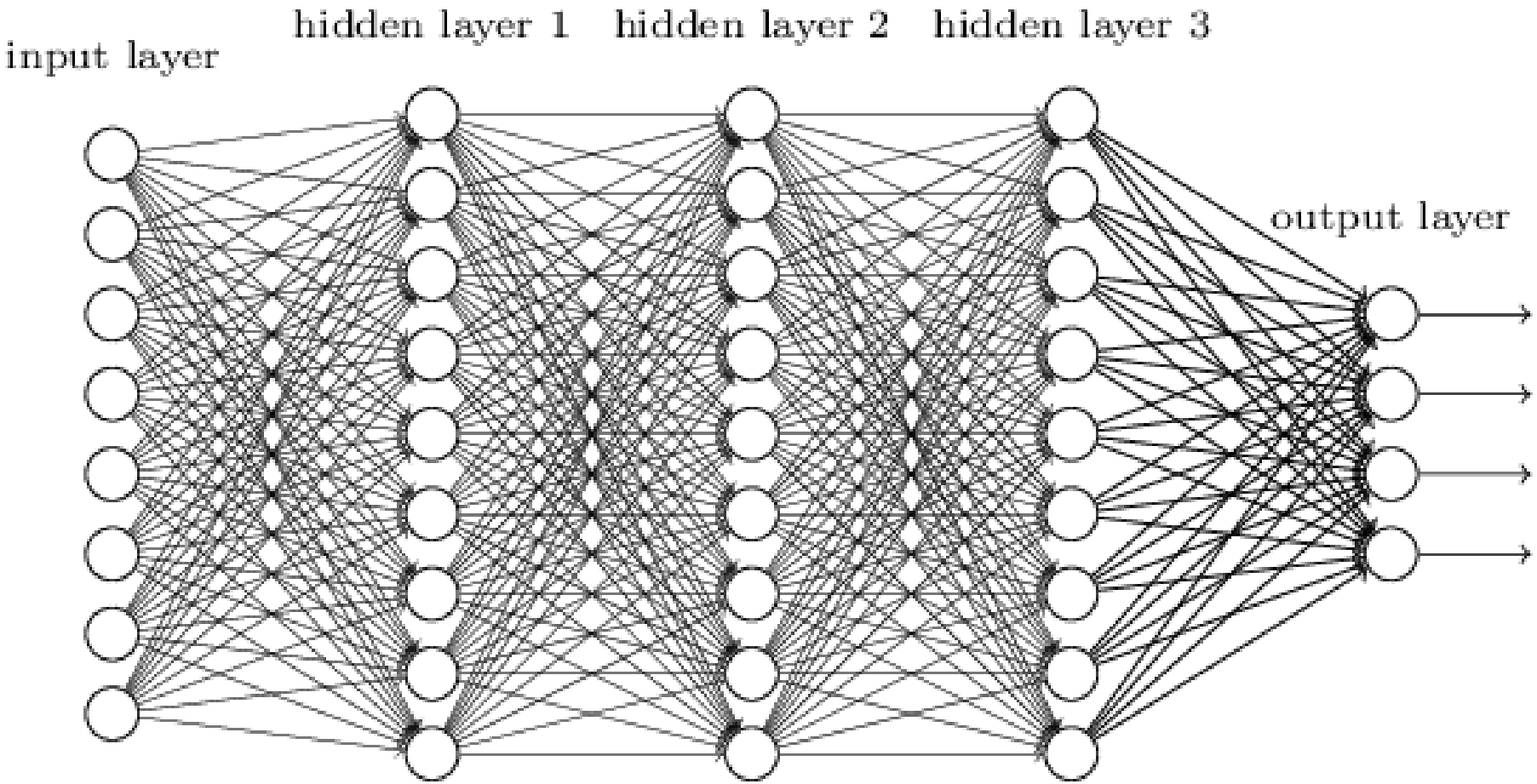
How to compute $\nabla_\theta \ell_i$ efficiently?

A Clever Gradient Descent Implementation

- Popularized by Rumelhart, McClelland, Hinton in 1986.
- Can be traced back to Werbos in 1974.
- Nothing but the use of chain rule derivation with a touch of dynamic programing.

- Key ingredient to make the Neural Networks work!
- Still at the core of Deep Learning algorithm.

Backpropagation idea



Backpropagation equations

Neural network with L layers, with vector output, with quadratic cost

$$C = \frac{1}{2}\|y - a^{(L)}\|^2.$$

By definition,

$$\delta_j^{(\ell)} = \frac{\partial C}{\partial z_j^{(\ell)}}.$$

The four fundamental equations of backpropagation are given by

$$\begin{aligned}
\delta^{(L)} &= \nabla_a C \odot \sigma'(z^{(L)}), \\
\delta^{(\ell)} &= ((w^{(\ell+1)})^T \delta^{(\ell+1)}) \odot \sigma'(z^{(\ell)}) \\
\frac{\partial C}{\partial b_j^{(\ell)}} &= \delta_j^{(\ell)} \\
\frac{\partial C}{\partial w_{j,k}^{(\ell)}} &= a_k^{(\ell-1)} \delta_j^{(\ell)}.
\end{aligned}$$

Proof

We start with the first equality

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

Applying the chain rule gives

$$\delta_j^{(L)} = \sum_k \frac{\partial C}{\partial a_k^{(L)}} \frac{\partial a_k^{(L)}}{\partial z_j^{(L)}},$$

where $z_j^{(L)}$ is the input of the j neuron of the layer L. Since the activation $a_k^{(L)}$ depends only on the input $z_j^{(L)}$, we have

$$\delta_j^{(L)} = \frac{\partial C}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}}.$$

Besides, since $a_j^{(L)} = \sigma(z_j^{(L)})$, we have

$$\delta_j^{(L)} = \frac{\partial C}{\partial a_j^{(L)}} \sigma'(z_j^{(L)}),$$

which is the component wise version of the first equality.

Proof

Now, we want to prove

$$\delta^{(\ell)} = ((w^{(\ell+1)})^T \delta^{(\ell+1)}) \odot \sigma'(z^{(\ell)}).$$

Again, using the chain rule,

$$\begin{aligned}
\delta_j^{(\ell)} &= \frac{\partial C}{\partial z_j^{(\ell)}} \\
&= \sum_k \frac{\partial C}{\partial z_k^{(\ell+1)}} \frac{\partial z_k^{(\ell+1)}}{\partial z_j^{(\ell)}} \\
&= \sum_k \delta_k^{(\ell+1)} \frac{\partial z_k^{(\ell+1)}}{\partial z_j^{(\ell)}}.
\end{aligned}$$

Recalling that $z_k^{(\ell+1)} = \sum_j w_{kj}^{(\ell+1)} \sigma(z_j^{(\ell)}) + b_k^{\ell+1}$, we get

$$\delta_j^{(\ell)} = \sum_k \delta_k^{(\ell+1)} w_{kj}^{(\ell+1)} \sigma'(z_j^{(\ell)}),$$

which concludes the proof.

Proof

Now, we want to prove

$$\frac{\partial C}{\partial b_j^{(\ell)}} = \delta_j^{(\ell)}.$$

Using the chain rule,

$$\frac{\partial C}{\partial b_j^{\ell}} = \sum_k \frac{\partial C}{\partial z_k^{(\ell)}} \frac{\partial z_k^{(\ell)}}{\partial b_j^{\ell}}.$$

However, only $z_j^{(\ell)}$ depends on b_j^{ℓ} and $z_j^{(\ell)} = \sum_k w_{jk}^{(\ell)} \sigma(z_k^{(\ell-1)}) + b_j^{\ell}$. Therefore,

$$\frac{\partial C}{\partial b_j^{\ell}} = \delta_j^{(\ell)}.$$

Proof

Finally, we want to prove

$$\frac{\partial C}{\partial w_{j,k}^{(\ell)}} = a_k^{(\ell-1)} \delta_j^{(\ell)}.$$

By the chain rule,

$$\frac{\partial C}{\partial w_{j,k}^{(\ell)}} = \sum_m \frac{\partial C}{\partial z_m^{(\ell)}} \frac{\partial z_m^{(\ell)}}{\partial w_{j,k}^{(\ell)}}.$$

Since $z_m^{(\ell)} = \sum_k w_{mk}^{(\ell)} \sigma(z_k^{(\ell-1)}) + b_m^{\ell}$, we have

$$\frac{\partial z_m^{(\ell)}}{\partial w_{j,k}^{(\ell)}} = \sigma(z_k^{(\ell-1)}) \mathbb{1}_{m=j}.$$

Consequently,

$$\frac{\partial C}{\partial w_{j,k}^{(\ell)}} = \delta_j^{(\ell)} \sigma(z_k^{(\ell-1)}) = a_k^{(\ell-1)} \delta_j^{(\ell)}.$$

Backpropagation Algorithm

Let

$$\delta_j^{(\ell)} = \frac{\partial C}{\partial z_j^{(\ell)}},$$

where $z_j^{(\ell)}$ is the entry of the neuron j of the layer ℓ.

Backpropagation Algorithm

- Initialize randomly weights and bias in the network.
- For each training sample x_i,
  1. **Feedforward:** let x_i go through the network and store the value of activation function and its derivative, for each neuron.
  2. **Output error:** compute the neural network error for x_i.
  3. **Backpropagation:** compute recursively the vectors $\delta^{(\ell)}$ starting from $\ell = L$ to $\ell = 1$.
- Update the weights and bias using the backpropagation equations.

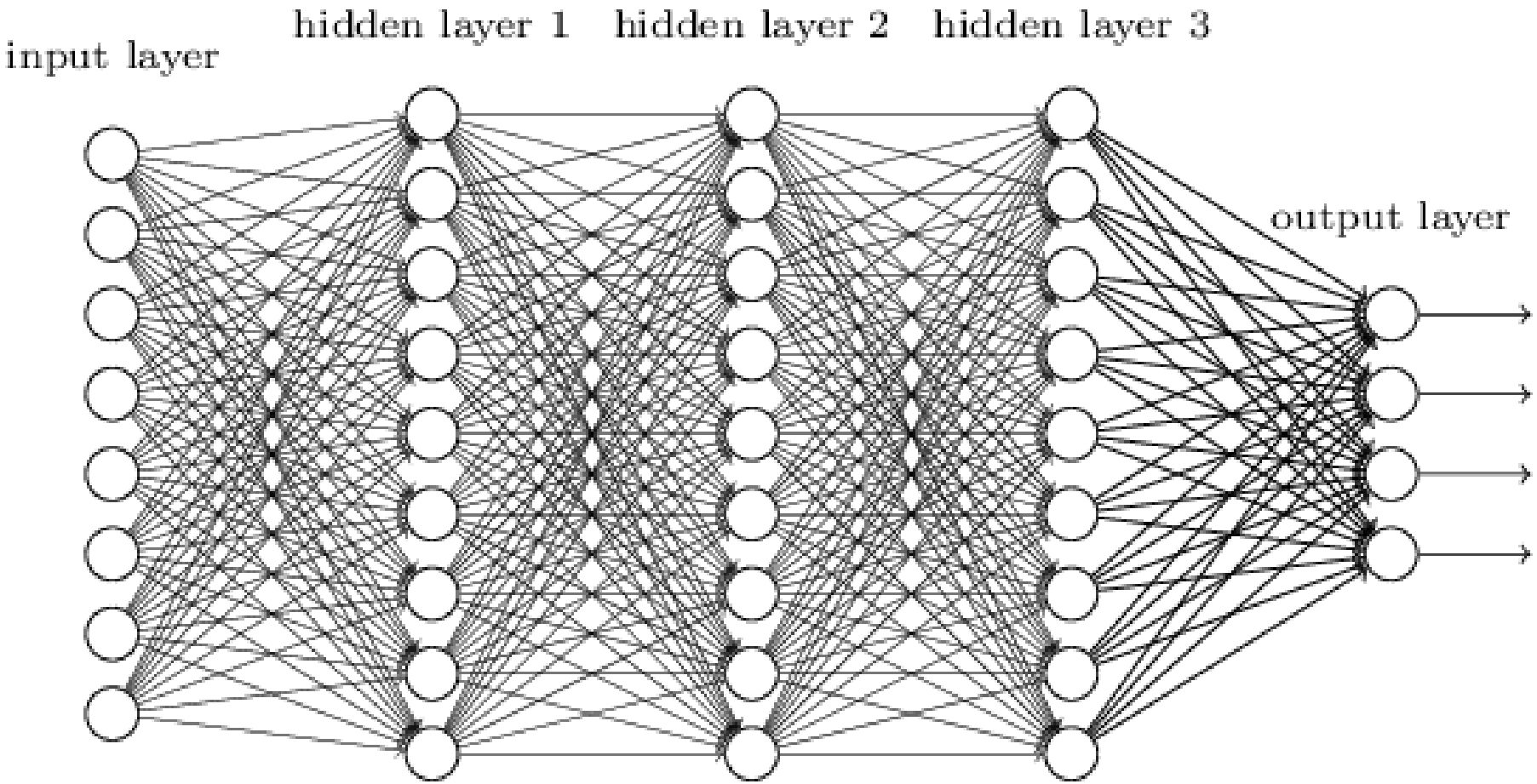
Neural Network terminology

- Epoch: one forward pass and one backward pass of all training examples.
- (Mini) Batch size: number of training examples in one forward/backward pass. The higher the batch size is, the more memory space you'll need.
- Number of iterations: number of passes, each pass using [batch size] number of examples. To be clear, one pass = one forward pass + one backward pass (we do not count the forward pass and backward pass as two different passes).

For example, for 1000 training examples, if you set the batch size at 500, it will take 2 iterations to complete 1 epoch.

Outline

What to set in a neural network?



Outline

Sigmoid activation function

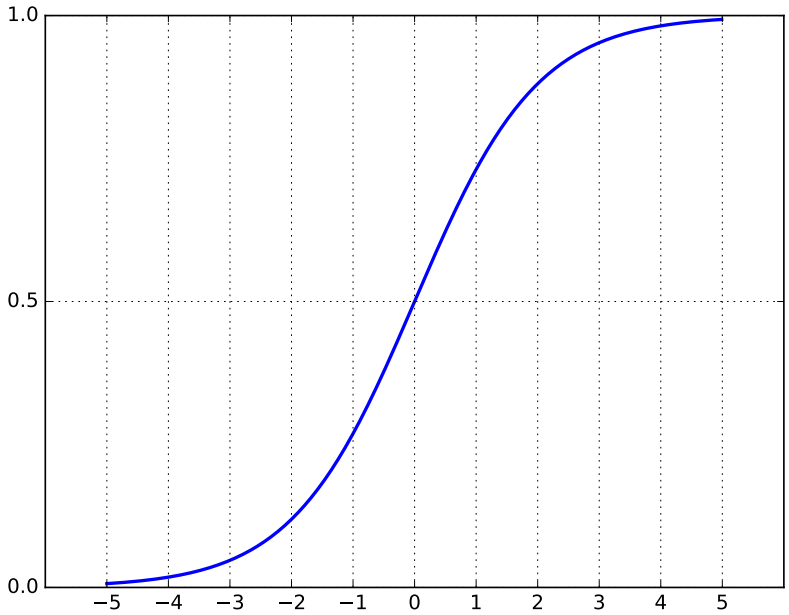


Figure: Sigmoid activation function σ

$$\sigma : x \mapsto \frac{\exp(x)}{1 + \exp(x)}$$

Comments:

- Saturated function due to horizontal asymptotes:
  - Gradient is close to zero in these two areas ($\pm\infty$)
  - Rescaling the inputs of each layer can help to avoid these areas.
- Sigmoid is not a zero-centered function
  - Rescaling data
- Computing $\exp(x)$ is a bit costly

Hyperbolic tangent

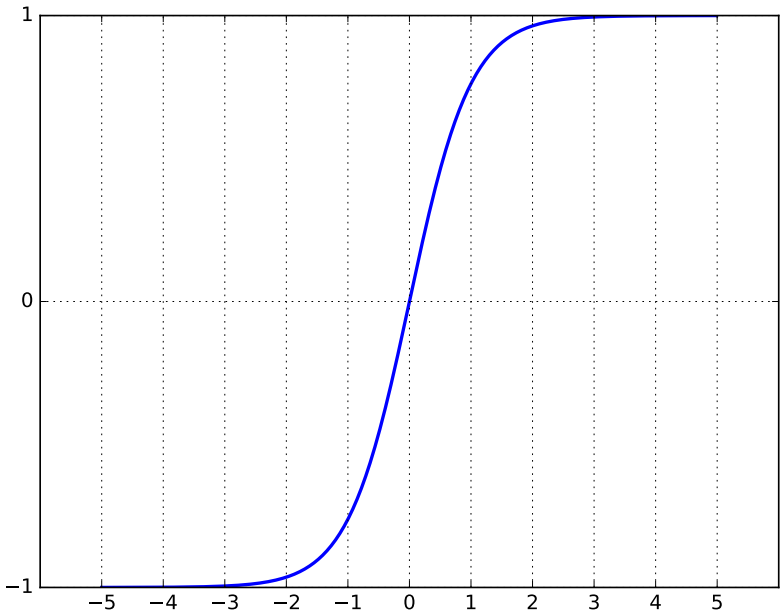


Figure: Hyperbolic tangent (tanh)

$$\tanh : x \mapsto \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

Comments:

- The function tanh is zero-centered
  - No need for rescaling data
- Saturated function due to horizontal asymptotes:
  - Gradient is close to zero in these two areas ($\pm\infty$)
  - Rescaling the inputs of each layer can help to avoid these areas.
- Computing $\exp(x)$ is a bit costly

Note that $\tanh(x) = 2\sigma(2x) - 1$.

Rectified Linear Unit (ReLU)

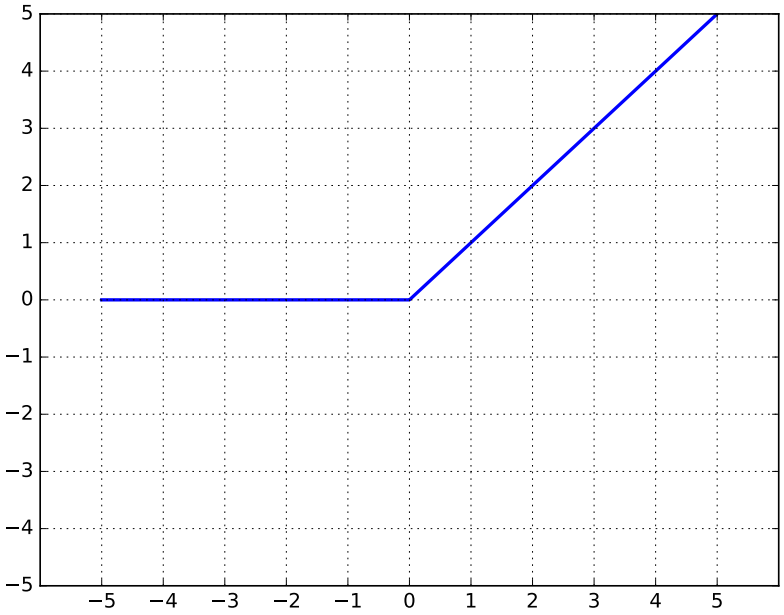


Figure: Rectified Linear Unit (ReLU)

$$ReLU : x \mapsto \max(0, x)$$

Comments:

- Not a saturated function in $+\infty$
- But saturated (and null!) in the region $x \leq 0$
- Computationally efficient
- Empirically, convergence is faster than sigmoid/tanh.
- Plus: biologically plausible

More on ReLU

The idea of ReLU in neural networks seems to appear in ["Cognitron: A self-organizing multilayered neural network"; "Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition", Fukushima 1975; Fukushima and Miyake 1982].

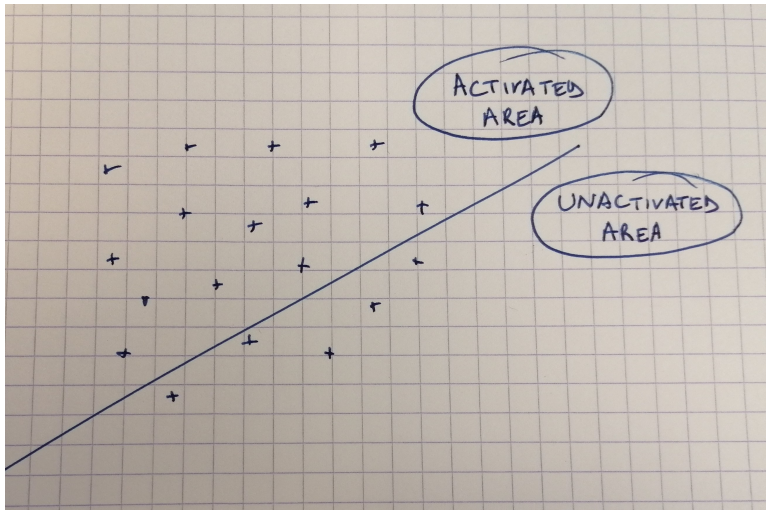


Figure: Good parameter initialization - ReLU is active

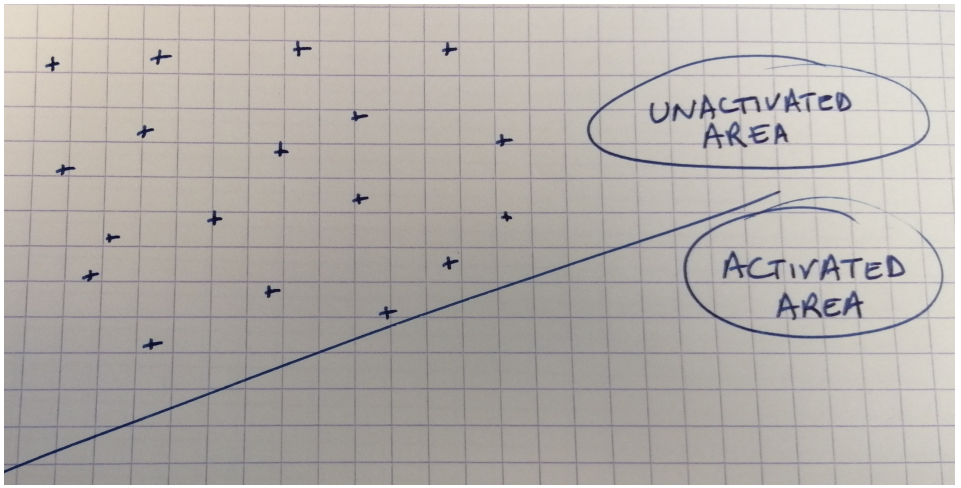


Figure: Bad parameter initialization - ReLU outputs zero

ReLU output can be zero but positive initial bias can help.

Related to biology ["Deep sparse rectifier neural networks", Glorot, Bordes, et al. 2011]:

- Most of the time, neurons are inactive.
- when they activate, their activation is proportional to their input.

Parametric ReLU

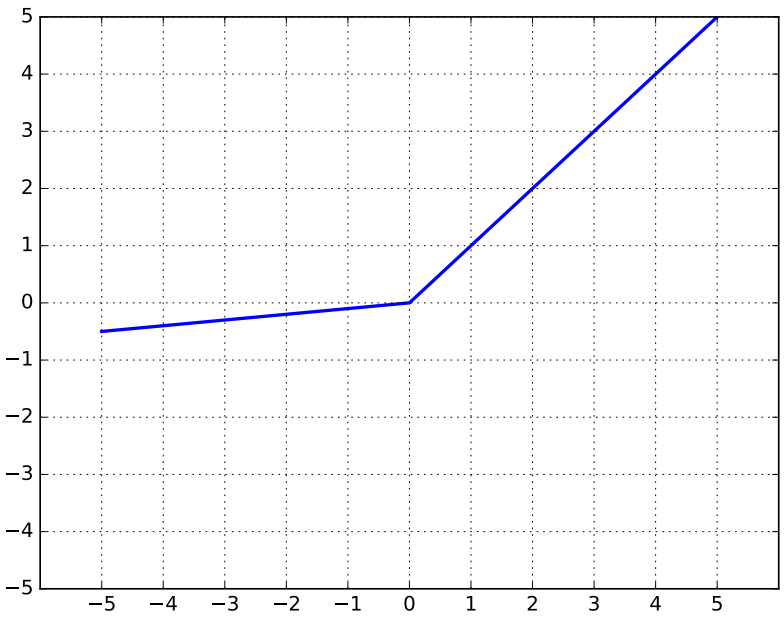


Figure: Parametric ReLU

- Leaky ReLU: $\alpha = 0.1$

  ["Rectifier nonlinearities improve neural network acoustic models", Maas et al. 2013]

- Absolute Value Rectification:

  $$\alpha = -1$$

  ["What is the best multi-stage architecture for object recognition?", Jarrett et al. 2009]

- Parametric ReLU: α optimized during backpropagation. Activation function is learned.

  ["Empirical evaluation of rectified activations in convolutional network", Xu et al. 2015]

$$\text{Parametric ReLU} : x \mapsto \max(\alpha x, x)$$

Exponential Linear Unit (ELU)

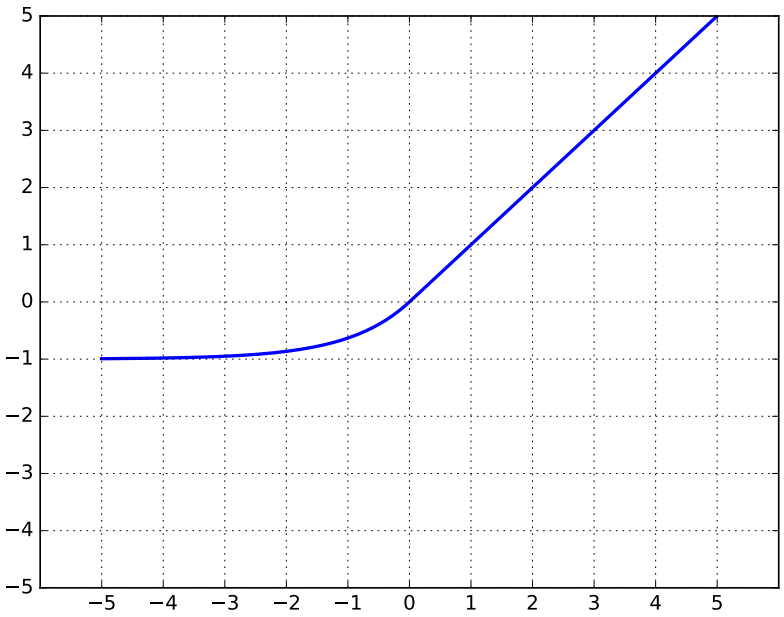


Figure: Exponential Linear Unit (ELU)

Comments:

- Close to ReLU but differentiable everywhere
- Closer to zero mean output.
- α is set to 1.0.
- Robustness to noise

  ["Fast and accurate deep network learning by exponential linear units (elus)", Clevert et al. 2015]

$$ELU : x \mapsto \begin{cases} x & \text{if } x \geq 0 \\ \alpha(\exp(x) - 1) & \text{otherwise} \end{cases}$$

Maxout

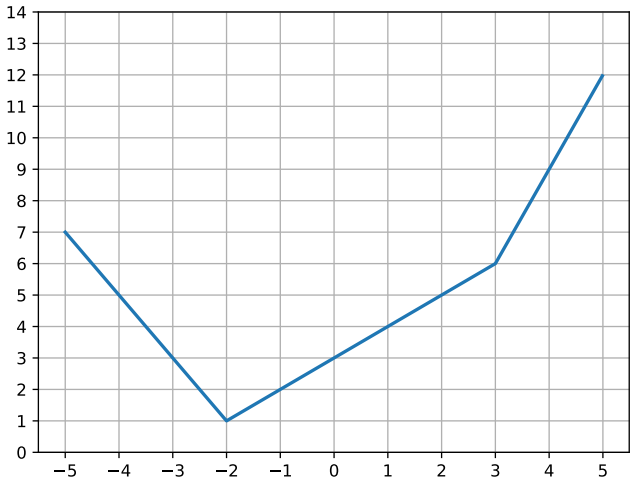


Figure: Maxout activation function, with $k = 3$ pieces

Maxout

$$x \mapsto \max(w_1 x + b_1, w_2 x + b_2, w_3 x + b_3)$$

Comments:

- Learn piecewise linear functions with k pieces.
- Linear regime with k pieces: no saturation.
- Number of parameters multiplied by k

  ["Maxout networks", Goodfellow, Warde-Farley, et al. 2013] ["Deep maxout neural networks for speech recognition", Cai et al. 2013]

- Resist to catastrophic forgetting

  ["An empirical investigation of catastrophic forgetting in gradient-based neural networks", Goodfellow, Mirza, et al. 2013]

Swish

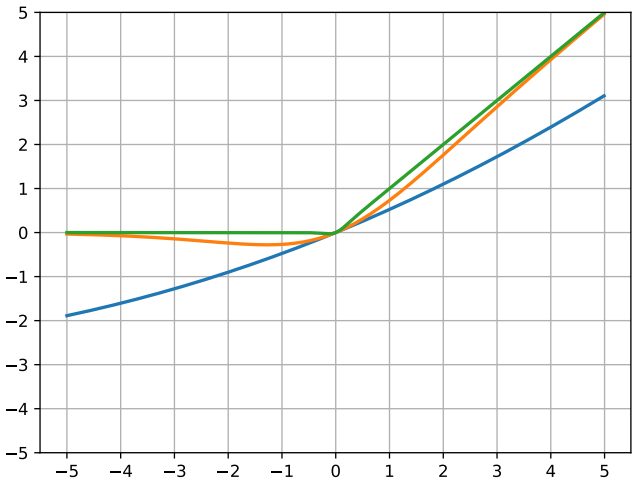


Figure: Swish function for $\beta = 0.1, 1, 10$

- Swish interpolates between the linear function and ReLU.
- ["Searching for activation functions", Ramachandran et al. 2017]
- Non-monotonic function which seems to be an important feature.

$$Swish : x \mapsto x \frac{\exp(\beta x)}{1 + \exp(\beta x)}$$

Conclusion on activation functions

- Use ReLU (or Swish).
- Test Leaky ReLU, maxout, ELU.
- Try out Tanh, but do not expect too much.
- Do not use sigmoid.

Outline

Output units

- Linear output unit:
$$\hat{y} = W^T h + b$$
$\rightarrow$ Linear regression based on the new variables h.

- Sigmoid output unit, used to predict $\{0, 1\}$ outputs:
$$\mathbb{P}(Y = 1|h) = \sigma(W^T h + b),$$
where $\sigma(t) = e^t/(1 + e^t)$.
$\rightarrow$ Logistic regression based on the new variables h.

- Softmax output unit, used to predict $\{1, \ldots, K\}$:
$$\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}$$
where, each z_i is the activation of one neuron of the previous layer, given by $z_i = W_i^T h + b_i$.
$\rightarrow$ Multinomial logistic regression based on the new variables h.

Multinomial logistic regression

Generalization of logistic regression for multiclass outputs: for all $1 \leq k \leq K$,

$$\log\left(\frac{\mathbb{P}[Y_i = k]}{Z}\right) = \beta_k X_i,$$

Hence, for all $1 \leq k \leq K$,

$$\mathbb{P}[Y_i = k] = Z e^{\beta_k X_i},$$

where

$$Z = \frac{1}{\sum_{k=1}^{K} e^{\beta_k X_i}}.$$

Thus,

$$\mathbb{P}[Y_i = k] = \frac{e^{\beta_k X_i}}{\sum_{\ell=1}^{K} e^{\beta_\ell X_i}}.$$

Biology bonus

Softmax, used with cross-entropy:

$$\begin{aligned} -\log(\mathbb{P}(Y=y|z)) &= -\log \operatorname{softmax}(z)_y \\ &= -z_y + \log\Big(\sum_j \exp(z_j)\Big) \\ &\simeq \max_j z_j - z_y, \end{aligned}$$

No contribution to the cost when $\operatorname{softmax}(z)_{\hat{y}}$ is maximal.

Lateral inhibition: believed to exist between nearby neurons in the cortex. When the difference between the max and the other is large, winner takes all: one neuron is set to 1 and the others go to zero.

More complex models: Conditional Gaussian Mixture: y is multimodal

["On supervised learning from sequential data with applications for speech recognition"; "Generating sequences with recurrent neural networks", Schuster 1999; Graves 2013].

Outline

Cost functions

- Mean Square Error (MSE)

$$\frac{1}{n}\sum_{i=1}^{n}\ell(Y_i, f_\theta(\mathbf{X}_i)) = \frac{1}{n}\sum_{i=1}^{n}(Y_i - f_\theta(\mathbf{X}_i))^2$$

- Mean Absolute Error

$$\frac{1}{n}\sum_{i=1}^{n}\ell(Y_i, f_\theta(\mathbf{X}_i)) = \frac{1}{n}\sum_{i=1}^{n}|Y_i - f_\theta(\mathbf{X}_i)|$$

- $0-1$ Error

$$\frac{1}{n}\sum_{i=1}^{n}\ell(Y_i, f_\theta(\mathbf{X}_i)) = \frac{1}{n}\sum_{i=1}^{n}\mathbb{1}_{Y_i \neq f_\theta(\mathbf{X}_i)}$$

Cost functions

- Cross entropy (or negative log-likelihood):

$$\frac{1}{n}\sum_{i=1}^{n}\ell(Y_i, f_\theta(\mathbf{X}_i)) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K}\mathbb{1}_{y_i=k}\log\left([f_\theta(\mathbf{X}_i)]_k\right)$$

Cross-entropy:

- Very popular!
- Should help to prevent saturation phenomenon compared to MSE:

$$-\log(\mathbb{P}(Y = y_i|X)) = -\log(\sigma((2y-1)(W^T h + b))),$$

with

$$\sigma(t) = \frac{e^t}{1+e^t}$$

Usually, saturation occurs when $(2y-1)(W^T h+b) \ll -1$. In that case, $-\log(\mathbb{P}(Y = y_i|X))$ is linear in W and b which makes the gradient easy to compute, and the gradient descent easy to implement.

Mean Square Error should not be used with softmax output units

["Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition", Bridle 1990]

Outline

Weight initialization

First idea: Set all weights and bias to the same value.

Small or big weights?

1. First idea: small random numbers, typically
$$0.01 \times \mathcal{N}(0, 10^{-4}).$$
   - work for small networks
   - for big networks ($\sim$ 10 layers) output become dirac in 0: there is no activation at all.

2. Second idea: "big random numbers"
$$\mathcal{N}(0, 10^{-4}).$$
   $\rightarrow$ Saturating phenomenon

In any case, no need to tune the bias: they can be initially set to zero.

Other initialization

Idea: the variance of the input should be the same as the variance of the output.

1. Xavier initialization
   Initialize bias to zero and weights randomly using
   $$\mathcal{U}\left[-\frac{\sqrt{6}}{\sqrt{n_j+n_{j+1}}},\frac{\sqrt{6}}{\sqrt{n_j+n_{j+1}}}\right],$$
   where n_j is the size of layer j
   ["Understanding the difficulty of training deep feedforward neural networks", Glorot and Bengio 2010].
   $\rightarrow$ Sadly, it does not work for ReLU (non activated neurons)

2. He et al. initialization
   Initialize bias to zero and weights randomly using
   $$\mathcal{N}\left(0,\frac{\sqrt{2}}{n_j}\right),$$
   where n_j is the size of layer j.
   ["Delving deep into rectifiers: Surpassing human-level performance on imagenet classification", He et al. 2015].

Bonus: ["All you need is a good init", Mishkin and Matas 2015]

Exercise

1. Consider a neural network with two hidden layers (containing n_1 and n_2 neurons respectively) and equipped with linear hidden units. Find a simple sufficient condition on the weights so that the variance of the hidden units stay constant accross layers.

2. Considering the same network, find a simple sufficient condition on the weights so that the gradient stay constant across layers when applying backpropagation procedure.

3. Based on previous questions, propose a simple way to initialize weights.

Solution

1. Consider the neuron i of the second hidden layer. The output of this neuron is

$$a_i^{(2)} = \sum_{j=1}^{n_1} W_{ij}^{(2)} z_j^{(1)} + b_i^{(2)}$$
$$= \sum_{j=1}^{n_1} W_{ij}^{(2)} \Big(\sum_{k=1}^{n_0} W_{jk}^{(1)} x_k + b_j^{(1)} \Big) + b_i^{(2)}$$
$$= \sum_{j=1}^{n_1} W_{ij}^{(2)} \Big(\sum_{k=1}^{n_0+1} W_{jk}^{(1)} x_k \Big) + b_i^{(2)},$$

where we set $x_{n_0+1} = 1$, and $W_{j,n_0+1}^{(1)} = b_j^{(1)}$ to ease notations. At fixed x, since the weights and bias are centred, we have

$$\mathbb{V}\Big[a_i^{(2)}\Big] = \mathbb{V}\Big[\sum_{j=1}^{n_1} \sum_{k=1}^{n_0+1} W_{ij}^{(2)} W_{jk}^{(1)} x_k + b_i^{(2)}\Big]$$
$$= \sum_{j=1}^{n_1} \sum_{k=1}^{n_0+1} x_k^2 \mathbb{V}\Big[W_{ij}^{(2)} W_{jk}^{(1)}\Big]$$

Solution

$$= \sum_{j=1}^{n_1} \sigma_2^2 \sum_{k=1}^{n_0+1} x_k^2 \sigma_1^2,$$

where σ_1^2 (resp. σ_2^2) is the shared variance of all weights between layers 0 and 1 (resp. layers 1 and 2). Since we want that $\mathbb{V}\left[a_i^{(2)}\right] = \mathbb{V}\left[a_i^{(1)}\right]$, it is sufficient that

$$n_1 \sigma_2^2 = 1.$$

Solution

2. Consider a neural network where the cost function is given, for one training example, by

$$C = \Big(y - \frac{1}{n_{d+1}} \sum_{j=1}^{n_{d+1}} z_j^{(d)}\Big)^2,$$

which implies

$$\frac{\partial C}{\partial z_j^{(d)}} = -\frac{2W_j}{n_{d+1}} \Big(y - \frac{1}{n_{d+1}} \sum_{i=1}^{n_d} W_i z_i^{(d)}\Big).$$

Hence,

$$\begin{aligned}\mathbb{E}\Big[\frac{\partial C}{\partial z_j^{(d)}}\Big] &= \mathbb{E}\Big[\frac{2W_j^2}{n_{d+1}^2} z_j^{(d)}\Big] \\ &= \frac{2\sigma_{d+1}^2}{n_{d+1}^2} \mathbb{E}\Big[z_j^{(d)}\Big].\end{aligned}$$

Besides, according to the chain rule, we have

$$\begin{aligned}\frac{\partial C}{\partial z_k^{(d-1)}} &= \sum_{j=1}^{n_{d+1}} \frac{\partial C}{\partial z_j^{(d)}} \frac{\partial z_j^{(d)}}{\partial z_k^{(d-1)}} \\ &= \sum_{j=1}^{n_{d+1}} \frac{\partial C}{\partial z_j^{(d)}} W_{jk}^{(d)}.\end{aligned}$$

Solution

More precisely,

$$\mathbb{E}\Big[\frac{\partial C}{\partial z_j^{(d)}} W_{jk}^{(d)}\Big] = \mathbb{E}\Big[\frac{\partial C}{\partial z_j^{(d)}} W_{jk}^{(d)}\Big]$$

$$= \mathbb{E}\Big[- \frac{2W_j}{n_{d+1}} y W_{jk}^{(d)} + \frac{2W_j}{n_{d+1}^2} W_{jk}^{(d)} \sum_i W_i \Big(\sum_\ell W_{i\ell}^{(d)} z_\ell^{(d-1)} + b_i^{(d)} \Big) \Big]$$

$$= \mathbb{E}\Big[\frac{2W_j^2}{n_{d+1}^2} W_{jk}^{(d)} \Big(\sum_\ell W_{j\ell}^{(d)} z_\ell^{(d-1)} + b_j^{(d)} \Big) \Big]$$

$$= \mathbb{E}\Big[\frac{2W_j^2}{n_{d+1}^2} (W_{jk}^{(d)})^2 z_\ell^{(d-1)} \Big]$$

$$= \frac{2\sigma_{d+1}^2 \sigma_d^2}{n_{d+1}^2} \mathbb{E}\Big[z_\ell^{(d-1)} \Big].$$

Using the chain rule, we get

$$\mathbb{E}\Big[\frac{\partial C}{\partial z_k^{(d-1)}}\Big] = \mathbb{E}\Big[\frac{\partial C}{\partial z_j^{(d)}}\Big]$$

$$\Leftrightarrow n_{d+1} \frac{2\sigma_{d+1}^2 \sigma_d^2}{n_{d+1}^2} \mathbb{E}\Big[z_\ell^{(d-1)} \Big] = \frac{2\sigma_{d+1}^2}{n_{d+1}^2} \mathbb{E}\Big[z_j^{(d)} \Big]$$

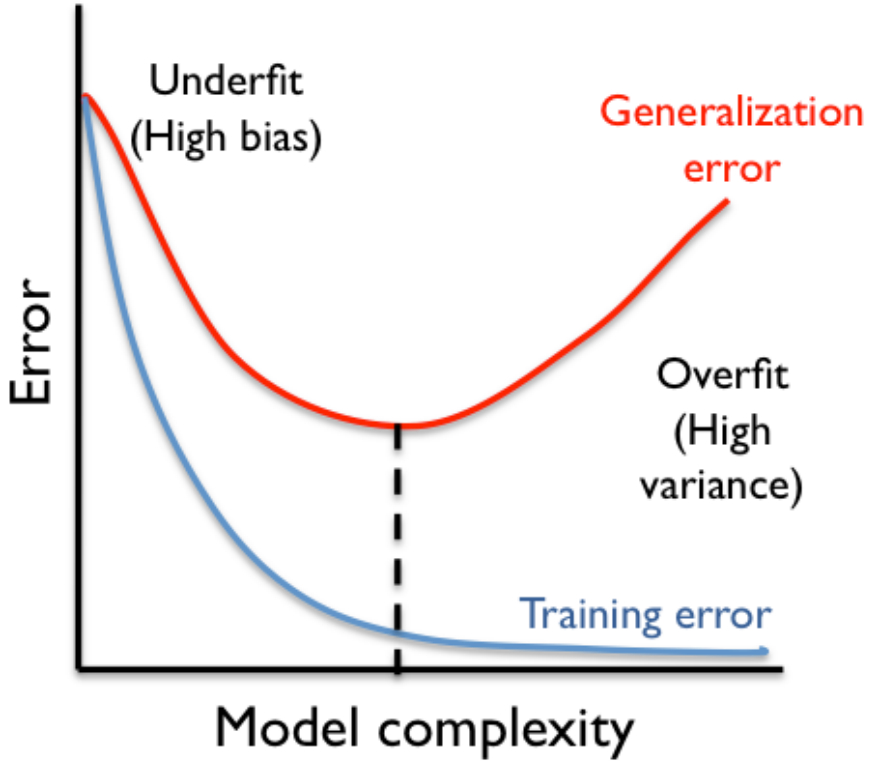
Solution

$$\Leftrightarrow n_{d+1}\sigma_d^2 = \frac{\mathbb{E}\left[z_j^{(d)}\right]}{\mathbb{E}\left[z_\ell^{(d-1)}\right]}.$$

This gives the sufficient condition $n_{d+1}\sigma_d^2 \sim 1$ assuming that the activations of two consecutive hidden layers are similar.

Outline

Regularizing to avoid overfitting



Avoid overfitting by imposing some constraints over the parameter space.

Reducing variance and increasing bias.

Overfitting

Many different manners to avoid overfitting:

- Penalization (L1 or L2)
  Replacing the cost function $\mathcal{L}$ by $\tilde{\mathcal{L}}(\theta, X, y) = \mathcal{L}(\theta, X, y) + \text{pen}(\theta)$.
- Soft weight sharing - see CNN lecture
  Reduce the parameter space artificially by imposing explicit constraints.
- Dropout
  Randomly kill some neurons during optimization and predict with the full network.
- Batch normalization
  Renormalize a layer inside a batch, so that the network does not overfit on this particular batch.
- Early stopping
  Stop the gradient descent procedure when the error on the validation set increases.

Outline

Constraint the optimization problem

$$\min_{\theta} \mathcal{L}(\theta, X, y), \quad \text{s.t. } \text{pen}(\theta) \leq \textit{cste}.$$

Using Lagrangian formulation, this problem is equivalent to:

$$\min_{\theta} \mathcal{L}(\theta, X, y) + \lambda \, \text{pen}(\theta),$$

where

- $\mathcal{L}(\theta, X, y)$ is the loss function (data-driven term)
- pen is a function that increases when θ becomes more *complex* (penalty term)
- $\lambda \geq 0$ is a constant standing for the strength of the penalty term.

For Neural Networks, pen only penalizes the weights and not the bias: the later being easier to estimate than weights.

Example of penalization

$$\min_{\theta} \mathcal{L}(\theta, X, y) + \text{pen}(\theta),$$

- Ridge

$$\text{pen}(\theta) = \lambda \|\theta\|_2^2$$

["Ridge regression: Biased estimation for nonorthogonal problems", Hoerl and Kennard 1970].
See also ["Lecture notes on ridge regression", Wieringen 2015]

- Lasso

$$\text{pen}(\theta) = \lambda \|\theta\|_1$$

["Regression shrinkage and selection via the lasso", Tibshirani 1996]

- Elastic Net

$$\text{pen}(\theta) = \lambda \|\theta\|_2^2 + \mu \|\theta\|_1$$

["Regularization and variable selection via the elastic net", Zou and Hastie 2005]

Simple case: linear regression

Linear regression

The estimate of linear regression $\hat{\beta}$ is given by

$$\hat{\beta} \in \underset{\beta \in \mathbb{R}^d}{\text{argmin}} \sum_{i=1}^{n} (Y_i - \sum_{j=1}^{d} \beta_j x_i^{(j)})^2,$$

which can be written as

$$\hat{\beta} \in \underset{\beta \in \mathbb{R}^d}{\text{argmin}} \| Y - \mathbb{X}\beta \|_2^2,$$

where $\mathbb{X} \in M_{n,d}(\mathbb{R})$.

Solution:

$$\hat{\beta} = (\mathbb{X}'\mathbb{X})^{-1}\mathbb{X}'Y.$$

Penalized regression

Penalized linear regression

The estimate of linear regression $\hat{\beta}_{\lambda,q}$ is given by

$$\hat{\beta}_{\lambda,q} \in \underset{\beta \in \mathbb{R}^d}{\text{argmin}} \|Y - \mathbb{X}\beta\|_2^2 + \lambda \|\beta\|_q^q.$$

- $q = 2$: Ridge linear regression
- $q = 1$: LASSO

Ridge regression, $q = 2$

Ridge linear regression

The ridge estimate $\hat{\beta}_{\lambda,2}$ is given by

$$\hat{\beta}_{\lambda,2} \in \underset{\beta \in \mathbb{R}^d}{\operatorname{argmin}} \|Y - \mathbb{X}\beta\|_2^2 + \lambda \|\beta\|_2^2.$$

Solution:

$$\hat{\beta}_{\lambda,2} = (\mathbb{X}'\mathbb{X} + \lambda I)^{-1}\mathbb{X}'Y.$$

This estimate has a bias equal to $-\lambda(\mathbb{X}'\mathbb{X} + \lambda I)^{-1}\beta$, and a variance $\sigma^2(\mathbb{X}'\mathbb{X} + \lambda I)^{-1}\mathbb{X}'\mathbb{X}(\mathbb{X}'\mathbb{X} + \lambda I)^{-1}$. Note that

$$\mathbb{V}[\hat{\beta}_{\lambda,2}] \leq \mathbb{V}[\hat{\beta}].$$

In the case of orthonormal design ($\mathbb{X}'\mathbb{X} = I$), we have

$$\hat{\beta}_{\lambda,2} = \frac{\hat{\beta}}{1+\lambda} = \frac{1}{1+\lambda} X_j' Y.$$

Sparsity

There is another desirable property on $\hat{\beta}$

If $\hat{\beta}_j = 0$, then feature j has no impact on the prediction:

$$\hat{y} = \text{sign}(x^\top \hat{\beta} + \hat{b})$$

If we have many features (d is large), it would be nice if $\hat{\beta}$ contained **zeros**, and many of them

- Means that only **few** features are statistically relevant.
- Means that only **few** features are useful to predict the label

Leads to a simpler model, with a "reduced" dimension

How do we enforce sparsity in β?

Sparsity

Tempting to solve

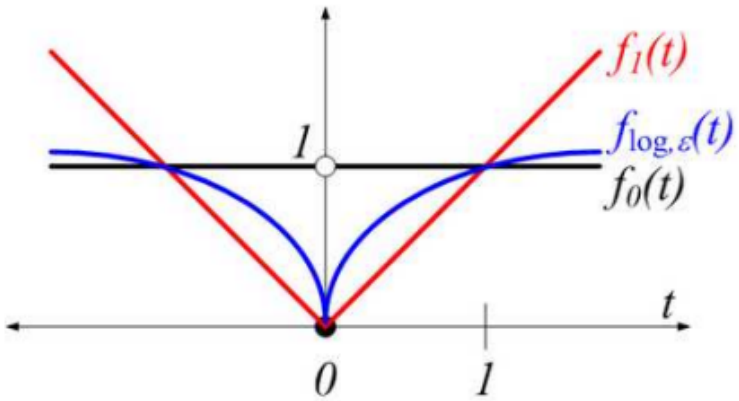
$$\hat{\beta}_{\lambda,0} \in \underset{\beta \in \mathbb{R}^d}{\operatorname{argmin}} \|Y - \mathbb{X}\beta\|_2^2 + \lambda \|\beta\|_0.$$

where

$$\|\beta\|_0 = \#\{j \in \{1, \dots, d\} : \beta_j \neq 0\}.$$

To solve this, explore **all** possible supports of β. Too long! (NP-hard)

Find a convex proxy of $\|\|_0$: the ℓ_1-**norm** $\|\beta\|_1 = \sum_{j=1}^d |\beta_j|$



LASSO

Lasso linear regression

The LASSO (Least Absolute Selection and Shrinkage Operator) estimate of linear regression $\hat{\beta}_{\lambda,1}$ is given by

$$\hat{\beta}_{\lambda,1} \in \underset{\beta \in \mathbb{R}^d}{\text{argmin}} \|Y - \mathbb{X}\beta\|_2^2 + \lambda \|\beta\|_1.$$

Solution: No close form in the general case

If the X_j are orthonormal then

$$\hat{\beta}_{\lambda,1} = X_j' Y \left(1 - \frac{\lambda}{2|X_j' Y|}\right)_+,$$

where $(x)_+ = \max(0, x)$.

Thus, in the very specific case of orthogonal design, we can easily show that L1 penalization implies a sparse vector if the parameter λ is properly tuned.

Sparsity - a picture

Why ℓ_2 (ridge) does not induce sparsity?

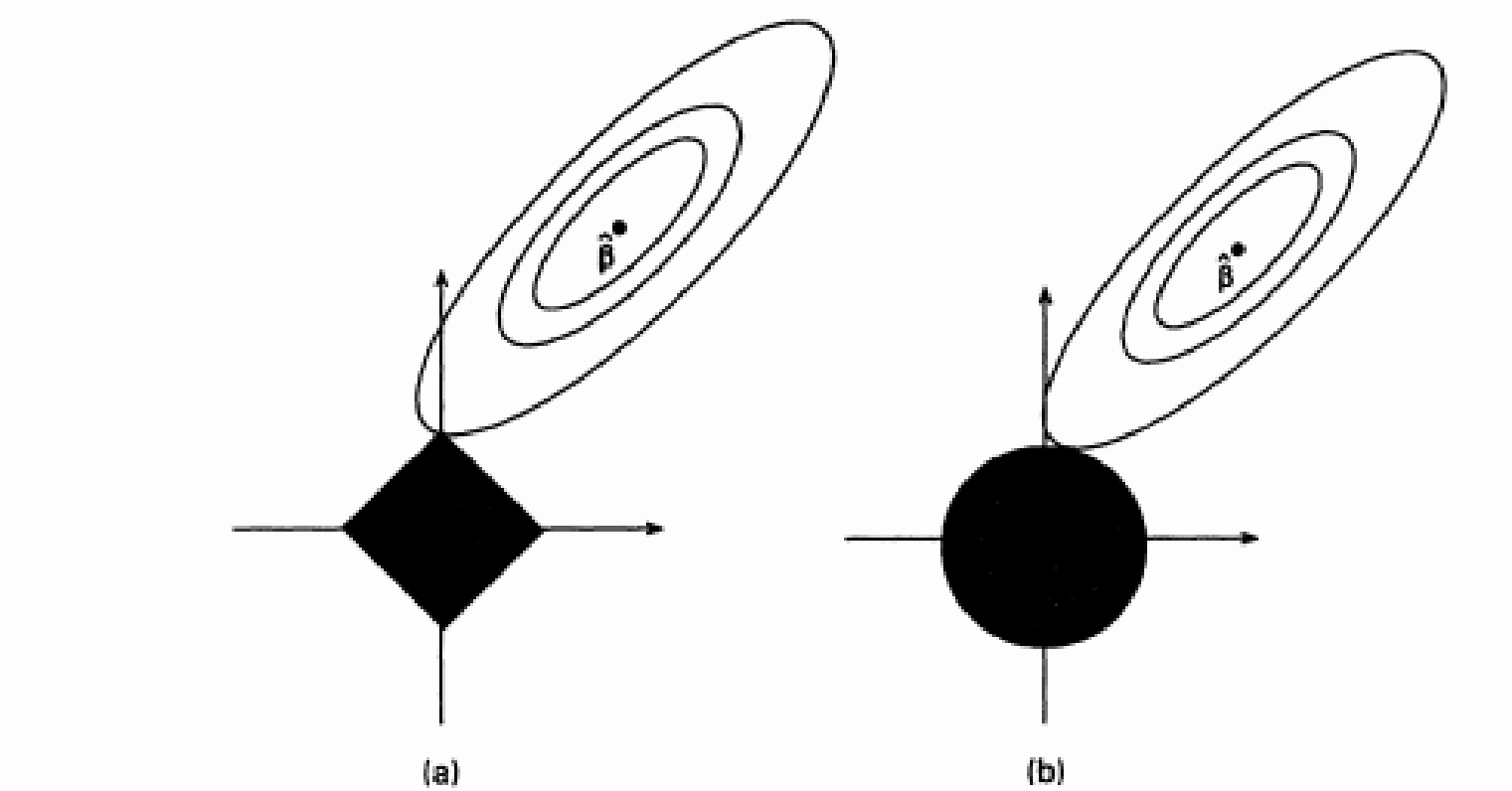


Fig. 2. Estimation picture for (a) the lasso and (b) ridge regression

Outline

Dropout

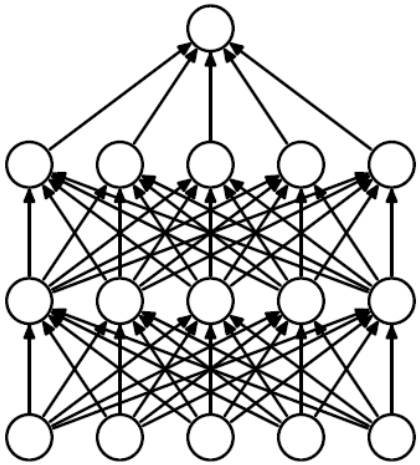
Dropout refers to dropping out units (hidden and visible) in a neural network, i.e., temporarily removing it from the network, along with all its incoming and outgoing connections.

Each unit is independently retained with probability

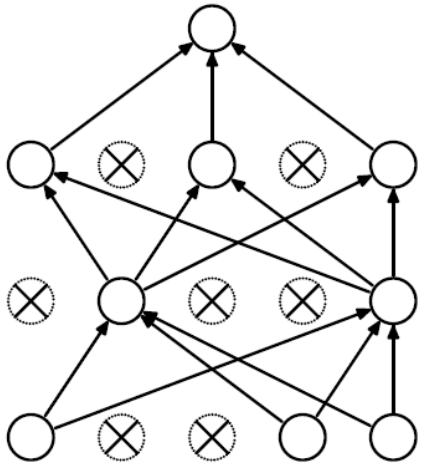
- $p = 0.5$ for hidden units
- $p \in [0.5, 1]$ for input units, usually $p = 0.8$.

["Improving neural networks by preventing co-adaptation of feature detectors", Hinton et al. 2012]

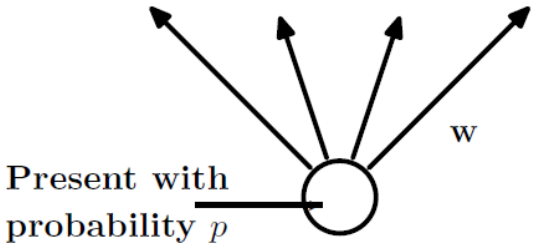
Dropout



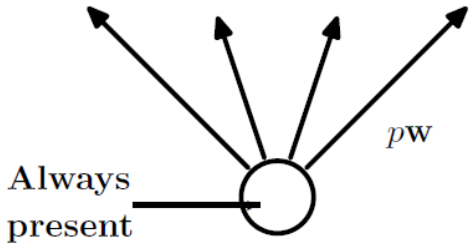
(a) Standard Neural Net



(b) After applying dropout.



(a) At training time



(b) At test time

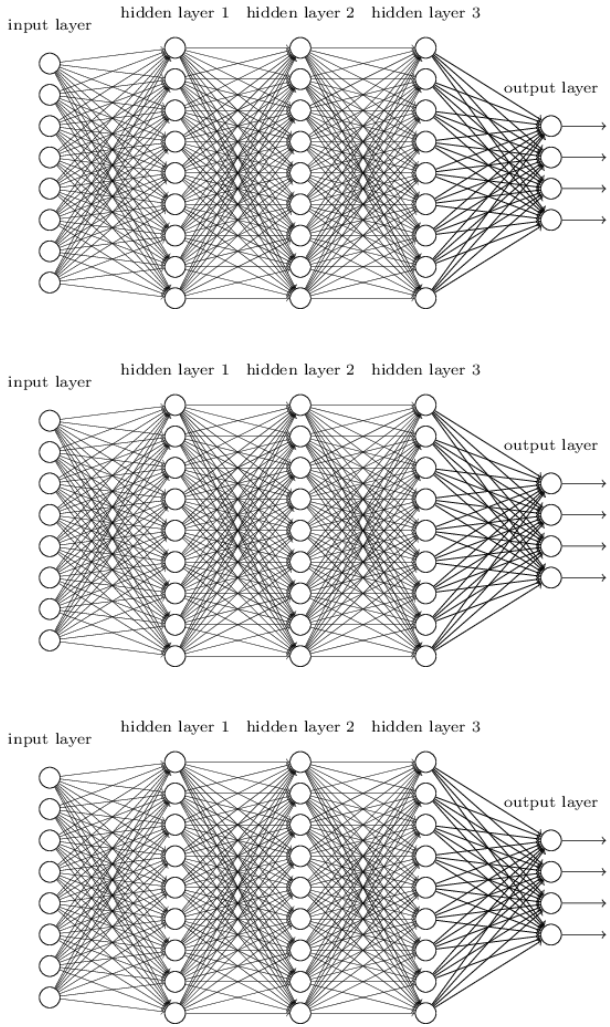
Dropout algorithm

Training step. While *not convergence*

1. Inside one epoch, for each mini-batch of size m,
   1. Sample m different mask. A mask consists in one Bernoulli per node of the network (inner and entry nodes but not output nodes). These Bernoulli variables are *i.i.d.*. Usually
      - ★ the probability of selecting an hidden node is 0.5
      - ★ the probability of selecting an input node is 0.8
   2. For each one of the m observation in the mini-batch,
      - ★ Do a forward pass on the masked network
      - ★ Compute backpropagation in the masked network
      - ★ Compute the average gradient
   3. Update the parameter according to the usual formula.

Prediction step.
Use all neurons in the network with weights given by the previous optimization procedure, times the probability p of being selected (0.5 for inner nodes, 0.8 for input nodes).

Another way of seeing dropout - Ensemble method



Averaging many different neural networks.

Different can mean either:

- randomizing the data set on which we train each network (via subsampling)

  Problem: not enough data to obtain good performance...

- building different network architectures and train each large network separately on the whole training set

  Problem: computationally prohibitive at training time and test time!

["Fast dropout training", Wang and Manning 2013]

["Dropout: A simple way to prevent neural networks from overfitting", Srivastava et al. 2014]

Dropping weights instead of the whole neurons: ["Regularization of neural networks using dropconnect", Wan et al. 2013]

Exercise: linear units

1. Consider a neural networks with linear activation functions. Prove that dropout can be seen as a model averaging method.
2. Given one training example, consider the error of the ensemble of neural network and that of one random neural network sample with dropout:

$$E_{ens} = \frac{1}{2}(y - a_{ens})^2 = \frac{1}{2}(y - \sum_{j=1}^{d} p_i w_i x_i)^2$$

$$E_{single} = \frac{1}{2}(y - a_{single})^2 = \frac{1}{2}(y - \sum_{j=1}^{d} \delta_i w_i x_i)^2,$$

where $\delta_i \in \{0, 1\}$ represents the presence ($\delta_i = 1$) or the absence of a connexion between the input x_i and the output.

Prove that

$$\mathbb{E}[\nabla_w E_{single}] = \nabla_w E_{ens} + \frac{1}{2} \sum_{j=1}^{d} w_i^2 x_i^2 \mathbb{V}(\delta_i).$$

Comment.

Solution

1. Simple calculations...
2. The gradient of the ensemble is given by

$$\frac{\partial E_{ens}}{\partial w_i} = -(y - a_{ens})\frac{\partial a_{ens}}{\partial w_i} = -(y - a_{ens})p_i I_i.$$

and that of a single (random) network satisfies

$$\begin{aligned}\frac{\partial E_{single}}{\partial w_i} &= -(y - a_{single})\frac{\partial a_{single}}{\partial w_i} \\ &= -(y - a_{single})\delta_i I_i \\ &= -t\delta_i x_i + w_i \delta_i^2 x_i^2 + \sum_{j \neq i} w_j \delta_i \delta_j x_i x_j.\end{aligned}$$

Taking the expectation of the last expression gives

$$\begin{aligned}\mathbb{E}\left[\frac{\partial E_{single}}{\partial w_i}\right] &= -tp_i\delta_i + w_i p_i x_i^2 + \sum_{j \neq i} w_i p_i p_j x_i x_j \\ &= -(t - a_{ens})p_i x_i + w_i x_i^2 p_i (1 - p_i).\end{aligned}$$

Therefore, we have

$$\mathbb{E}\left[\frac{\partial E_{single}}{\partial w_i}\right] = \frac{\partial E_{ens}}{\partial w_i} + w_i \mathbb{V}[\delta_i x_i].$$

Solution

In terms of error, we have

$$\mathbb{E}[E_{single}] = E_{ens} + \frac{1}{2}\sum_{j=1}^{d} w_i^2 x_i^2 \mathbb{V}[\delta_i].$$

The regularization term is maximized for $p = 0.5$.

Outline

Batch normalization

The network converges faster if its input are scaled (mean, variance) and decorrelated.

["Efficient backprop", LeCun et al. 1998]

Hard to decorrelate variables: requiring to compute covariance matrix.

["Batch normalization: Accelerating deep network training by reducing internal covariate shift", Ioffe and Szegedy 2015]

Ideas:

- Improving gradient flows
- Allowing higher learning rates
- Reducing strond dependence on initialization
- Related to regularization (maybe slightly reduces the need for Dropout)

Algorithm

See ["Batch normalization: Accelerating deep network training by reducing internal covariate shift", Ioffe and Szegedy 2015]

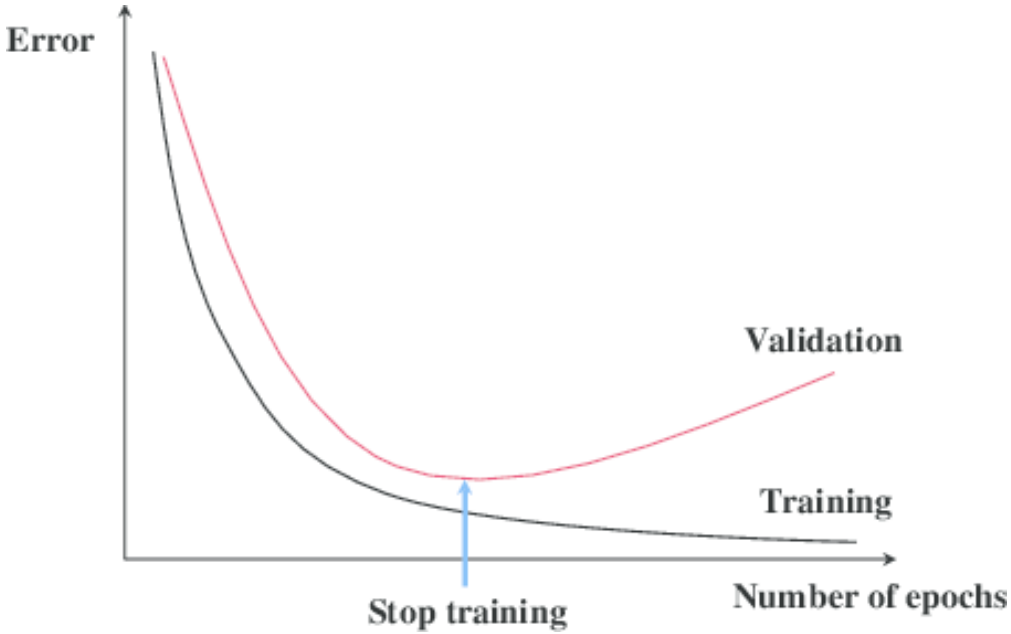
1. For every unit in the first layer,
   1. $\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$
   2. $\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$
   3. $\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$
   4. $y_i = \gamma \hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i)$
2. y_i is fed to the next layer and the procedure iterates.
3. Backpropagation is performed on the network parameters including $(\gamma^{(k)}, \beta^{(k)})$. This returns a network.
4. For inference, compute the average over many training batches $\mathcal{B}$:
$$\mathbb{E}_{\mathcal{B}}[x] = \mathbb{E}_{\mathcal{B}}[\mu_{\mathcal{B}}] \quad \text{and} \quad \mathbb{V}_{\mathcal{B}}[x] = \frac{m}{m-1}\mathbb{E}_{\mathcal{B}}[\sigma_{\mathcal{B}}^2].$$
5. For inference, replace every function $x \mapsto NB_{\gamma,\beta}(x)$ in the network by
$$x \mapsto \frac{\gamma}{\sqrt{\mathbb{V}_{\mathcal{B}}[x] + \varepsilon}} x + \left(\beta - \frac{\gamma \mathbb{E}_{\mathcal{B}}[x]}{\sqrt{\mathbb{V}_{\mathcal{B}}[x] + \varepsilon}}\right).$$

Outline

Early stopping

Idea:

- Store the parameter values that lead to the lowest error on the validation set
- Return these values rather than the latest ones.



Early stopping algorithm

Parameters:

- patience p of the algorithm: number of times to observe no improvement on the validation set error before giving up;
- the number of steps n between evaluations.

1. Start with initial random values θ_0.
2. Let $\theta^\star = \theta_0$, $\mathrm{Err}^\star = \infty$, $j = 0$, $i = 0$.
3. While $j < p$
   1. Update θ by running the training algorithm for n steps
   2. $i = i + n$
   3. Compute the error $\mathrm{Err}(\theta)$ on the validation set
   4. If $\mathrm{Err}(\theta) < \mathrm{Err}^\star$
      - ★ $\theta^\star = \theta$
      - ★ $\mathrm{Err}^\star = \mathrm{Err}(\theta)$
      - ★ $j = 0$

      else $j = j + 1$.
4. Return $\theta^\star$ and the overall number of steps $i^\star = i - np$.

How to leverage on early stopping?

First idea: use early stopping to determine the best number of iterations $i^\star$ and train on the whole data set for $i^\star$ iterations.

Let $X^{(train)}, y^{(train)}$ be the training set.

- Split $X^{(train)}, y^{(train)}$ into $X^{(subtrain)}, y^{(subtrain)}$ and $X^{(valid)}, y^{(valid)}$.
- Run early stopping algorithm starting from random θ using $X^{(subtrain)}, y^{(subtrain)}$ for training data and $X^{(valid)}, y^{(valid)}$ for validation data. This returns $i^\star$ the optimal number of steps.
- Set θ to random values again.
- Train on $X^{(train)}, y^{(train)}$ for $i^\star$ steps.

How to leverage on early stopping?

Second idea: use early stopping to determine the best parameters and the training error at the best number of iterations. Starting from $\theta^\star$, train on the whole data set until the error matches the previous early stopping error.

Let $X^{(train)}, y^{(train)}$ be the training set.

- Split $X^{(train)}, y^{(train)}$ into $X^{(subtrain)}, y^{(subtrain)}$ and $X^{(valid)}, y^{(valid)}$.
- Run early stopping algorithm starting from random θ using $X^{(subtrain)}, y^{(subtrain)}$ for training data and $X^{(valid)}, y^{(valid)}$ for validation data. This returns the optimal parameters $\theta^\star$.
- Set $\varepsilon = \mathcal{L}(\theta^\star, X^{(subtrain)}, y^{(subtrain)})$.
- While $\mathcal{L}(\theta^\star, X^{(valid)}, y^{(valid)}) > \varepsilon$, train on $X^{(train)}, y^{(train)}$ for n steps.

To go further

- Early stopping is a very old idea
  - ["Three topics in ill-posed problems", Wahba 1987]
  - ["A formal comparison of methods proposed for the numerical solution of first kind integral equations", Anderssen and Prenter 1981]
  - ["Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping", Caruana et al. 2001]
- But also an active area of research
  - ["Adaboost is consistent", Bartlett and Traskin 2007]
  - ["Boosting algorithms as gradient descent", Mason et al. 2000]
  - ["On early stopping in gradient descent learning", Yao et al. 2007]
  - ["Boosting with early stopping: Convergence and consistency", Zhang, Yu, et al. 2005]
  - ["Early stopping for kernel boosting algorithms: A general analysis with localized complexities", Wei et al. 2017]

More on reducing overfitting averaging

- Soft-weight sharing:
  ["Simplifying neural networks by soft weight-sharing", Nowlan and Hinton 1992]
- Model averaging:
  Average over: random initialization, random selection of minibatches, hyperparameters, or outcomes of nondeterministic neural networks.
- Boosting neural networks by incrementally adding neural networks to the ensemble
  ["Training methods for adaptive boosting of neural networks", Schwenk and Bengio 1998]
- Boosting has also been applied interpreting an individual neural network as an ensemble, incrementally adding hidden units to the networks
  ["Convex neural networks", Bengio et al. 2006]

Outline

Pipeline for neural networks

- Step 1: Preprocessing the data (substract mean, divide by standard deviation). More complex if data are images.
- Step 2: Choose the architecture (number of layers, number of nodes per layer)
- Step 3:
  1. First, run the network and see if the loss is reasonable (compare with dumb classifier: uniform for classification, mean for regression)
  2. Add some regularization and check that the error on the training set increases.
  3. On a small portion of data, make sure you can overfit when turning down the regularization.
  4. Find the best learning rate
     1. The error does not change too much $\rightarrow$ learning rate too small
     2. The error explodes, NaN $\rightarrow$ learning rate too high
     3. Find a rough range $[10^{-5}, 10^{-3}]$.

Playing with neural network:

`http://playground.tensorflow.org/`

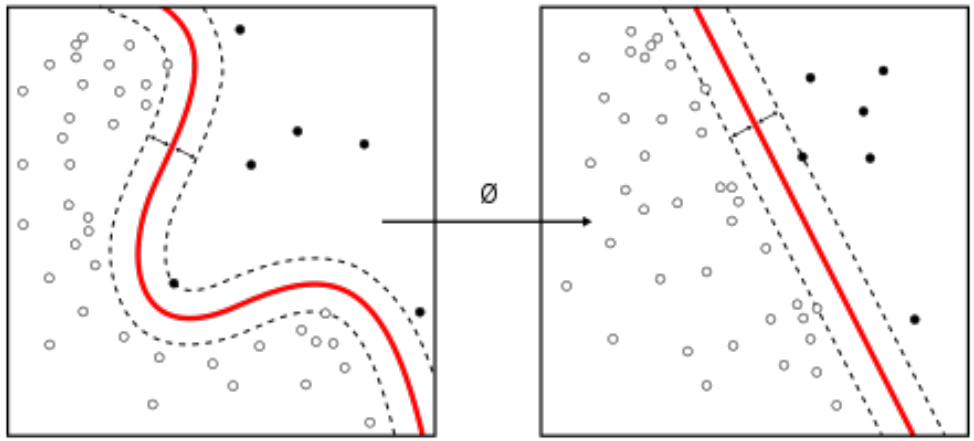
To go further

The kernel perceptron algorithm was already introduced by

["Theoretical foundations of the potential function method in pattern recognition learning", Aizerman 1964].

General idea (work for all methods using only dot product): replace the dot product by a more complex kernel function.

Linear separation in the original space becomes a linear separation in a more complex space, i.e., a non linear separation in the original space.



Margin bounds for the Perceptron algorithm in the general non-separable case were proven by

["Large margin classification using the perceptron algorithm", Freund and Schapire 1999]

and then by

["Perceptron mistake bounds", Mohri and Rostamizadeh 2013]

who extended existing results and gave new L1 bounds.


Mark A Aizerman. "Theoretical foundations of the potential function method in pattern recognition learning". In: *Automation and remote control* 25 (1964), pp. 821–837.


RS Anderssen and PM Prenter. "A formal comparison of methods proposed for the numerical solution of first kind integral equations". In: *The ANZIAM Journal* 22.4 (1981), pp. 488–500.


Yoshua Bengio et al. "Convex neural networks". In: *Advances in neural information processing systems*. 2006, pp. 123–130.


Hans-Dieter Block. "The perceptron: A model for brain functioning. i". In: *Reviews of Modern Physics* 34.1 (1962), p. 123.


John S Bridle. "Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition". In: *Neurocomputing*. Springer, 1990, pp. 227–236.


Widrow Bernard and D Stearns Samuel. "Adaptive signal processing". In: *Englewood Cliffs, NJ, Prentice-Hall, Inc* 1 (1985), p. 491.


Peter L Bartlett and Mikhail Traskin. "Adaboost is consistent". In: *Journal of Machine Learning Research* 8.Oct (2007), pp. 2347–2368.


Rich Caruana, Steve Lawrence, and C Lee Giles. "Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping". In: *Advances in neural information processing systems*. 2001, pp. 402–408.


Meng Cai, Yongzhe Shi, and Jia Liu. "Deep maxout neural networks for speech recognition". In: *Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on*. IEEE. 2013, pp. 291–296.


Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. "Fast and accurate deep network learning by exponential linear units (elus)". In: *arXiv preprint arXiv:1511.07289* (2015).


Kunihiko Fukushima and Sei Miyake. "Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition". In: *Competition and cooperation in neural nets*. Springer, 1982, pp. 267–285.


Yoav Freund and Robert E Schapire. "Large margin classification using the perceptron algorithm". In: *Machine learning* 37.3 (1999), pp. 277–296.


Kunihiko Fukushima. "Cognitron: A self-organizing multilayered neural network". In: *Biological cybernetics* 20.3-4 (1975), pp. 121–136.


Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010, pp. 249–256.


Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep sparse rectifier neural networks". In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. 2011, pp. 315–323.


Ian J Goodfellow, Mehdi Mirza, et al. "An empirical investigation of catastrophic forgetting in gradient-based neural networks". In: *arXiv preprint arXiv:1312.6211* (2013).


Ian J Goodfellow, David Warde-Farley, et al. "Maxout networks". In: *arXiv preprint arXiv:1302.4389* (2013).


Alex Graves. "Generating sequences with recurrent neural networks". In: *arXiv preprint arXiv:1308.0850* (2013).


Kaiming He et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.


DO Hebb. *The organization of behavior: a neuropsychological theory*. Wiley, 1949.


Geoffrey E Hinton et al. “Improving neural networks by preventing co-adaptation of feature detectors”. In: *arXiv preprint arXiv:1207.0580* (2012).


Arthur E Hoerl and Robert W Kennard. “Ridge regression: Biased estimation for nonorthogonal problems”. In: *Technometrics* 12.1 (1970), pp. 55–67.


Michael Jen-Chao Hu. “Application of the adaline system to weather forecasting”. PhD thesis. Department of Electrical Engineering, Stanford University, 1964.


Sergey Ioffe and Christian Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *arXiv preprint arXiv:1502.03167* (2015).


Kevin Jarrett, Koray Kavukcuoglu, Yann LeCun, et al. “What is the best multi-stage architecture for object recognition?” In: *Computer Vision, 2009 IEEE 12th International Conference on*. IEEE. 2009, pp. 2146–2153.


Yann LeCun et al. “Efficient backprop”. In: *Neural networks: Tricks of the trade*. Springer, 1998, pp. 9–50.


Llew Mason et al. “Boosting algorithms as gradient descent”. In: *Advances in neural information processing systems*. 2000, pp. 512–518.


Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. "Rectifier nonlinearities improve neural network acoustic models". In: *Proc. icml*. Vol. 30. 1. 2013, p. 3.


Dmytro Mishkin and Jiri Matas. "All you need is a good init". In: *arXiv preprint arXiv:1511.06422* (2015).


Warren S McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133.


Marvin Minsky and Seymour Papert. "Perceptrons." In: (1969).


Mehryar Mohri and Afshin Rostamizadeh. "Perceptron mistake bounds". In: *arXiv preprint arXiv:1305.0208* (2013).


Steven J Nowlan and Geoffrey E Hinton. "Simplifying neural networks by soft weight-sharing". In: *Neural computation* 4.4 (1992), pp. 473–493.


Albert B Novikoff. *On convergence proofs for perceptrons*. Tech. rep. STANFORD RESEARCH INST MENLO PARK CA, 1963.


Mikel Olazaran. "A sociological study of the official history of the perceptrons controversy". In: *Social Studies of Science* 26.3 (1996), pp. 611–659.


Frank Rosenblatt. “Perceptron simulation experiments”. In: *Proceedings of the IRE* 48.3 (1960), pp. 301–309.


Frank Rosenblatt. *Principles of neurodynamics. perceptrons and the theory of brain mechanisms*. Tech. rep. CORNELL AERONAUTICAL LAB INC BUFFALO NY, 1961.


Prajit Ramachandran, Barret Zoph, and Quoc V Le. “Searching for activation functions”. In: *arXiv preprint arXiv:1710.05941* (2017).


Holger Schwenk and Yoshua Bengio. “Training methods for adaptive boosting of neural networks”. In: *Advances in neural information processing systems*. 1998, pp. 647–653.


Michael Schuster. “On supervised learning from sequential data with applications for speech recognition”. In: *Daktaro disertacija, Nara Institute of Science and Technology* 45 (1999).


Nitish Srivastava et al. “Dropout: A simple way to prevent neural networks from overfitting”. In: *The Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958.


LR Talbert, GF Groner, and JS Koford. “Real-Time Adaptive Speech-Recognition System”. In: *The Journal of the Acoustical Society of America* 35.5 (1963), pp. 807–807.


Robert Tibshirani. “Regression shrinkage and selection via the lasso”. In: *Journal of the Royal Statistical Society. Series B (Methodological)* (1996), pp. 267–288.


Grace Wahba. “Three topics in ill-posed problems”. In: *Inverse and ill-posed problems*. Elsevier, 1987, pp. 37–51.


Li Wan et al. “Regularization of neural networks using dropconnect”. In: *International conference on machine learning*. 2013, pp. 1058–1066.


Bernard Widrow and Marcian E Hoff. *Adaptive switching circuits*. Tech. rep. Stanford Univ Ca Stanford Electronics Labs, 1960.


Wessel N van Wieringen. “Lecture notes on ridge regression”. In: *arXiv preprint arXiv:1509.09169* (2015).


Sida Wang and Christopher Manning. “Fast dropout training”. In: *international conference on machine learning*. 2013, pp. 118–126.


Capt Rodney Winter and B Widrow. “Madaline Rule II: a training algorithm for neural networks”. In: *Second Annual International Conference on Neural Networks*. 1988, pp. 1–401.


Yuting Wei, Fanny Yang, and Martin J Wainwright. "Early stopping for kernel boosting algorithms: A general analysis with localized complexities". In: *Advances in Neural Information Processing Systems*. 2017, pp. 6067–6077.


Bing Xu et al. "Empirical evaluation of rectified activations in convolutional network". In: *arXiv preprint arXiv:1505.00853* (2015).


Yuan Yao, Lorenzo Rosasco, and Andrea Caponnetto. "On early stopping in gradient descent learning". In: *Constructive Approximation* 26.2 (2007), pp. 289–315.


Tong Zhang, Bin Yu, et al. "Boosting with early stopping: Convergence and consistency". In: *The Annals of Statistics* 33.4 (2005), pp. 1538–1579.


Hui Zou and Trevor Hastie. "Regularization and variable selection via the elastic net". In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 67.2 (2005), pp. 301–320.