TDT4171 Artificial Intelligence Methods

Lecture 7 – Learning from Observations

Norwegian University of Science and Technology

Helge Langseth
Gamle Fysikk 255
helge.langseth@ntnu.no

NTNU

Outline

Summary from last time

- **Sequential decision problems**
  - **Assumptions:** Stationarity, Markov assumption, Additive rewards, infinite horizon with discount
  - **Model classes:** Markov decision processes/Partially Observable Markov Decision Processes
  - **Algorithm:** Value iteration / policy iteration
- Intuitively, MDPs combine **probabilistic models over time** (filtering, prediction) with **maximum expected utility principle**.

Reference Group meeting:
We'll have a meeting in the RefGrp next week. Send them an email if you have feedback.

Learning

This is the second part of the course:

- We have learned about **representations** for uncertain knowledge
- **Inference** in these representations
- Making **decisions** based on the inferences

Now we will talk about learning the representations:

- Supervised learning – When focus is on learning "a mapping"
- Reinforcement learning – When focus is on learning "to behave"
- There is a third category of learning, unsupervised learning, that we won't cover in this course.

Plan for Part II of the course

Today: Recap: Machine learning, neural nets, deep nets

March 7th: **Instance based learning and CBR.**
Guest lecture, Kerstin Bach

March 14th: More deep learning

March 21st: (Deep) Reinforcement Learning

March 28th: NLP including RLHF, and Transformers

April 4th: Summary

April 11th: Class trip. **No lecture**

April 18th: Easter. **No lecture**

April 25th: Buffer. Hopefully **No lecture**

Assignments: We keep releasing them according to current schedule. Last assignment (Assignment 10) has planned deadline April 3rd. 7 out of 10 needed.

Learning goals for today

Being familiar with:

- **Motivation** and **Formalization** for learning
- **Fundamental issues** like learning bias, overfitting
- **Neural net** basics and main idea for learning
- **Deep Learning** basics

Recap: Learning from observations

Why do Learning?

Arthur Samuel playing Checkers, 1956

Well-defined learning problem

What "parts" are needed to formally define a learning problem?
. . . and how can Arthur Samuel describe his problem in that way?

Discuss with your neighbour for a couple of minutes.

Well-defined learning problem

What "parts" are needed to formally define a learning problem?
... and how can Arthur Samuel describe his problem in that way?

One classically relates a learning-problem to three objects: **Task** T, **Performance measure** P, and **Experience** E:

- Improve over task T:
  - Playing checkers.
- ... with respect to performance measure P:
  - Games (out of 100, say) won against a fixed opponent.
- ... based on experience E:
  - Playing against itself to generate experience to learn from.

Why do Learning?

- Learning **modifies the agent's decision mechanisms** to improve performance
- Learning is essential for **unknown environments** (when designer lacks omniscience)
- Learning is useful as a **system construction** method (expose the agent to reality rather than trying to write it down)

Currently we see lots of work in machine learning, due to:

- Availability of data massively increasing
- Increased utilization of hardware architectures (like GPUs)
- Method development (like deep learning)
- Clever definition of new tasks as machine learning problems

Inductive learning

Simplest form: Learn a function from examples

f is the **target function**

An **example** is a pair $\{x, f(x)\}$, e.g.,

$$\left\{ \begin{array}{c|c|c} O & O & X \\ \hline & X & \\ \hline X & & \end{array} \; , \quad +1 \right\}$$

Problem:
Find **hypothesis** $h \in H$ s.t. $h \approx f$ given a **training set** of examples

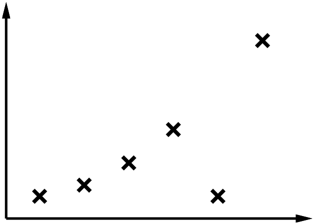
This is a highly simplified model of real learning:

- Ignores **prior knowledge**
- Assumes a **deterministic**, **observable** "environment"
- Assumes examples are **given**

Inductive learning method

- Construct/adjust h to agree with f on training set
- h is **consistent** if it agrees with f on all examples

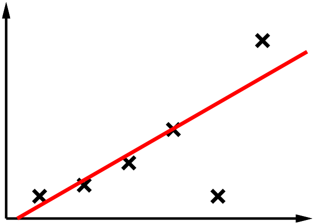
Example – curve fitting:



Inductive learning method

- Construct/adjust h to agree with f on training set
- h is **consistent** if it agrees with f on all examples

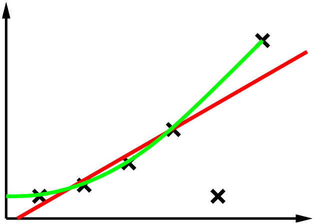
Example – curve fitting:



Inductive learning method

- Construct/adjust h to agree with f on training set
- h is **consistent** if it agrees with f on all examples

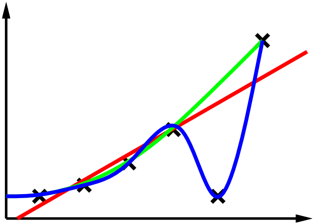
Example – curve fitting:



Inductive learning method

- Construct/adjust h to agree with f on training set
- h is **consistent** if it agrees with f on all examples

Example – curve fitting:

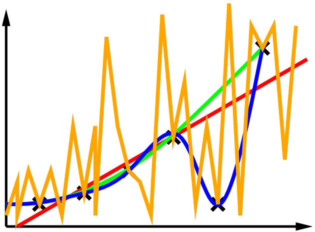


Which curve is better? – and WHY?
Can we make an operational definition?
Discuss with your neighbour for a couple of minutes.

Inductive learning method

- Construct/adjust h to agree with f on training set
- h is **consistent** if it agrees with f on all examples

Example – curve fitting:

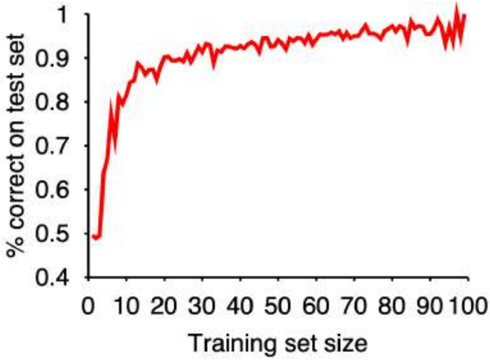


Ockham's razor: maximize consistency and simplicity.
Key insight: We don't necessarily aim to be consistent.
Rather, we typically aim to make good predictions!

Performance measurement

Question: How do we know that $h \approx f$?
Answer: Try h on a new **test set** of examples (use **same distribution over example space** as training set)

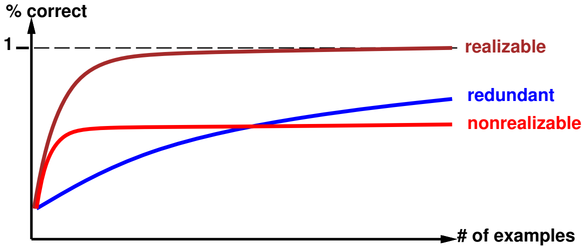
Learning curve = % correct on **test set** as a function of training set size



Performance measurement contd.

Learning curve depends on. . .

- **realizable** (can express target function) vs. **non-realizable**
- **non-realizability** can be due to missing attributes or restricted hypothesis class (e.g., thresholded linear function)
- **redundant expressiveness** (e.g., loads of irrelevant attributes)



Overfitting

Consider error of hypothesis h over

- Training data: $\text{error}_t(h)$
- Entire distribution $\mathcal{D}$ of data (often approximated by measurement on test-set): $\text{error}_{\mathcal{D}}(h)$

Overfitting

Hypothesis $h \in H$ **overfits** training data if there is an alternative hypothesis $h' \in H$ such that

$$\text{error}_t(h) < \text{error}_t(h') \text{ and } \text{error}_{\mathcal{D}}(h) > \text{error}_{\mathcal{D}}(h')$$

Avoiding Overfitting

- Overfitting often occur for flexible learning representations (like high-order polynomials, ...)
- Overfitting harms the usefulness of the machine learning system, because it is the **generalization ability** (score on a test-set) that is important!

What techniques can be used to prevent overfitting for ML in general (not specific to particular learning algorithm)?

Discuss with your neighbour for a couple of minutes.

Avoiding Overfitting

- Overfitting often occur for flexible learning representations (like high-order polynomials, . . .)
- Overfitting harms the usefulness of the machine learning system, because it is the **generalization ability** (score on a test-set) that is important!

What techniques can be used to prevent overfitting for ML in general (not specific to particular learning algorithm)?

- Compare models' actual **generalization ability** using a validation set or some statistical technique on training data.
- Use some heursitic, like bias model search towards **simplicity** (e.g., linear over high order polynomial fit)

Recap: Basics of neural nets

Connectionist Models

Some facts about the human brain:

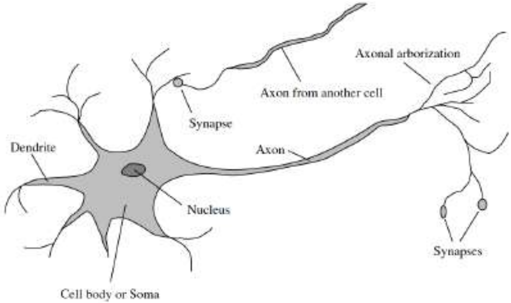
- Number of neurons about 10^{11}
- Connections per neuron about 10^4 – 10^5
- Scene recognition time about $.1$ second
- Neuron switching time about $.001$ second
- 100 inference steps doesn't seem like enough for scene recognition $\rightarrow$ much parallel computation

Properties of artificial neural nets (ANN's):

- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
- Emphasis on tuning weights automatically

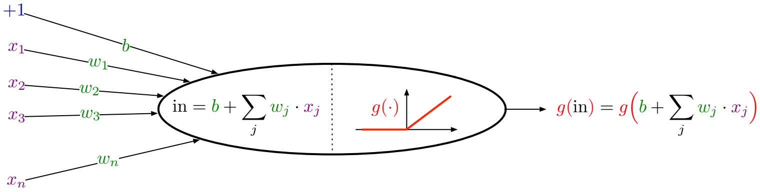
Brains

10^{11} neurons of > 20 types, 10^{14} synapses
Signals are noisy “spike trains” of electrical potential



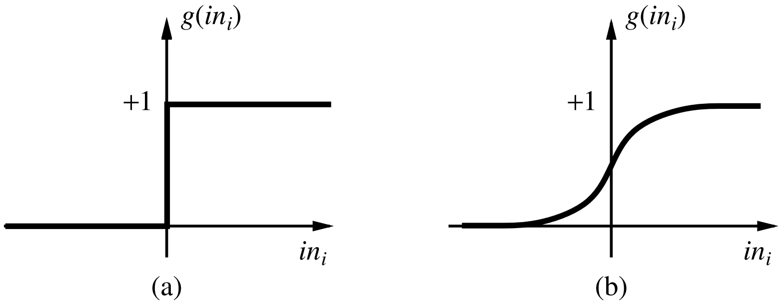
Perceptron

Output is a **nonlinear function** of the inputs with offset:



A gross **oversimplification** of real neurons, but its purpose is to develop understanding of what networks of simple units can do.

Activation functions



(a) is a **step function** or **threshold function**

(b) is a **sigmoid** function $1/(1+e^{-x})$

(c) We will also consider **linear** functions and **rectified linear units** (ReLUs).

Note! Changing the bias b shifts the output along the x-axis

Finding the optimal weights

Let's start with a **linear unit**, where input is $\vec{x}$ gives output

$$o = \underbrace{w_0 \cdot \overbrace{x_0}^{x_0=1}}_{\text{per convention: } b} + w_1 x_1 + \cdots + w_n x_n = \vec{\boldsymbol{w}}^{\mathsf{T}} \vec{\boldsymbol{x}}$$

We learn w_i's that minimise some loss. For regression models it makes sense to use the squared error

$$\mathcal{L}[\vec{\boldsymbol{w}}] = \sum_{d \in \mathcal{D}} (t_d - o_d)^2,$$

where $\mathcal{D}$ is set of training examples, each of the form $d = \langle \boldsymbol{x}_d, t_d \rangle$.

Finding optimal weights – requirements

Description of our situation:

- We have a function $\mathcal{L}[\vec{w}]$ we want to minimise (wrt $\vec{w}$).
- **Why not just try a number of weight configurations $\vec{w}_i$, calculate $\mathcal{L}[\vec{w}_i]$ and see what happens?**
- There are infinitely many (even uncountably many) weight configurations.
- Minimization is typically in very high dimensional space.
- Evaluating $\mathcal{L}[\vec{w}]$ involves summing over all training examples – can be very expensive.

We cannot use a standard trial & error approach, but must devise a local search method. **What can we do instead?**

Discuss with your neighbour for a couple of minutes.

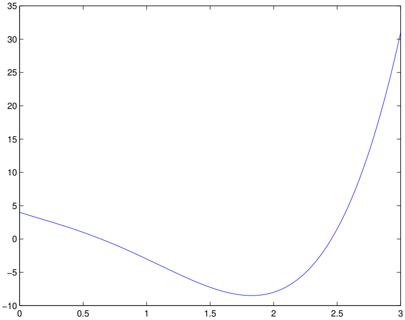
Gradient Descent – The setup

- We want to find the value x which minimizes $f(x)$.
  Yes, $f(x)$ will be replaced by $\mathcal{L}[\vec{\boldsymbol{w}}]$ later on!
- To avoid evaluating the whole function we use an iterative approach:
  - Guess a value for x
  - Calculate the derivative at x.
  - Make a new guess for x based on the calculated information
  - . . . and keep going.
- **Intuition:**
  - If the derivative is zero then we are done
  - If it is small (in absolute value) we are close to the minimizing point
  - If it is large we are not that close

Solution:
Use update rule $x_{i+1} \leftarrow x_i - \eta \cdot f'(x_i)$. $\eta > 0$ is the **learning rate**.

Gradient Descent – Example

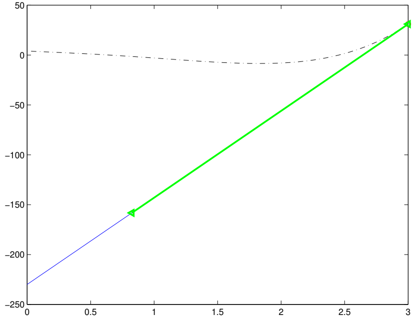
Minimize $f(x) = 2x^4 - 5x^3 + 2x^2 - 6x + 4$ with $\eta = 0.025$.



The $f(x)$ has a minimum at $x = 1.8261$. Let's try to find it. . .

Gradient Descent – Example

Minimize $f(x) = 2x^4 - 5x^3 + 2x^2 - 6x + 4$ with $\eta = 0.025$.

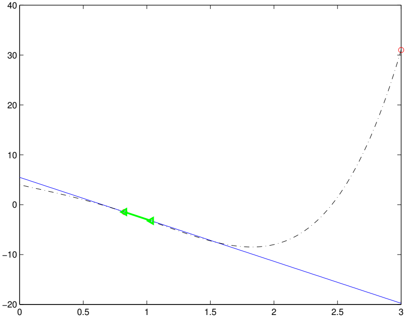


Starting from $x_0 = 3$ and finding $f'(3) = 87$:

$$\begin{aligned} x_1 &= x_0 - \eta f'(x_0) \\ &= 3 - 0.025 \cdot 87 = 0.8250 \end{aligned}$$

Gradient Descent – Example

Minimize $f(x) = 2x^4 - 5x^3 + 2x^2 - 6x + 4$ with $\eta = 0.025$.

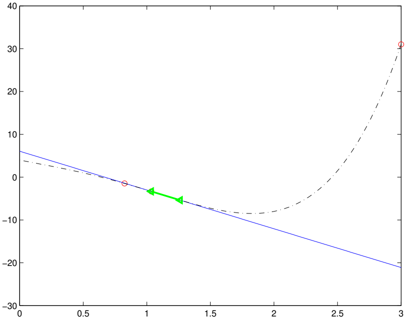


Going from $x_1 = 0.8250$ with $f'(0.8250) = -8.4172$:

$$\begin{aligned} x_2 &= x_1 - \eta f'(x_1) \\ &= 0.825 - 0.025 \cdot (-8.4172) = 1.0354 \end{aligned}$$

Gradient Descent – Example

Minimize $f(x) = 2x^4 - 5x^3 + 2x^2 - 6x + 4$ with $\eta = 0.025$.

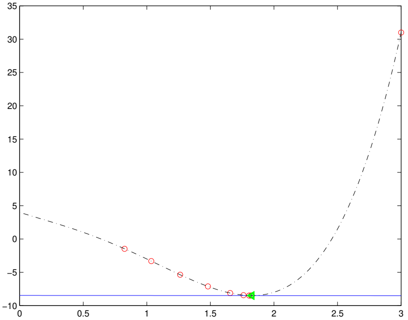


Going from $x_2 = 1.0354$ with $f'(1.0354) = -9.0592$:

$$x_3 = 1.0354 - 0.025 \cdot (-9.0592) = 1.2619$$

Gradient Descent – Example

Minimize $f(x) = 2x^4 - 5x^3 + 2x^2 - 6x + 4$ with $\eta = 0.025$.

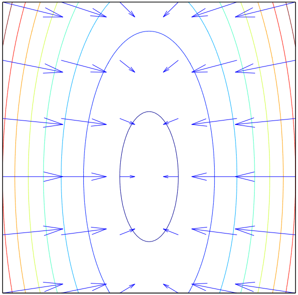


…and finally going from $x_{10} = 1.8260$ with $f'(1.8260) = -0.0034$:

$$x_{11} = 1.8260 - 0.025 \cdot (-0.0034) = 1.8261$$

…and we are done.

Gradient Descent – in higher dimensions



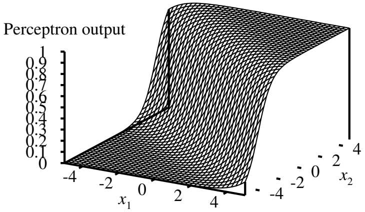
Recall that

- The **gradient** of a surface $\mathcal{L}[\vec{w}]$ is a vector in the direction the curve grows the most (calculated at $\vec{w}$).
- The gradient is calculated as $\nabla \mathcal{L}[\vec{w}] \equiv \left[\frac{\partial \mathcal{L}}{\partial w_0}, \frac{\partial \mathcal{L}}{\partial w_1}, \cdots \frac{\partial \mathcal{L}}{\partial w_n}\right]$.

Training rule: $\Delta \vec{w} = -\eta \cdot \nabla \mathcal{L}[\vec{w}]$, i.e., $\Delta w_i = -\eta \cdot \frac{\partial \mathcal{L}}{\partial w_i}$.

The perceptron's problem: Expressibility

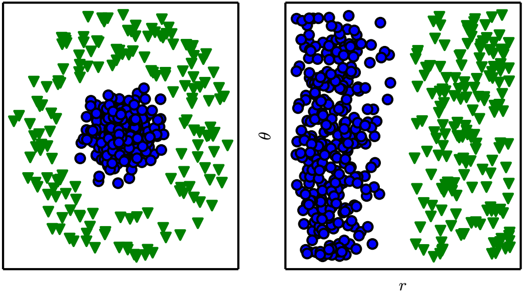
Output from perceptron on input (x_1, x_2)



- Adjusting weights moves the location, orientation, and steepness of cliff
- Cannot tackle "correlation effects" of non-separable targets
- Solution: Make **layers** of nodes. **All continuous functions representable w/ 1 hidden layer, all functions w/ 2**

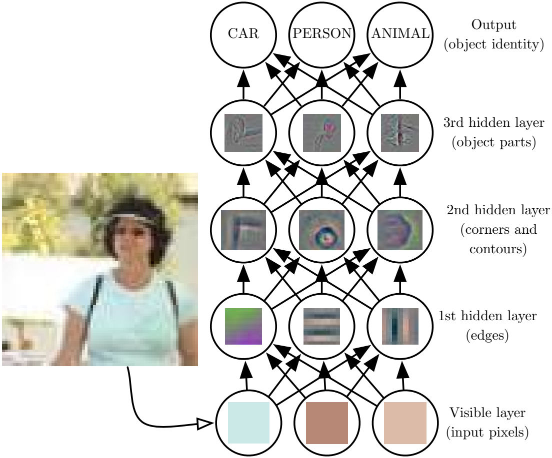
Recap: Deep Learning basics

Representation matters

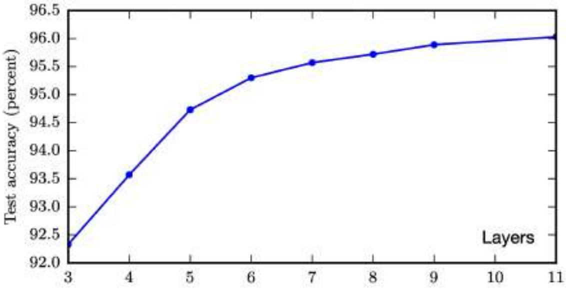


- Representation in cartesian coordinates (x and y) is "difficult": Not linearly separable.
- Representation in polar coordinates (r and θ) is "easy". **But how do we find this representation?**

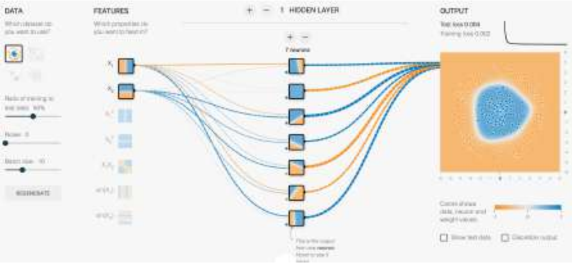
Depth: Compositions repeated



Depth: Compositions repeated



Tensorflow playground



https://playground.tensorflow.org/

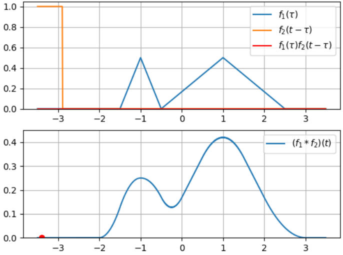
Convolutional Neural nets

- **Goal:** Scale up to process very large images/videos
  - Sparse connections
  - Parameter sharing
  - Automatically generalize across spatial translations of inputs
  - Applicable to any input laid out on a grid (1-D, 2-D, 3-D, ...) and other data with spatial structure

- **Key idea:** Replace/replicate flattened representations and matrix multiplication with convolution that respect locality of information!
  - Everything else stays the same
  - Optimization criteria
  - Training algorithm
  - And so on

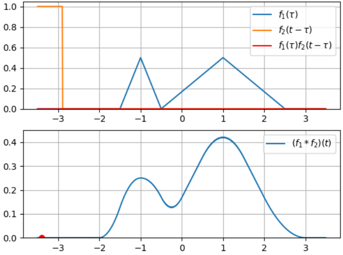
Convolutions

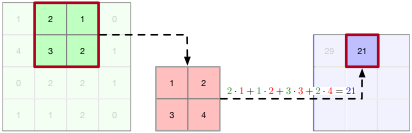
Mathematical definition: $(f_1 * f_2)(t) = \int_{-\infty}^{\infty} f_1(\tau) \cdot f_2(t-\tau)\mathrm{d}\tau$



Convolutions

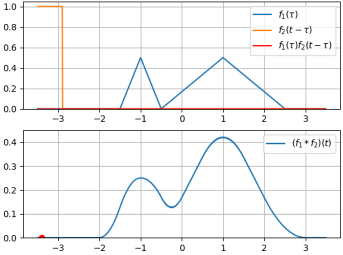
Mathematical definition: $(f_1 * f_2)(t) = \int_{-\infty}^{\infty} f_1(\tau) \cdot f_2(t-\tau)\mathrm{d}\tau$

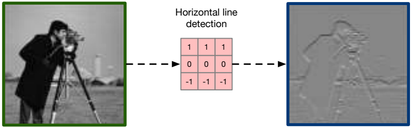




Convolutions

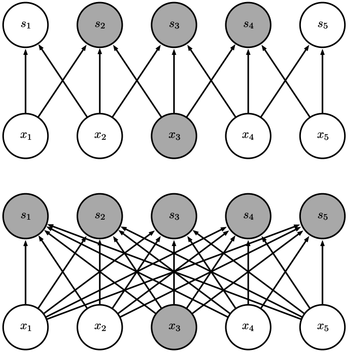
Mathematical definition: $(f_1 * f_2)(t) = \int_{-\infty}^{\infty} f_1(\tau) \cdot f_2(t-\tau)\mathrm{d}\tau$



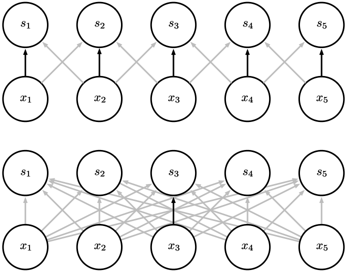


Convolutional Neural nets: Efficiency

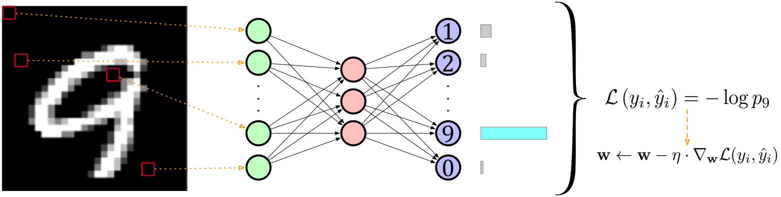
Limited "spread":



Reuse of params:



Implementation of DL models



Implementation frameworks:

- **tensorflow.keras:** High level of abstraction, easy to get going, used in enterprise systems and cloud platforms.
- **pytorch:** More coding required, far easier to customize for semi-advanced extensions.

Model types:

- **Feed forward:** Simple but ineffective
- **Conv.nets:** Parameter efficient; similar (slightly better) quality

Expressive Capabilities of ANNs

Boolean functions:

- Every boolean function can be represented by network with a single hidden layer
- Note: We might require exponential (in number of inputs) hidden units

Continuous functions:

- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers

Depth adds to expressiveness

- Overfitting can be a massive issue
- Techniques for **regularization** becomes very improtant

Regularization in deep learning

Remember the def of overfitting

Hypothesis $h \in H$ **overfits** training data if there is an alternative hypothesis $h' \in H$ such that

$$\text{error}_t(h) < \text{error}_t(h') \text{ and } \text{error}_{\mathcal{D}}(h) > \text{error}_{\mathcal{D}}(h')$$

And now: Regularization

Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.

Types of regularization:

- Norm penalty
- Dropout
- ...

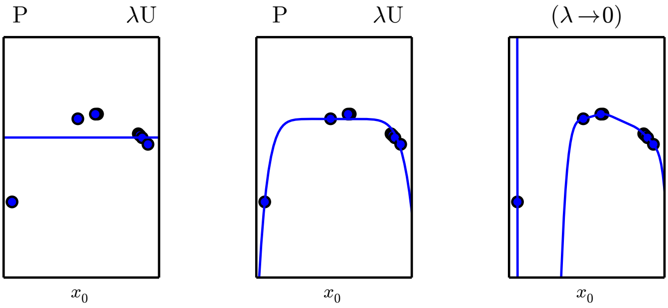
Norm penalties

- Remember our objective: Optimize some loss $\mathcal{L}_{\text{data}}(\boldsymbol{w})$ where $\boldsymbol{w}$ are the weights in the deep net
- Norm penalty (a.k.a. *weight regularization*) penalizes "long $\boldsymbol{w}$-vectors":

$$\mathcal{L}_{\text{norm}}(\boldsymbol{w}) = \|\boldsymbol{w}\|_p = \left(\sum_j w_j^p\right)^{1/p}$$

- Total loss: $\mathcal{L}(\boldsymbol{w}) = \mathcal{L}_{\text{data}}(\boldsymbol{w}) + \lambda \cdot \mathcal{L}_{\text{norm}}(\boldsymbol{w})$. Note the λ, which balances the two losses.
- Typical examples:
  - $p = 1$: Encourages sparsity (some weights "exactly" zero)
  - $p = 2$: Typically results in small but non-zero weights

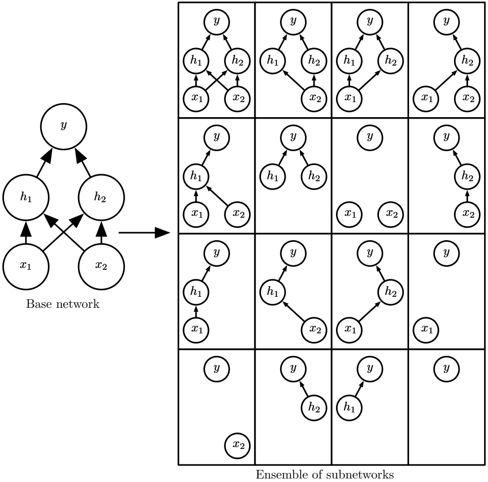
Norm penalties



$$\mathcal{L}(\boldsymbol{w}) = \mathcal{L}_{\text{data}}(\boldsymbol{w}) + \lambda \cdot \|\boldsymbol{w}\|_p$$

Dropout

Dropout: To randomly drop a fraction of neurons (or sometimes also edges) at each training iteration.



Summary

- **Learning:** Use data to generate (or improve) a representation.
- **Inductive learning hypothesis:** "Old data has relevance for new problems"
- **Neural networks:** Capable structures defined by combining many simple computational units
  - Simple perceptrons
  - Layered models $\rightarrow$ Feed forward networks
  - DL models can also contain more complicated structures
- **Learning:** Define a loss, minimize using gradient descent
- **Overfitting:** A model overfits if it does well on the training data, but fails to generalize
  - Prefer simplicity
  - Can be enforced using, e.g., norm penalties; dropout

Next week: Guest lecture by Kerstin Bach: CBR