

TDT4171 Artificial Intelligence Methods

Lecture 8 – Basic ANNs

Norwegian University of Science and Technology

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





1 Summary from last time

2 Introduction to ANNs

- Background
- Perceptrons
- Gradient descent for perceptrons

3 Multilayer networks

- Setup
- Backpropagation
- XAI: What does the model actually encode?
- Overfitting

4 Summary



Summary from last time

- Learning needed for **unknown environments**, **lazy designers**
- **Learning agent = performance element + learning element**
- Learning method depends on **type of performance element**, available **feedback**, type of **component to be improved**, and its **representation**
- For **supervised learning**, the aim is to find a simple hypothesis that is approximately consistent with training examples
- Decision tree learning using **information gain**
- **Learning performance = prediction accuracy measured on test set**

Reference group meeting:

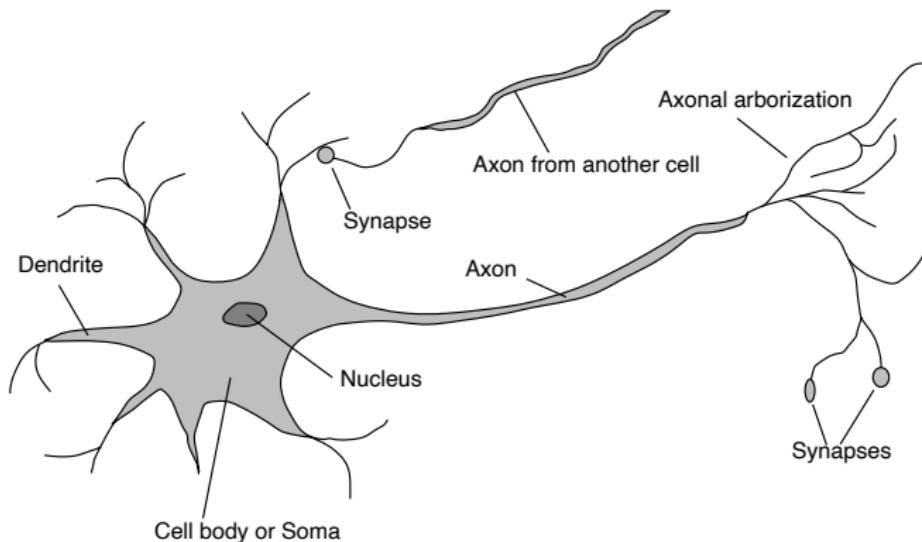
Meeting held some days ago. Nothing major came up.

Brains



10^{11} neurons of > 20 types, 10^{14} synapses

Signals are noisy “spike trains” of electrical potential





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 → 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



When to Consider Neural Networks

- Input is high-dimensional discrete or real-valued (e.g. raw sensor input)
- Output is discrete or real valued
- Output is a vector of values
- Possibly noisy data
- Form of target function is unknown
- Human readability of result is unimportant

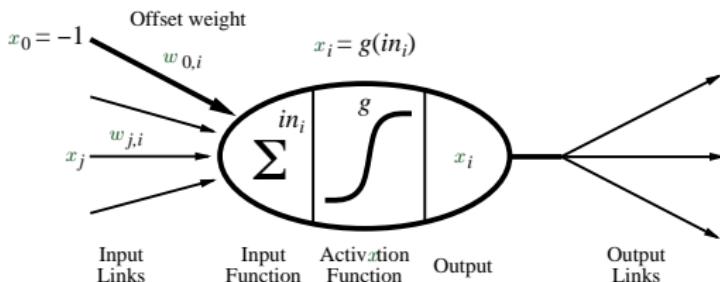
Examples:

- Speech recognition
- Image classification
- Text understanding
- Autonomous drivers
- Financial prediction

Perceptron

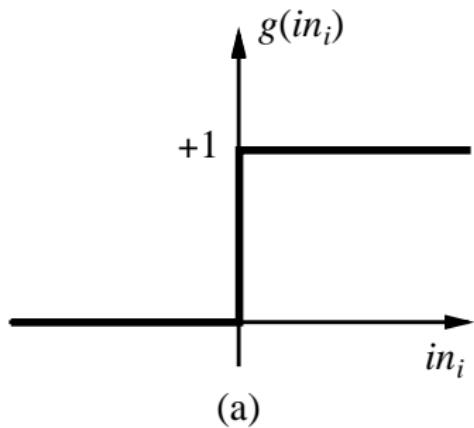
Output is typically a **non-linear function** of the inputs (e.g., a “squashed” one):

$$x_i \leftarrow g(in_i) = g \left(\sum_j w_{ji} x_j \right)$$

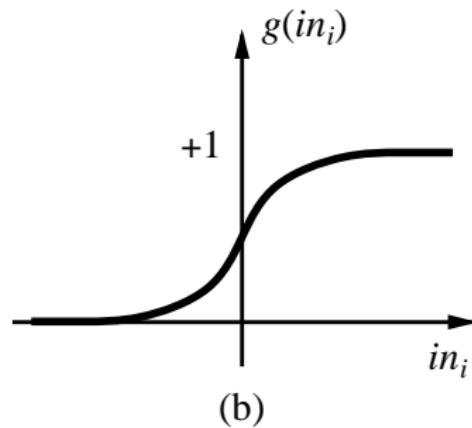


A gross **oversimplification** of real neurons, but its purpose is to develop understanding of what networks of simple units can do.

Activation functions



(a)



(b)

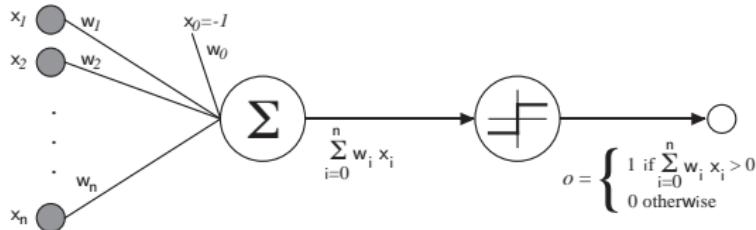
- (a) is a **step function** or **threshold function**
- (b) is a **sigmoid** function $1/(1 + e^{-x})$
- (c) We will also consider **linear** functions today, and **rectified linear units** (ReLUs) when we get to deep learning.

Note! Changing the bias weight $w_{0,i}$ moves the threshold location

Perceptron activated by step-function



Start by looking at **single perceptron with step-function activation:**



$$\text{output}(x_1, \dots, x_n) = \begin{cases} 1 & \text{if } w_0 x_0 + w_1 x_1 + \dots + w_n x_n > 0 \\ 0 & \text{otherwise.} \end{cases}$$

Alternative vector notation:

$$\text{output}(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w}^\top \vec{x} > 0 \\ 0 & \text{otherwise.} \end{cases}$$

Decision Surface of a Perceptron



A perceptron activated by a **step-function** can represent some useful binary functions.

Discuss with your neighbour:

What weights (if any) represent $\text{output}(x_1, x_2) = \text{AND}(x_1, x_2)$?

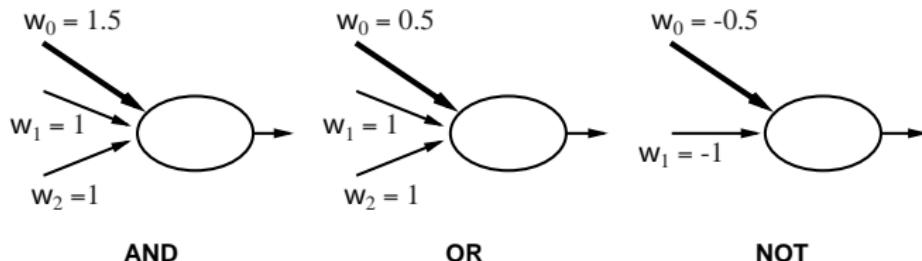
$\text{output}(x_1, x_2) = \text{OR}(x_1, x_2)$?

$\text{output}(x_1, x_2) = \text{XOR}(x_1, x_2)$?

$\text{output}(x_1) = \text{NOT}(x_1)$?



Implementing logical functions



- Some functions **not representable** by perceptrons, namely those that are **not linearly separable** (e.g., XOR).
- Can make it work by “**inventing**” **new inputs** to make it linearly separable (e.g., $x_3 \leftarrow |x_1 - x_2|$ for XOR).
- **Equivalently**, if we have **networks** of perceptrons, **any Boolean function** can be implemented



XOR: Why only linearly separable decisions?

Let us consider the formulation for the “step” (i.e., the decision boundary) defined by $w_0x_0 + w_1x_1 + \cdots + w_nx_n = 0$.

- Pick out one of the data dimensions with non-zero weight, x_i (so $w_i \neq 0$; this is not a big deal because an x_i with corresponding $w_i = 0$ plays no role in the decision and can be thrown out).
- Wrt x_i , the decision boundary is given by

$$x_i = \sum_{j \neq i} \frac{-w_j}{w_i} x_j \quad \underbrace{=}_{\gamma_j := -w_j/w_i} \sum_{j \neq i} \gamma_j x_j$$

... meaning that the decision boundary is a linear function in x_i , and this is true for any choice of i .



Perceptron training rule (The “Delta-rule”)

First we look at **learning weights of single perceptrons**:

From data $\langle \vec{x}, t \rangle$ (t is “target”), we want to tune \vec{w} automatically.

Let us consider simpler **linear unit**, where input is \vec{x} gives output $o = \vec{w}^T \vec{x}$. The correct (observed) value is t .

Desiderata:

- Each weight w_i should be tuned separately.
- If $o = t$ there is no need to change any of the weights.
- If $o < t$ (too low):
 - If $x_i > 0$, we should **increase** w_i
 - If $x_i < 0$, we should **decrease** w_i
- If $o > t$ (too high):
 - If $x_i > 0$, we should **decrease** w_i
 - If $x_i < 0$, we should **increase** w_i
- The **more** we are off, the **larger** the change to the weights.



Perceptron training rule (The “Delta-rule”)

First we look at **learning weights of single perceptrons**:

From data $\langle \vec{x}, t \rangle$ (t is “target”), we want to tune \vec{w} automatically.

Let us consider simpler **linear unit**, where input is \vec{x} gives output $o = \vec{w}^T \vec{x}$. The correct (observed) value is t .

Learning rule:

$$w_i \leftarrow w_i + \Delta w_i, \text{ where } \Delta w_i = \eta(t - o)x_i$$

Where:

- t is target value and o is perceptron output (as before)
- η is small constant called **the learning rate**.

Perceptron training rule (The “Delta-rule”)



First we look at **learning weights of single perceptrons**:

From data $\langle \vec{x}, t \rangle$ (t is “target”), we want to tune \vec{w} automatically.

Let us consider simpler **linear unit**, where input is \vec{x} gives output $o = \vec{w}^T \vec{x}$. The correct (observed) value is t .

Learning rule:

$$w_i \leftarrow w_i + \Delta w_i, \text{ where } \Delta w_i = \eta(t - o)x_i$$

Where:

- t is target value and o is perceptron output (as before)
- η is small constant called **the learning rate**.

Convergence:

- **Converges** if training data is linearly separable and η sufficiently small.
- **Result unclear when NOT linearly separable.**



Finding the optimal weights

We still consider **linear unit**, where input is \vec{x} gives output

$$o = w_0x_0 + w_1x_1 + \cdots + w_nx_n = \vec{w}^\top \vec{x}$$

We learn w_i 's that minimise the squared error

$$\mathbb{E}[\vec{w}] \equiv \sum_{d \in \mathcal{D}} (t_d - o_d)^2,$$

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

Finding optimal weights – requirements



Description of our situation:

- We have a function $\mathbb{E}[\vec{w}]$ we want to minimise (wrt \vec{w}).
- Why not just try a number of weight configurations \vec{w}_i , calculate $\mathbb{E}[\vec{w}_i]$ and see what happens?

Finding optimal weights – requirements



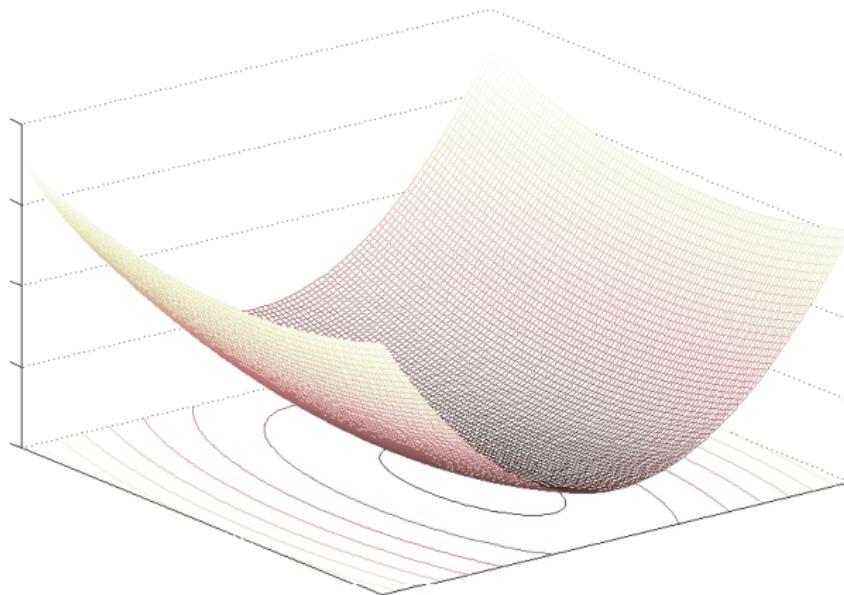
Description of our situation:

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

Thus, we cannot use a standard (“silly”) Trial & Error approach here, but must devise a local search method.



Solution procedure



Discuss with your neighbour:

How can we minimise this function without knowing the whole shape?
We want a **general**, **quick**, and **robust** technique.

Gradient Descent – The setup



Let's begin by considering a function of a single variable:

- We want to find the value x which minimizes $f(x)$.
- 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:
If $f'(x) > 0$ our guess is too high; if $f'(x) < 0$ we are too low,
and if $f'(x) \equiv 0$ we are just perfect.
 - ... and keep going.

Gradient Descent – The setup



Let's begin by considering a function of a single variable:

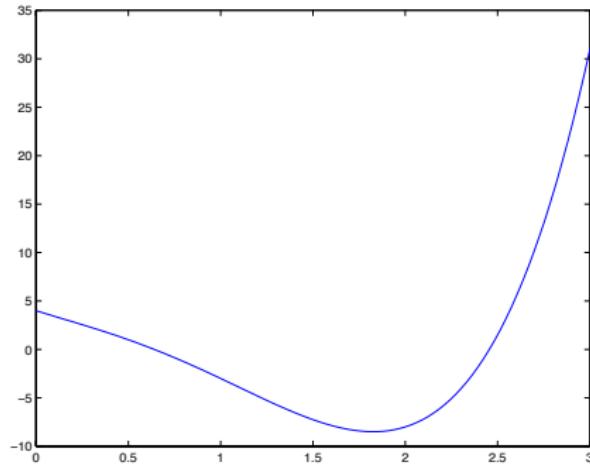
- We want to find the value x which minimizes $f(x)$.
- To avoid evaluating the whole function we use an iterative approach:
 - Guess a value for x
 - Calculate the derivative at x .
 - Make a new guess for x based on the calculated information:
If $f'(x) > 0$ our guess is too high; if $f'(x) < 0$ we are too low,
and if $f'(x) \equiv 0$ we are just perfect.
 - ... and keep going.
- Intuition:
 - If the derivative is small (in absolute value) we are close to the minimizing point
 - ... and if it is zero we are done
 - If it is large we are far away

Solution:

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

Gradient Descent – Example

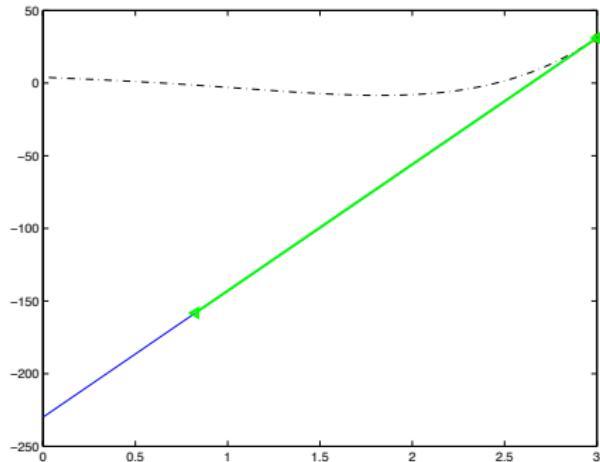
Minimize $f(x) = 2x^4 - 5x^3 + 2x^2 - 6x + 4$ with $\eta = 0.025$.
Note that $f'(x) = 8x^3 - 15x^2 + 4x - 6$



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 $\eta = 0.025$.
Note that $f'(x) = 8x^3 - 15x^2 + 4x - 6$

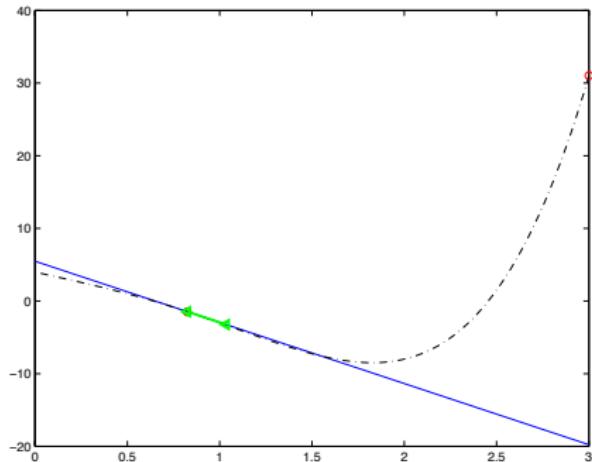


Starting from $x_0 = 3$ and finding $f'(3) = 87$:

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

Gradient Descent – Example

Minimize $f(x) = 2x^4 - 5x^3 + 2x^2 - 6x + 4$ with $\eta = 0.025$.
Note that $f'(x) = 8x^3 - 15x^2 + 4x - 6$

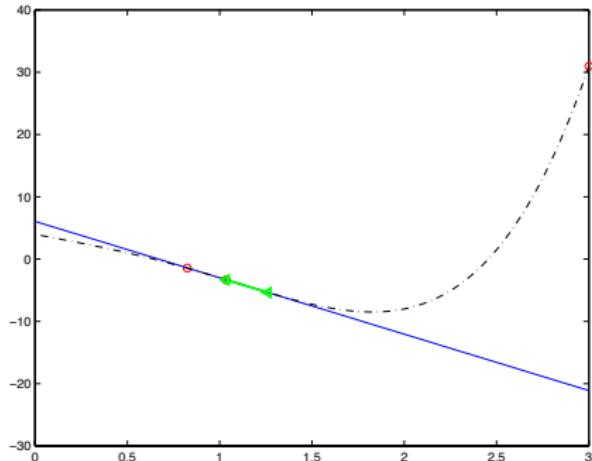


Going from $x_1 = 0.8250$ with $f'(0.8250) = -8.4172$:

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

Gradient Descent – Example

Minimize $f(x) = 2x^4 - 5x^3 + 2x^2 - 6x + 4$ with $\eta = 0.025$.
Note that $f'(x) = 8x^3 - 15x^2 + 4x - 6$

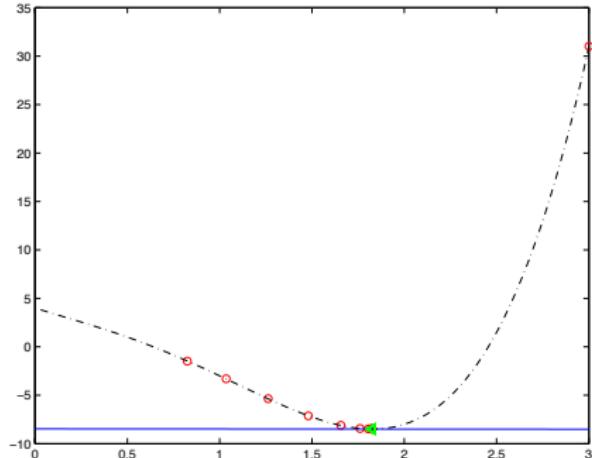


Going from $x_2 = 1.0354$ with $f'(1.0354) = -9.0592$:

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

Gradient Descent – Example

Minimize $f(x) = 2x^4 - 5x^3 + 2x^2 - 6x + 4$ with $\eta = 0.025$.
Note that $f'(x) = 8x^3 - 15x^2 + 4x - 6$

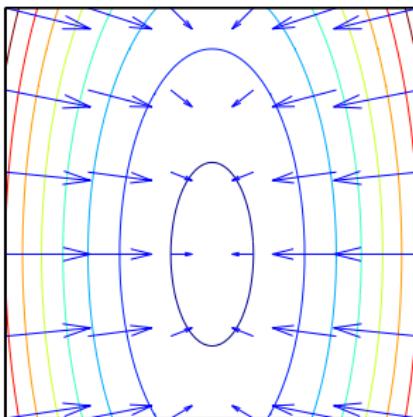


... and finally going from $x_{10} = 1.8260$ with $f'(1.8260) = -0.0034$:

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

... and we are done because $f'(1.8261) = 0.000$.

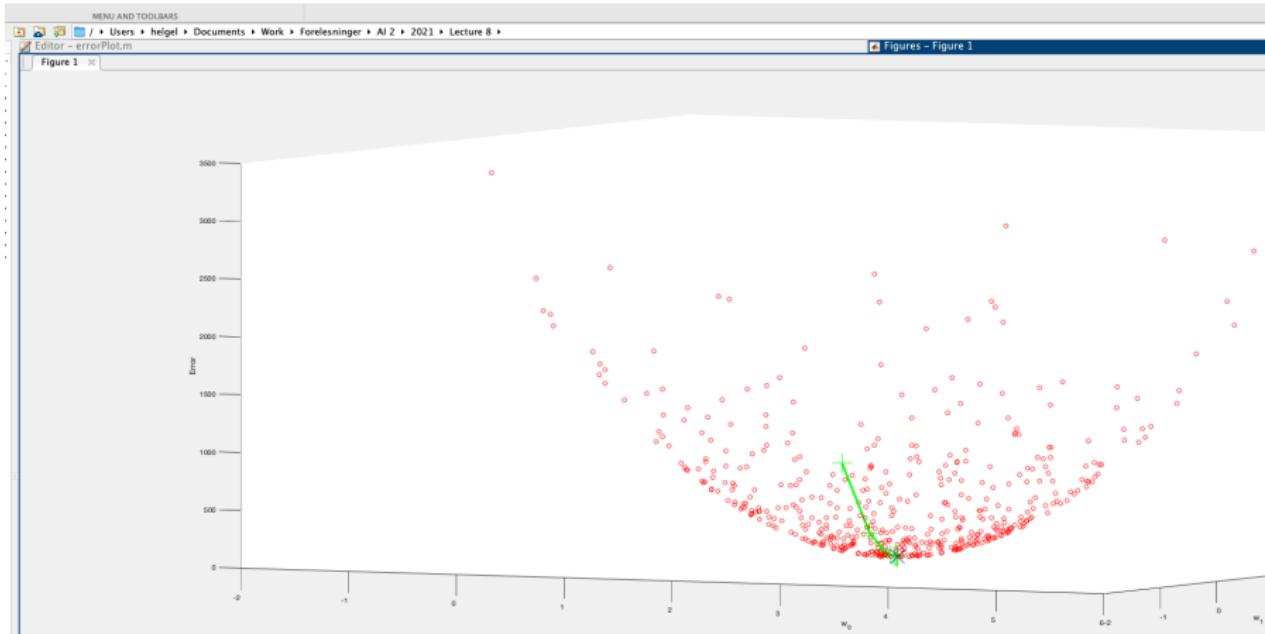
Gradient Descent – in higher dimensions



Recall that

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

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



Command Window

```
First, do random evaluations
*****
After 588 evaluations: w_0 = 0.96 and w_1 = 2.92 with error 0.1725
```

Now, do gradient descent

```
*****
Step 1: Error = 761.3351
Step 2: Error = 184.7905
Step 3: Error = 44.8960
Step 4: Error = 10.9185
Step 5: Error = 2.6579
Step 6: Error = 0.6477
Step 7: Error = 0.1580 --> Now we beat the result of random search
Step 8: Error = 0.0365
Step 9: Error = 0.0094
Step 10: Error = 0.0023
```

Gradient Descent for the perceptron



We have

- A database of inputs and targets, $\langle \vec{x}_d, t_d \rangle, d \in \mathcal{D}$.
 - Inputs padded with “-1-s”: $\vec{x}_d = \langle -1, x_{1,d}, x_{2,d}, \dots, x_{n,d} \rangle$.
- A set of trainable weights: $\vec{w} = \langle w_0, \dots, w_n \rangle$.
- A pre-defined transfer-function $g(\cdot)$, so that $o_d = g(\vec{w}^\top \vec{x})$.
 - We believe that there exist weights so that $o_d \approx t_d$.
- An error-measure $\mathbb{E}[\vec{w}]$ to quantify the how far off we are. For now we use

$$\mathbb{E}[\vec{w}] \equiv \sum_{d \in \mathcal{D}} (t_d - o_d)^2.$$

We have decided to use the GD update-rule:

$$\vec{w} \leftarrow \vec{w} - \eta \cdot \nabla \mathbb{E}[\vec{w}],$$

so we need to calculate $\nabla \mathbb{E}[\vec{w}] \equiv \left[\frac{\partial \mathbb{E}}{\partial w_0}, \frac{\partial \mathbb{E}}{\partial w_1}, \dots, \frac{\partial \mathbb{E}}{\partial w_n} \right]$.

Gradient Descent – IDENTITY activation function



$$\begin{aligned}
 \frac{\partial \mathbb{E}}{\partial w_i} &= \frac{\partial}{\partial w_i} \sum_d (t_d - o_d)^2 \\
 &= \sum_d \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\
 &= \sum_d 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\
 &= 2 \sum_d (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w}^\top \vec{x}_d) \quad g(\cdot) \text{ is identity} \\
 \frac{\partial \mathbb{E}}{\partial w_i} &= -2 \sum_d x_{i,d} \cdot (t_d - o_d)
 \end{aligned}$$

For the last identity, recall $\frac{\partial}{\partial w_i} \vec{w}^\top \vec{x}_d = \frac{\partial}{\partial w_i} \sum_{j=0}^n w_j \cdot x_{j,d} = x_{i,d}$.

Gradient Descent – The algorithm



Gradient-Descent(\mathcal{D} , η)

Each training example is a pair of the form $\langle \vec{x}, t \rangle$, where \vec{x} is the vector of input values, and t is the target output value. η is the learning rate (e.g., .05).

- ➊ Initialize each w_i to some small random value
- ➋ Until the termination condition is met:
 - ➌ Initialize: $\Delta w_i \leftarrow 0$.
 - ➍ For each $\langle \vec{x}, t \rangle$ in \mathcal{D} :
 - ➎ Input the instance \vec{x} to the unit and compute the output o
 - ➏ For each w_i : $\Delta w_i \leftarrow \Delta w_i - \eta \cdot 2 (-x_i (t - o))$
 - ➐ For each w_i : $w_i \leftarrow w_i + \Delta w_i$

Same algorithm skeleton works for other activation functions, as long as we can calculate $\frac{\partial E}{\partial w_i}$.



Gradient Descent with LOGISTIC activation functions

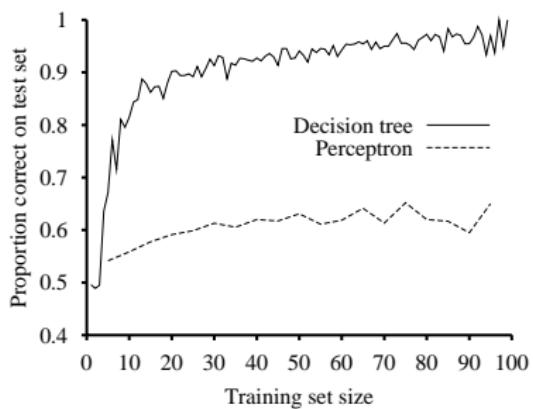
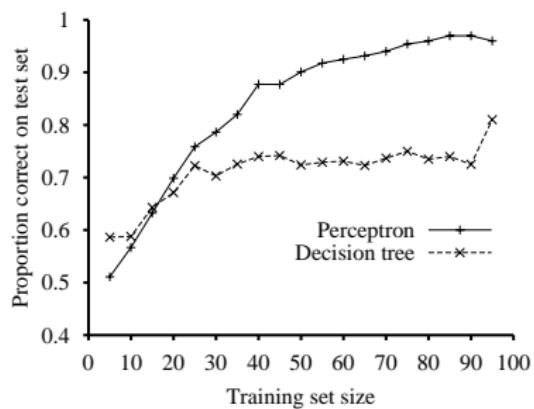
Assume the logistic activation function:

$$g(t) = \frac{1}{1 + \exp(-t)}, \quad g'(t) = g(t) \cdot [1 - g(t)].$$

$$\begin{aligned}\frac{\partial \mathbb{E}}{\partial w_i} &= \frac{\partial}{\partial w_i} \sum_d (t_d - o_d)^2 \\ &= 2 \sum_d (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - g(\vec{w}^\top \vec{x}_d)) \\ &= 2 \sum_d (t_d - o_d) \cdot (-1) \cdot x_{i,d} \cdot g'(\vec{w}^\top \vec{x}_d) \\ \frac{\partial \mathbb{E}}{\partial w_i} &= -2 \sum_d x_{i,d} \cdot (t_d - o_d) \cdot g(\vec{w}^\top \vec{x}_d) [1 - g(\vec{w}^\top \vec{x}_d)]\end{aligned}$$

Perceptron learning – Does it work?

Perceptron learning rule converges to a consistent function **for any linearly separable data set**



majority_11: **Perceptron** OK, **DTL** hopeless

restaurant: **DTL** OK, **perceptron** cannot represent it

Summary - Perceptron



Perceptron delta-rule guaranteed to succeed if

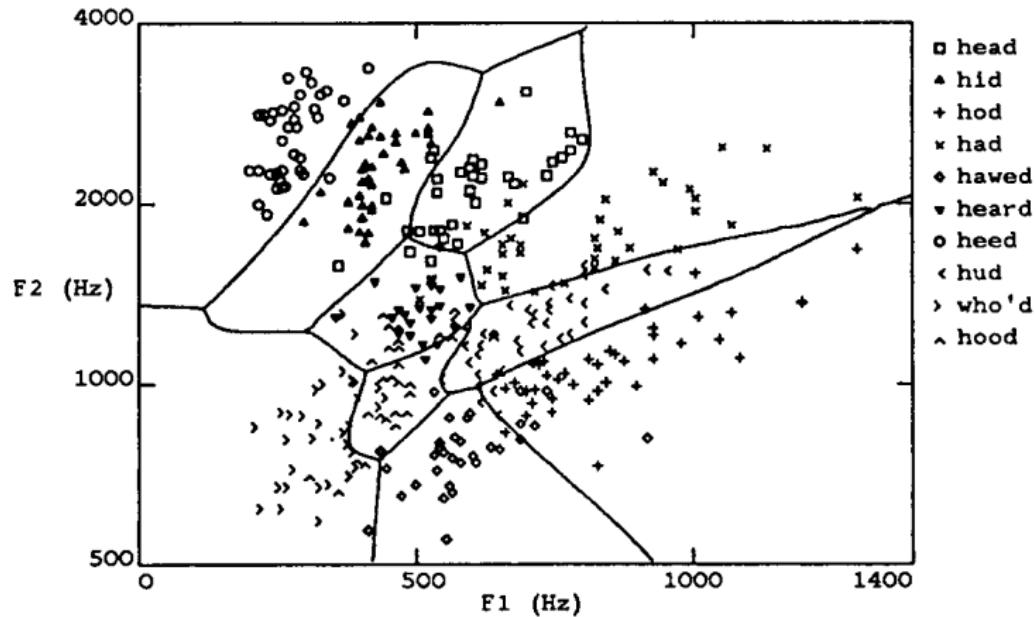
- Training examples are linearly separable
- Sufficiently small learning rate η

Linear unit training rule uses gradient descent

- Guaranteed to converge to w with minimum squared error
- Given sufficiently small learning rate η
- This is true even when training data contains **noise**, or the data is **not linearly separable**.
- Gradient descent also works for other activation functions.

Perceptrons are unable to capture intricate patterns

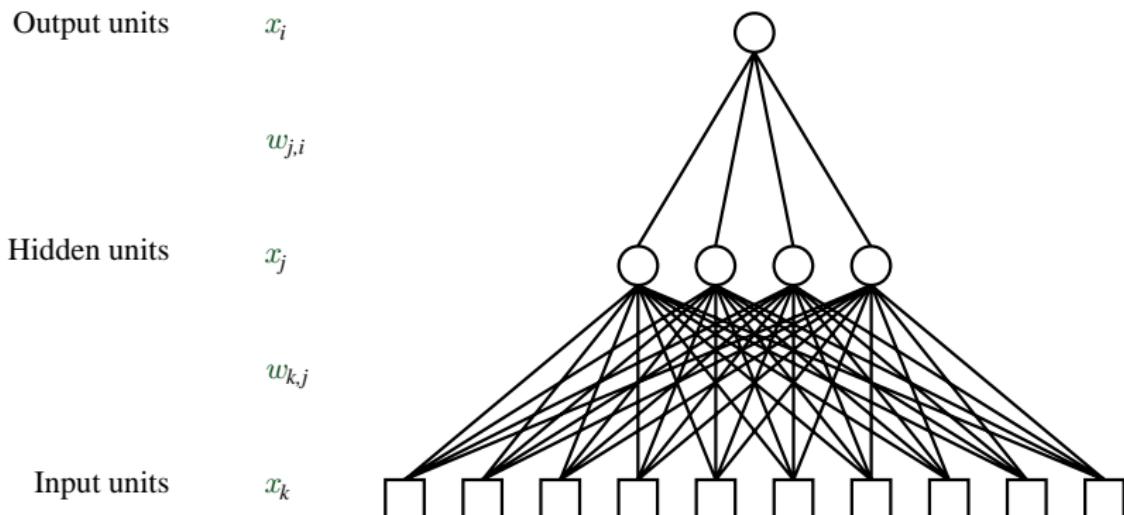
Multilayer Networks of Sigmoid Units



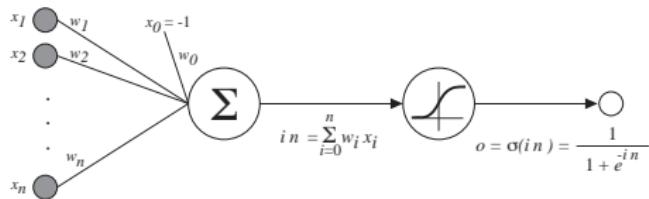
Multilayer perceptrons



Layers are usually fully connected; numbers of **hidden units** typically chosen “by hand” – doing a hyper-parameter search.



Sigmoid Unit Networks



Remember the sigmoid activation, often called $\sigma(\cdot)$, $\sigma(y) = \frac{1}{1+e^{-y}}$.

Remember also that it has the nice property that

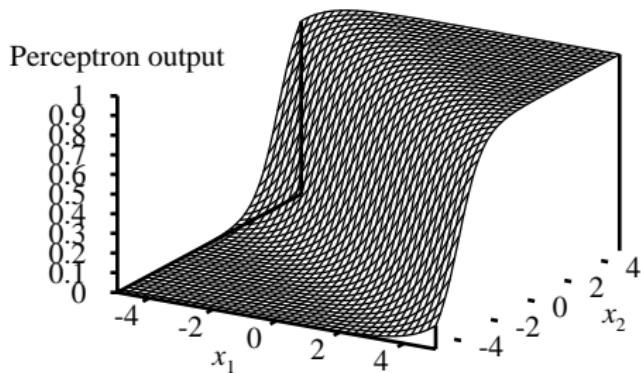
$$\frac{d\sigma(y)}{dy} = \sigma(y)(1 - \sigma(y)).$$

We can derive gradient decent rules to train

- One sigmoid unit → We (should) know how to do that.
- Multilayer networks of sigmoid units → Backpropagation

Sigmoids - expressibility

No hidden layer, input (x_1, x_2) :

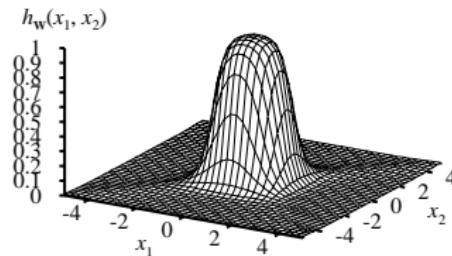
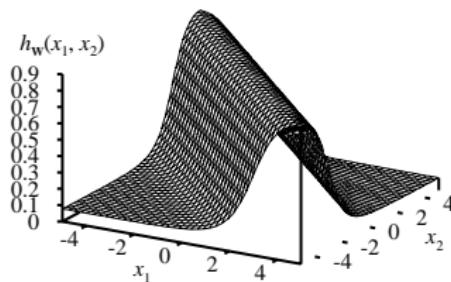


- Adjusting weights moves the location, orientation, and steepness of cliff
- Cannot tackle “correlation effects” of non-separable targets

Sigmoids - expressibility



Hidden layers:



- Combine two opposite-facing threshold functions to make a ridge
- Combine two perpendicular ridges to make a bump
- Add bumps of various sizes and locations to fit any surface

All continuous functions w/ 1 hidden layer, all functions w/ 2



Expressive Capabilities of ANNs

Boolean functions:

- Every boolean function can be represented by network with single hidden layer
- but 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



Recall: Error Gradient for a Sigmoid Unit

Remember the perceptron learning, where we did the partial derivative of error wrt. weights:

$$\frac{\partial \mathbb{E}(\vec{w})}{\partial w_i} = -2 \sum_{d \in \mathcal{D}} (t_d - o_d) \frac{\partial o_d}{\partial \text{in}_d} \frac{\partial \text{in}_d}{\partial w_i}$$

Now, activation is sigmoid, so $\frac{\partial o_d}{\partial \text{in}_d} = o_d(1 - o_d)$ and $\frac{\partial \text{in}_d}{\partial w_i} = x_{i,d}$. Therefore, $\frac{\partial \mathbb{E}}{\partial w_i}$ can easily be calculated in closed form.

Idea of Backprop:

Do Gradient Descent for the full model:

$$w_i \leftarrow w_i - \eta \cdot \frac{\partial \mathbb{E}(\vec{w})}{\partial w_i}$$

Weight updates are (typically, but depending on $g(\cdot)$) easily available due to the restrictive structure of the feed forward model.



Backpropagation – General idea

- Denote by $w_{i,j}$ the weight on the connection between node i and node j . Then:

$$w_{i,j} \leftarrow w_{i,j} - \eta \cdot \frac{\partial \mathbb{E}(\vec{w})}{\partial w_{i,j}}.$$

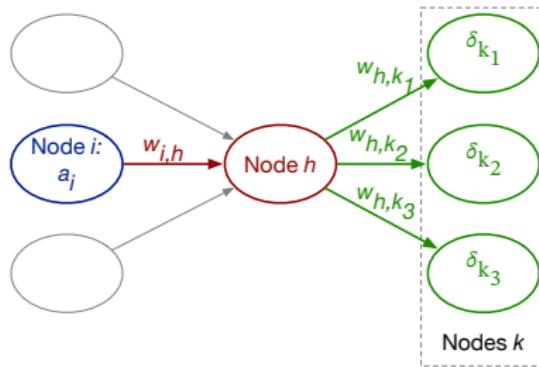
- If we are willing to calculate $\frac{\partial \mathbb{E}(\vec{w})}{\partial w_{i,j}}$ for all weights $w_{i,j}$ by hand, we have an algorithm that always works!
- The layered architecture helps simplify the algorithm:
 - Do calculations layer by layer.
 - Update for the weight $w_{i,j}$ will be determined by the activation of node i , and the “loss contribution” from node j .

Backpropagation Algorithm – Sigmoid units



- ➊ Initialize all weights to small random numbers.
 - ➋ Until satisfied:
 - ➌ For each training example
 - ➍ Input the training example to the network and compute the network outputs and node activations along the way
 - ➎ For each output unit k : $\delta_k \leftarrow \underbrace{o_k(1 - o_k)}_{g'(\text{in}_k)} \times \underbrace{(t_k - o_k)}_{\text{Output error}}$
 - ➏ For each hidden unit h :
- $$\delta_h \leftarrow \underbrace{o_h(1 - o_h)}_{g'(\text{in}_h)} \times \underbrace{\sum_{k \in \text{outputs}} w_{h,k} \cdot \delta_k}_{\text{Unit } h \text{'s contribution to next layer's error}}$$
- ➐ Update each network weight $w_{i,j}$: $w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$, where $\Delta w_{i,j} = \eta \cdot 2\delta_j \cdot a_i$ and a_i is the activation of the node pushing information into $w_{i,j}$.

Backprop details



$$\delta_h \leftarrow o_h(1 - o_h) \times \sum_k w_{h,k} \delta_k \quad \text{where} \quad \Delta w_{i,h} \leftarrow \eta \cdot 2 \delta_h \cdot a_i$$

Note the “locality” of the calculations:

- $\Delta w_{i,h}$ only cares about Node h , its parent (Node i), and its children (Nodes k).
- The calculation of δ_h does not care about the parent (Node i), so we can use the same value for all parents!



More on Backpropagation

- Gradient descent over entire *network* weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
 - In practice, often works well
- Minimizes error over *training* examples
 - Will it generalize well to subsequent examples? **Overfitting...**
- Training can take thousands of iterations → **slow!**
- Using network after training is very fast

Convergence of Backpropagation



Gradient descent to some local minimum:

- Perhaps not the *global* minimum ...
 - Include weight *momentum* $\alpha > 0$:

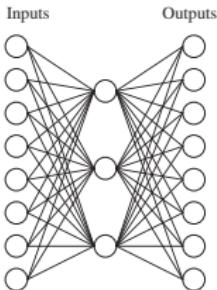
$$\Delta w_{i,j}(n) = \eta \cdot \left(\underbrace{2\delta_j \cdot a_i}_{\text{The "standard"}} + \underbrace{\alpha \cdot \Delta w_{i,j}(n-1)}_{\text{Scaled last move}} \right)$$

- Train multiple nets with different initial weights
- Batching/stochastic gradient descent

Nature of convergence – depending on g (here: logistic):

- Initialize weights near zero
→ Therefore, initial networks *near-linear*.
- Increasingly non-linear functions possible as training progresses.

Understanding Hidden Layer Representations – “XAI Light”



A target function:

Input		Output
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001

Can this “auto-encoder” be learned with sigmoid activations??



Learning Hidden Layer Representations

Learned hidden layer representation:

Input	Hidden Values			Output	
10000000	→	.89	.04	.08	→ 10000000
01000000	→	.15	.99	.99	→ 01000000
00100000	→	.01	.97	.27	→ 00100000
00010000	→	.99	.97	.71	→ 00010000
00001000	→	.03	.05	.02	→ 00001000
00000100	→	.01	.11	.88	→ 00000100
00000010	→	.80	.01	.98	→ 00000010
00000001	→	.60	.94	.01	→ 00000001

Do you recognize the encoding?



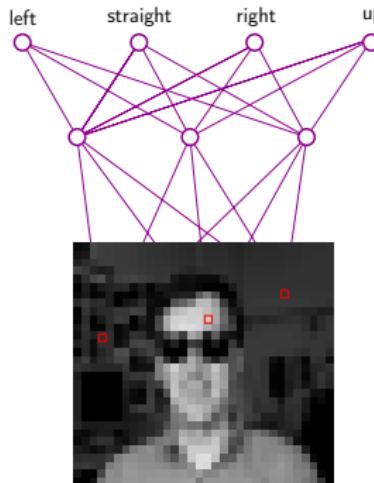
Learning Hidden Layer Representations

Learned hidden layer representation:

Input	Hidden Values			Output	
	.89	.04	.08	→	10000000
10000000	→ .15	.99	.99	→	01000000
01000000	→ .01	.97	.27	→	00100000
00100000	→ .99	.97	.71	→	00010000
00010000	→ .03	.05	.02	→	00001000
00001000	→ .01	.11	.88	→	00000100
00000100	→ .80	.01	.98	→	00000010
00000010	→ .60	.94	.01	→	00000001

Do you recognize the encoding?

Neural Nets for Recognition of Viewing Direction

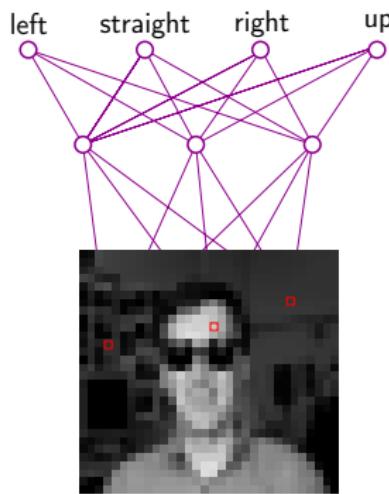


Typical input images (30×32 pixels)

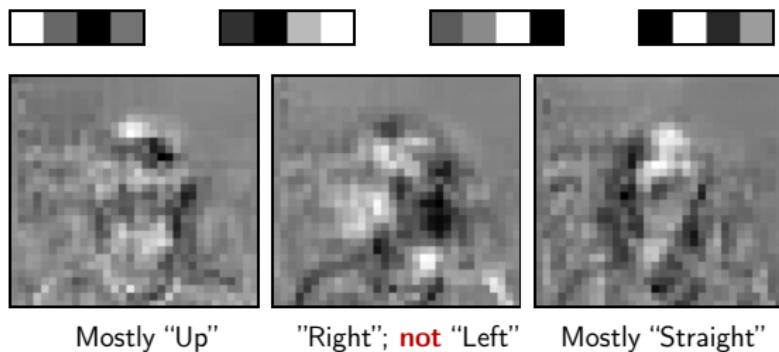
90% accurate learning head pose, and recognizing 1-of-20 faces



Learned hidden unit weights



Learned Weights

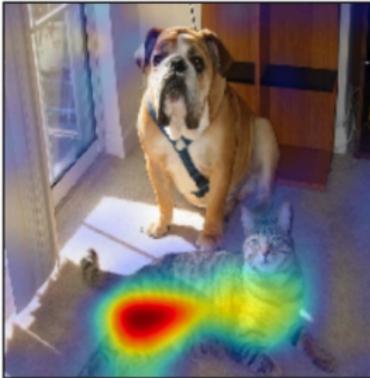
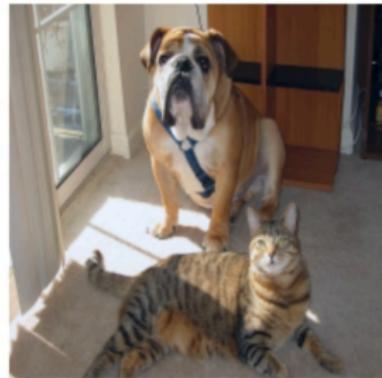


Typical input images



More descriptive models

More flexible models (typically deep learning, next time) can give even more impressive explanations:



Examples using GradCAM over CNN models; asked to explain animal classification.

Overfitting in ANNs – Why, and what to do...



The problem:

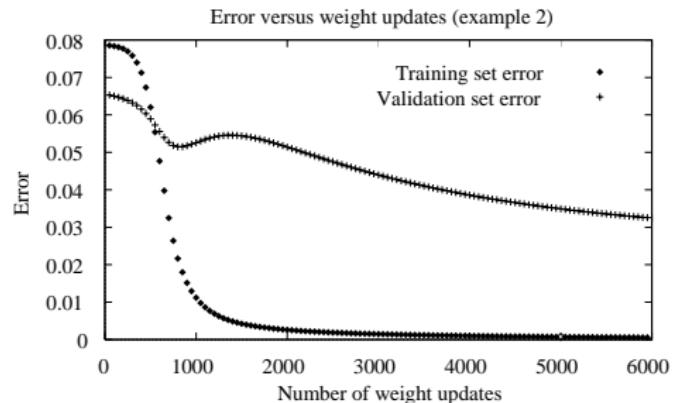
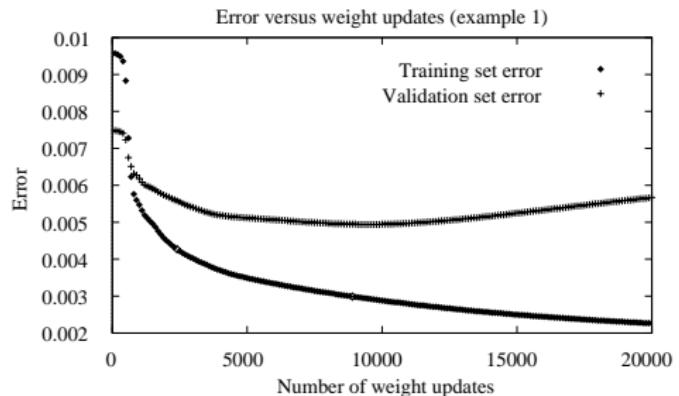
- Due to ANNs **universal approximation theorem**, overfitting is prominent in ANNs.
- Even a “relatively small” network has, if weights grow large in absolute value, the ability to construct **consistent hypothesis by memorizing** the training data.

Some popular solutions:

Regularization: Define alternative error/loss functions, where we punish model complexity and/or extreme weights.

Validation: Monitor convergence/overfitting on a separate validation-set.

Overfitting in ANNs – validation loss



Summary



- Most brains have lots of neurons; each neuron \approx linear-threshold unit (?)
- Perceptrons (one-layer networks) insufficiently expressive
- Multi-layer networks are sufficiently expressive; can be trained by gradient descent, i.e., error back-propagation
- Many applications: speech, driving, handwriting, fraud detection, etc.
- Engineering, cognitive modelling, and neural system modelling subfields have largely diverged