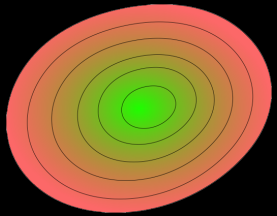


UNIVERSITY OF TWENTE.

Deep Learning for 3D Medical Image Analysis

Lecture 2: Perceptrons, filters, and CNNs

February 11, 2025

Jelmer Wolterink

Mathematics of Imaging & AI group

Last week

Lecture 1

- Course organization
- Project
  - Don’t forget to make teams
  - Mentors on Canvas
  - Plan of approach

Tutorial 1

- Getting started with Python
- Working with images
- Try out MevisLab

This week

‘Crash course’ into

- Neural networks
- Image filters
- Convolutional neural networks
- Tutorial (**Thursday**): Training your first neural network

This week

‘Crash course’ into

- Neural networks
- Image filters
- Convolutional neural networks
- Tutorial (**Thursday**): Training your first neural network

This week

‘Crash course’ into

- Neural networks
- Image filters
- Convolutional neural networks
- Tutorial (**Thursday**): Training your first neural network

This week

‘Crash course’ into

- Neural networks
- Image filters
- Convolutional neural networks
- Tutorial (**Thursday**): Training your first neural network

Disambiguation

Deep learning $\subset$ Machine learning $\subset$ Artificial intelligence

- *Artificial intelligence*: Programs with the ability to learn and reason like humans
- *Machine learning*: Algorithms with the ability to learn without being explicitly programmed
- *Deep learning*: Machine learning with artificial neural networks

Machine learning

Mitchell, 1997

A 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 T, as measured by P, improves with experience E.

- Tasks T are, for example, classification, denoising, segmentation, registration, …
- The performance measure P is task-dependent
- Experience E is what is provided during learning

Machine learning

Mitchell, 1997

A 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 T, as measured by P, improves with experience E.

- Tasks T are, for example, classification, denoising, segmentation, registration, …
- The performance measure P is task-dependent
- Experience E is what is provided during learning

Machine learning

Mitchell, 1997

A 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 T, as measured by P, improves with experience E.

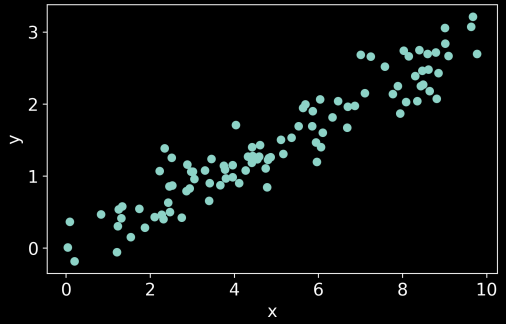
- Tasks T are, for example, classification, denoising, segmentation, registration, …
- The performance measure P is task-dependent
- Experience E is what is provided during learning

Example problem: linear regression

- Input is a vector $\mathbf{x} \in \mathbb{R}^n$, output is a scalar $y \in \mathbb{R}$
- Task T: predict $\hat{y} = \mathbf{w}\mathbf{x} + b$
- Performance metric P: MSE $= \frac{1}{n} \sum_i^n (\widehat{\mathbf{y}}_\mathbf{i} - \mathbf{y}_i)^2$
- Goal: Change the weights $\mathbf{w}$ s.t. the metric P is optimized

Example: linear regression

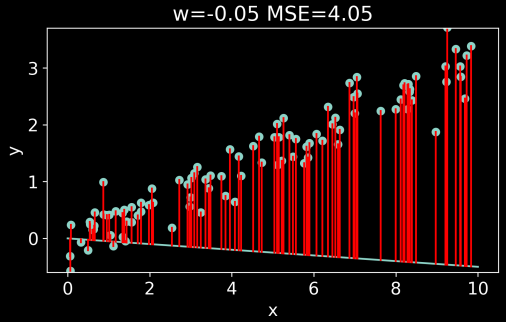
With one independent variable x, one dependent variable y, and one weight w we get



And we want to find a minimizer $\hat{w} = \arg\min_w \frac{1}{n} \sum_i^n (w\mathbf{x}_i - \mathbf{y}_i)^2$

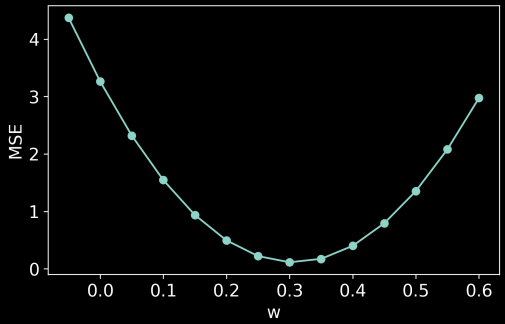
Example: linear regression

We can compute for a range of w values what the error would be



Example: linear regression

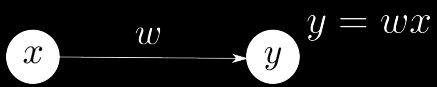
If we plot the error as a function of w, it's clear what the optimal value $\hat{w}$ is



In this case, we could have found $\hat{w}$ analytically using the normal equation

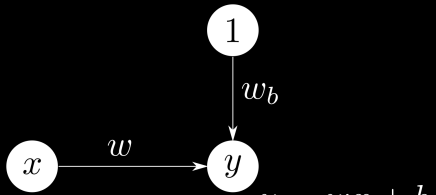
Neural network

We could also represent this function $y = wx$ in a *neural network*



Neural network

And to make it a *real* linear regression problem, add a bias b term, $y = wx + b$



Concatenate the parameters in a parameter vector $\theta = [w, b]$

$$\hat{\theta} = \arg\min_{\theta} \frac{1}{n} \sum_{i}^{n} ((b + w\mathbf{x}_i) - \mathbf{y}_i)^2$$

Gradient-based optimization

There are (at least) three ways to find the optimal parameters for this optimization problem

- Brute force $\rightarrow$ inefficient
- Use the normal equations $\rightarrow$ works for convex problems
- Gradient descent $\rightarrow$ puts the 'learning' in machine learning

Gradient-based optimization

There are (at least) three ways to find the optimal parameters for this optimization problem

- Brute force $\rightarrow$ inefficient
- Use the normal equations $\rightarrow$ works for convex problems
- Gradient descent $\rightarrow$ puts the 'learning' in machine learning

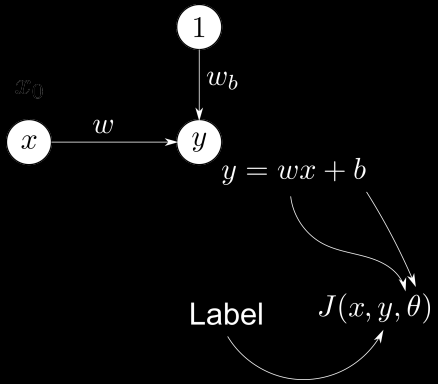
Gradient-based optimization

There are (at least) three ways to find the optimal parameters for this optimization problem

- Brute force $\rightarrow$ inefficient
- Use the normal equations $\rightarrow$ works for convex problems
- Gradient descent $\rightarrow$ puts the ‘learning‘ in machine learning

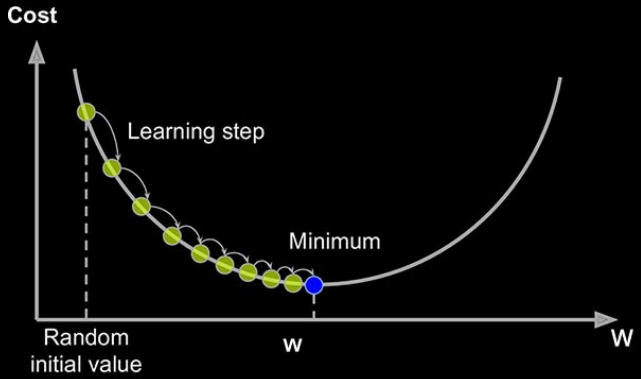
Problem definition

Find the parameter values for θ that minimize a cost function $J(\mathbf{x}, \mathbf{y}, \theta)$



Gradient descent

- **Goal**: find weights θ so that a loss or cost is minimized
- Start at some *initial* random values for the weights
- In each update, take a small step
- **Challenge:** How to find the direction and magnitude of the learning step?

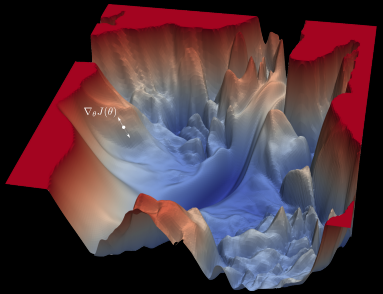


Gradient descent

Gradient descent

We iteratively update the weights θ in the inverse direction of the gradient of J w.r.t. θ

$$\theta_{n+1} = \theta_n - \alpha \nabla_\theta J(\theta)$$



Recap: gradient

- Let $f : \mathbb{R}^n \mapsto \mathbb{R}$ be a function that takes $\mathbb{R}^n$ vector $\boldsymbol{x}$ as input and returns a number (scalar) y.
- We obtain the gradient

$$\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_m} \end{bmatrix}$$

Back to our example

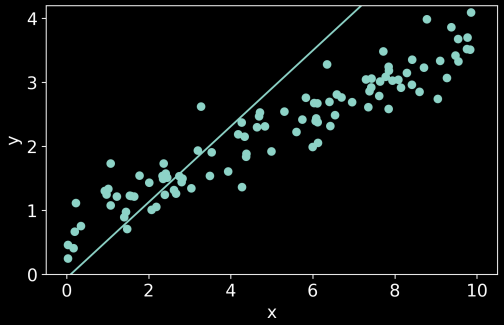
- Given an input scalar x, we have our prediction y
- Our objective (for many inputs) $J(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \left((b + w\mathbf{x}_i) - \mathbf{y}_i \right)^2$
- We want to compute the gradient of the output w.r.t. the parameters θ

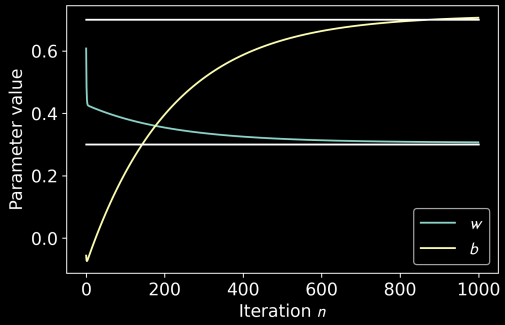
$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial J}{\partial w} \\ \frac{\partial J}{\partial b} \end{bmatrix}$$

- The partial derivative for the weight $\frac{\partial J}{\partial w} = \frac{2}{m} \sum_{i=1}^{m} (b + wx_i - y_i) x_i$
- The partial derivative for the bias $\frac{\partial J}{\partial b} = \frac{2}{m} \sum_{i=1}^{m} (b + wx_i - y_i)$

Our example

- Generate some data with $w = 0.3$ and $b = 0.7$
- Initialize estimates for w and b
- Optimize for 1000 iterations



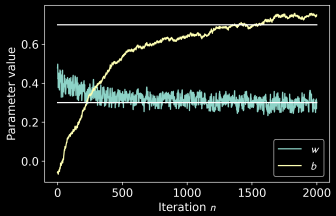


Gradient descent

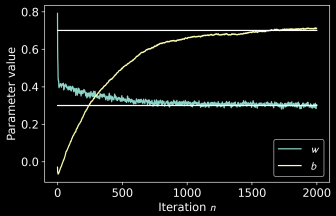
Given a data set with n samples, there are three common variants

- *Batch* gradient descent: feed the whole data set each time
- *Stochastic* gradient descent: feed one sample at a time
- *Mini-batch* gradient descent: feed a subset of the data each time
- Walking down a hill vs. stumbling down a hill

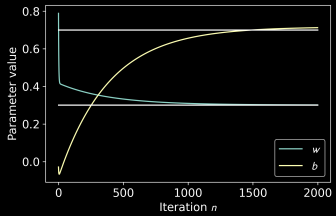
Stochastic



Mini-batch



Batch

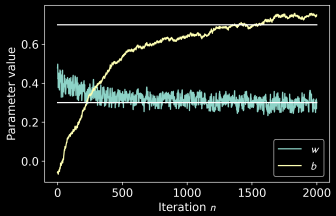


Gradient descent

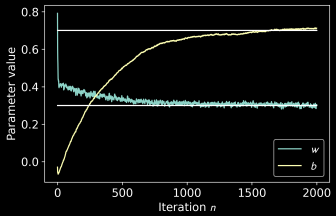
Given a data set with n samples, there are three common variants

- *Batch* gradient descent: feed the whole data set each time
- *Stochastic* gradient descent: feed one sample at a time
- *Mini-batch* gradient descent: feed a subset of the data each time
- Walking down a hill vs. stumbling down a hill

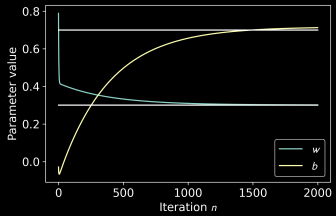
Stochastic



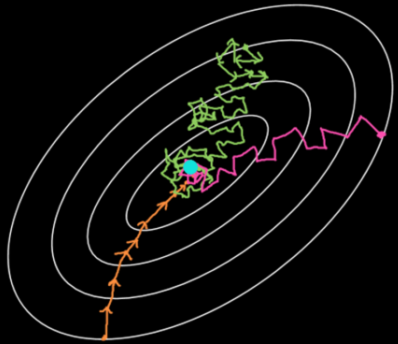
Mini-batch



Batch



Gradient descent



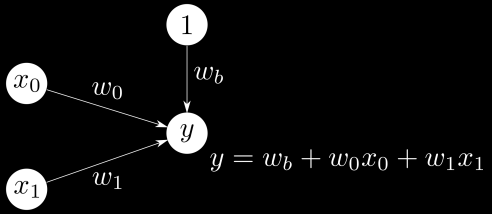
Batch gradient descent
Mini-batch gradient Descent
Stochastic gradient descent

Epochs and iterations

- **Iteration** One forward, one backward, one update
- **Epoch** One pass through all data samples
- **Updates** Number of backward passes through the network

Two input nodes

If the input dimensionality increases, we connect multiple input nodes to one output node

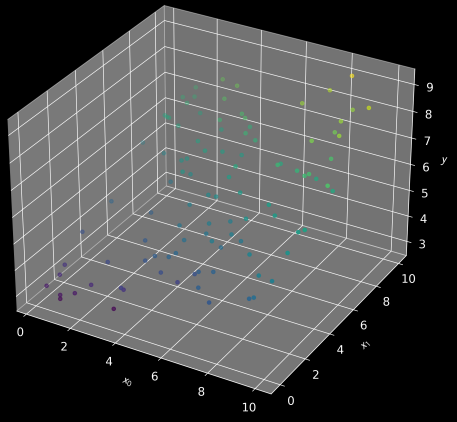


- So now our parameter vector grows to $\theta = [w_0, w_1, w_b]$.
- In PyTorch:

```
perceptron = torch.nn.Linear(in_features=2, out_features=1, bias=True)
```

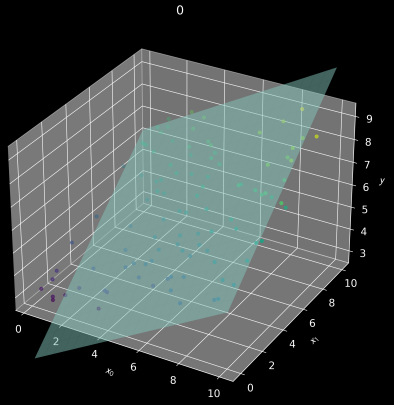
Regression

- What would this look like in data space?
- We want to find a 2D plane through a 3D pointcloud



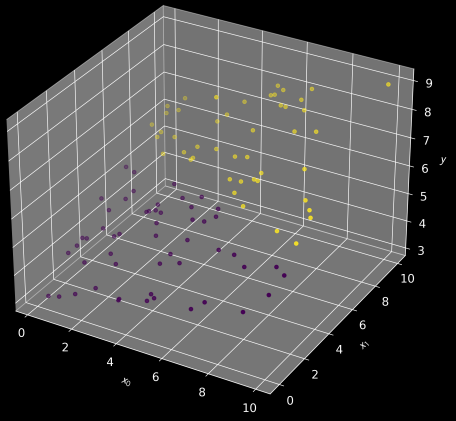
Back to data

- What would this look like in data space?
- We want to find a 2D plane through a 3D pointcloud
- Just like before, we can optimize or solve and find weights $w_0 = 0.2, w_1 = 0.4, b = 3$



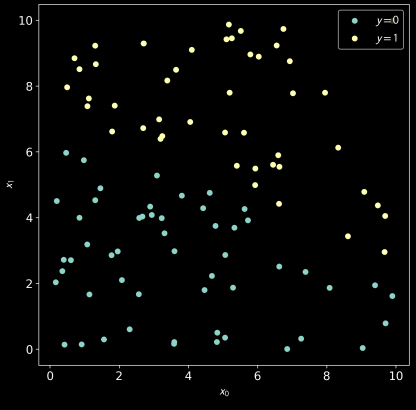
From regression to classification

- Now, y is no longer a scalar, but a binary class label, i.e. $y \in \{0, 1\}$
- For example, we might *threshold* y at some value to assign classes



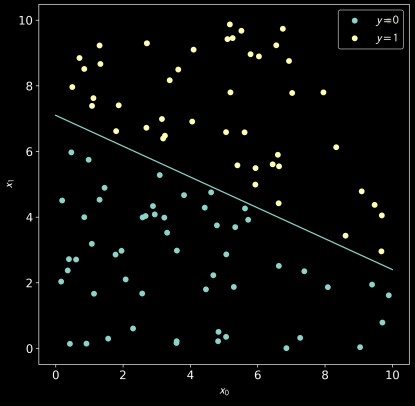
From regression to classification

- Now, y is no longer a scalar, but a binary class label, i.e. $y \in \{0, 1\}$
- This we can also visualize in 2D



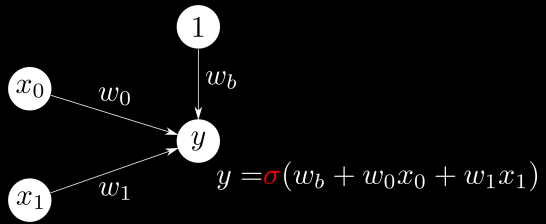
From regression to classification

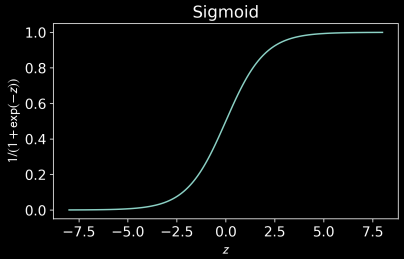
- Now, y is no longer a scalar, but a binary class label, i.e. $y \in Y = \{0, 1\}$
- We aim to find a separating line between class $y = 0$ and $y = 1$



From regression to classification

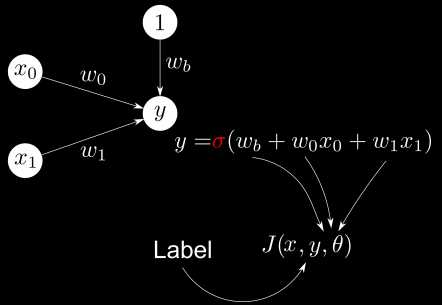
- The output is no longer just a linear combination of inputs + an intercept/bias
- We wrap a non-linear **activation function** around the output
- A useful output activation function is the logistic sigmoid $\sigma(z) = \frac{1}{1+\exp(-z)}$





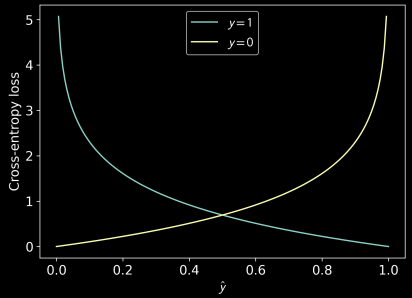
Problem definition

- The output of our network is $\sigma(z) = \frac{1}{1+\exp(-z)}$
- **Goal**: Find the parameter values for θ that minimize a cost function J

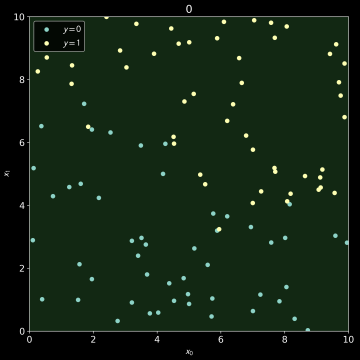


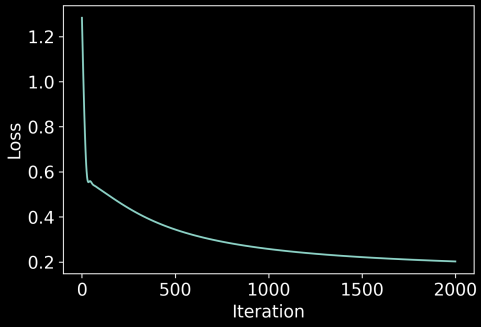
Optimization

- Can we still optimize for θ? Yes, we can!
- A common loss function for classification is the binary cross-entropy
  $J(y, \hat{y}) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$
- Simply compute $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ like before and optimize using gradient descent



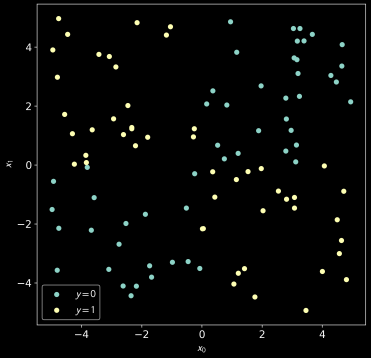
Training a classifier





XOR

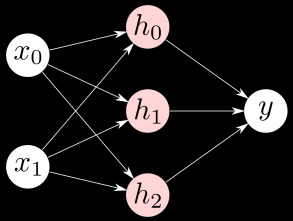
- The single-layer perceptron works well for problems that are linearly separable
- The weights define a separating hyperplane
- But a single-layer perceptron will never work on, e.g., the data below



XOR

- The single-layer perceptron works well for problems that are linearly separable
- The weights define a separating hyperplane
- Most *real-world* problems are not linearly separable
- If we have non-linearly separable problem, we need to find a mapping ϕ that turns our problem into a linearly separable one. Options are
  - We define some very generic kernel $\phi(\mathbf{x})$ such as the radial basis function.
  - We manually engineer ϕ.
  - We learn ϕ! E.g., $y = f(\mathbf{x}; \theta; \mathbf{w}) = \phi(\mathbf{x}; \theta)$
- In practice, this means adding more layers $\rightarrow$ **multi-layer perceptron (MLP)**

Multi-layer perceptron

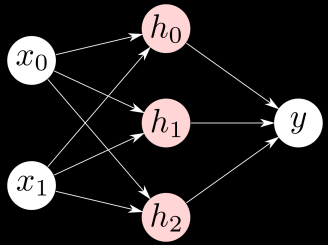


$$y = \mathbf{W}\mathbf{h} + w_b$$

- Now we have added three nodes in a *hidden* layer
- These are *fully connected* to the input and output layer
- ⚠ We assume that each node has a bias weight, but we don't draw it

Discussion point

How many trainable weights does this small network have, including biases?



Parameter space

- It's important to realize that our parameter space is high-dimensional
- We cannot visualize this, but we're looking for an optimal point in m-dimensional space

Optimizing a multilayer network

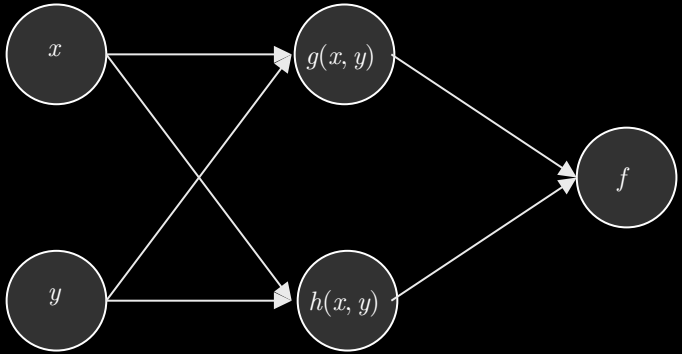
- The neural network $f(\mathbf{x})$ is a sequence of layers: $f(\mathbf{x}) = f_d(f_3(f_2(f_1(\mathbf{x}))))$
- We can use gradient descent as before $\theta_{n+1} = \theta_n - \alpha \nabla_\theta J(\theta)$
- The *only* challenge is how to compute $\nabla_\theta J(\theta)$
- It's infeasible to compute all partial derivatives by hand
- We use a process called backpropagation of errors or *backprop*

Chain rule

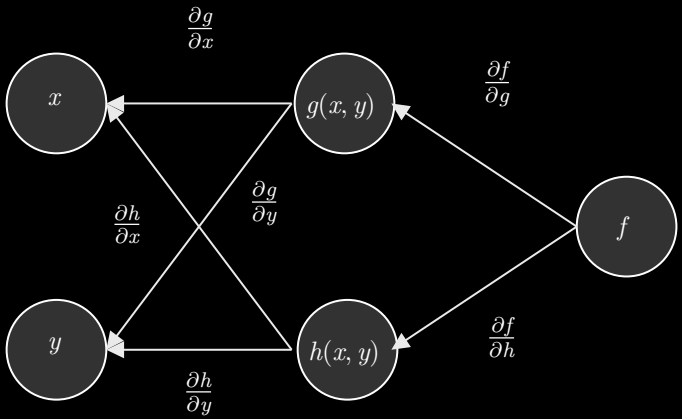
Remember the chain rule from calculus

- Suppose that $y = g(x)$ and $z = f(g(x)) = f(y)$
- Then $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$

Example



Example



$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x} + \frac{\partial f}{\partial h}\frac{\partial h}{\partial x}$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial y} + \frac{\partial f}{\partial h}\frac{\partial h}{\partial y}$$

Jacobian

- For more than one dimension
  - $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{y} \in \mathbf{R}^n$, g maps from $\mathbb{R}^m$ to $\mathbb{R}^n$ and f from $\mathbb{R}^n$ to $\mathbb{R}$
  - Then $\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$
  - Or $\nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}} \right)^{\mathsf{T}} \nabla_{\mathbf{y}} z$, where $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ is the *Jacobian*

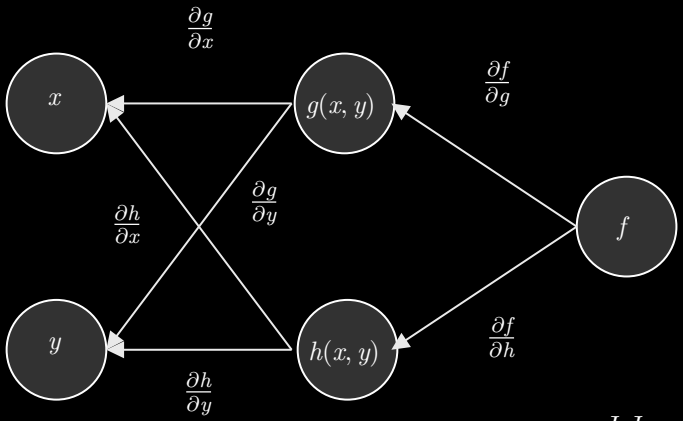
Jacobian

- The Jacobian $\boldsymbol{J_f}$ is a generalization of the gradient for vector valued functions.
- Let $\boldsymbol{f}: \mathbb{R}^n \mapsto \mathbb{R}^m$ be a function that takes n-dimensional vector $\boldsymbol{x}$ as input and returns a m-dimensional vector as an output.
- The Jacobian $\boldsymbol{J_f}$ is defined as

$$\boldsymbol{J_f} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

- For the special case of a scalar-valued function, the Jacobian is the transpose of the gradient.

Example



$$J_f = \begin{bmatrix} \frac{\partial f}{\partial g} & \frac{\partial f}{\partial h} \end{bmatrix}$$

$$J_{\mathbf{h}} = \begin{bmatrix} \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} \\ \frac{\partial h}{\partial x} & \frac{\partial h}{\partial y} \end{bmatrix}$$

$$J_f J_{\mathbf{h}} = \begin{bmatrix} \frac{\partial f}{\partial g}\frac{\partial g}{\partial x} + \frac{\partial f}{\partial h}\frac{\partial h}{\partial x} & \frac{\partial f}{\partial g}\frac{\partial g}{\partial y} + \frac{\partial f}{\partial h}\frac{\partial h}{\partial y} \end{bmatrix}$$

Backprop algorithm

- Start at the output z, set $\frac{dz}{dz} = 1$
- Compute the Jacobian of the operation between each parent of z and z
- Multiply $\frac{dz}{dz}$ with the Jacobian
- Proceed until you reach the input $\mathbf{x}$
- If a node has two children, sum the gradients

More info: *Dive into deep learning* Sec. 5.3, this video or the PyTorch website

Differentiability

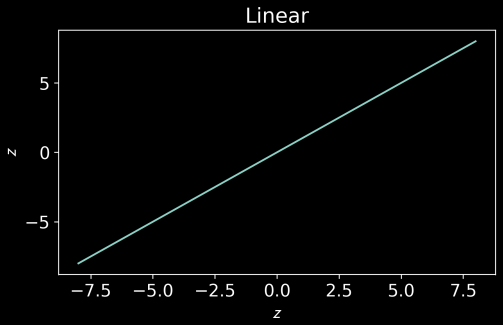
As long as we can compute the derivative of a function, it can be a part of the optimization process! $\longrightarrow$ Errors can be backpropagated!

Vanishing and exploding gradients

- Backpropagation uses the Jacobian matrices $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ in multiplication
- It's problematic if values are very low or high in these matrices
- **Low values** $\rightarrow$ vanishing gradients
- **High values** $\rightarrow$ exploding gradients
- How can we prevent this?

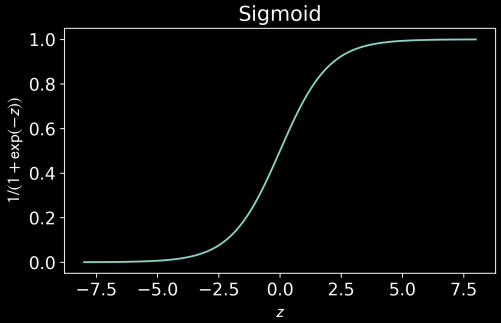
Non-linear activation functions

- Each (hidden) node has some activation following $\sigma(\mathbf{W}\mathbf{x} + b)$.
- The choice of σ results in different behaviors



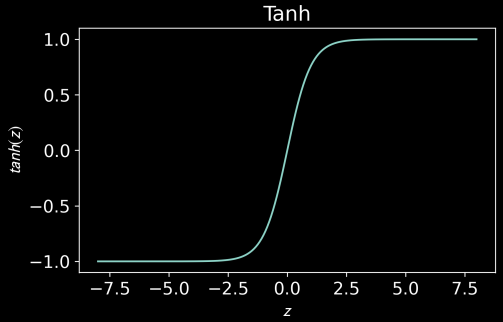
(Logistic) sigmoid

- $\sigma(z) = \frac{1}{1+\exp\{-z\}}$
- The *classical* activation function, squashes outputs between 0 and 1
- Also useful for outputs



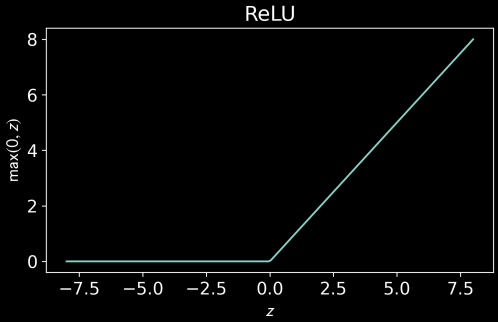
Tanh

- $\sigma(z) = \tanh z$
- Has stronger gradients around $z = 0$



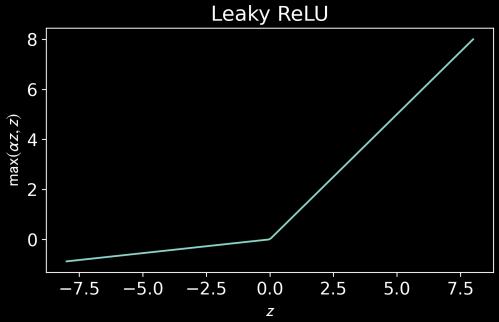
ReLU

- Rectified linear unit, $\sigma(z) = \max(0, z)$
- Faster, sparser networks (more nodes ≤ 0), better gradients if $z > 0$

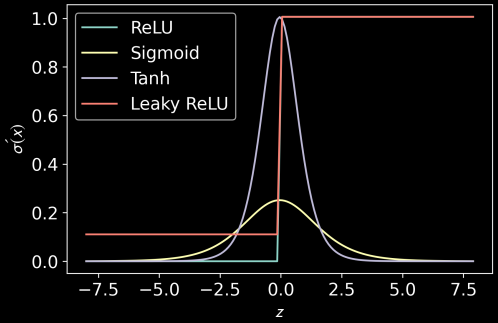


Leaky ReLU / PReLU

- Variant on ReLU, $\sigma(z) = \max(\alpha z, z)$
- Allows some gradients if $z < 0$, α can be fixed (Leaky ReLU) or optimized (PReLU)



Derivatives



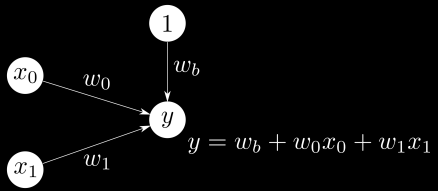
The derivatives of ReLU ($z > 0$) and LeakyReLU do not saturate.

Rules of thumb

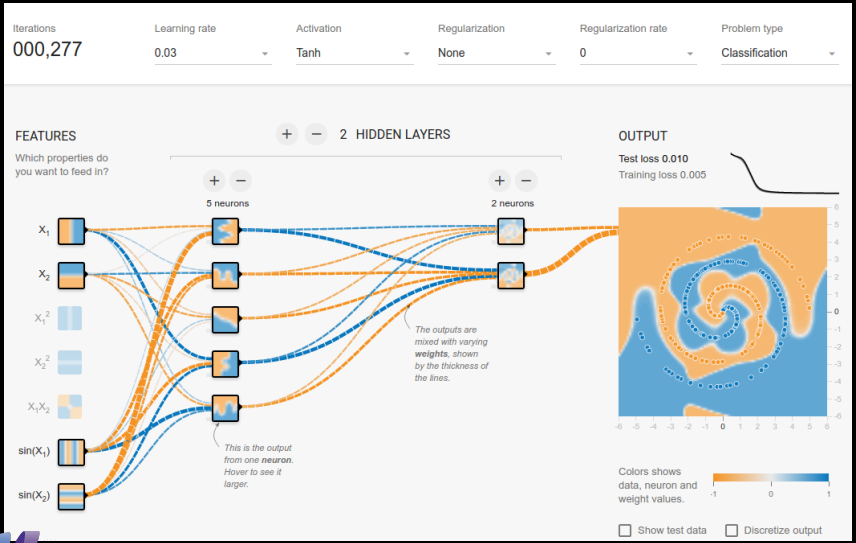
- Your network will train better and faster if the input to the activation functions is around 0
- You can achieve this by initializing the parameters well
- You can also normalize the input to, e.g., zero-mean unit-variance
- You can use techniques such as batch normalization and layer normalization (later in this course)

A minimal PyTorch example

```
model = torch.nn.Linear(in_features=2, out_features=1, bias=True)
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
loss_function = torch.nn.MSELoss()
for iteration in range(1000):
    optimizer.zero_grad()
    output = model(x)
    loss = loss_function(output, y)
    loss.backward()
    optimizer.step()
```



Demo

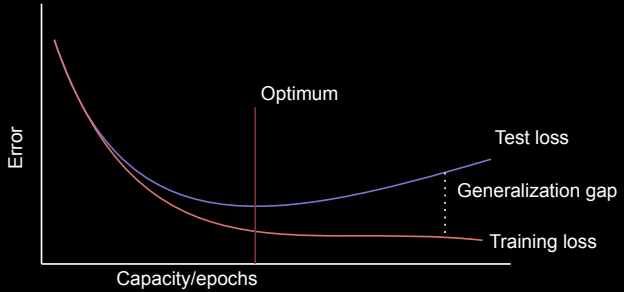

Iterations
000,277
Learning rate
0.03
Activation
Tanh
Regularization
None
Regularization rate
0
Problem type
Classification
FEATURES
Which properties do you want to feed in?
2 HIDDEN LAYERS
5 neurons
2 neurons
X1
X2
X1²
X2²
X1X2
sin(X1)
sin(X2)
This is the output from one neuron. Hover to see it larger.
The outputs are mixed with varying weights, shown by the thickness of the lines.
OUTPUT
Test loss 0.010
Training loss 0.005
Colors shows data, neuron and weight values.
Show test data
Discretize output

Generalization

- We can now optimize a network for known data, but what about new data?
- Central to machine learning is designing algorithms that will perform well on new data
- This ability is called **generalization**
- **Training error** is the error computed on the training set
- During the training (learning) we aim to reduce the training error
- If that is the end goal, we only have an optimization problem, not a machine learning one

Generalization error

- **Generalization error** is defined as the expected error on new, previously unseen data
- Unlike in optimization, in machine learning our main goal is to minimize the **generalization error**
- Usually the generalization error is estimated by measuring the performance on a **test data set** which must be independent from the training set



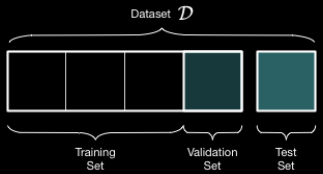
Assumptions

- We often make assumptions about the data sets
  - The training and test data are generated by drawing from a probability distribution over data sets. We refer to that as **data-generating process**
  - **i.i.d. assumptions**
    - Examples are **independent** from each other
    - The training data set and the test data set are **identically distributed**, i.e., drawn from the same probability distribution
- **Question:** Can you name scenarios in medical image analysis where the i.i.d. assumptions are bound to be broken?

Data splits

Split data into multiple sets

- **Training** set that will be used to optimize your *parameters/weights*
- **Validation** set that will be used to tweak hyperparameters such as the learning rate
- **Test** set that will be used for final evaluation as a surrogate of new and unseen data



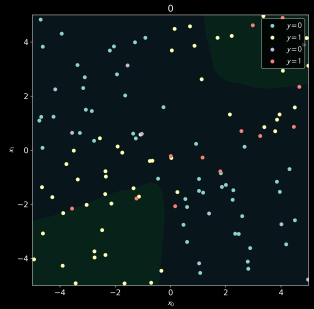
Cross-validation $\rightarrow$ Next lecture

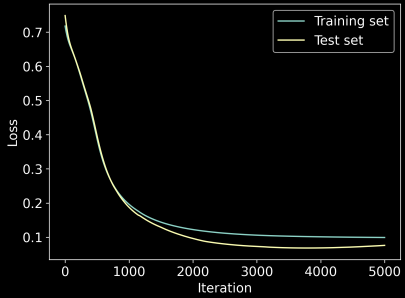
Important

Splitting data

Always split medical image data sets at the patient level!

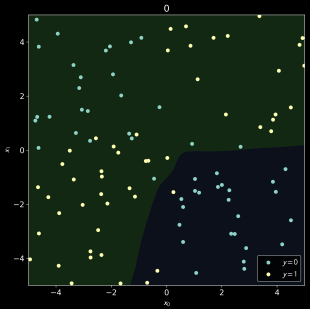
No overfitting: low-complexity model

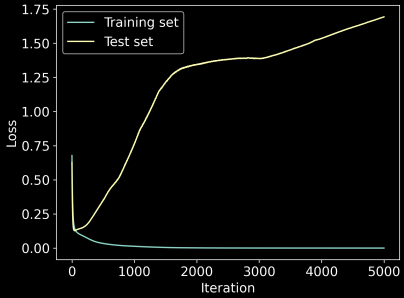





```
model = nn.Sequential(
          nn.Linear(in_features=2, out_features=4),
          nn.ReLU(),
          nn.Linear(4, 1)
        )
```

Overfitting: high-complexity model






```
model = nn.Sequential(
          nn.Linear(in_features=2, out_features=128),
          nn.ReLU(),
          nn.Linear(128, 128),
          nn.ReLU(),
          nn.Linear(128, 1)
        )
```

Model evaluation

- To optimize a model we use a *loss*
- To quantitatively evaluate a machine learning algorithm we need to define a **metric**
- Usually this **metric** is specific to the task carried out by the algorithm.
- For classification tasks a natural measure is the model **accuracy**
- **Accuracy** is defined as the proportion of examples for which the model produces the correct output
- Many more metrics can be computed based on the confusion matrix

Confusion matrix

For a given binary classifier there are four possible outcomes

		True class	
		1	0
Predicted class	1	True positives (TP)	False positives (FP)
	0	False negatives (FN)	True negatives (TN)

Binary classifications metrics

- For instance the accuracy can be defined as

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN}$$

- Also quite frequently used measures are

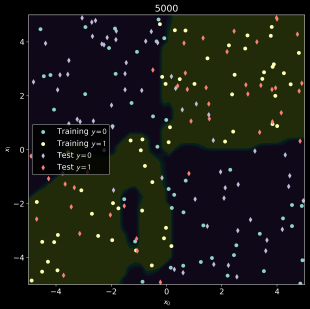
$$Sensitivity = \frac{TP}{TP + FN} \quad Specificity = \frac{TN}{TN + FP} \quad Precision = \frac{TP}{TP + FP}$$

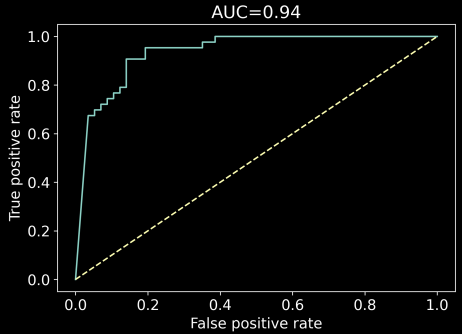
Confusion matrix confusion

		Predicted condition			
	Total population = P + N	**Positive (PP)**	**Negative (FN)**	Informedness, bookmaker informedness (BM) = TPR + TNR − 1	Prevalence threshold (PT) $= \frac{\sqrt{TPR \times FPR} - FPR}{TPR - FPR}$
Actual condition	**Positive (P)**	**True positive (TP)**, hit	**False negative (FN)**, type II error, miss, underestimation	True positive rate (TPR), recall, sensitivity (SEN), probability of detection, hit rate, power $= \frac{TP}{P} = 1 - FNR$	False negative rate (FNR), miss rate $= \frac{FN}{P} = 1 - TPR$
	Negative (N)	**False positive (FP)**, type I error, false alarm, overestimation	**True negative (TN)**, correct rejection	False positive rate (FPR), probability of false alarm, fall-out $= \frac{FP}{N} = 1 - TNR$	True negative rate (TNR), specificity (SPC), selectivity $= \frac{TN}{N} = 1 - FPR$
	Prevalence $= \frac{P}{P + N}$	Positive predictive value (PPV), precision $= \frac{TP}{PP} = 1 - FDR$	False omission rate (FOR) $= \frac{FN}{PN} = 1 - NPV$	Positive likelihood ratio (LR+) $= \frac{TPR}{FPR}$	Negative likelihood ratio (LR−) $= \frac{FNR}{TNR}$
	Accuracy (ACC) = $\frac{TP + TN}{P + N}$	False discovery rate (FDR) $= \frac{FP}{PP} = 1 - PPV$	Negative predictive value (NPV) = $\frac{TN}{PN} = 1 - FOR$	Markedness (MK), deltaP (Δp) = PPV + NPV − 1	Diagnostic odds ratio (DOR) = $\frac{LR+}{LR-}$
	Balanced accuracy (BA) = $\frac{TPR + TNR}{2}$	F_1 score $= \frac{2PPV \times TPR}{PPV + TPR} = \frac{2TP}{2TP + FP + FN}$	Fowlkes–Mallows index (FM) = $\sqrt{PPV \times TPR}$	Matthews correlation coefficient (MCC) $= \sqrt{TPR \times TNR \times PPV \times NPV} - \sqrt{FNR \times FPR \times FOR \times FDR}$	Threat score (TS), critical success index (CSI), Jaccard index = $\frac{TP}{TP + FN + FP}$

ROC curve

- A very common way to visualize the trade-off between TPR and FPR is the ROC curve
- The area under this curve (AUC) quantifies performance of a classifier (higher = better)





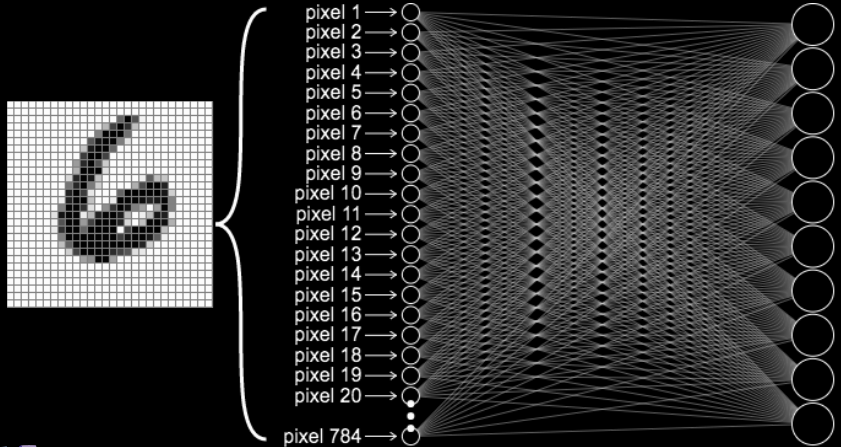
Discussion point

You have developed a method for some image analysis diagnostic task that has very high sensitivity (e.g. 0.99) but relatively low specificity (e.g. 0.25). Can this be still a useful tool for clinicians and if so in what context?

Higher-dimensional inputs

- All previous toy examples had either one or two input values in $\mathbf{x}$
- We classified samples in a two-dimensional space
- Real sample spaces can have many more dimensions
  - A car can be described by color, brand, type, year etc.
  - A person can be described by age, color of hair, gender, length, height, etc.
- Images are a function $f(\mathbf{x})$ in 2D or 3D, but we typically consider them on grids
- These grids are extremely high-dimensional: every pixel/voxel is a dimension!

High-dimensional input

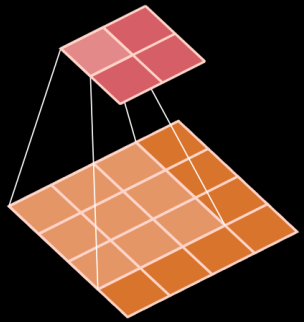


Neural networks on images

- Image analysis tasks
  - Image classification
  - Image segmentation
  - Object detection
  - ...
- A medical image easily has $>$ million pixel/voxels
- E.g., a CT image has $512 \times 512 \times 200 > 50,000,000$ voxel values
- The simplest fully connected classification network would *also* have $> 50,000,000$ weights
- Such a neural network will overfit to noise in the data

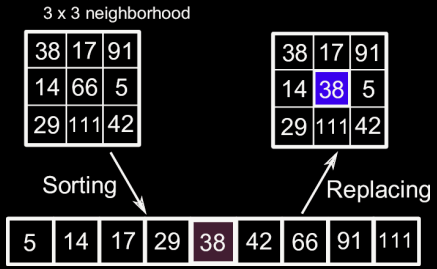
Image filters

- Filters take an image as input and provide another image
- Extreme case: a filter is an $n_{pixel} \times n_{pixel}$ multiplication matrix
- In practice, a filter is much smaller
- We shift a filter window over the image and apply the same operations at each pixel
- This results in a *filtered* image

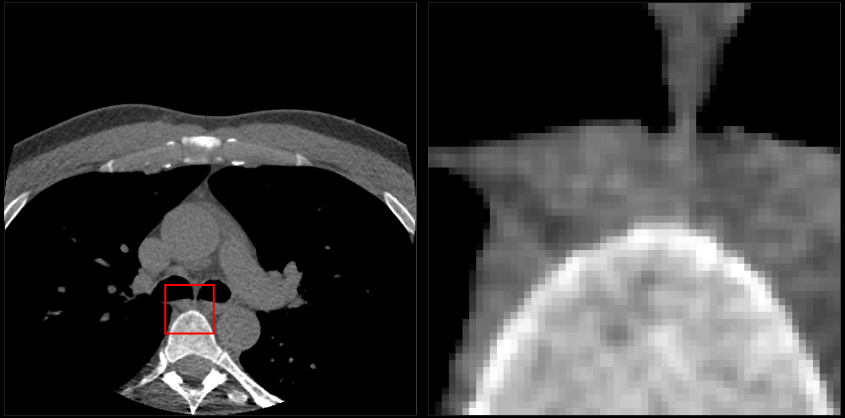


Non-linear filter: median filter

- At each pixel, put values in a local neighborhood in an array
- Sort those values
- Replace the pixel intensity with the median of the sorted values



Non-linear filter: median filter



Linear filters: convolution

In 2D: Let f the image, h be a kernel of size $2k+1 \times 2k+1$, and g the output image

$$g(i,j) = \sum_{u=-k}^{k} \sum_{v=-k}^{k} h(u,v) f(i-u, j-v)$$

Cross-correlation

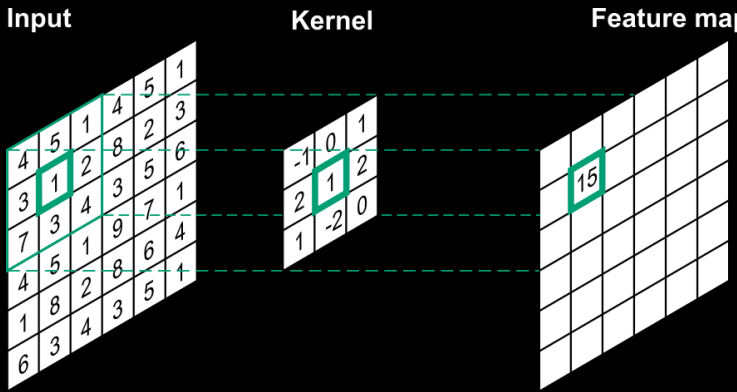
Convolution $g * h$

$$f(i,j) = \sum_m \sum_n h(i-m, j-n)\, g(m,n)$$

Cross-correlation $g \otimes h$

$$f(i,j) = \sum_m \sum_n h(i+m, j+n)\, g(m,n)$$

Linear filters: cross-correlation

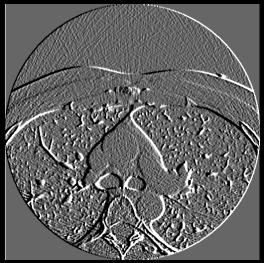


$$g(i,j) = \sum_{u=-k}^{k} \sum_{v=-k}^{k} h(u,v) f(i+u, j+v)$$

Image filtering

Which filter leads to which image?

-1	-1	-1
-1	9	-1
-1	-1	-1

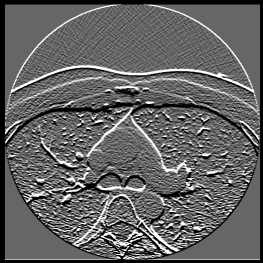


1/9

1	1	1
1	1	1
1	1	1

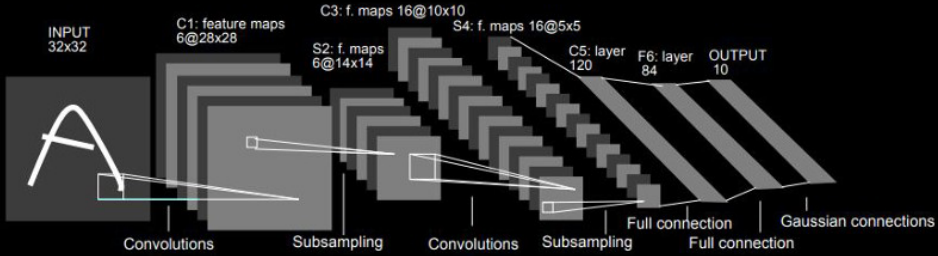
-1	0	1
2	0	2
-1	0	1

1	2	1
0	0	0
-1	-2	-1



Convolutional neural networks

- Combine fully connected layers for classification with *convolution layers*
- Use the structure of the image as an inductive bias
- Use downsampling layers to reduce complexity
- Hidden convolution and downsampling layers are now the feature extractor ϕ



LeCun et al., 1998

Convolutional neural networks

Key idea

Convolution kernels are not predefined, but are weights that are optimized using backprop and gradient descent.

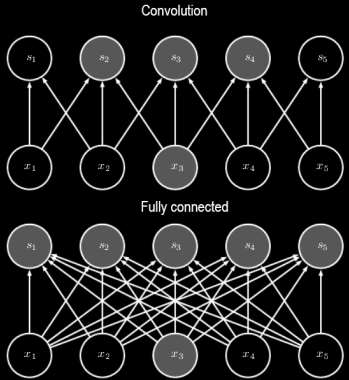
Why convolution?

Adding convolution to neural networks has three benefits

- Sparse interactions
- Parameter sharing
- Equivariant representations

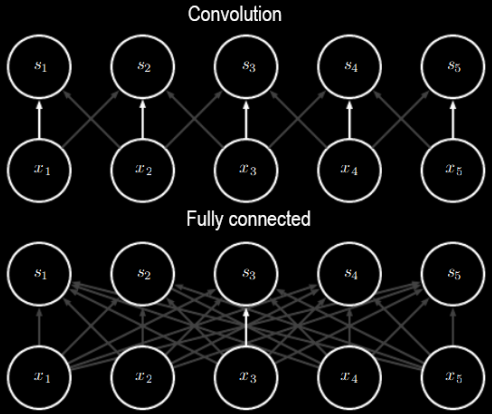
Sparse interactions

- Fewer weights
- Faster computing
- Lower chance of overfitting



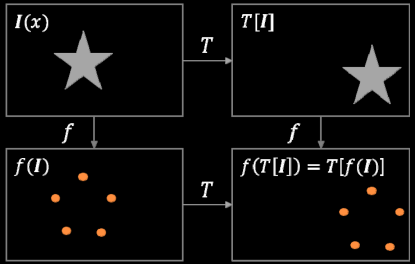
Parameter sharing

- The convolution kernel is the same everywhere in the image
- This drastically reduces the number of parameters
- Reduces the risk of overfitting

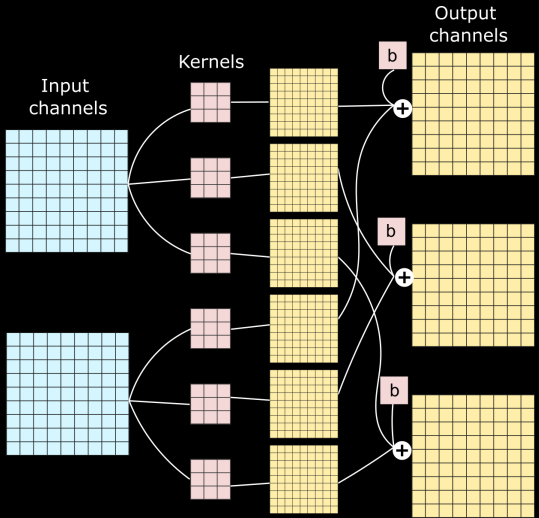


Equivariant properties

- Let's say we have two functions $f(x)$ and $g(x)$
- **Equivariance** of f to g says that $f(g(x)) = g(f(x))$
- **Invariance** of f to f that $f(g(x)) = f(x)$
- Convolution is translation equivariant (but not rotation, scale, ...)
- More in **Lecture 8**

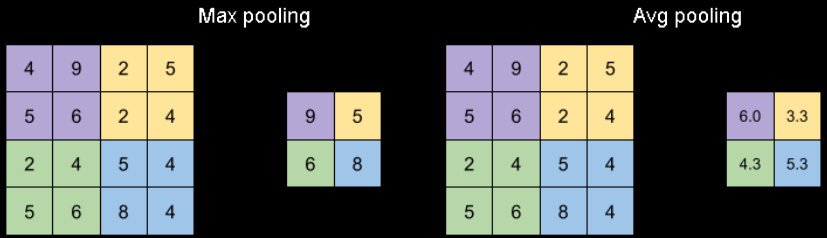


Weights

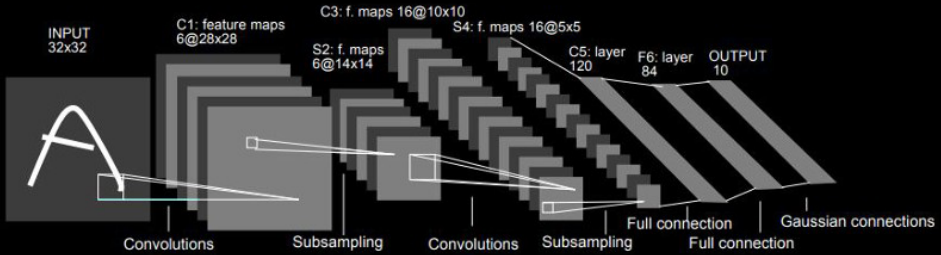


Downsampling

- So-called pooling layers reduce the size of the *feature maps*
- They are similar to non-linear (median) filters
- Not linear, so not differentiable, but can be used in backprop with some bookkeeping
- Adds translation *invariance*



Demo


INPUT
32x32
C1: feature maps
6@28x28
S2: f. maps
6@14x14
C3: f. maps 16@10x10
S4: f. maps 16@5x5
C5: layer
120
F6: layer
84
OUTPUT
10
Convolutions
Subsampling
Convolutions
Subsampling
Full connection
Full connection
Gaussian connections

Summary

We have discussed

- Feedforward neural networks
- Convolutional neural networks
- Evaluation *[Fawcett et al.]*

Don't forget

- Tutorial tomorrow
- Project plan due **Monday February 17, 5pm**

Materials

- *[Dive into deep learning book - see Canvas]*
- *[Fawcett et al.]*
- Slides Dr. Mitko Veta, TU/e