Machine Learning:

Diving Deep

Jan Kieseler

This is a very rich topic, with enough content for whole courses.

Outline and overview

Basic principles

- What is a feed-forward NN really
- Gradient descent and back propagation
- The training

Exploiting the structure

- CNNs

Lecture 1

- Attention and transformers
- Graph neural networks

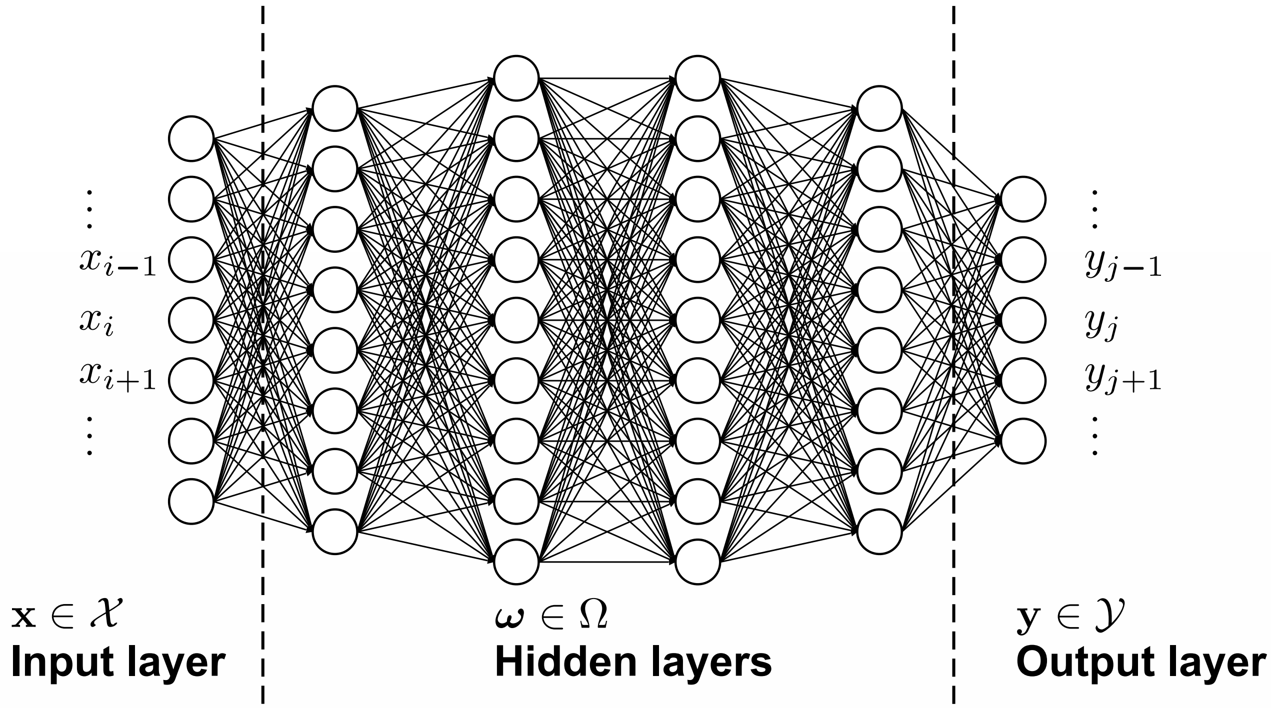
Lecture 2

Examples for advanced applications in HEP

- Low-level reconstruction
- Anomaly detection

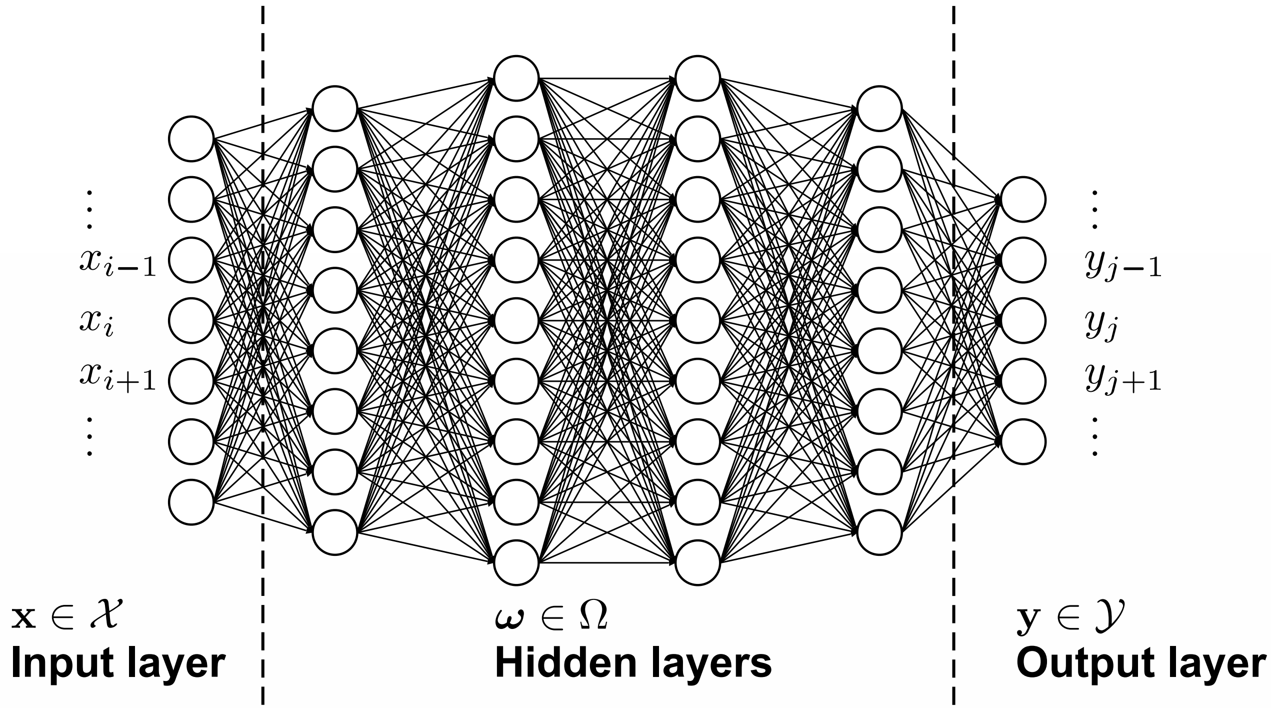
A list of things that are important, but that I could not cover

What is a DNN really?



- All nodes of consecutive layers are connected with each other
- Typically an ANN is called "deep" if it has >4 hidden layers
- Referred to as Multi-Layer Perceptron, Feed-Forward NN

What is a DNN really?



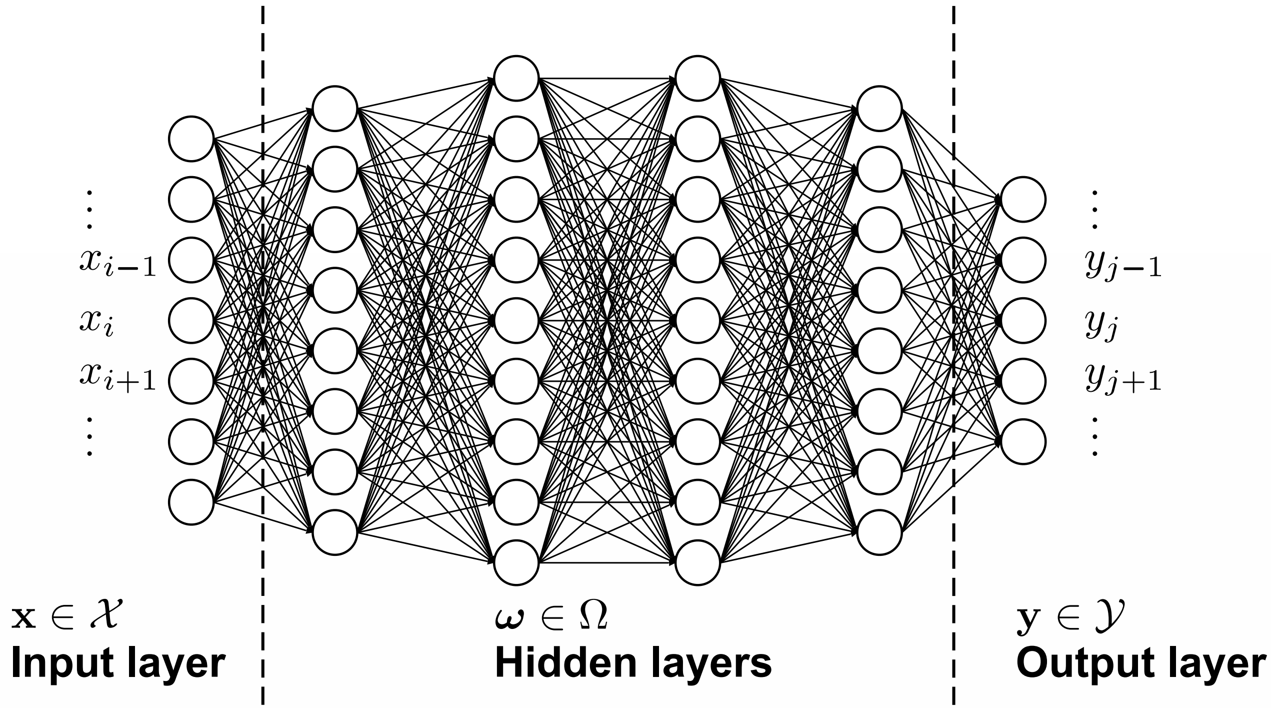
- One layer: $h^{(k+1)}(h^{(k)}) = \theta(\omega_k h^{(k)} + b_k)$

Activation function $\dim(h^{(k+1)}) \rightarrow \dim(h^{(k+1)})$

Weight matrix $\dim(h^{(k+1)}) \times \dim(h^{(k)})$

Bias vector: $\dim(h^{(k+1)})$

What is a DNN really?



- One layer: $h^{(l+1)}(h^{(l)}) = \theta(\omega_k h^{(l)} + b_l)$
- Full DNN: $y(x) = h^{(4)}(h^{(3)}(h^{(2)}(h^{(1)}(x))))$

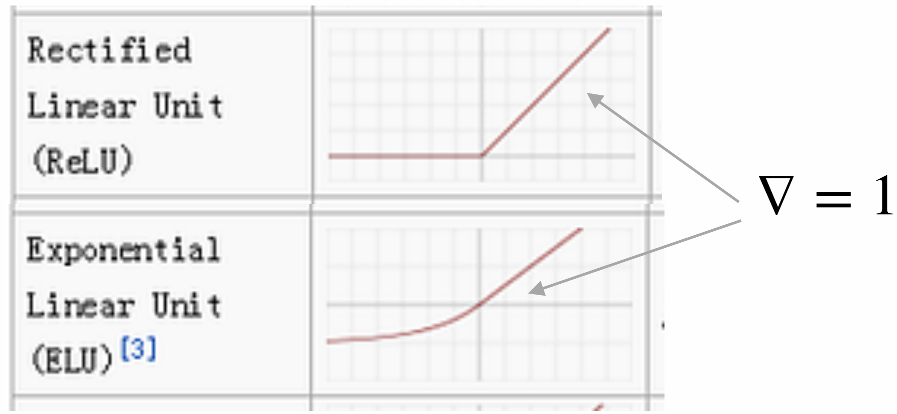
Activation functions: adding non-linearities

- One layer: $h^{(k+1)}(h^{(k)}) = \theta(\omega_k h^{(k)} + b_k)$

- Without non-linear activation:
  $y(x) = h^{(4)}(h^{(3)}(h^{(2)}(h^{(1)}(x)))) = \tilde{\omega} x + \tilde{b}$

Back-of-the envelope exercise

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$



- There is a whole zoo: theoretically, the choice does not matter for hidden layers
  - For the output it **does** matter as it restricts / shapes the output distribution
- In practice: vanishing/exploding gradients, initialisations, normalisation …
  - Suggestion: (s/r)elu

https://machinelearninggeek.com/activation-functions/

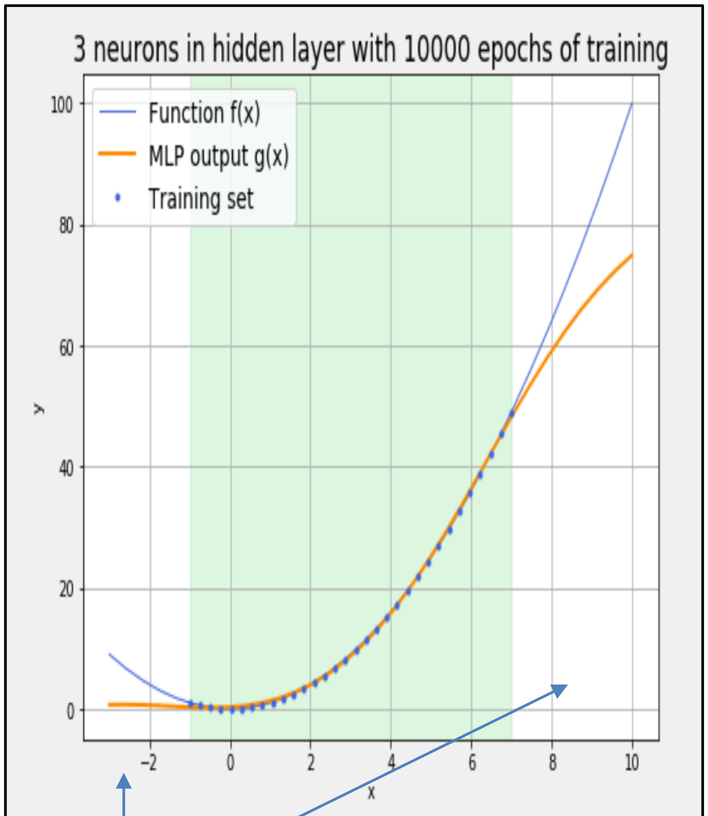
DNNs: very powerful universal function approximators

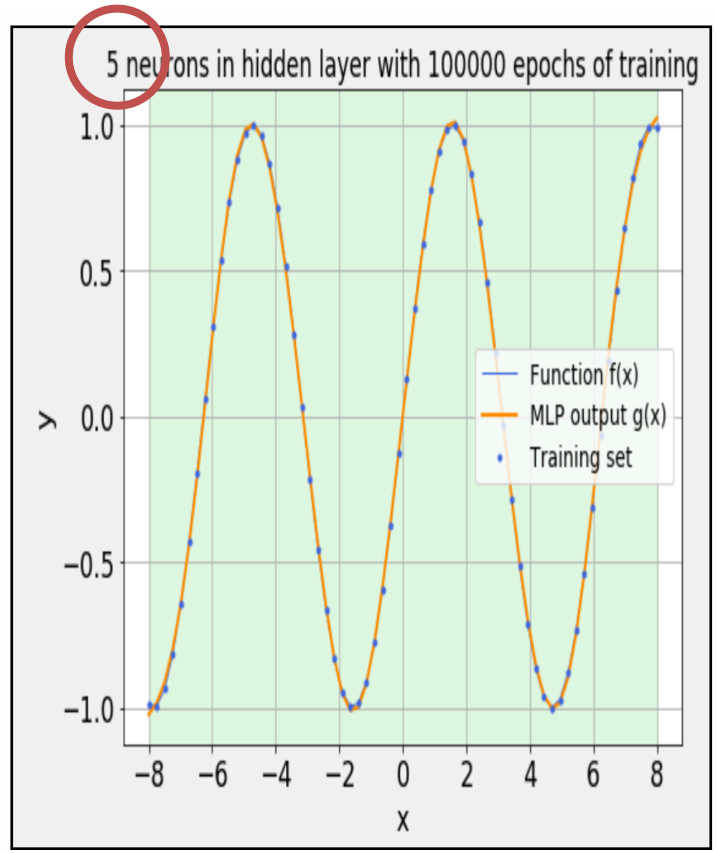
- Very simple NN: one hidden layer, one input, one output, $\tanh$ activation
  $\Phi(\omega, x) = \omega_1 \tanh(\omega_0 x + b)$

1 x 3 matrix

3 x 1 matrix

3 vector

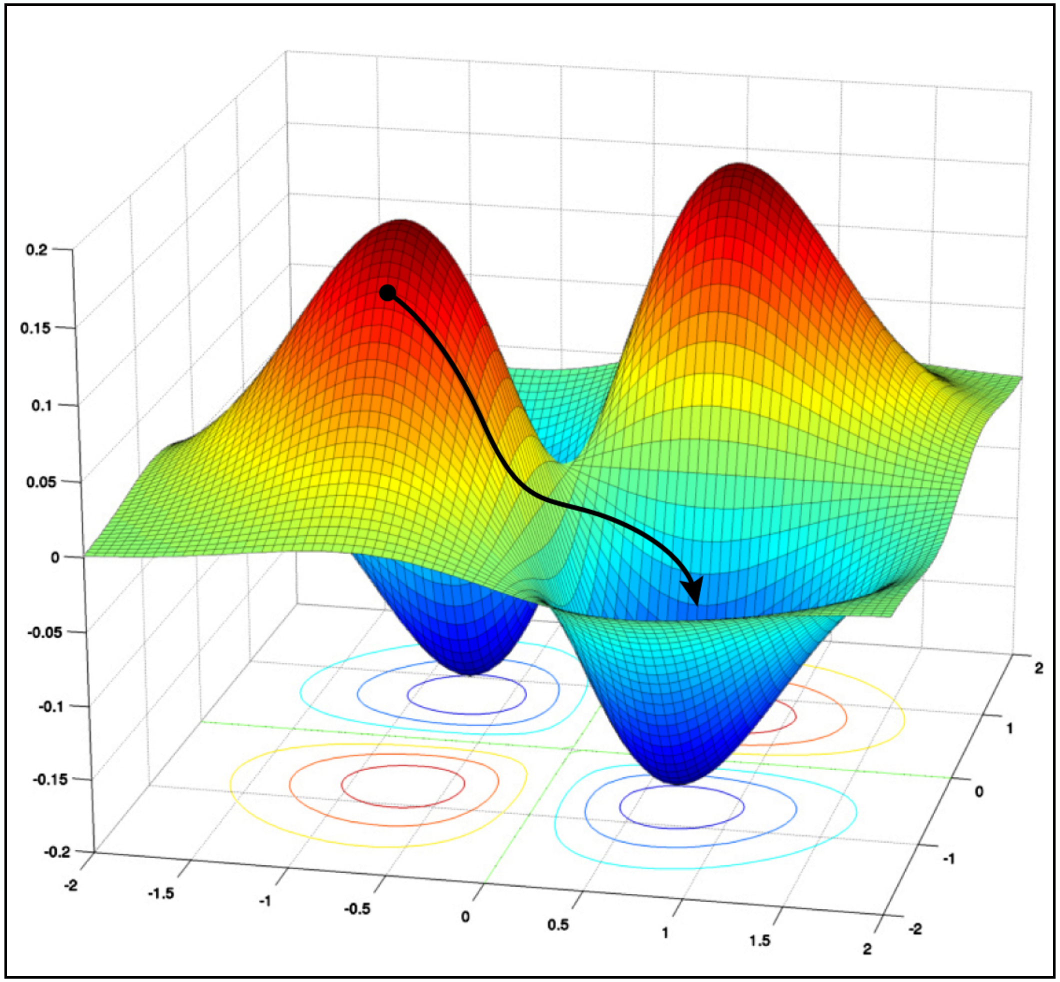




https://notebook.community/kit-cel/lecture-examples/mloc/ch3_Deep_Learning/pytorch/function_approximation_with_MLP

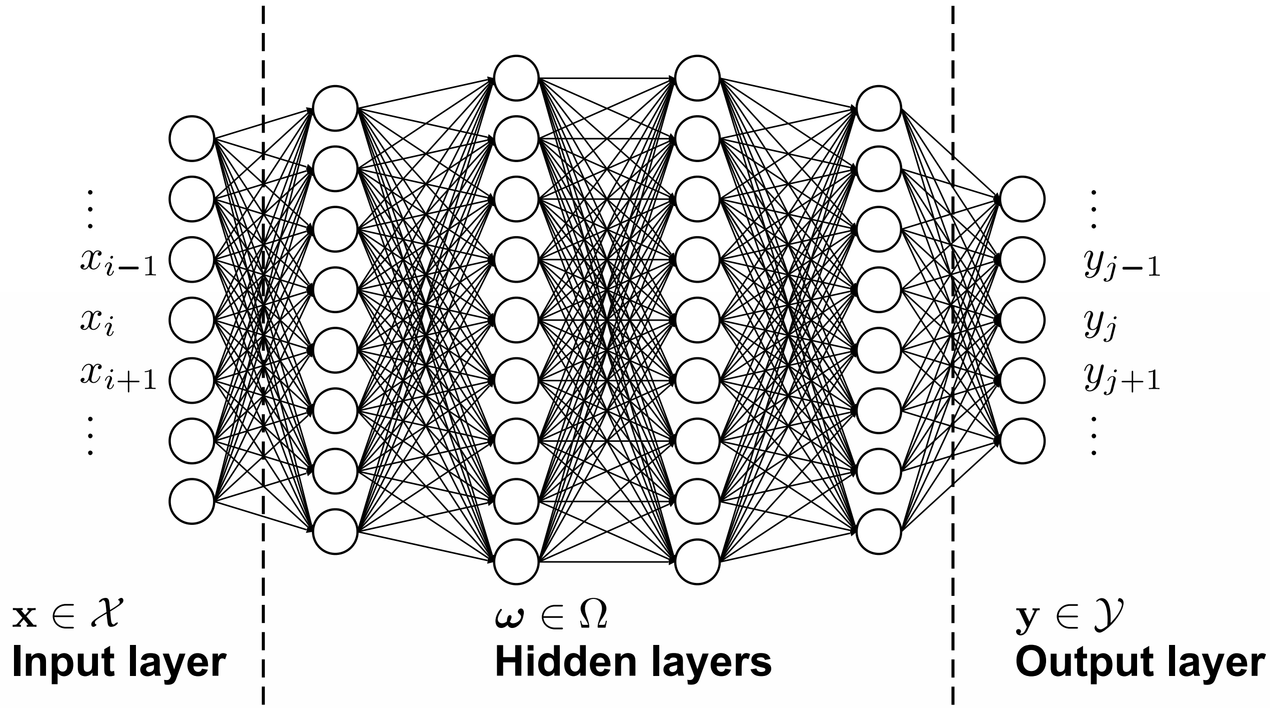
“Out-of-distribution”

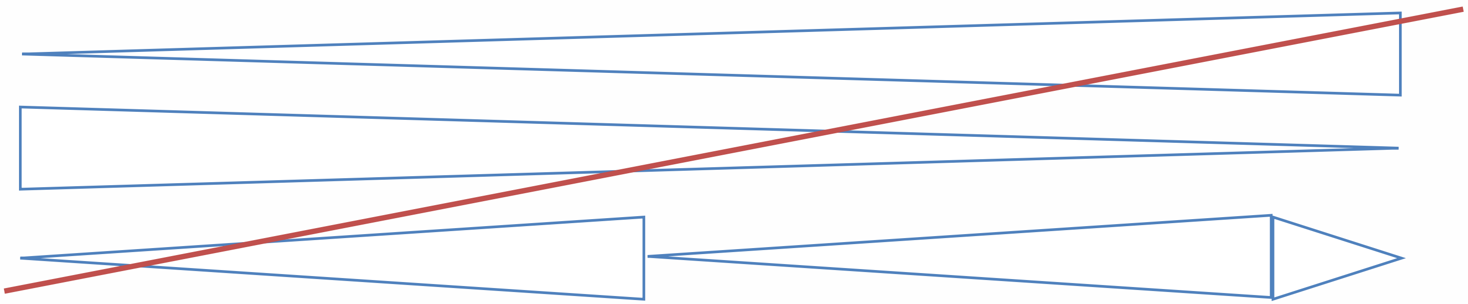
Training



https://medium.com/@tejovk311/optimization-challenges-in-deep-learning-a4b085d529b6

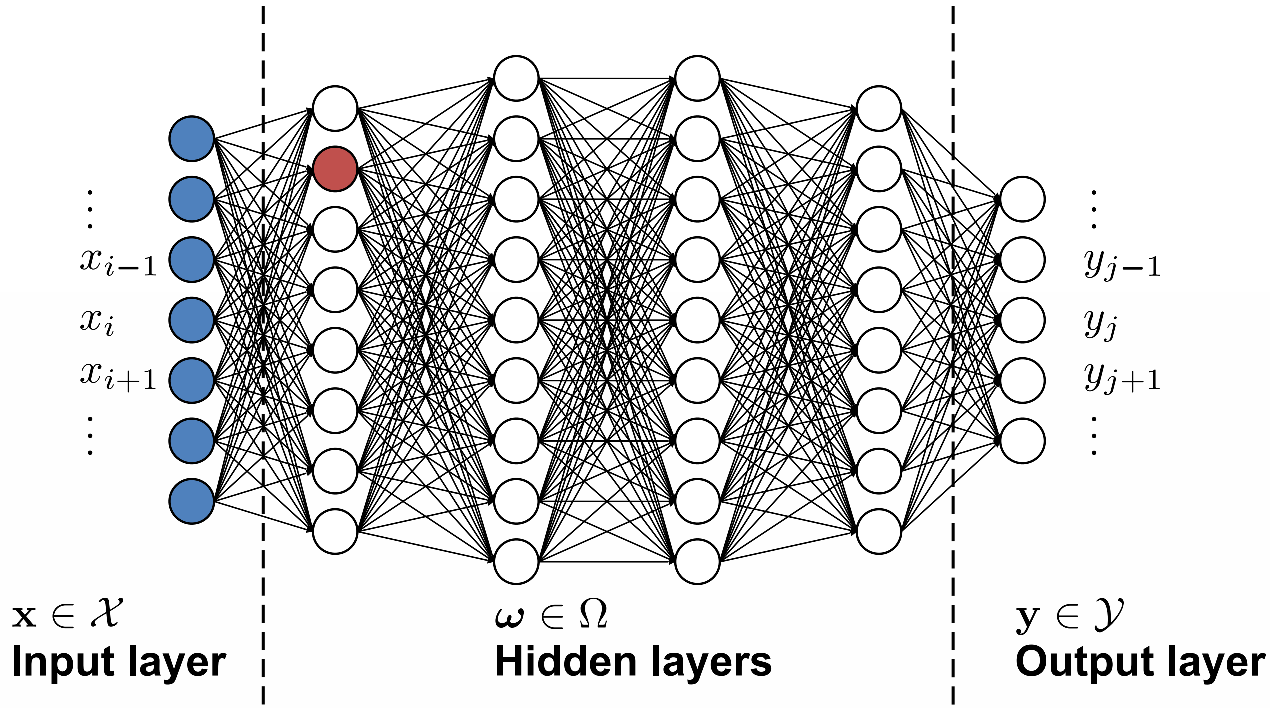
Parameter initialisation and preprocessing: super short





- Keep inputs, the expected outputs, and values within the network as much as possible close to distributions with mean = 0 and variance = 1

Parameter initialisation and preprocessing: super short



Truth

Keep truth normalised

Normalise

Initialise weights 'the right way'

- Each input uncorrelated, normal distributed ($\mu = 1,\ \sigma = 1$), **linear (no) activation**
- Then the red node is normal distributed with variance N = N_{inputs}
- Initialise $\omega^{(1)}$ normal distributed, scaled by $1/\sqrt{N}$: Glorot initialisation (keras standard)

- The best initialisation is **intertwined with the activation function used**
- They all aim for keeping the variance at 1

Loss (cost) function

- The loss function quantifies how well a model performs

- E.g. text book linear regression: we know the 'truth'
  - Model: $\Phi(\omega, x) = \omega_a x + \omega_b$

  - Least-square method:

$$\min 1/N \sum_{i}^{N} \left((\Phi(\omega, x_i) - y_i)^2 \right) = \min \mathrm{MSE}(\Phi(\omega, x), y)$$

Mean squared error loss

- The mean squared error loss is a standard loss for regression tasks

- It assumes a Gaussian distribution of the NN estimates (log(L))
- We want to map to the whole output range: linear output activation

Classification loss: binary cross-entropy

- For binary classification, we have two options: cat or not cat
  $\hat{y} =: \Phi(\omega, x)$

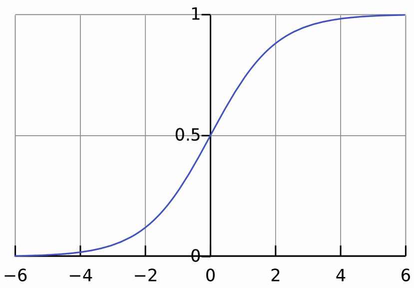
- Probability for a single sample to be identified by the NN (Bernoulli process)
  $P(\hat{y}, y) = \hat{y}^y (1 - \hat{y})^{1-y}$

- The likelihood for N processes factorises:
  $\Pi_{l=1}^{N} (\hat{y}^{(l)})^{y^{(l)}} (1 - \hat{y}^{(l)})^{(1-y^{(l)})}$

- Take log: get binary cross entropy loss:

$$\sum_{l}^{N} \left(y^{(l)} \log(\hat{y}^{(l)}) + (1 - y^{(l)}) \log(1 - \hat{y}^{(l)}) \right)$$

Sigmoid



$1/(1 + e^{-x})$

➡ The loss choice depends on the distribution you **expect** the network output to have

➡ Map to 0-1 → output activation: **sigmoid**

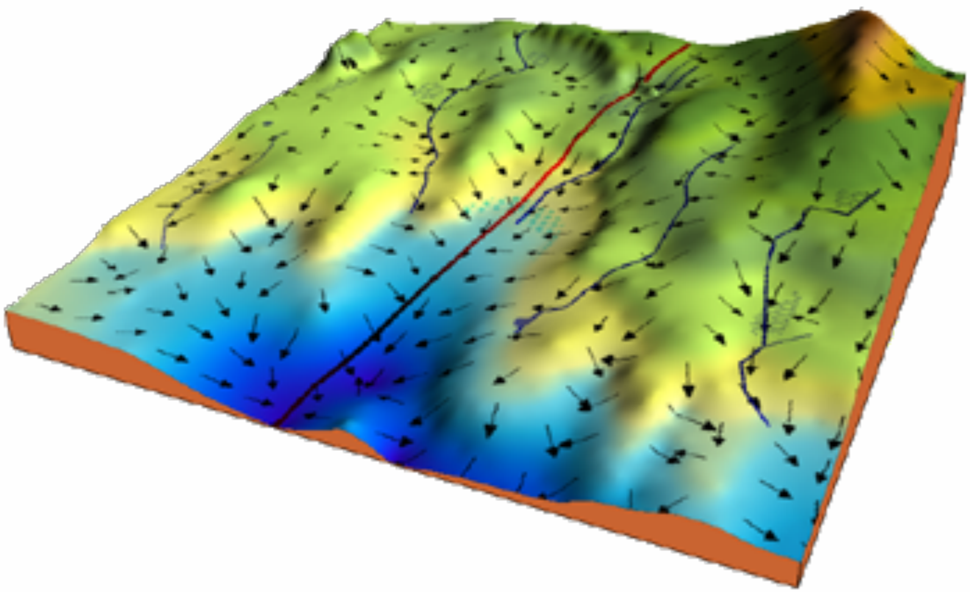
How do we train: gradient descent

- Well established, robust numerical minimisation procedure:

$$\omega^{(k+1)} = \omega^{(k)} - \eta \nabla_{\omega^{(k)}} L\left(\Phi(\omega, x), y\right)$$

Learning rate

- Update ω until $L\left(\Phi(\omega^{(k)}, x), y\right) - L\left(\Phi(\omega^{(k+1)}, x), y\right) < \epsilon$



https://ml-cheatsheet.readthedocs.io/en/latest/gradient_descent.html

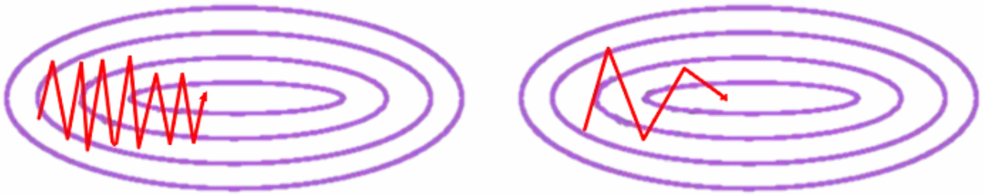
Stochastic gradient descent and momentum

- Stochastic gradient descent is gradient descent on (mini) batches instead of the full data set

$$\omega^{(k+1)} = \omega^{(k)} - \eta \nabla_{\omega^{(k)}} L\left(\Phi(\omega, x), y\right) \rightarrow \omega^{(k+1)} = \omega^{(k)} - \eta \nabla_{\omega^{(k)}} L\left(\Phi(\omega, \{x\}_k), \{y\}_k\right)$$

GD SGD

- Reduces computational burden: makes training feasible
- Introduces extra noise that can actually **help**



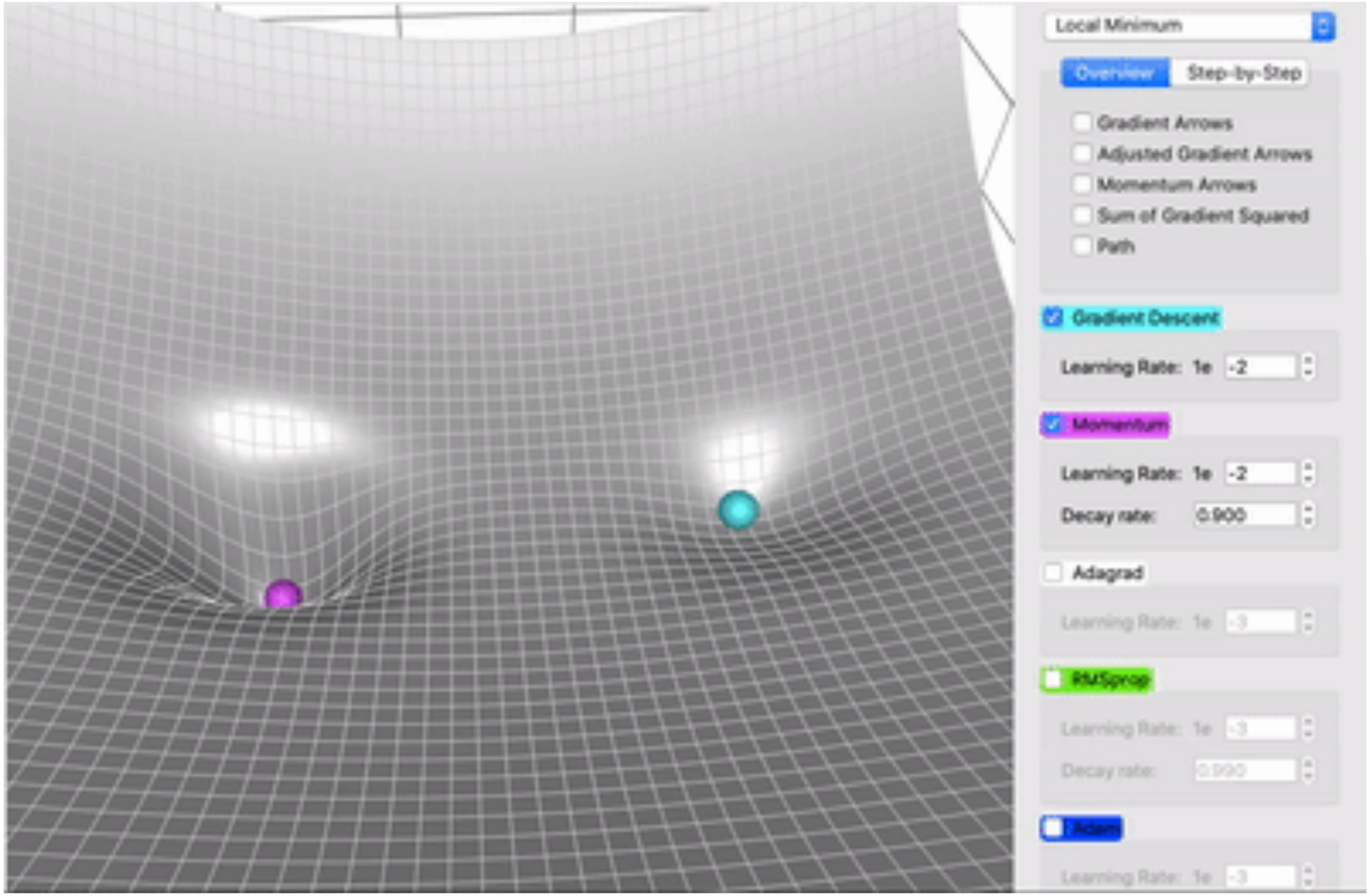
Goodfellow et al. (2016)

- Add a momentum/velocity that averages the general directions in parameter space

$$v^{(k)} = \alpha v^{(k-1)} - \eta \nabla_{\omega^{(k)}} L$$
$$\omega^{(k+1)} = \omega^{(k)} + v^{(k)}$$

➡The basis for most common optimisers that are in use

Momentum in action



The above and many more details (great page)
https://towardsdatascience.com/a-visual-explanation-of-gradient-descent-methods-momentum-adagrad-rmsprop-adam-f898b102325c

Getting the gradients: back propagation

- For each (mini) batch, we calculate a loss value numerically
- Simple "network": $\Phi(\omega, x) = \theta(\omega x)$, Loss $L = (\Phi - y)^2$
- Use chain rule; gradient for ω:

$$\left.\frac{\partial L}{\partial \omega}\right|_{\omega^{(k)},x^{(k)}} = \left.\frac{\partial \theta}{\partial \omega}\right|_{\omega^{(k)},x^{(k)}} \left.\frac{\partial L}{\partial \theta}\right|_{\omega^{(k)},x^{(k)}} = \left((x)\Big|_{\omega^{(k)},x^{(k)}} \cdot (\theta - y)\Big|_{\omega^{(k)},x^{(k)}} \right)$$

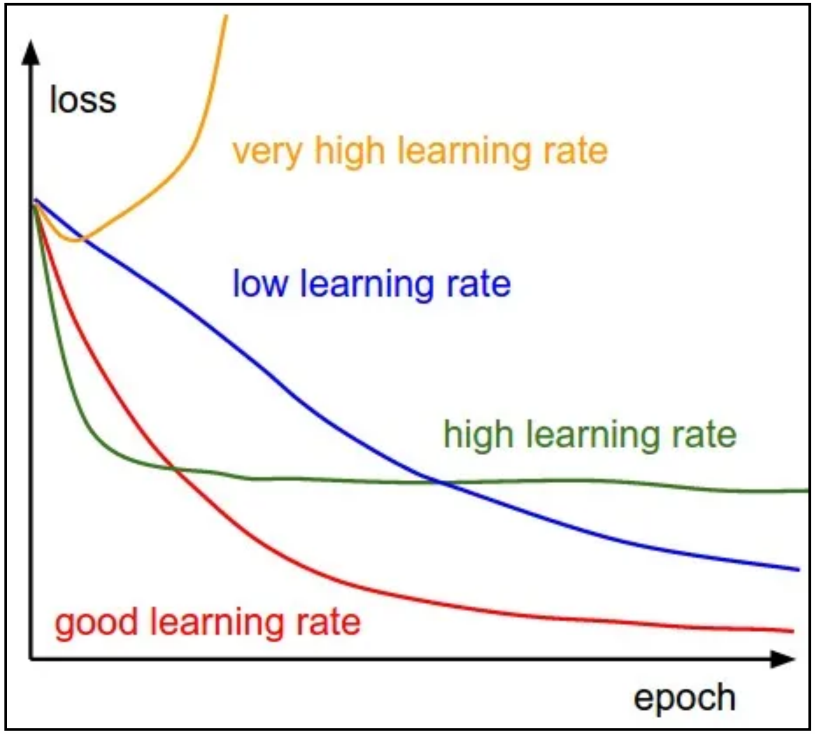
This could be the output of a **previous** layer: $x = h^{(l-1)}$

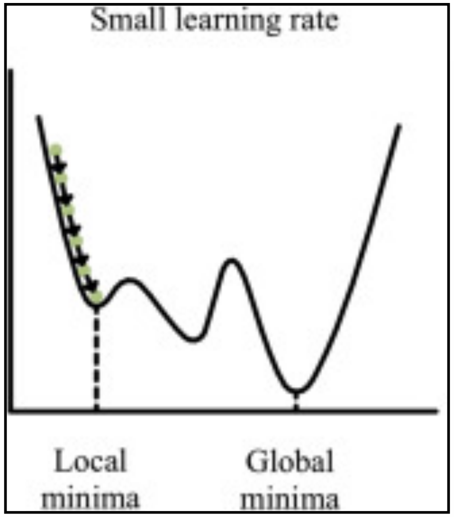
- Can be extended to arbitrary depth
  - The weight gradients for layer l depend on all layers closer to the loss in this simple manner, but **not** on layers $l - m, m > 0$
  - Each operation is simple (fast to calculate)
  - Can (has to) use intermediate results in hidden layers (that's why training takes much more GPU memory than inference)

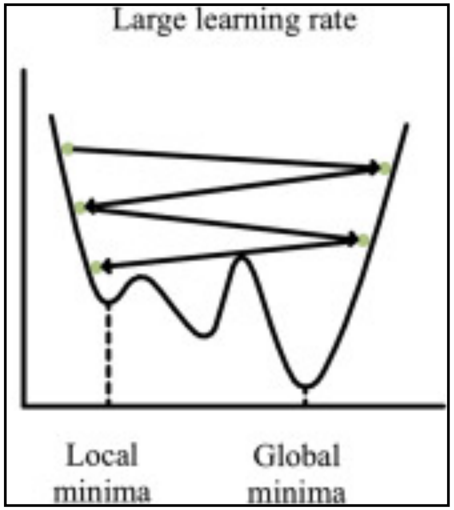
- Gradient calculations happen transparently in modern ML frameworks! (auto-differentiation)

https://alexcpn.github.io/html/NN/ml/8_backpropagation_full/

Learning rates

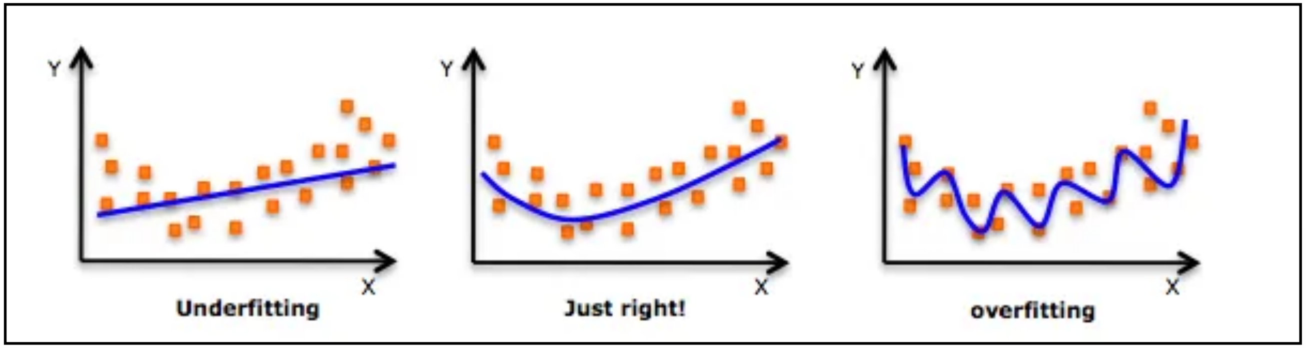


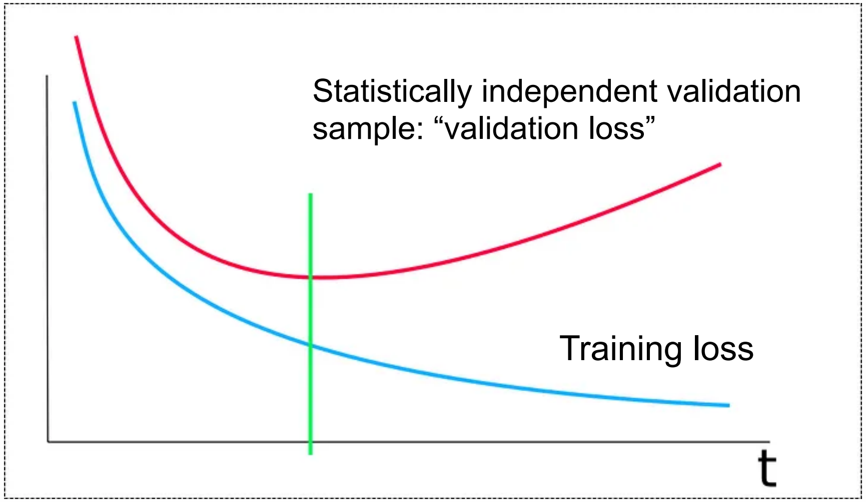




- There is no universally best learning rate - always needs to be adjusted
- Rule of thumb:
  - More parameters ↔ lower learning rate
  - Smaller batches ↔ lower learning rate

Quick interlude: overfitting / overtraining



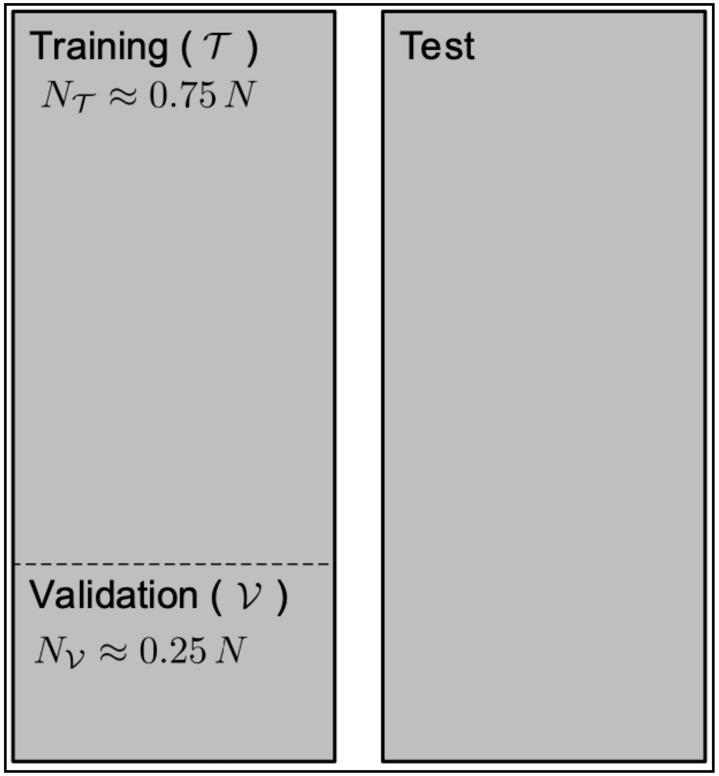


- More data **per weight**:
  - Simpler network
  - More data
- Lower learning rate
- Regularisation (weight regularisation, Dropout) *

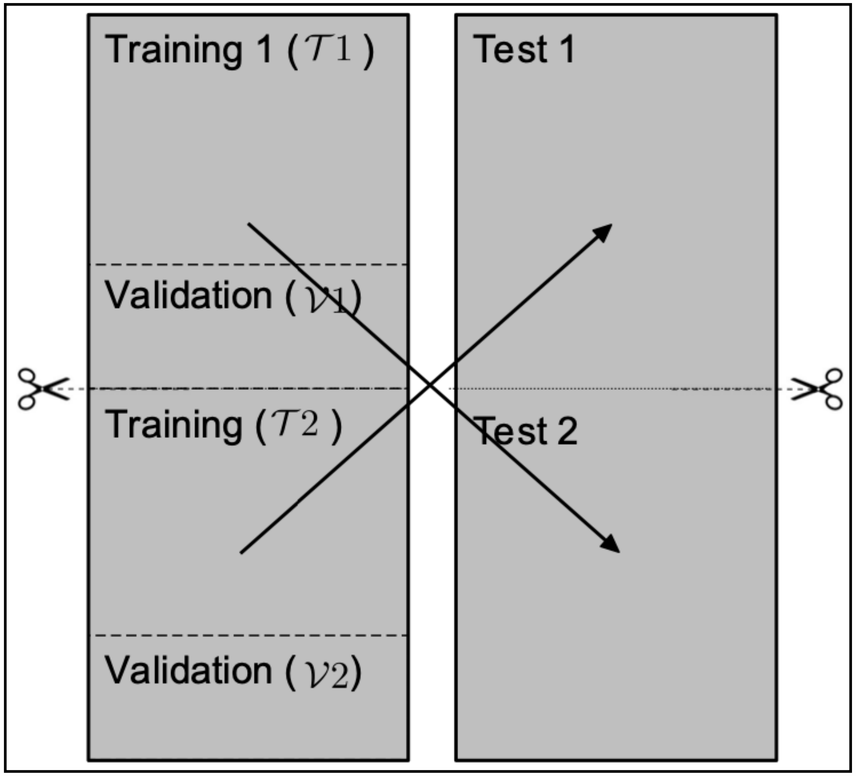
https://medium.com/analytics-vidhya/the-perfect-fit-for-a-dnn-596954c9ea39

Datasets

- The NN will learn from but also to represent the dataset (lossy compression)
- Strictly separate: training, test, validation



2-fold cross validation



- K-fold cross-validation can be very useful if we want to exploit the whole sample

What is different in HEP?

- For most tasks, we have a lot of labelled data at our fingertips: simulation

- Many techniques to deal with small amounts of data …
  - The best initialisation / activation function combination
  - Regularisation techniques
  - Data augmentation

- … are often not worth the effort for standard tasks in HEP

- So while the internet is full of great resources on ML, keep the above in mind

- When used in analyses, make sure inputs **and their correlations** are well modelled

- * There are also methods to dig deeper into how inputs relate to outputs, e.g. Layer-wise relevance propagation or Taylor expansions [arxiv:1803.08782, arXiv:1604.00825, …]

Learning rate

Momentum

Gradients

Expressivity

Time for questions

Losses

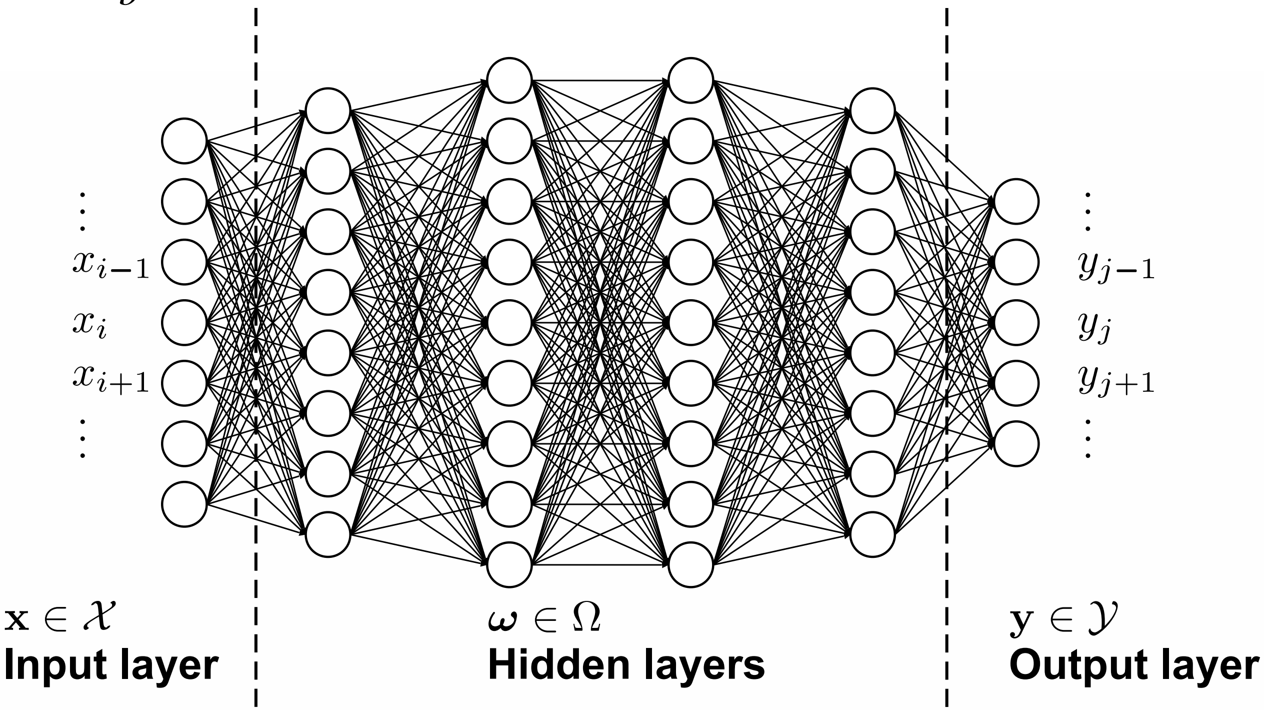
Normalisation

CNNs

Counting parameters

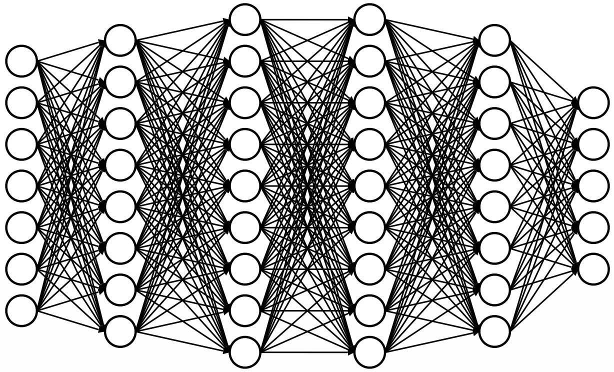
$$N_\omega = \ 7 \times 8 + 8 \times 9 + 9 \times 9 + 9 \times 8 + 8 \times 5 = 321$$

$$N_b \geq \ 8 + 9 + 9 + 8 + 0 = 34$$



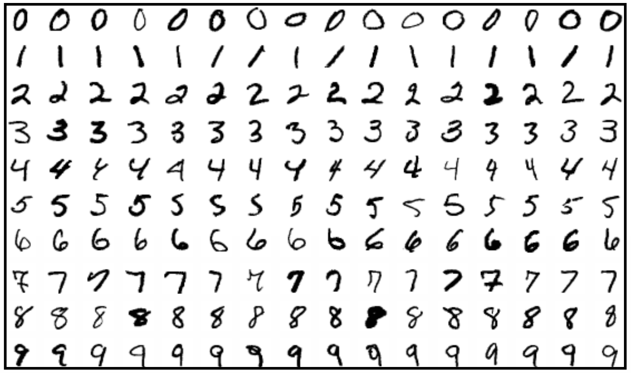
- Typical small MLPs: about 10k - 100k
- ChatGPT4: 1.5 Trillion?
- More free parameters → more expressivity

More parameters → more resources



~10k-100k parameters
Trains in minutes on your laptop
Uses ~10 Wh of electricity

↔



MNIST [L. Deng, IEEE 2012]
60k images

Not an MLP!

OpenAI
ChatGPT**4.0**

~1.5Trillion parameters
Trained 6 month
$100M for compute, roughly 10 000 MWh

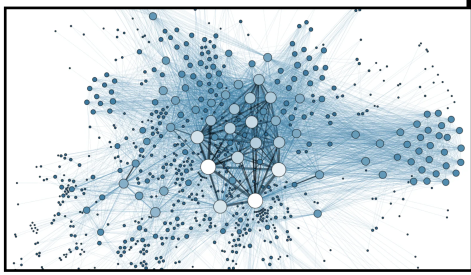
Only estimates, no official numbers

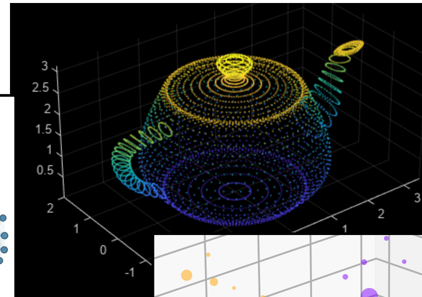
↔

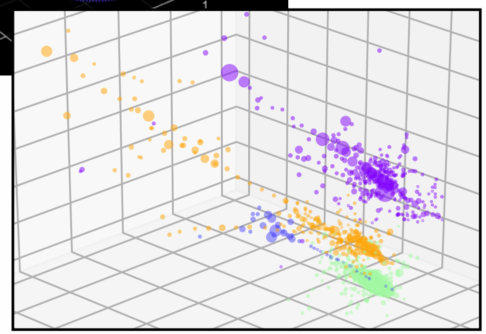
Common Crawl

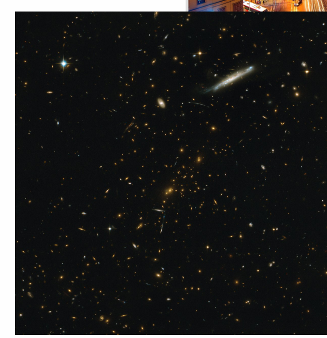
- More parameters:
  - More training data
  - More resources to evaluate
  - Even more resources to train

Structure matters







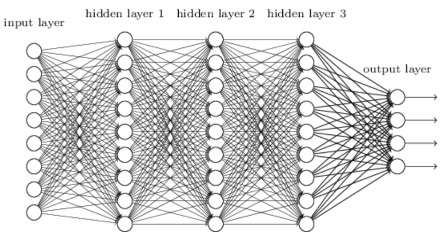


<table>
<tr><td>the</td><td>little</td><td>bear</td><td>saw</td><td>the</td><td>fine</td><td>fat</td><td>trout</td><td>in</td><td>the</td><td>brook</td></tr>
<tr><td>the</td><td colspan="2">bear</td><td>saw</td><td>the</td><td colspan="3">trout</td><td>in</td><td colspan="2">it</td></tr>
<tr><td colspan="3">He</td><td>saw</td><td colspan="4">it</td><td colspan="3">there</td></tr>
<tr><td colspan="3">He</td><td colspan="5">ran</td><td colspan="3">there</td></tr>
<tr><td colspan="3">He</td><td colspan="8">ran</td></tr>
</table>

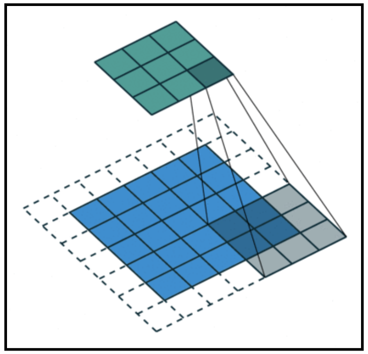
- Architecture needs to fit the desired output ✓
- Architecture needs to fit the **input data**

Main building blocks of architectures

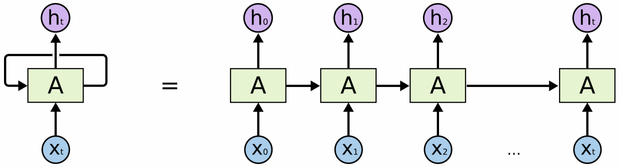
- MLP / Feed forward ✓



- CNNs

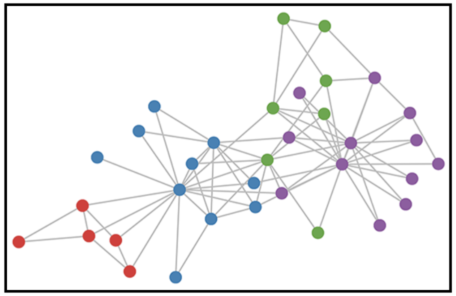


- RNNs   Next time



- Attention

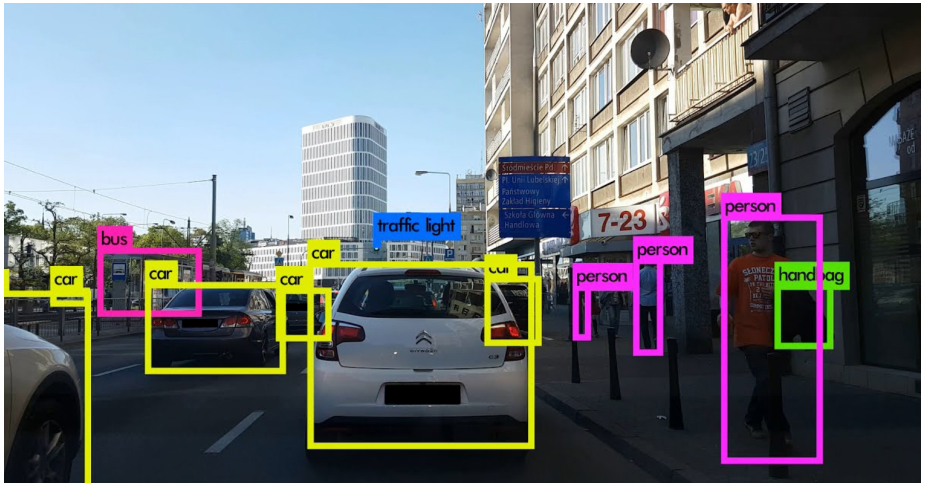
- GNNs

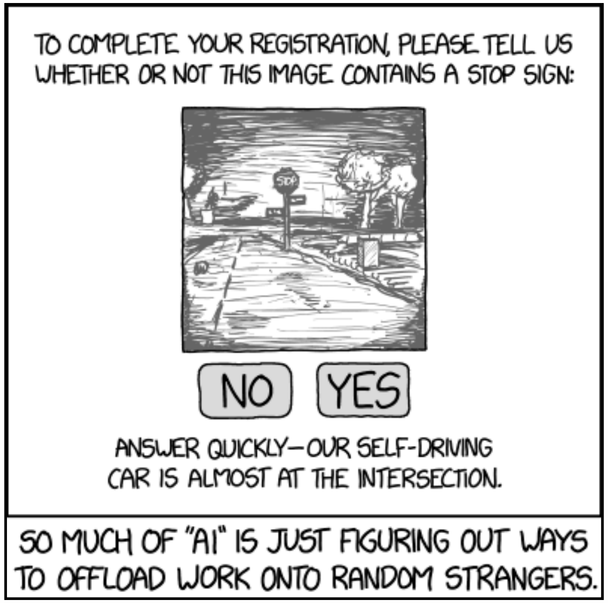


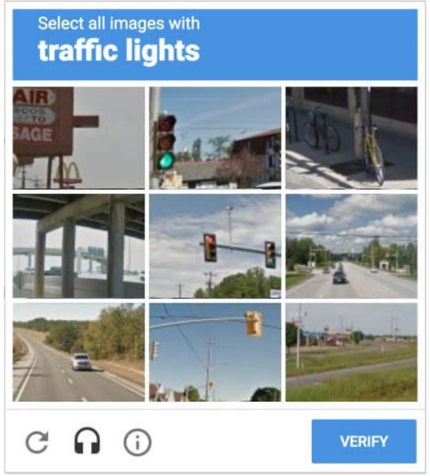
Convolutional Neural Networks

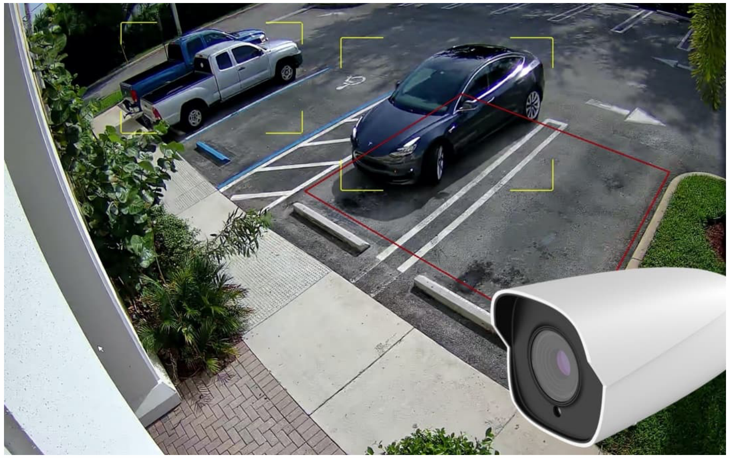
Image-like data

CNNs are everywhere and at the core of computer vision





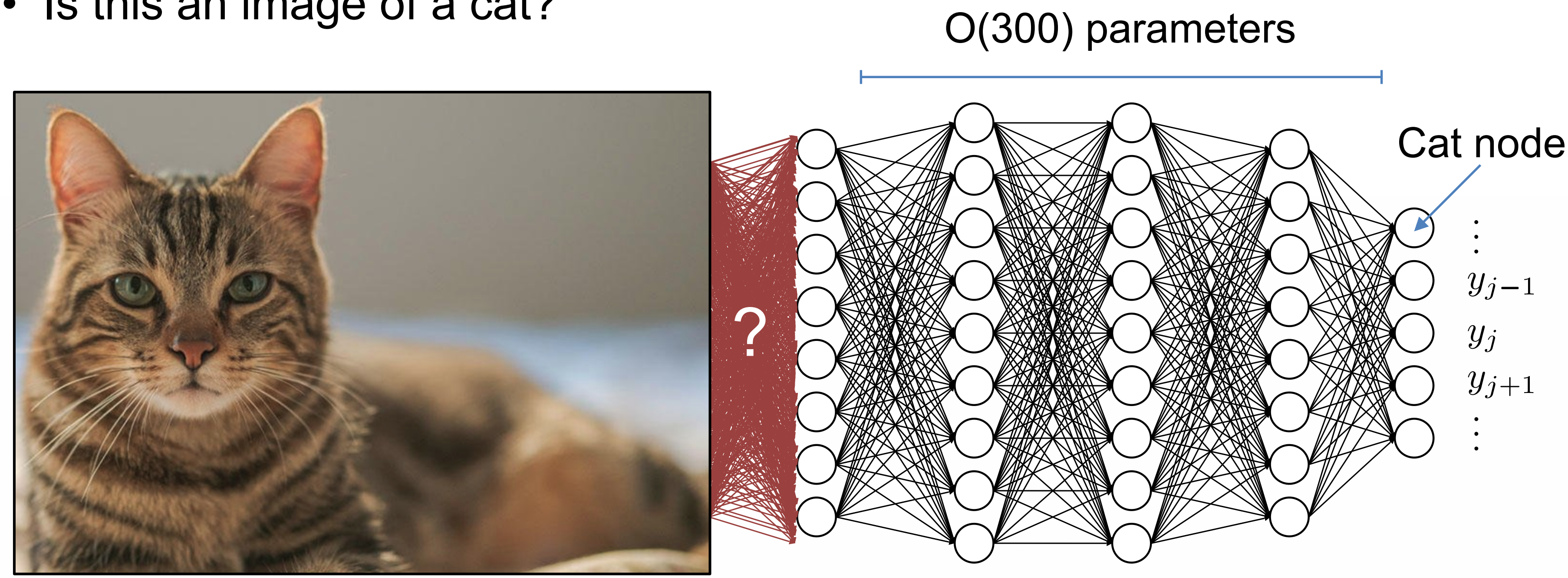




- Self-driving cars
- Surveillance
- Skin cancer detection
- …
- Particle physics

Structure counts

- Is this an image of a cat?



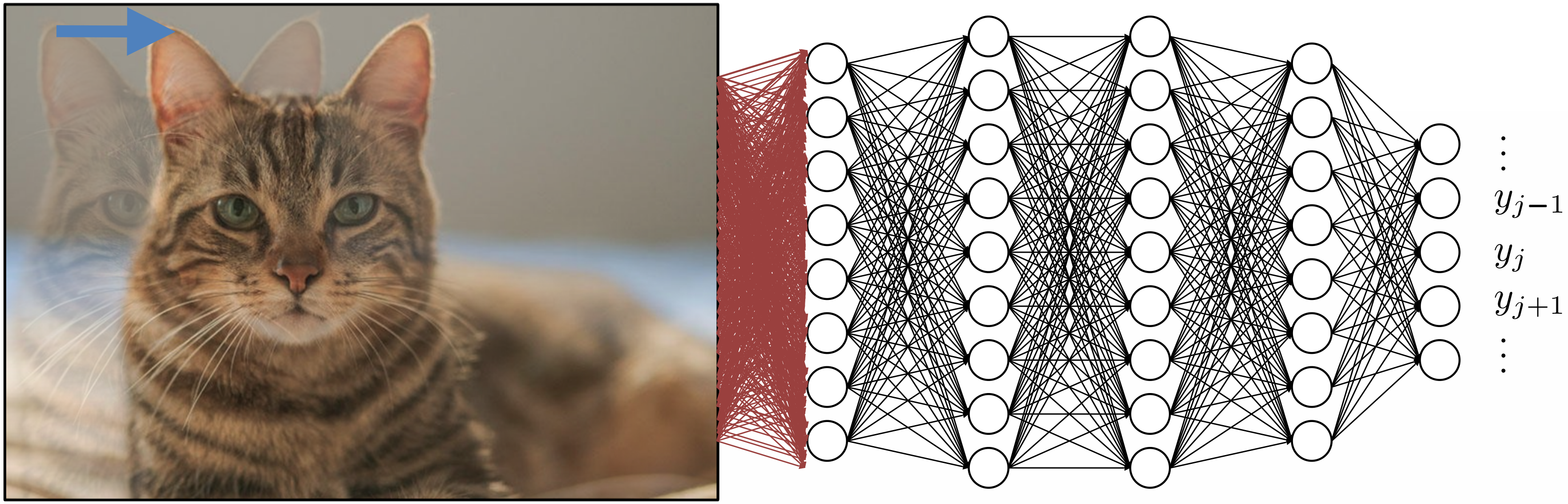
- Typical (phone) cameras 10-50 MP
- How many parameters does the first layer have?

- In this example: **80 - 400 million parameters** in first layer

- Also, this architecture will not perform well

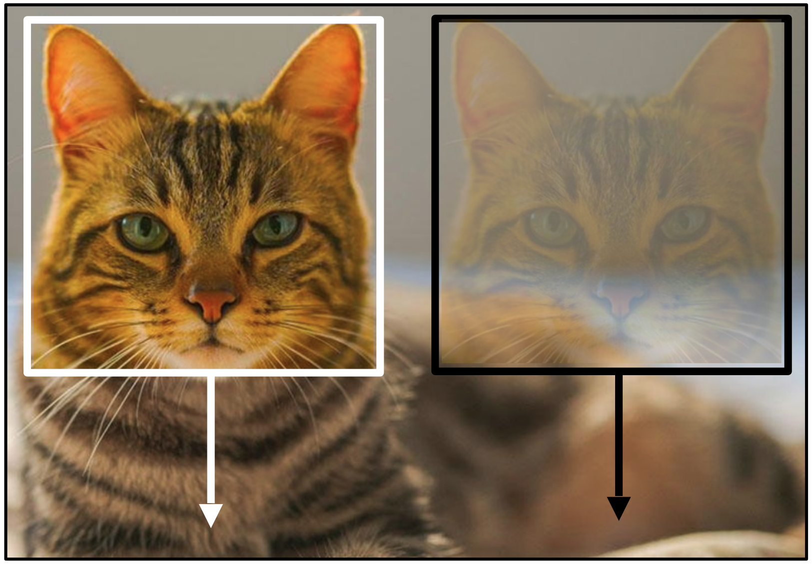
Structure counts

- What if the cat moved?



- Present **entirely different** input to the DNN
- This complexity cannot be captured by as little as 8 nodes
  - Lack of expressivity
- Solution: exploit the **structure** of the data

Introducing filters



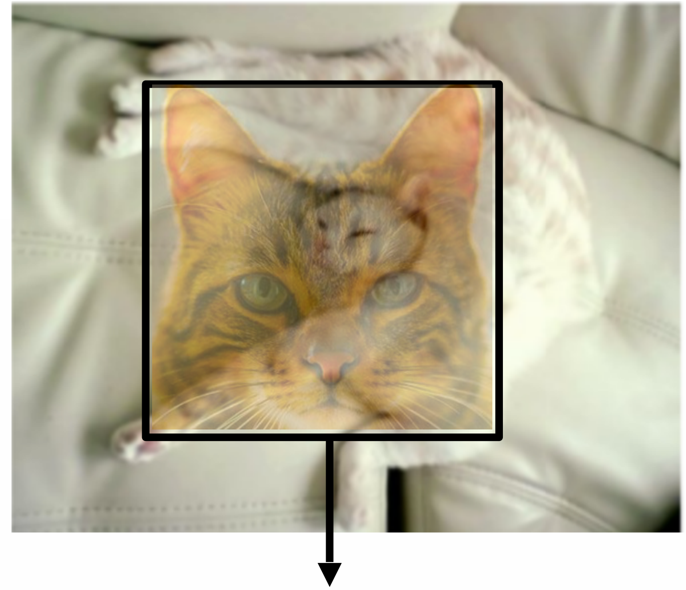
Very cat-like:
Score = 1

Not at all cat-like
Score = 0

- Create a cat-face filter (no ML here)

- Slide it over the image
- Take maximum of all cat scores: image cat score
- We found the cat

Cats come in different shapes

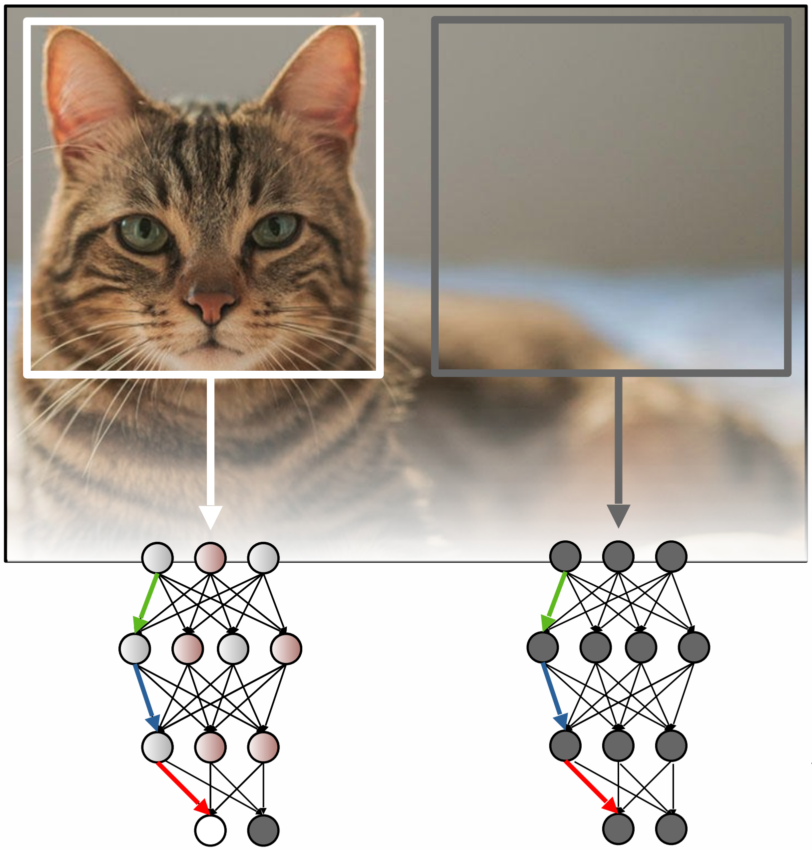


Not a cat

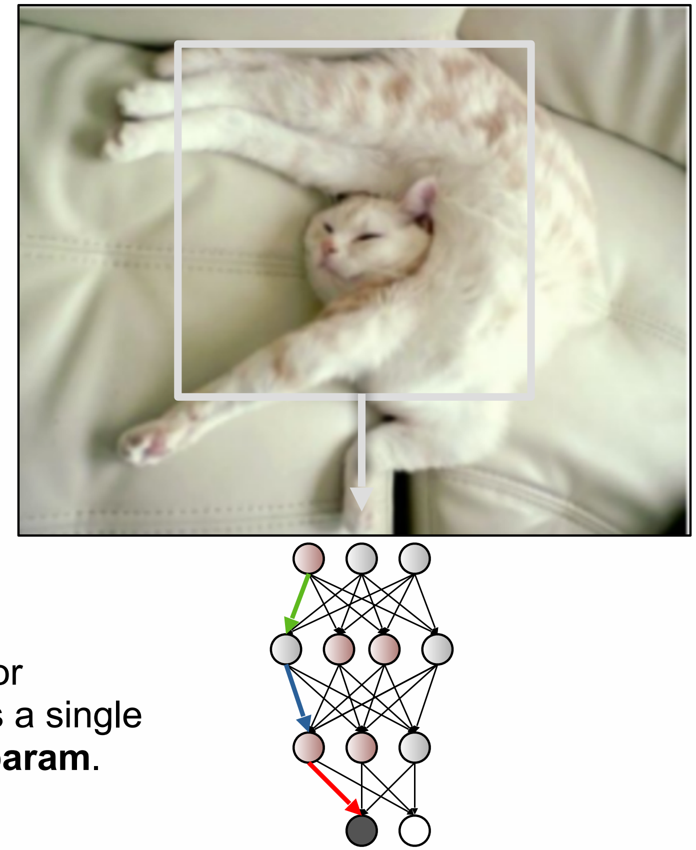
Not a cat

- Many different very complex filters are needed
- Can be solved by
    - **Learning filters from examples**
    - Abstraction

Learning the filters

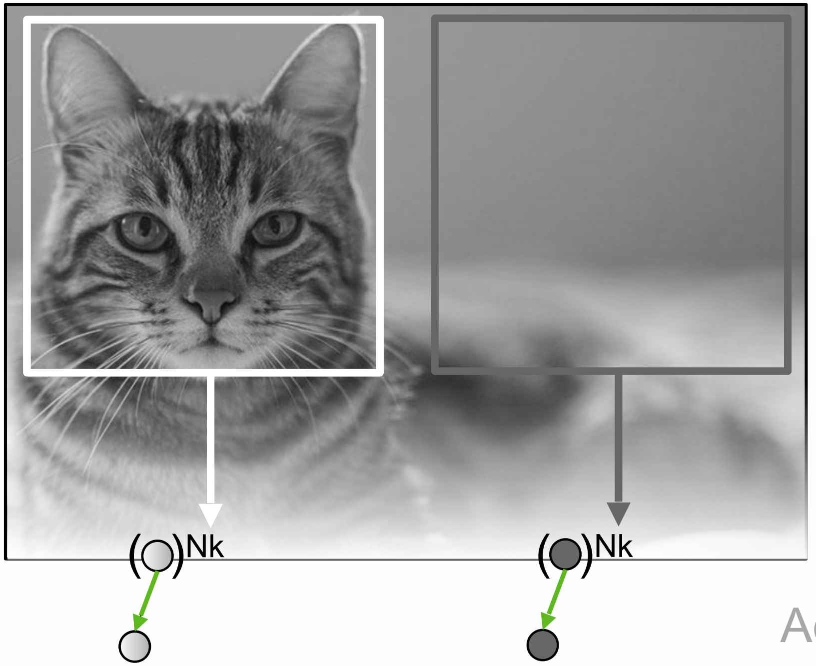


Each color highlights a single **shared param.**



- Learn (approximations of) different shapes
- Represent them by (combinations of) output nodes

A CNN kernel: step by step



- Inputs x
- For one *channel*:

Learnable bias

Kernel size

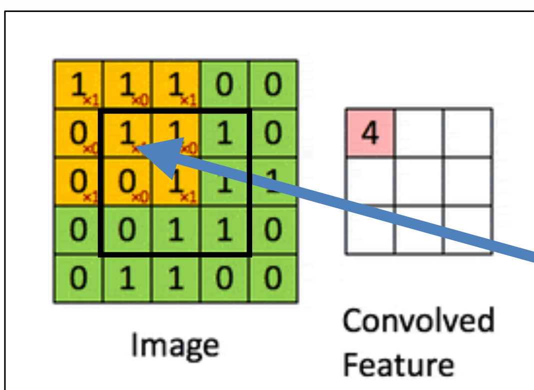
$$y_j = \theta\left(\sum_{i}^{N_k} \omega_i \; x_{I(j,i)} - T\right)$$

Activation function

Learnable weights:

Relative position to j

Index m of the pixel on the i-th place in the neighbourhood of j



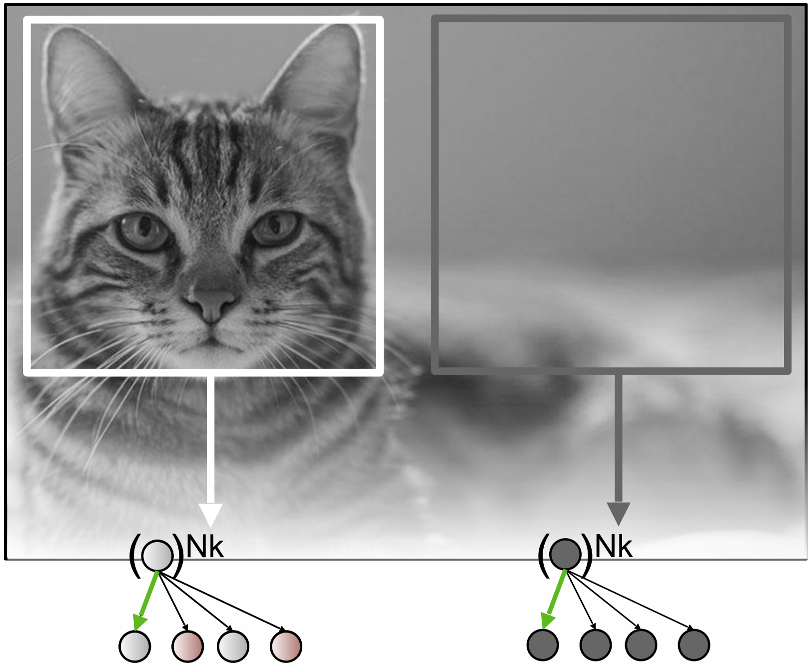
I(7,i) = { +2 for full row, +2 for full row

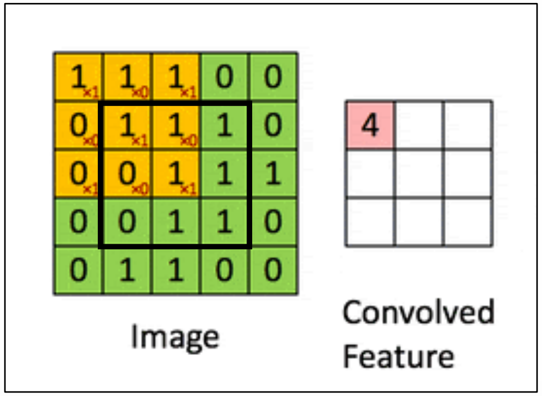
1	2	3
6	7	8
11	12	13

}

conditions at the edges → wait a few slides

Multiple output channels



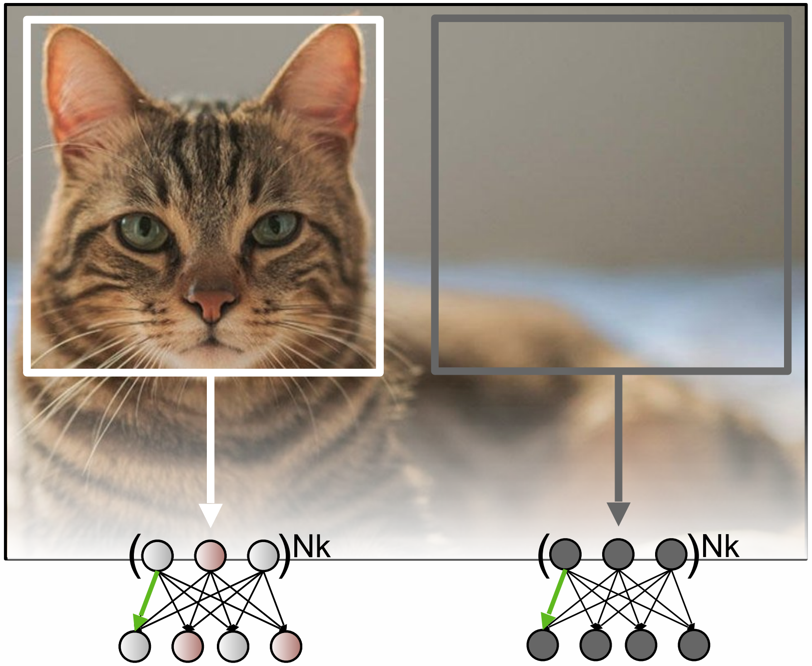


- Inputs x
- For N_c output channels (α)

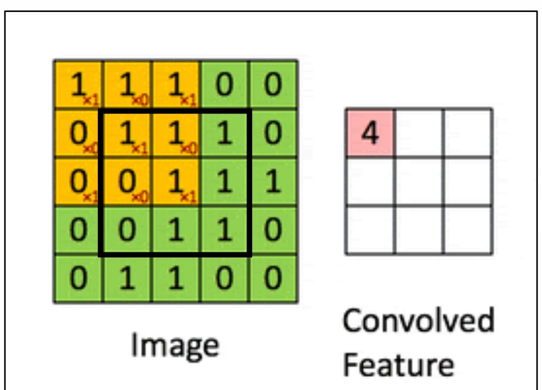
$$y_{j\alpha} = \theta\left(\sum_{i}^{N_k} \omega_{i\alpha}\, x_{I(j,i)} - T_{\alpha}\right)$$

The weights are still shared and depend only on **relative position** w.r.t. pixel j (and α)

Multiple input channels



One *kernel* ≙ one dense MLP layer



- Inputs x

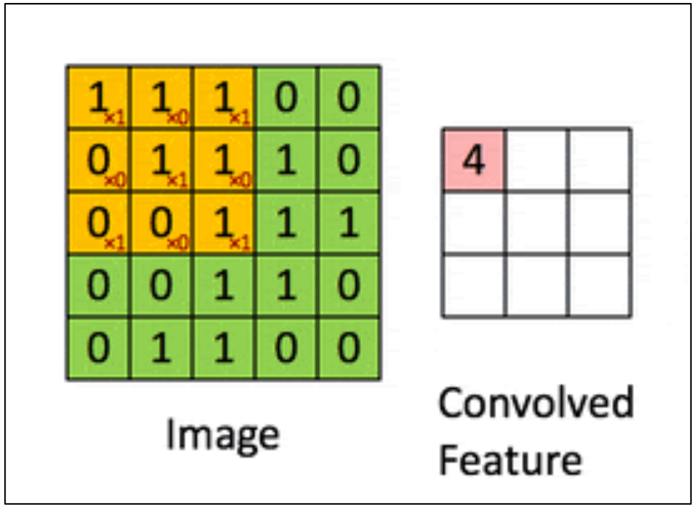
- For N_F input channels/features

$$y_{j\alpha} = \theta\left(\sum_{\beta}^{N_F} \sum_{i}^{N_k} \omega_{i\alpha\beta}\; x_{I(j,i)\beta} - T_\alpha \right)$$

Still strictly relative

- This is a complete convolutional layer

Example

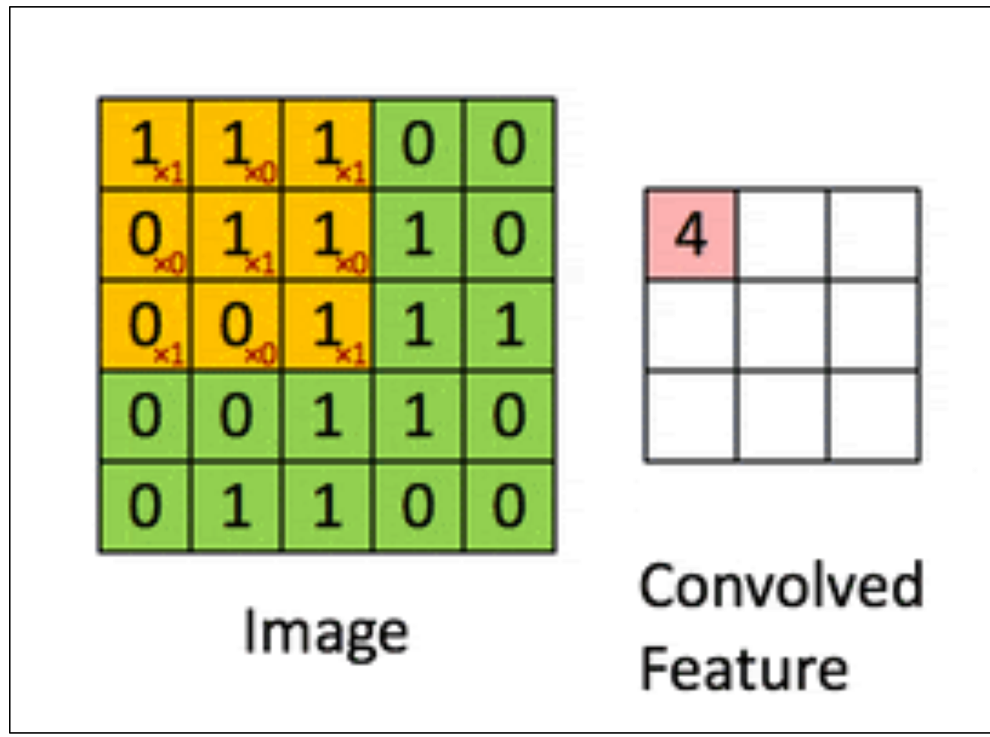


Kernel

1	0	1
0	1	0
1	0	1

- No activation
- No bias
- One input
- One output

$$y_j = \sum_{i}^{N_k} \omega_i \; x_{I(j,i)}$$



$$y_{j\alpha} = \theta\left(\sum_{\beta}^{N_F} \sum_{i}^{N_k} \omega_{i\alpha\beta}\ x_{I(j,i)\beta} - T_\alpha\right)$$

Parameters

Filter

Time for some (more) questions

Neighbourhood

Channels

Kernel

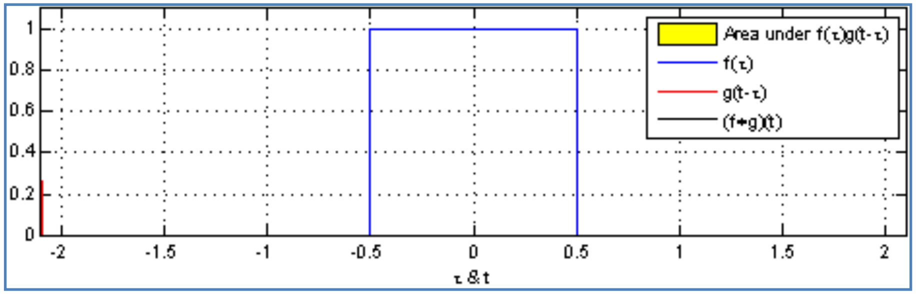
Bias

Longer side note: where is the convolution?

- CNN:

$$y_j = \sum_{i}^{N_k} \omega_i \; x_{I(j,i)}$$

'n-m' hidden here



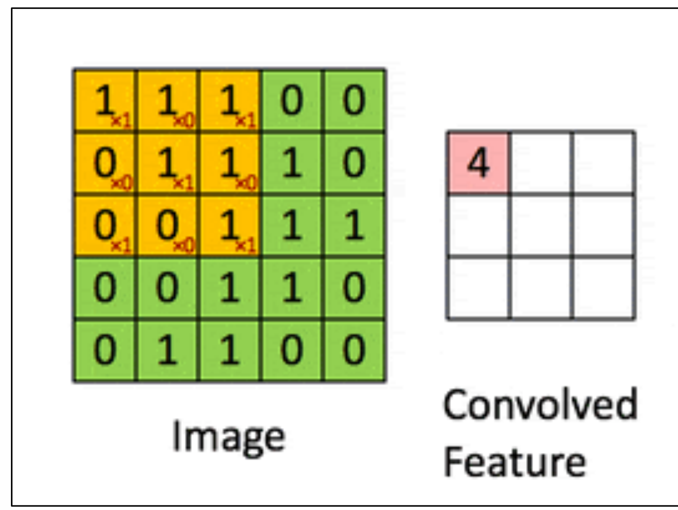
https://en.wikipedia.org/wiki/Convolution [accessed 13.7.23]

- Convolution:

$$(f * g)(t) = \int_{-\infty}^{+\infty} f(\tau)g(t-\tau)d\tau$$

- Discrete:

$$(f * g)[n] = \sum_{m=-\infty}^{+\infty} f[m]g[n-m]$$



Re-shuffle symbols

$$(f * g)[n] = \sum_{m=-\infty}^{+\infty} f[m]g[n-m]$$

$$y_j = \sum_{i}^{N_k} \omega_i \; x_{m=I(j,i)}$$

Index m of the pixel on the i-th place in the neighbourhood of j

Switch perspective

Pixels in image

$$= \sum_{m=1}^{N_p} \omega_{i=I^{-1}(j,m)} x_m$$

The i-th place for a pixel with index m in the neighbourhood of j

If not in neighbourhood: extend kernel such that $\omega = 0$

$$\begin{matrix} & & \vdots & & & \\ & 0 & 0 & 0 & 0 & 0 & \\ & 0 & 1 & 0 & 1 & 0 & \\ \cdots & 0 & 0 & 1 & 0 & 0 & \cdots \\ & 0 & 1 & 0 & 1 & 0 & \\ & 0 & 0 & 0 & 0 & 0 & \\ & & & \vdots & & & \end{matrix}$$

$$y_j = \sum_{m=1}^{N_p} x[m] \; \omega_{i=I^{-1}(j,m)}$$

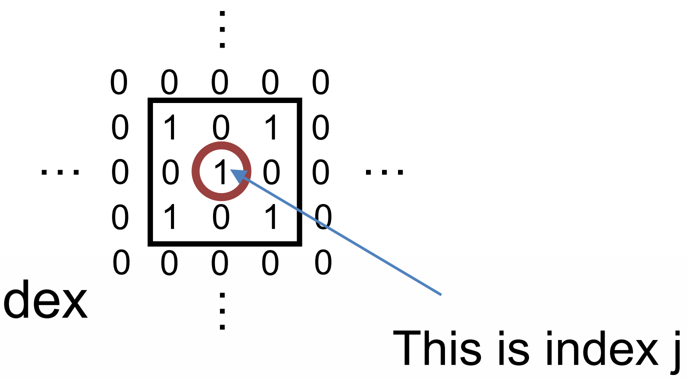
Simple replacement as $x[m] = x_m$

It is a convolution

$$(f * g)[n] = \sum_{m=-\infty}^{+\infty} f[m]g[n-m]$$

$$y_j = \sum_{m=1}^{N_p} x[m]\ \omega_{I^{-1}(j,m)}$$

$I^{-1}(j, m)$ can be rephrased as a distance index



Define $\tilde{\omega}[j-m] = \omega_{I^{-1}(j,m)}$ *

$$y_j = \sum_{m=1}^{N_p} x[m]\ \tilde{\omega}[j-m] \quad \leftrightarrow \quad (f * g)[n] = \sum_{m=-\infty}^{+\infty} f[m]g[n-m]$$

- **A convolutional neural network layer is indeed equivalent to a convolution**

* technically, depending on the definition, this could implement a convolution or cross correlation, possibly implementing a sign flip w.r.t. convolution. In practice this does not matter since ω_i are learnable and can re-absorb the flip. A detailed explanation can be found here: https://ai.stackexchange.com/questions/21999/do-convolutional-neural-networks-perform-convolution-or-cross-correlation

Translational equivariance as direct consequence

The convolution commutes with translations, meaning that

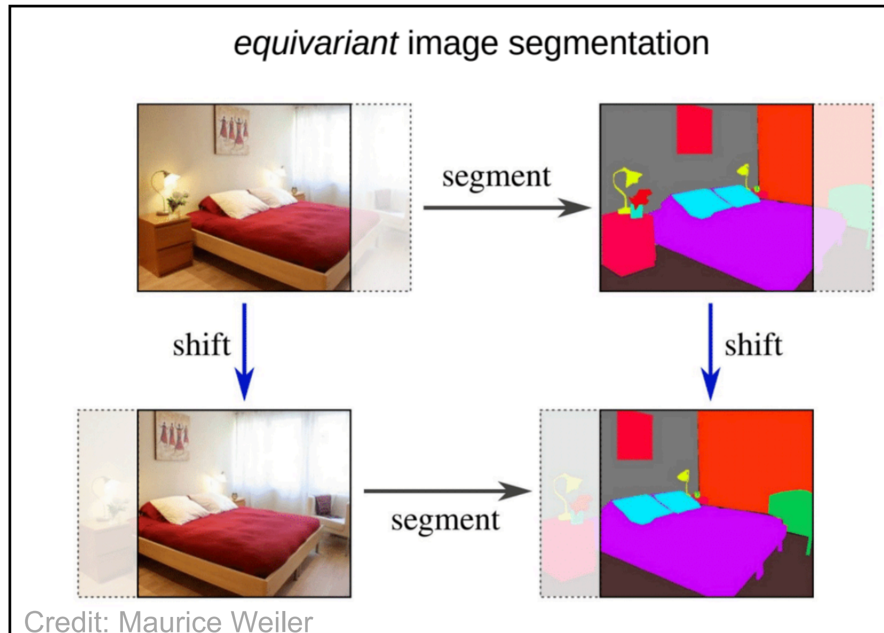
$$\tau_x(f * g) = (\tau_x f) * g = f * (\tau_x g)$$

where $\tau_x f$ is the translation of the function f by x defined by

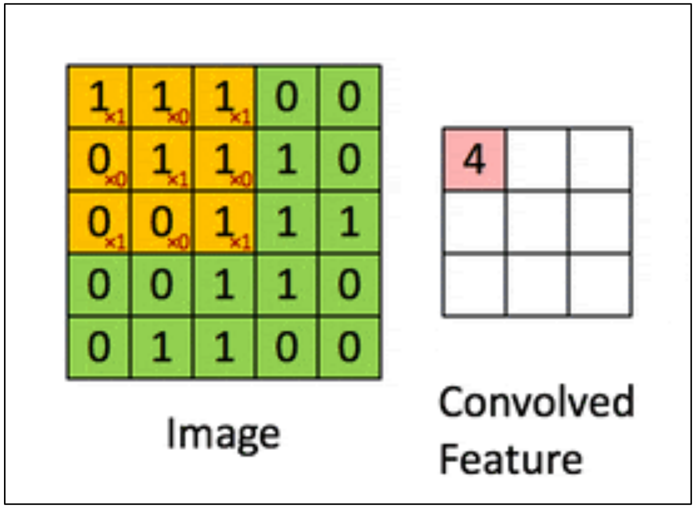
$$(\tau_x f)(y) = f(y - x).$$

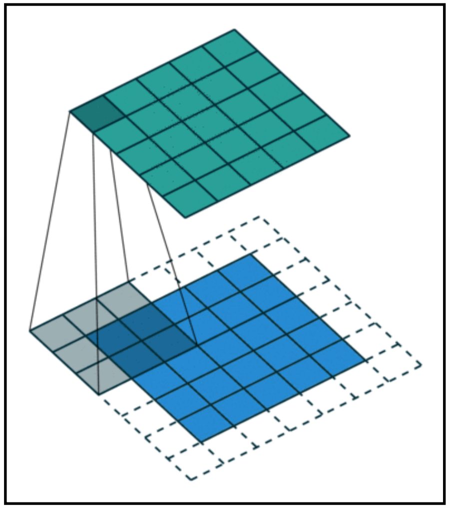
https://en.wikipedia.org/wiki/Convolution

- Convolutions and translation commute
- Shift + convolution is the same as convolution + shift
- This is referred to translation **equivariance** (not invariance)

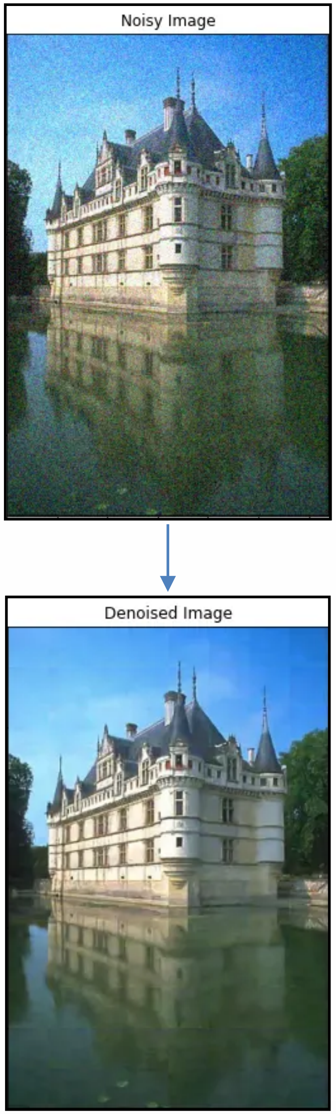


Conditions at the edges





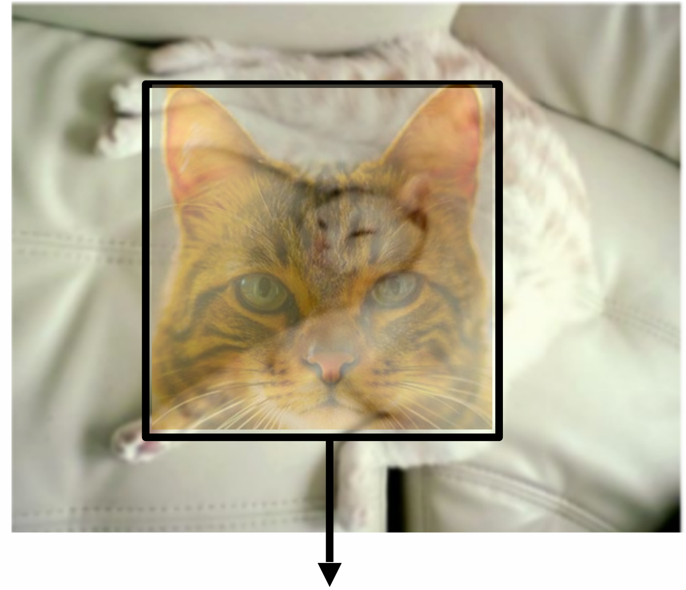
arxiv:1603.07285



- For a 3 x 3 kernel, the image size will be reduced by 2 pixels on top and bottom
  - For a 5 x 5 kernel?
- If this is not desired (zero) padding the image can help

https://medium.com/analytics-vidhya/noise-removal-in-images-using-deep-learning-models-3972544372d2

Cats (still) come in different shapes

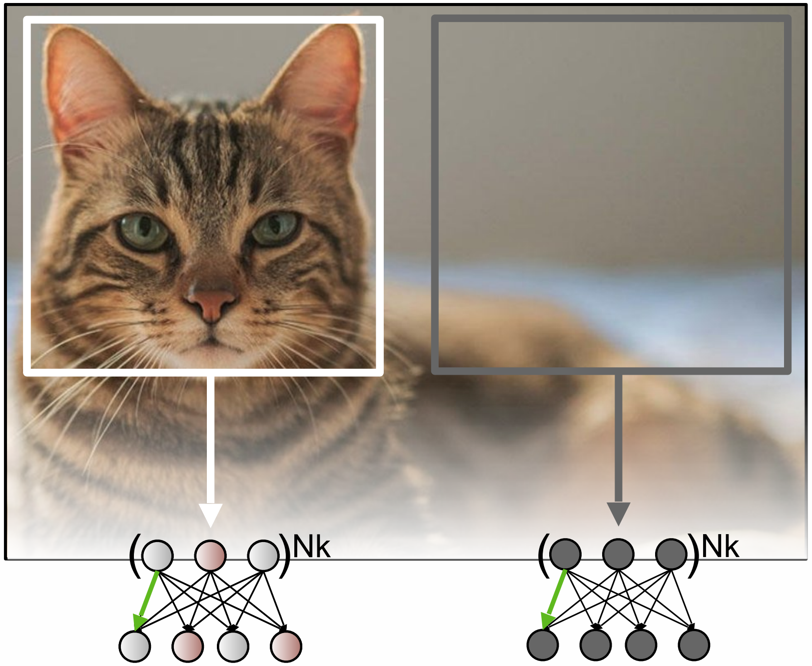


Not a cat

Not a cat

- Many different very complex filters are needed
- Can be solved by
  - Learning filters from examples ✓
  - **Abstraction**

Breaking up the problem into smaller parts



One *kernel* ≙ one dense MLP layer

$$y_{j\alpha} = \theta\left(\sum_{\beta}^{N_F}\sum_{i}^{N_k}\omega_{i\alpha\beta}\,x_{I(j,i)\beta} - T_\alpha\right)$$

- This is one complete convolutional layer with $\alpha \in \{1, \dots, N_C\}$

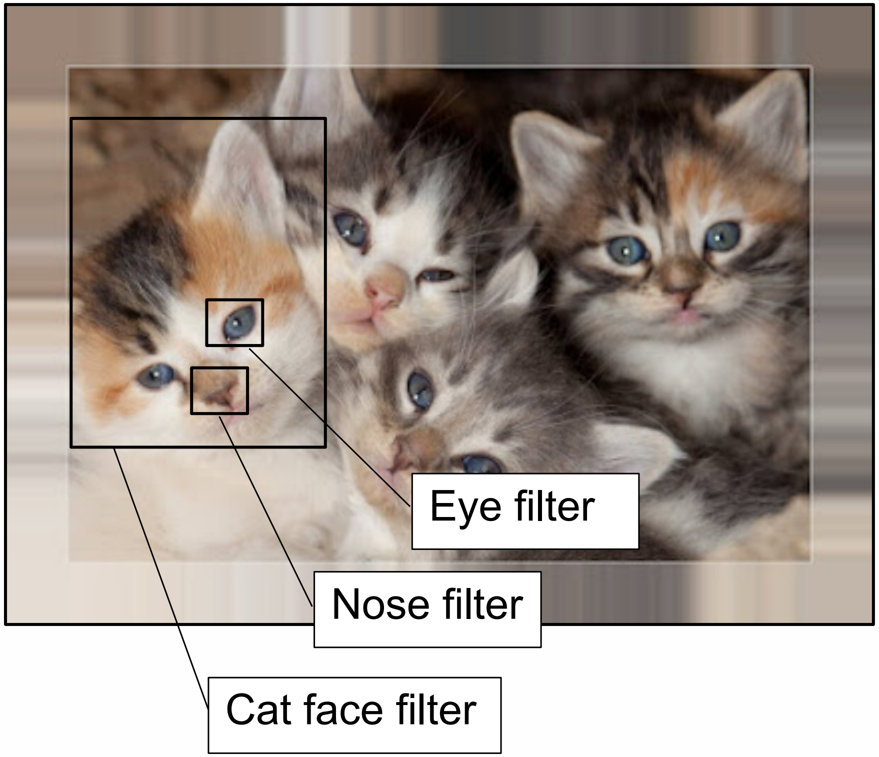
- Counting weights:
  how many do we have?

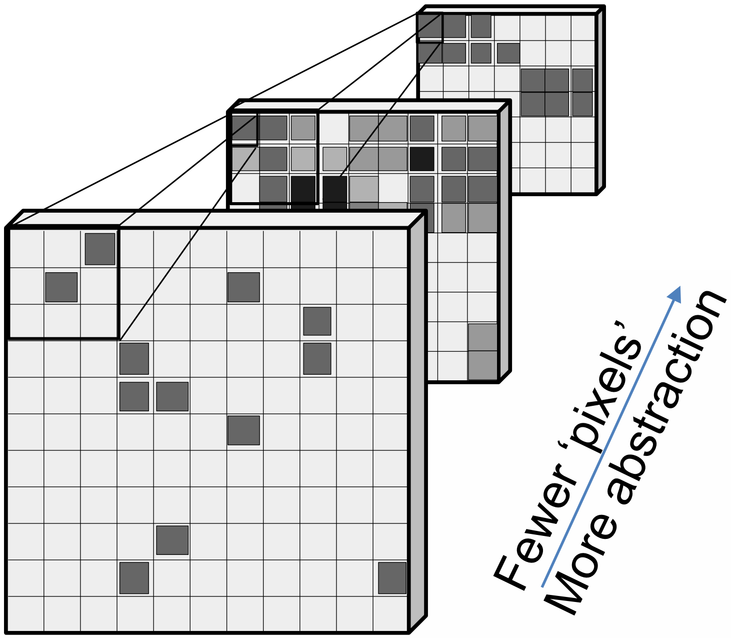
$$N_C \cdot N_F \cdot N_k$$

- With $N_k \approx H \otimes W$, kernels must not be too big
- Smaller kernels cannot capture a whole cat

- Break down problem: abstraction and pooling

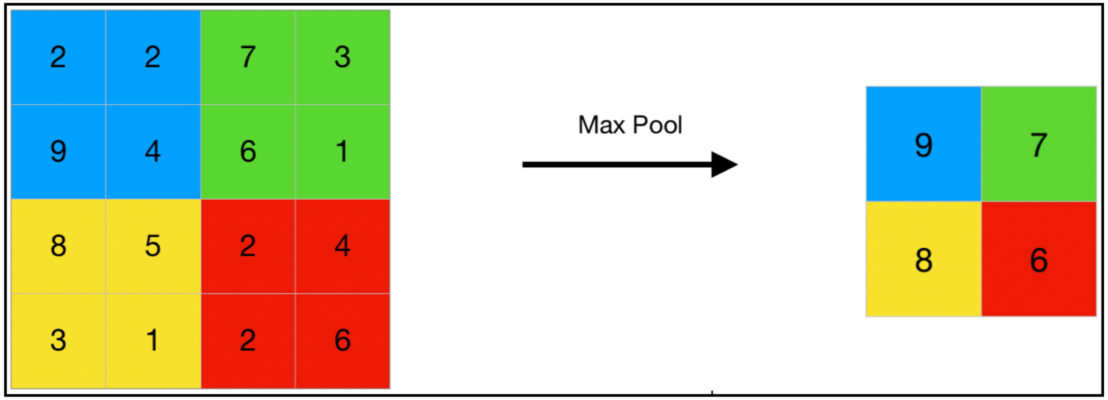
Abstraction and pooling



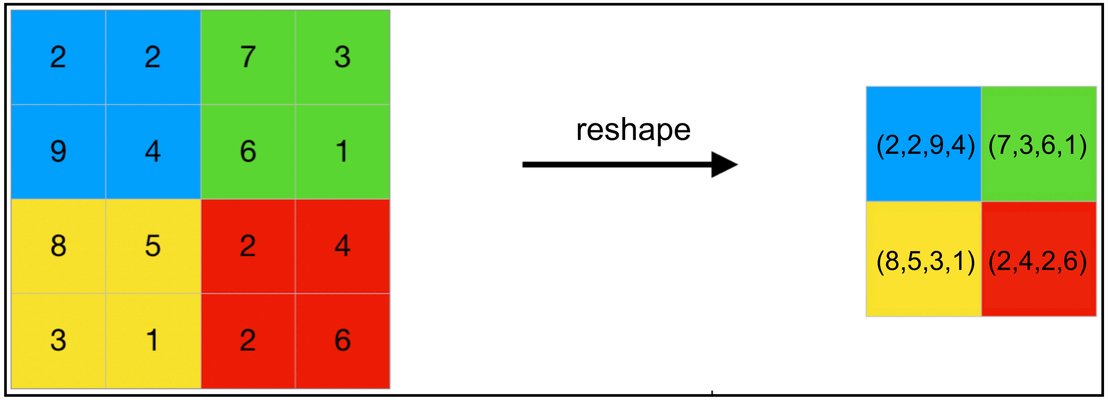


- Use smaller kernels to capture individual features
- Summarise (pool) the filter outputs of several neighbouring pixels
  - Take maximum (max pooling)
  - Take average/sum (average pooling)
  - Reshape tensor
- Go in bigger steps 'skipping' pixels: strides

Pooling

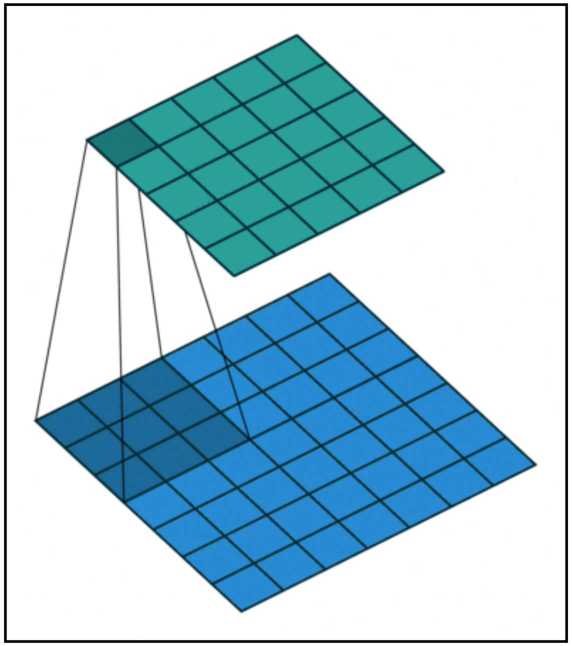


https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/

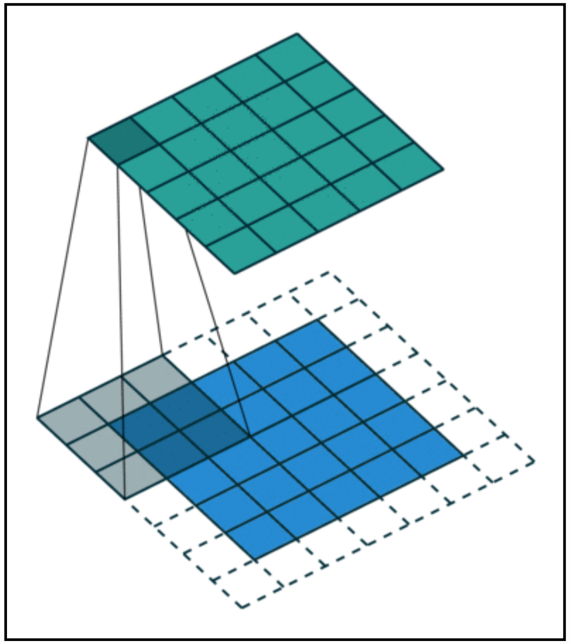


- Max pooling: which filter has triggered the largest output?
  - Is this more of an eye or a nose in that patch
- Reshaping: re-organise the information without removal of information
  - Not used so much, in particular for classification Why?

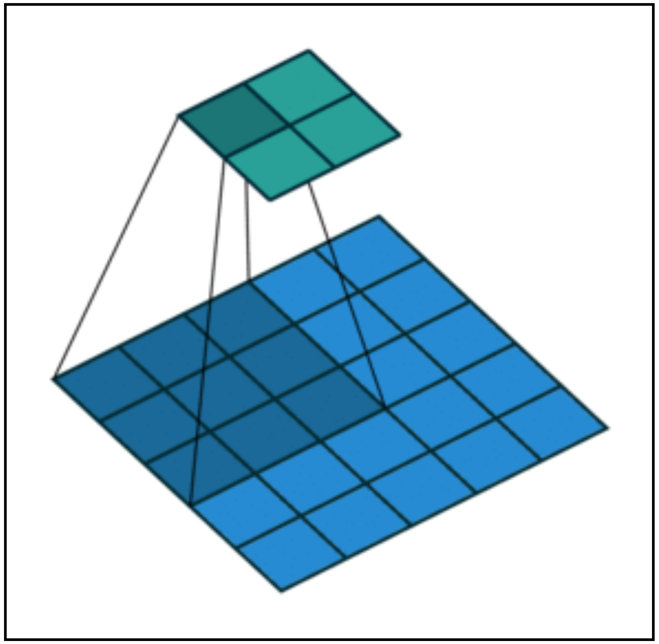
Strides



Stride 1, no padding



Stride 1, padding

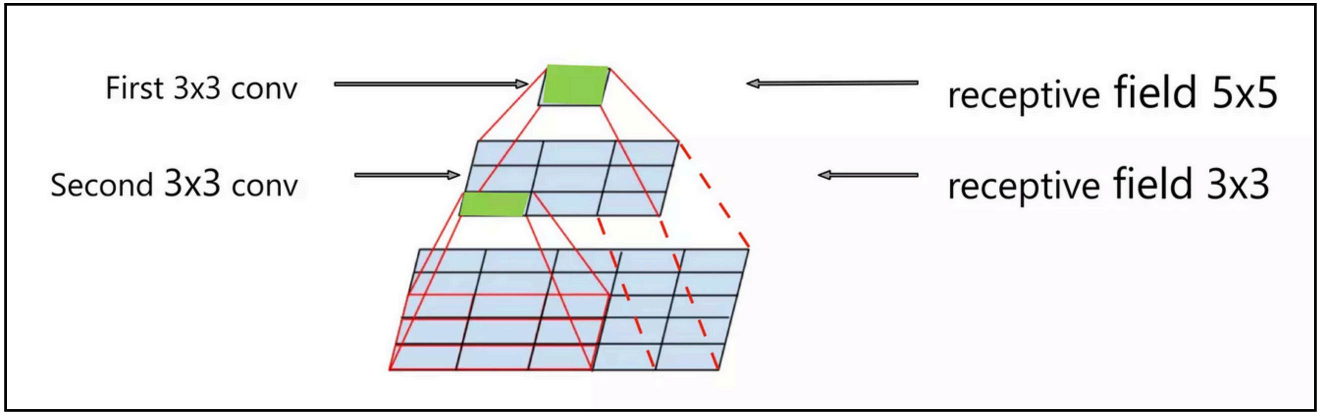


Stride 2

arxiv:1603.07285

- The stride is the amount the filter 'moves' at each step

The notion of the receptive field



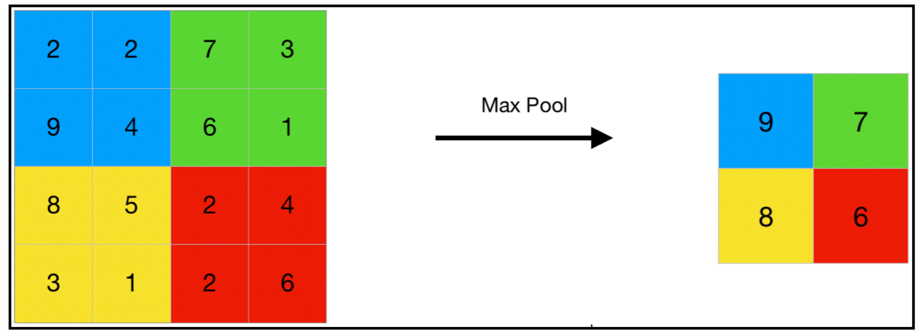
- For a given pixel, from how far away could it have accumulated information
- Central concept when designing neural networks in general
- Easily accessible for CNNs
- Needs to be big enough to capture the object

Our CNN toolbox

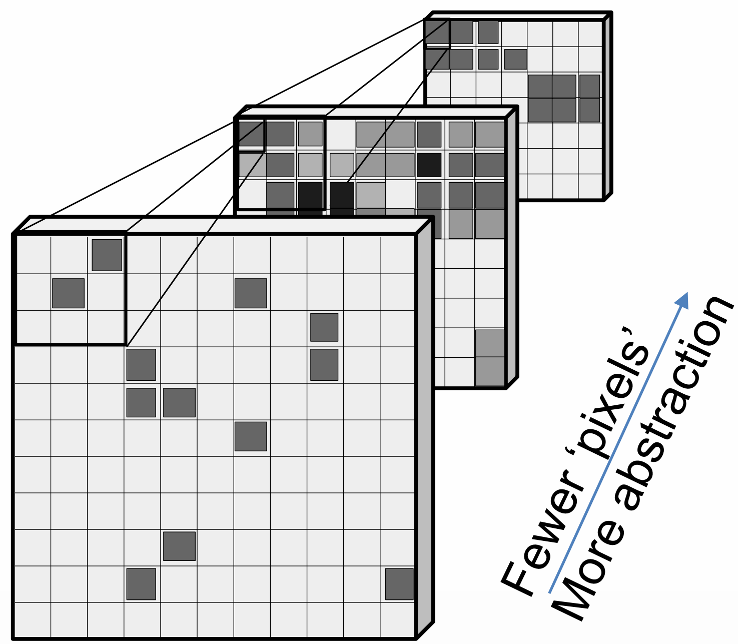
- CNN kernel
    - Learns filters

$$y_{j\alpha} = \theta\left(\sum_{\beta}^{N_F} \sum_{i}^{N_k} \omega_{i\alpha\beta}\, x_{I(j,i)\beta} - T_{\alpha} \right)$$

- Strides + Pooling
    - Build summaries

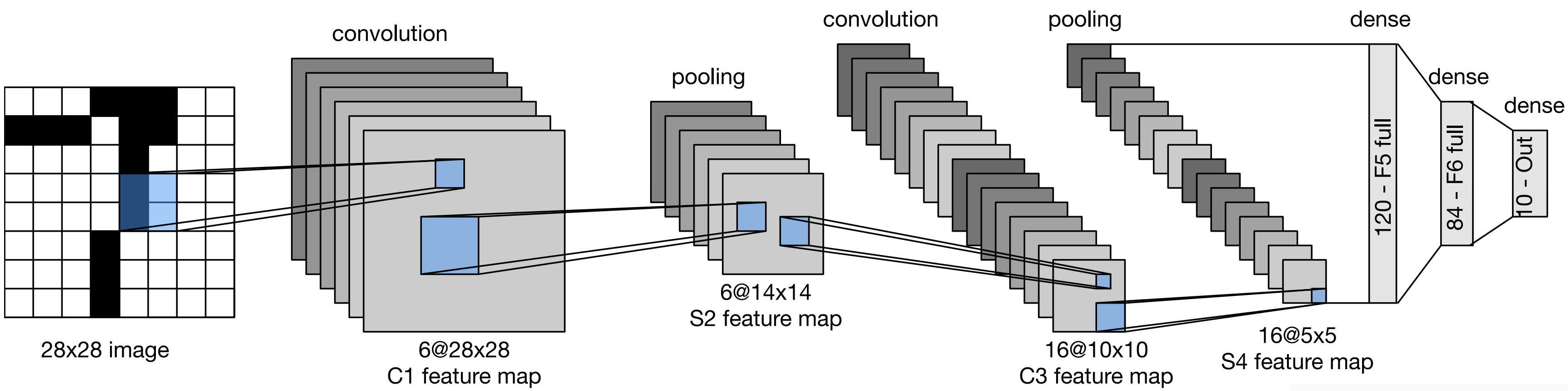


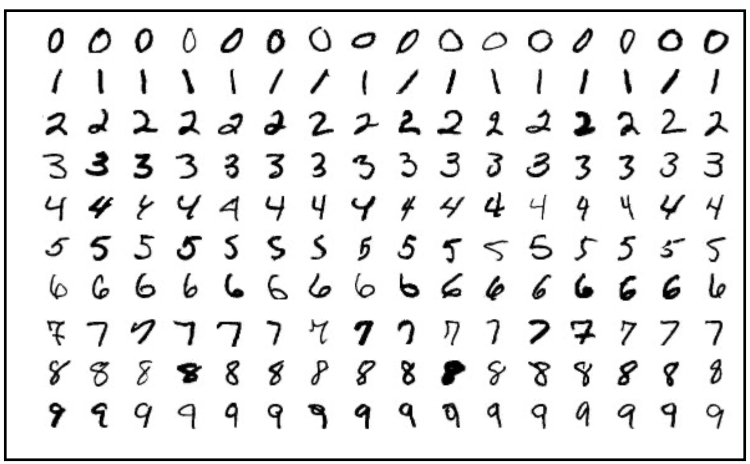
- Stack CNN layers
    - Abstraction



Example: LeNet (1998)

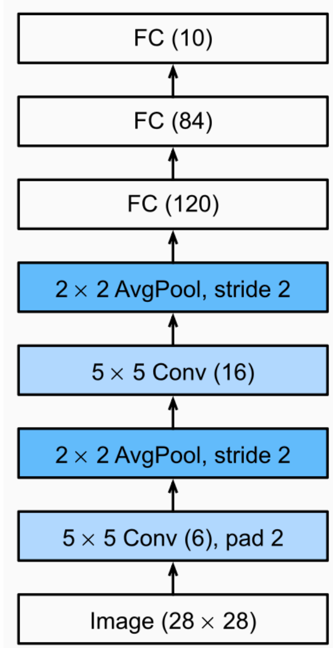
LeCun et al, Proceedings of the IEEE, 1998



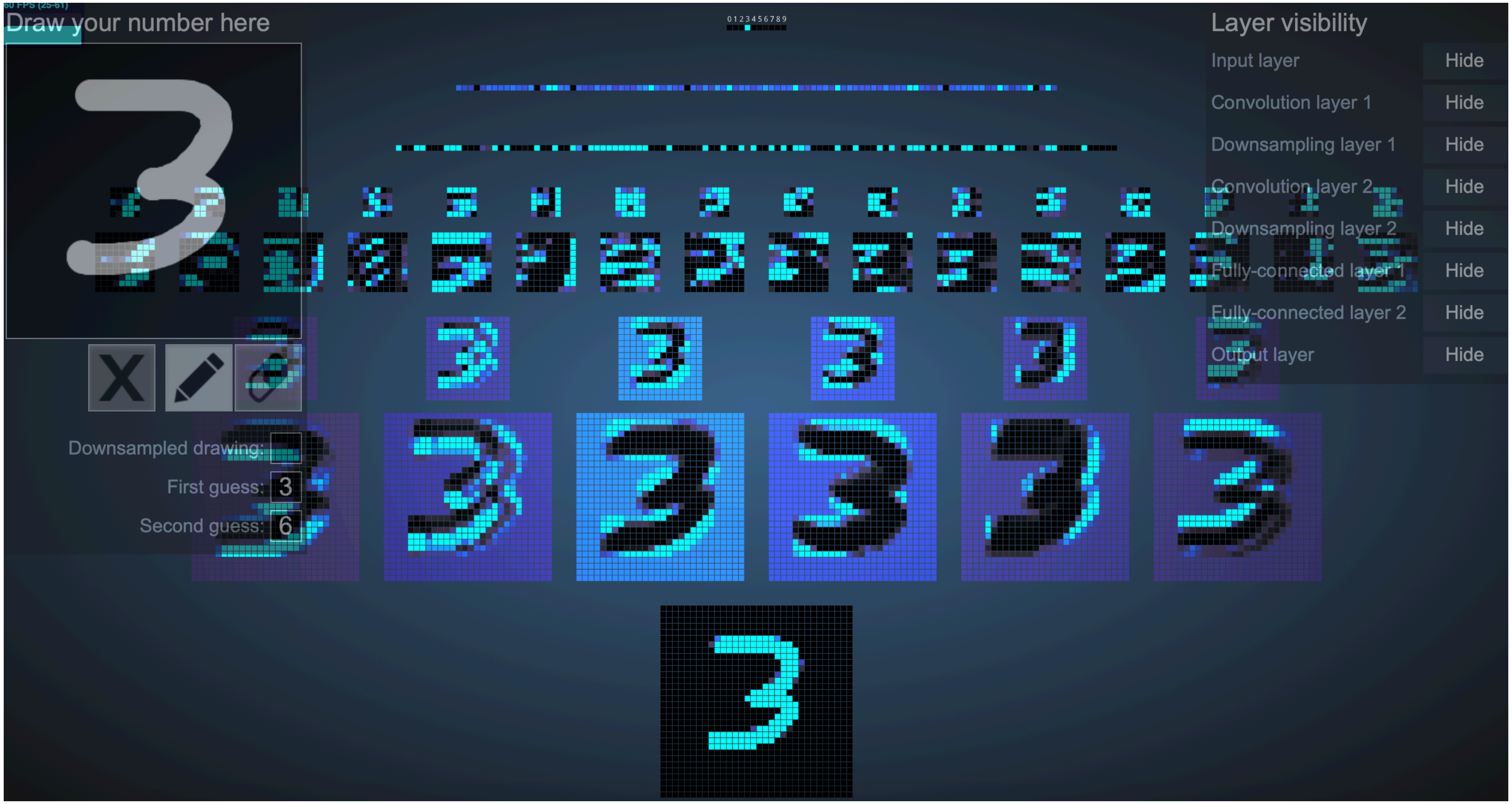


MNIST dataset

- Very early CNN ("the" CNN)
- Shows typical features of also modern classification CNNs: (pooling, pixel dims → feature dims, …)



Unboxing: we can directly visualise the filters



Try yourself: → https://adamharley.com/nn_vis/cnn/2d.html

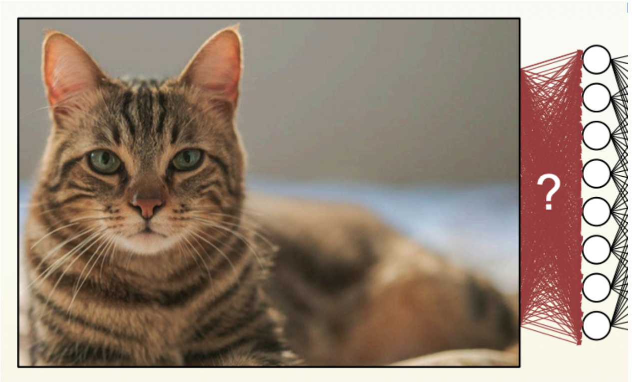
A. W. Harley, "An Interactive Node-Link Visualization of Convolutional Neural Networks," in ISVC, pages 867-877, 2015

CNNs are very powerful: fewer parameters

- In general the following statements hold:
  - The more TPs the higher the risk to overtrain.
  - The larger the training dataset the smaller the risk to overtrain.
  - It is therefore also always possible to reduce the risk of overtraining by increasing the training dataset.

- CNNs break down the large number of input pixels with a **much** smaller number of parameters

- Abstraction and pooling maintain expressivity



CNNs are very powerful: effective training sample

- In general the following statements hold:
  - The more TPs the higher the risk to overtrain.
  - The larger the training dataset the smaller the risk to overtrain.
  - It is therefore also always possible to reduce the risk of overtraining by increasing the training dataset.

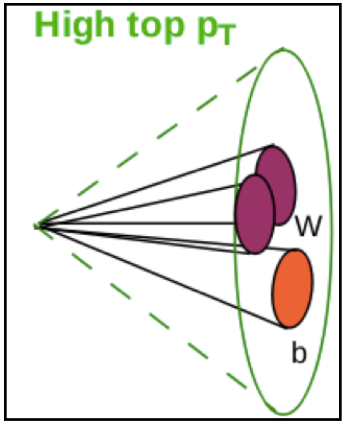
- The filter weights are shared for all j

$$y_{j\alpha} = \theta\left(\sum_{\beta}^{N_F} \sum_{i}^{N_k} \omega_{i\alpha\beta}\, x_{I(j,i)\beta} - T_\alpha\right)$$

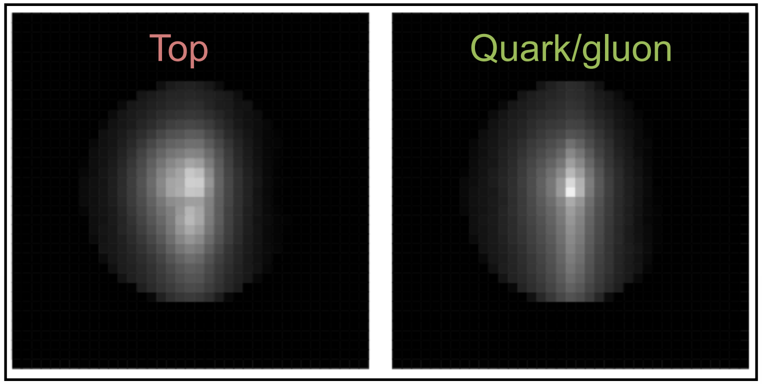
- They are trained for **every** y_j **:**
  - ω 'see' (sample size * number of pixels) training examples → Millions

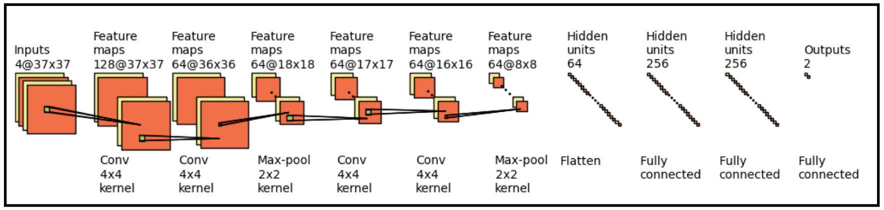
- There are (almost) always **multiple** benefits from using the structure of the data

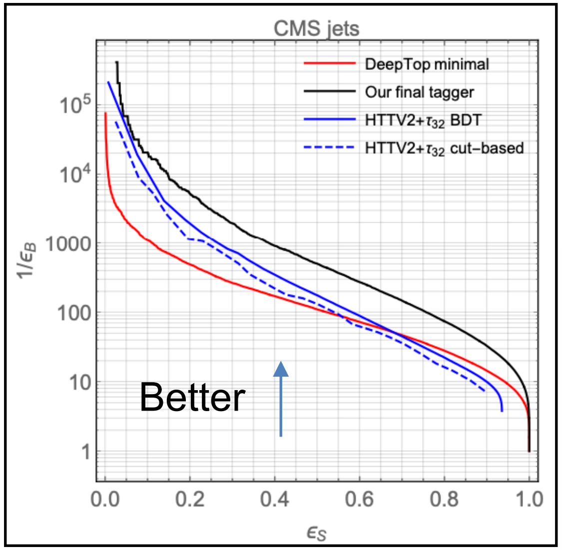
Physics examples: jet tagging



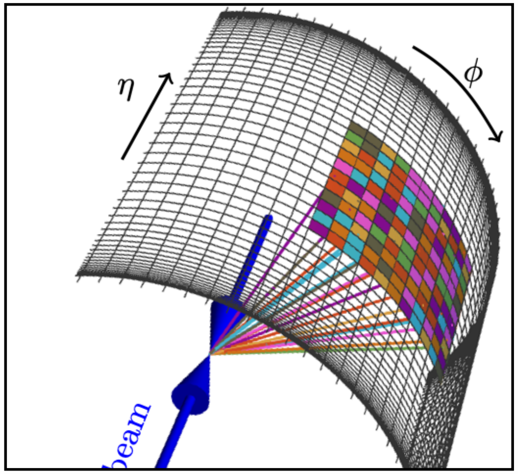
arxiv:1803.00107
(and many others)



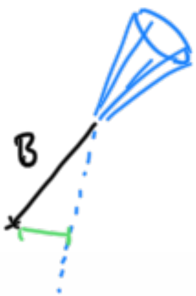


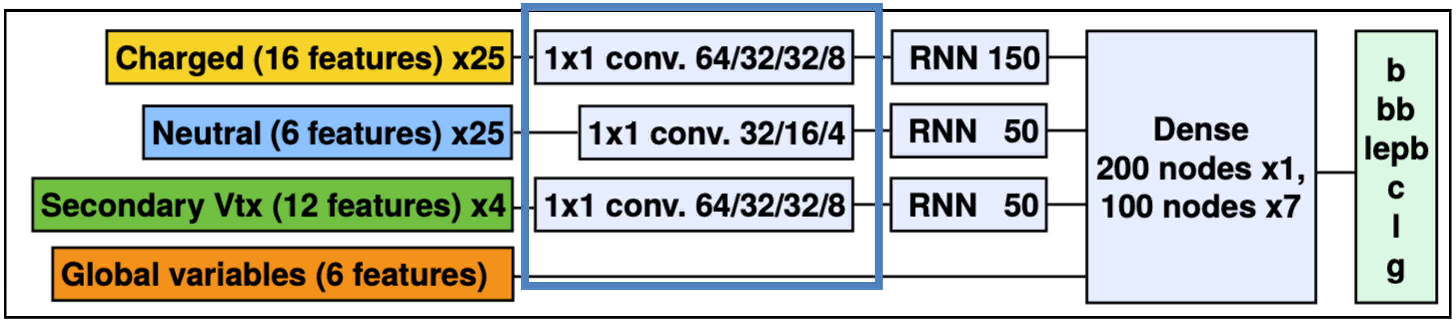


- Identifying origin of a jet very useful for many analyses
- Treat the jet deposits (e.g. in the calorimeter) as an image
- Performance gain over high-level variables

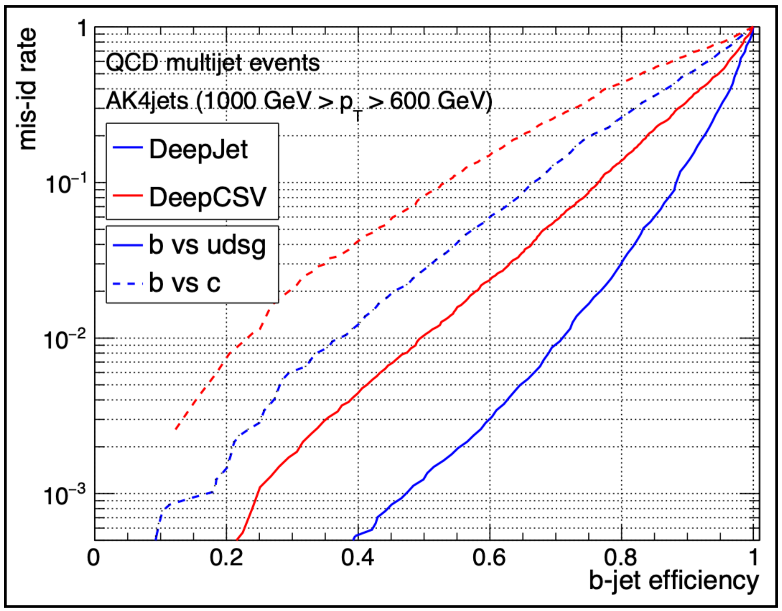


Structure matters: CNNs are not just for images





- Interpret all reconstructed **particles** in the jet as individual ‘pixels’ in a 1D image
- Pre-process using 1D ‘CNNs’
  - **Translation equivariance → particle equivariance**
  - Enabled to use **all** jet constituents for the first time
  - Enormous performance gain in particular at high momentum
- Standard tagger in CMS
  - >>100 analyses



arxiv:2008.10519

- **Gain ≈ up to decades more data taking for some analyses!**

Summary

- Feed-forward NN can be powerful classifiers and regressors
- With great power comes great responsibility
  understand the inputs and their correlations and beware of out-of-distribution effects
- Understanding and utilising the structure of the data is key for advanced tasks
- CNN architectures combine
  - translation equivariant feature detection
  - abstraction and pooling of information

BACKUP

What about uncertainties on NN?

- Some terminology from Machine Learning

Machine learning

Aleatoric uncertainties	**Epistemic** uncertainties
Inherent due to statistical variance	Due to insufficient knowledge of laboratory conditions
≈	≈
Statistical uncertainties	**Systematic** uncertainties

Physics

- This is a hot topic in machine learning

Aleatoric uncertainties

- Reminder: a DNN training consists of
  dataset + architecture + loss function + minimisation

Where are statistical processes
in the MLP training?

- Random initialisation of weights and biases
- Random choice of mini batches
- Stochastic minimisation procedures
- Random distinction of training, (test), and validation sample

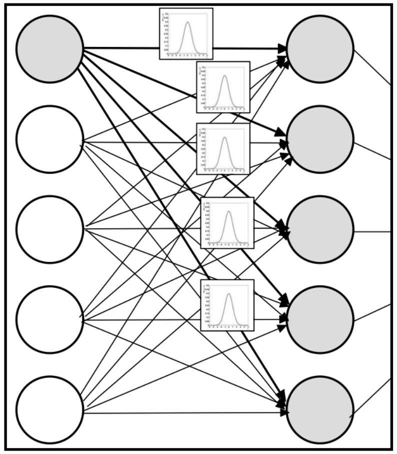
- The whole sample is sampled from the ground truth

Estimation of aleatoric uncertainties: some teasers

Deep Ensembles

- Initialise identical NNs with varying random seeds and check the distribution of outcomes
- Obvious frequentist approach

Bayesian methods



$$\omega \rightarrow p(\omega \,|\, \hat{y}(x))$$

- Learns probability distribution over *possible* neural networks
- Won't be covered here
- Resources and tutorial e.g. [arxiv:2007.06823]

Dropout

- Next slide

arXiv:1506.02142, >6k citations

Dropout to estimate uncertainty

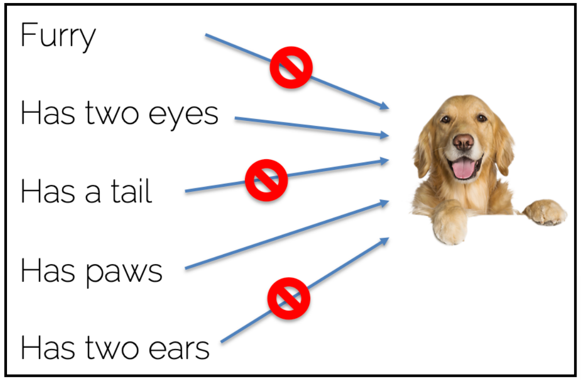
- Full proof too much for this lecture

- Dropout during training time forces the network to create redundant representations

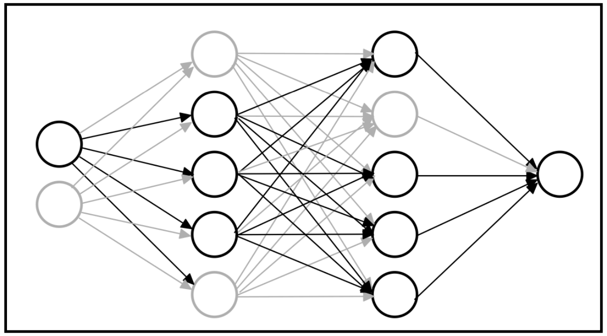
- Dropout during inference/test time (MC) samples from these redundant (but all different!) representations

- If dropout is placed before **every** MLP layer in the DNN, this sampling approximates a Bayesian FF NN → uncertainties can be estimated

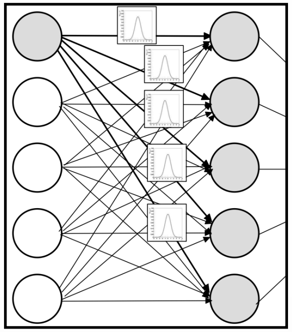
- Powerful and **easy to use** tool
- Can also cover epistemic uncertainties



Sample



≈ arXiv:1506.02142



Epistemic uncertainties

- The model does not have enough degrees of freedom to map the ground truth
  → underfitting

- The model systematically maps specific, non-general properties of the training sample
  → overfitting

- Differences between training and test sample
  → bias

- Much as systematic uncertainties, epistemic uncertainties can be reduced on the basis of additional information