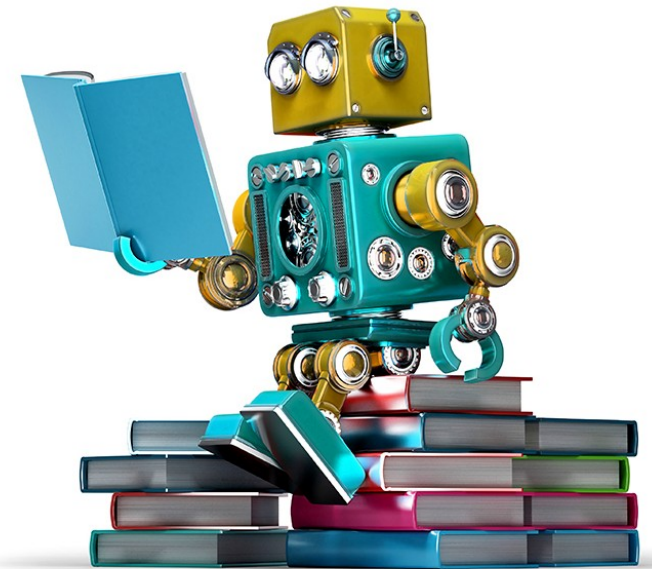


Machine Learning for Particle Physics – A Gentle, Hands-On Introduction

Wolfgang Waltenberger

The 28th Vietnam School of Physics,
July 25 – August 5, 2022



Introduction

- This introductory lecture series aims at giving a very **practical introduction to novel machine learning** techniques.
- The examples will be in **pytorch** and python. Please work in groups of two or three. People familiar with the subject should team up with people who are new to the topic. Skeleton code will help you get started.
- There will be four **exercises**, the last one being the “**challenge**” with data from a dark matter experiment. The **winning** team gets a **small prize**.

Syllabus

- Introduction
- Lesson 1: Logistic Regression
- [Exercise 1](#)
- Lesson 2: Multi-Layer Perceptron
- [Exercise 2](#)
- Lesson 3: Convolutional Neural Networks
- [Exercise 3](#)
- Lesson 4: Generative Models
- Lesson 5: Beyond

recurrent/recursive neural networks, explanation methods, uncertainty of network

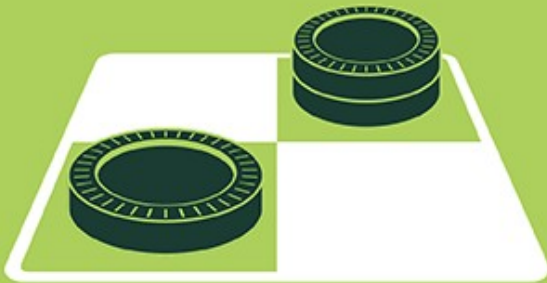
- **Challenge: learn discriminator for dark matter direct detection experiment**

Introduction

in which we mostly concern ourselves with
naming conventions and problem statements

ARTIFICIAL INTELLIGENCE

Early artificial intelligence stirs excitement.



MACHINE LEARNING

Machine learning begins to flourish.



DEEP LEARNING

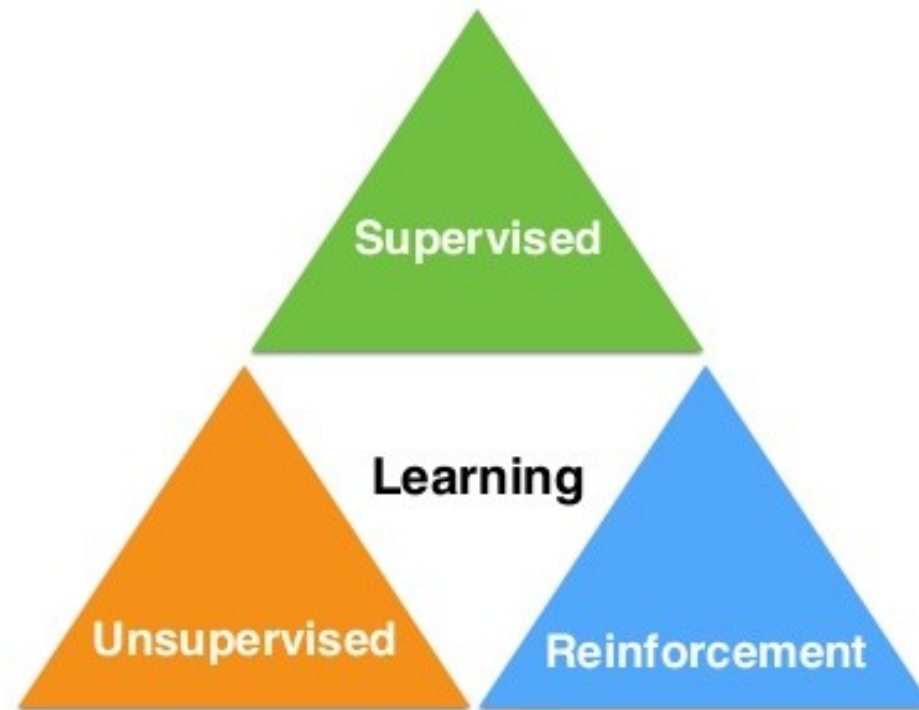
Deep learning breakthroughs drive AI boom.



Since an early flush of optimism in the 1950s, smaller subsets of artificial intelligence – first machine learning, then deep learning, a subset of machine learning – have created ever larger disruptions.

this course will be an introduction to the rightmost box – deep learning

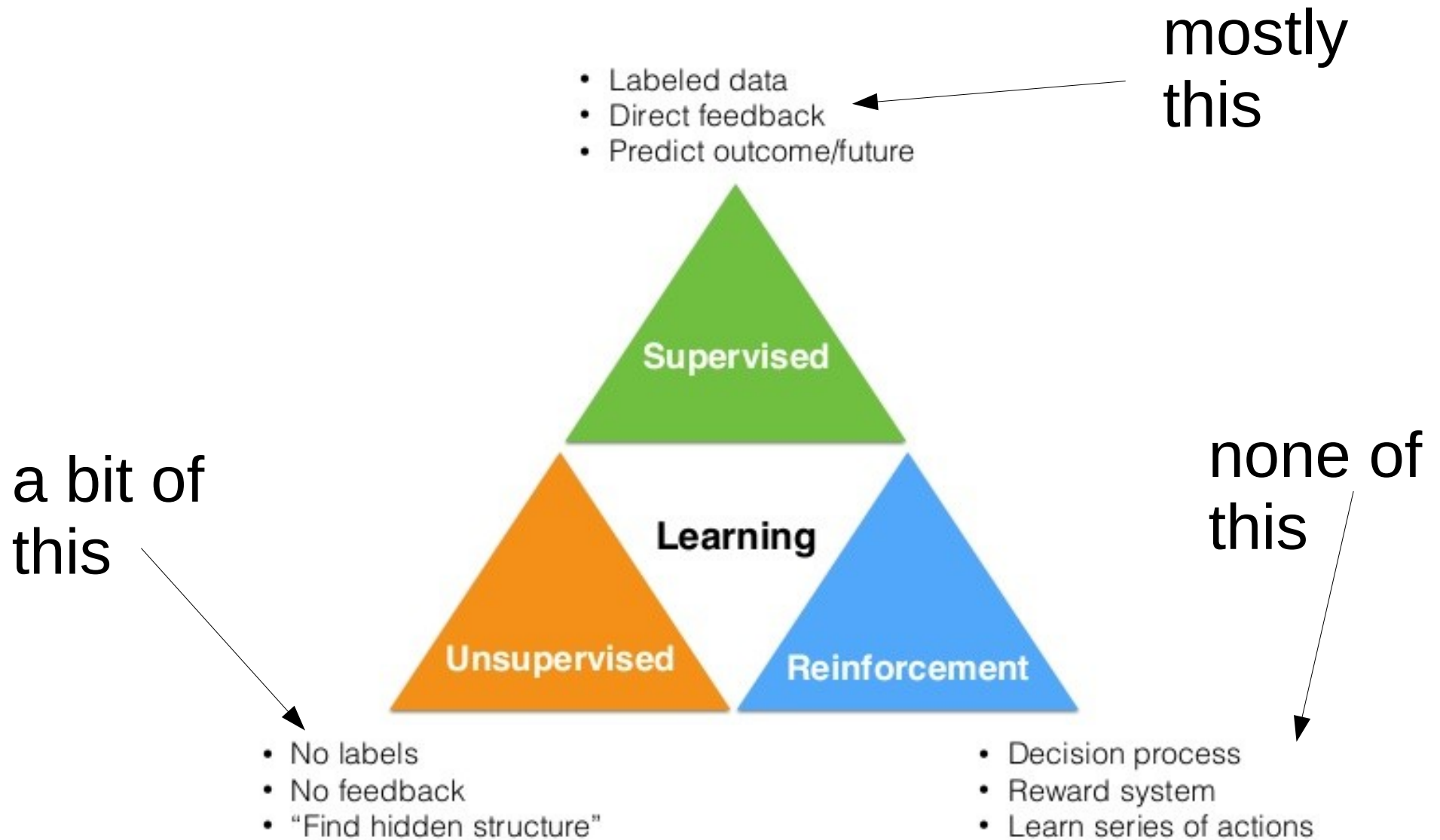
- Labeled data
- Direct feedback
- Predict outcome/future



- No labels
- No feedback
- "Find hidden structure"

- Decision process
- Reward system
- Learn series of actions

In this lecture series:



Supervised versus unsupervised

Naming convention:

x := **data** (a.k.a. “features”, “observables”, or “input”)

y := **labels** (a.k.a. “target”) = regression values, classifications

Supervised:

given the data x , and a “labeled” training sample (x,y) , train the label y from $x \rightarrow$ learn “probability” of y , given x : $p(y|x)$

If all possible y labels form a finite set: “classification”

If all possible y labels are form an (uncountably) infinite set: “regression”

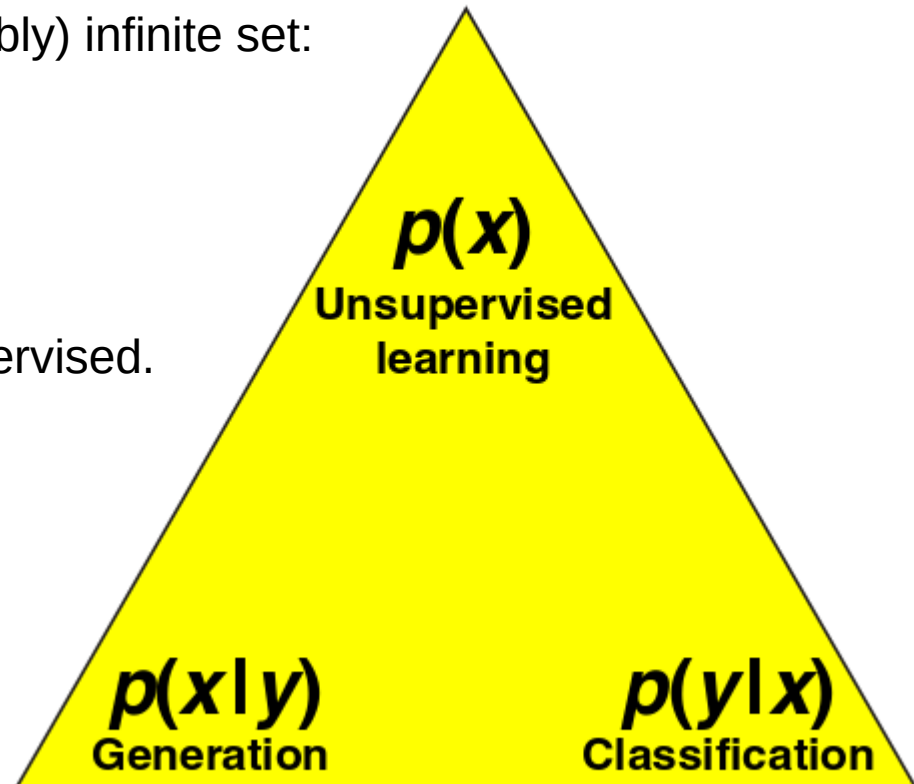
both: “discriminative models”

Unsupervised:

Usually more difficult, more vague, than supervised.
Find structure behind data: $p(x)$.

Generate “fake” data belonging to category y : $p(x|y)$: “generative model”

Plot from paper (written by physicists), “why does deep and cheap learning work so well”, [arXiv:1608.08225](https://arxiv.org/abs/1608.08225)



Machine learning versus statistics

- No clear boundary between machine learning / neural networks and statistics
- Rough distinction:

statistics deals with well-motivated low-dimensional models, is concerned with **inference** of the parameters of the models. Often **probabilistic**, “frequentist” interpretation of estimates

machine learning, especially deep learning, is more concerned with **learning “representations” and generalizing** from data, not so much about probabilistic interpretability. Neural networks are high-dimensional heuristic models, less well-motivated.

Correlation

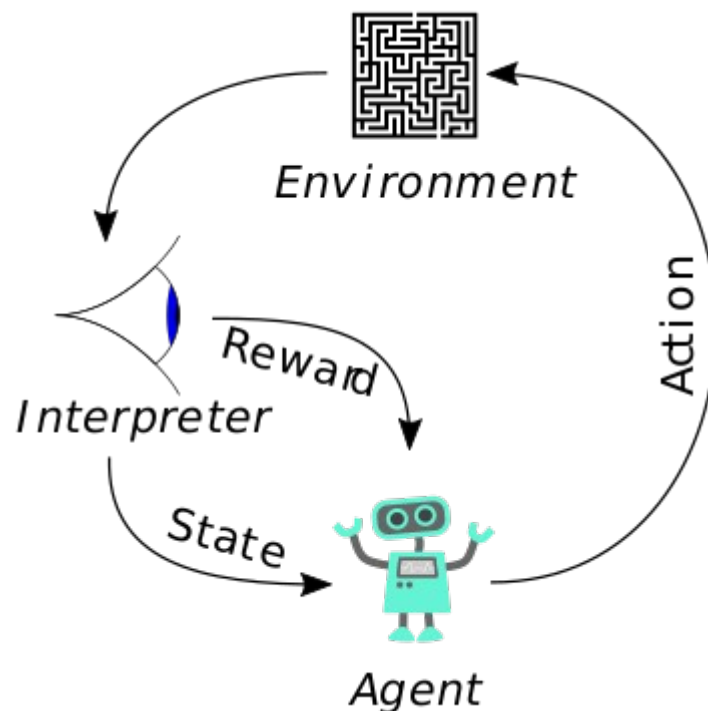


Causality

Deep learning versus other methods

- **Not all of machine learning is based on neural networks!**
- Other (“shallow”) methods are e.g.: boosted decision trees, support vector machines, Bayes statistics, and many more
- This course will however be an introduction to neural networks and deep learning.
- (we can of course discuss when to use neural networks, when not to use neural networks, when to combine methods)

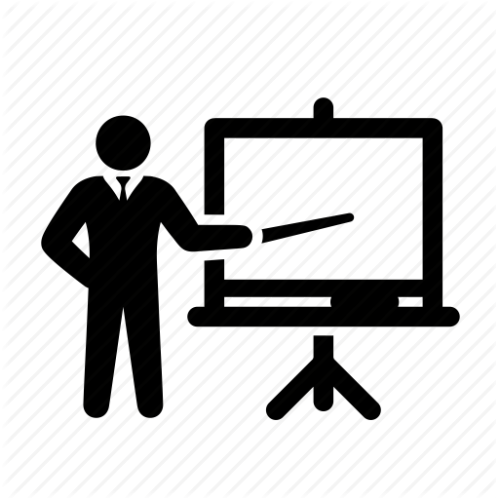
Reinforcement learning



There is an agent that can perform a series of actions in an environment. After each action $a \in A$, the agent is in a state $s \in S$. The outcome of the action is subject to reward or penalty.

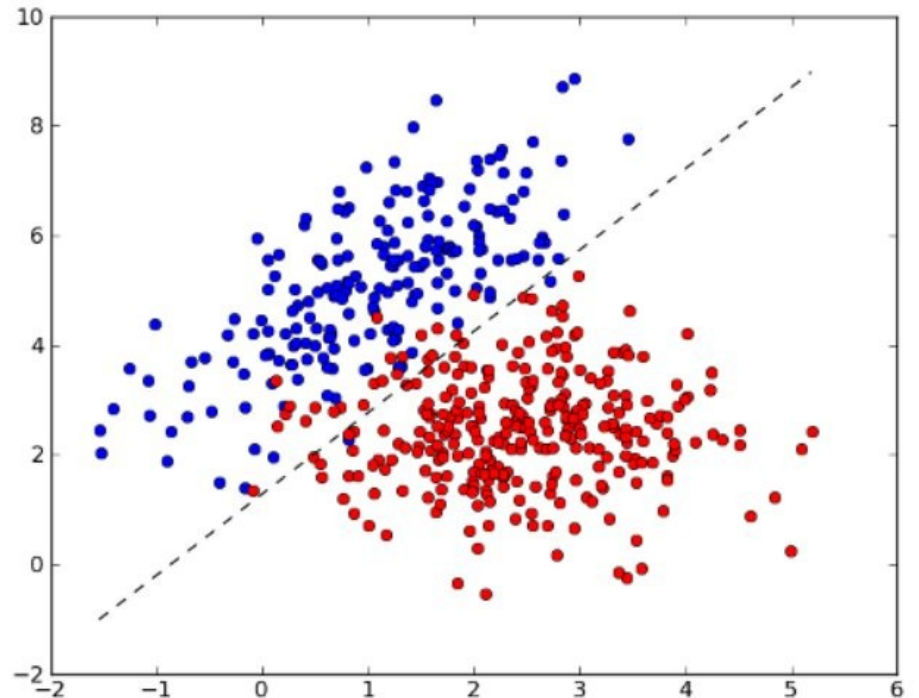
The goal is to learn a policy $p: S \rightarrow A$, that maps states to actions, maximizing reward.

("Given the state s , which action should I perform?")



Lesson 1

Logistic regression
and linear decision
boundaries

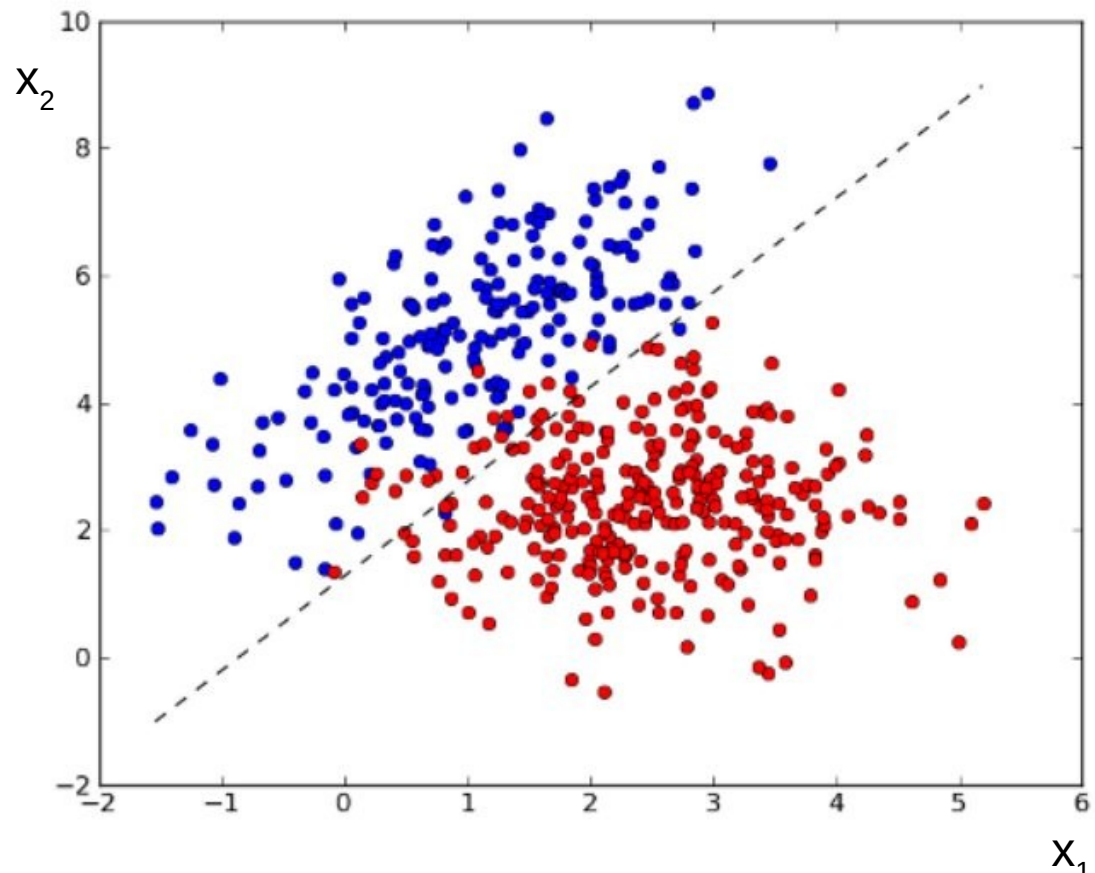


Logistic regression

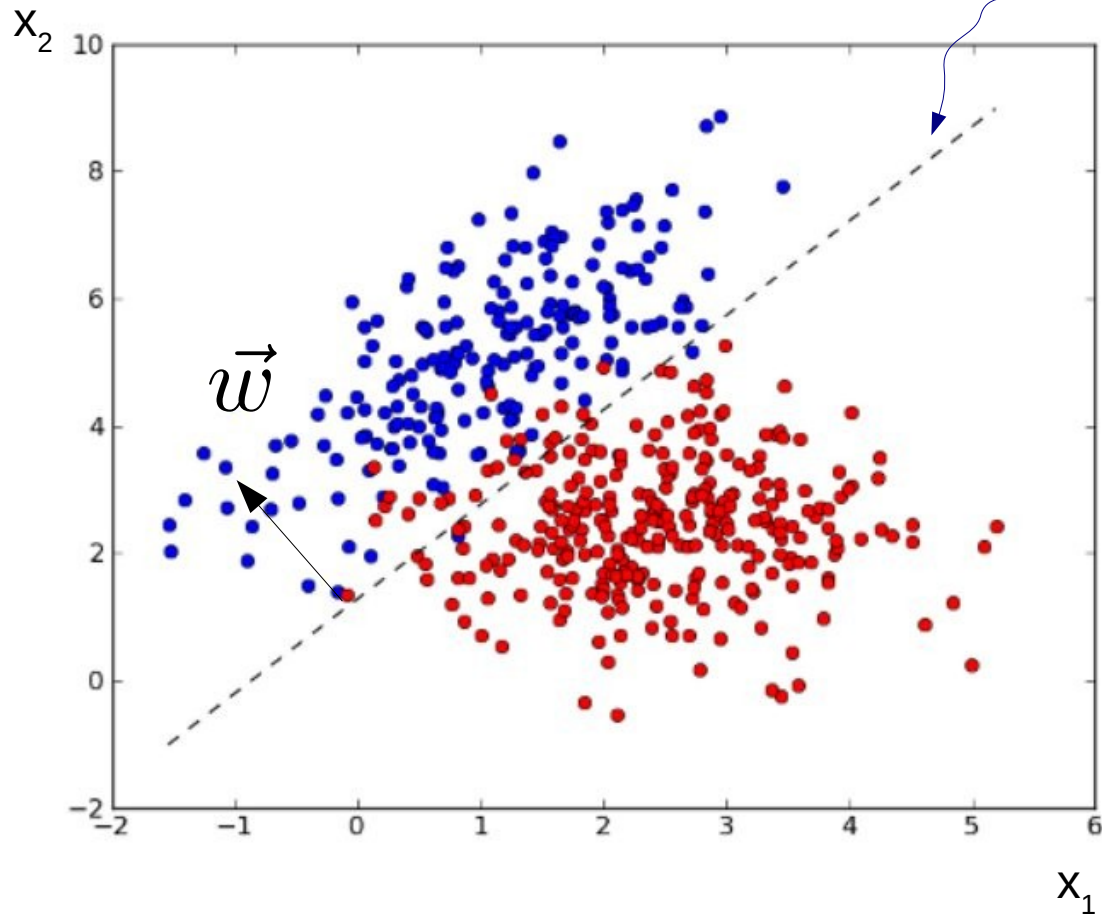
The (pedagogically) easiest way to understand neural networks is via “logistic regression”

Task: find straight line that optimally separates blue points from red points

x_1 and x_2 are “features” of your data



Logistic regression



$$x_2 = kx_1 + d$$

orthogonal
vector

$$\vec{w} := \begin{pmatrix} -k \\ 1 \end{pmatrix}$$

$$\text{net}(\vec{x}) := \vec{w} \cdot \vec{x} + d$$

$$\begin{cases} \text{net}(\vec{x}) > 0 \rightarrow \text{blue} \\ \text{net}(\vec{x}) < 0 \rightarrow \text{red} \end{cases}$$

Logistic regression

But $\text{net}(\vec{x})$ can take on any value between $-\infty$ and $+\infty$.

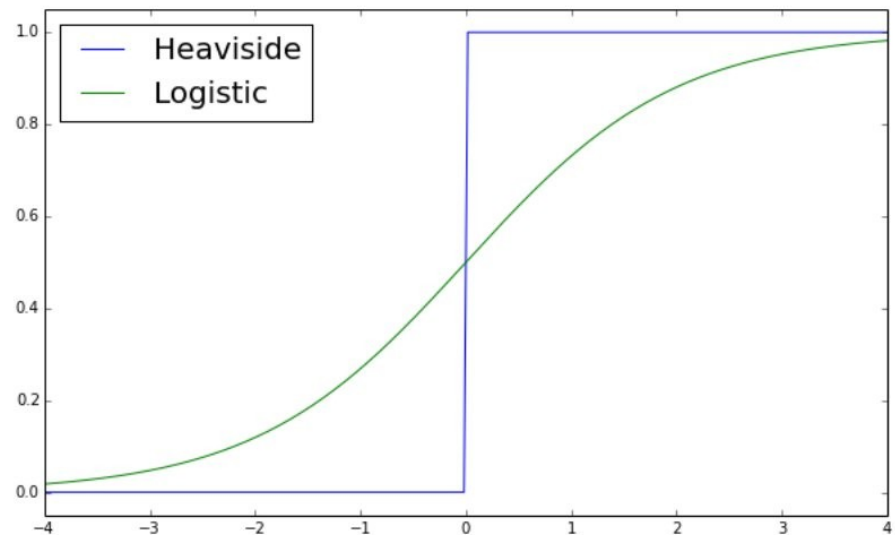
Lets map $(-\infty, \infty)$ to $[0, 1]$. Makes the handling easier. And it makes it look like a probability. One possible mapping:

Logistic function

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

Properties

1. monotonic, $\sigma(x) \in (0, 1)$
2. $\sigma(x) + \sigma(-x) = 1$
3. $\sigma'(x) = \sigma(x)(1 - \sigma(x))$
4. $2\sigma(x) = 1 + \tanh(x/2)$



$\sigma(x) = 1$ means “clearly blue”, $\sigma(x) = 0$ means “clearly red”

As physicists we take note that this looks a lot like the Fermi function

Let's construct a "loss" function

Now we build a function that tells us how much we are off:

$$\text{net}(\vec{x}) := \vec{w} \cdot \vec{x} + d$$

$$p(\vec{x}; \text{blue}) = \sigma(\text{net}(\vec{x}))$$

The point is blue.
How correct is my prediction?

$$p(\vec{x}; \text{red}) = 1 - \sigma(\text{net}(\vec{x}))$$

The point is red.
How "right" am I?

$$L(\vec{w}) = \sum_{i:\text{bluepoints}} 1 - p(\vec{x}_i; \text{blue}) + \sum_{i:\text{redpoints}} 1 - p(\vec{x}_i; \text{red})$$

Summing up over all points: how **wrong** am I?

Other possibilities

- The loss function on the last page takes the (mean) absolute value of the element-wise difference between prediction $p(x)$ and target y : **L1 Loss**.
- Let's turn the labels $y_i \in \{ \text{red, blue} \}$ into numbers: $y_i \in \{ \text{red}=0, \text{blue}=1 \}$.
- The “prediction” for label y_i

$$p(\vec{x}_i; y_i)$$

- Many other sensible choices for a loss function:

L2 Loss: (mean) squared error between prediction and target.

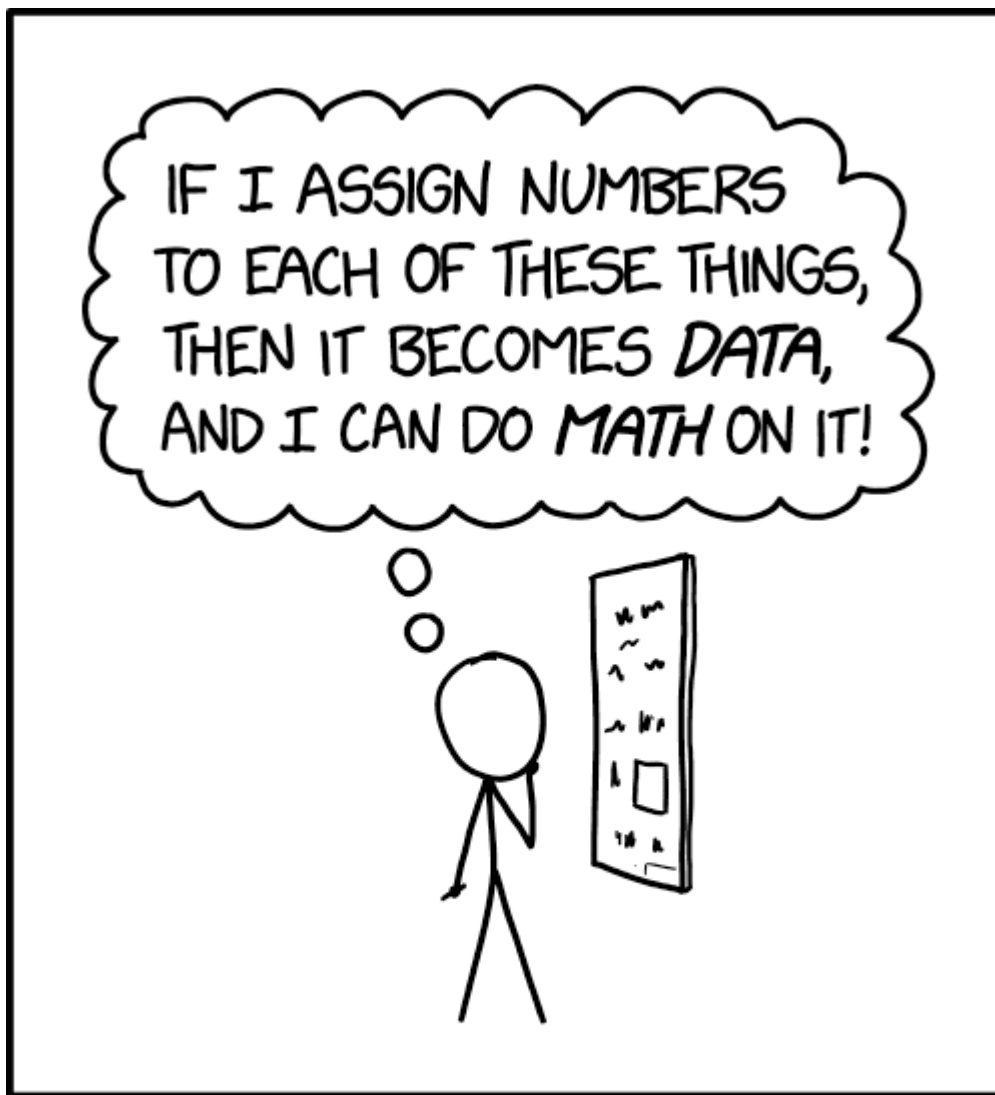
$$L(\vec{w}) = \sum_{i:\text{points}} (y_i - p(\vec{x}_i; y_i))^2$$

(Binary) Cross Entropy: the expected negative log likelihood of misclassification

$$L(\vec{w}) = - \sum_{i:\text{points}} y_i \ln(p(\vec{x}_i; y_i)) + (1 - y_i) \ln(1 - p(\vec{x}_i; y_i))$$

(*) note that I dropped a factor of $1/N$ in all losses, as they do not really matter

“Let’s turn the labels $y_i \in \{ \text{red, blue} \}$ into numbers: $y_i \in \{ \text{red}=0, \text{blue}=1 \}$.”



THE SAME BASIC IDEA UNDERLIES
GÖDEL'S INCOMPLETENESS THEOREM
AND ALL BAD DATA SCIENCE.

Loss function

- The loss function goes by many names:

energy function

objective function

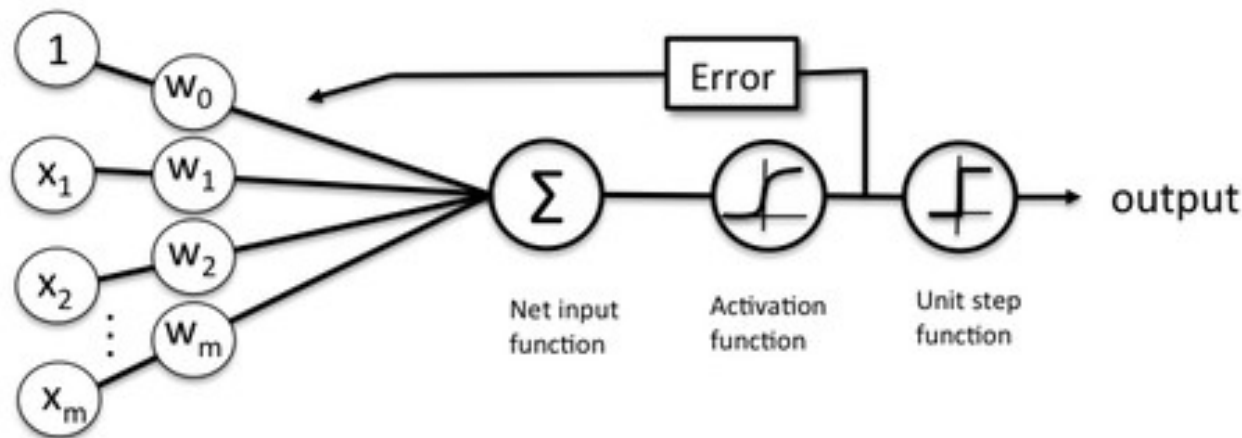
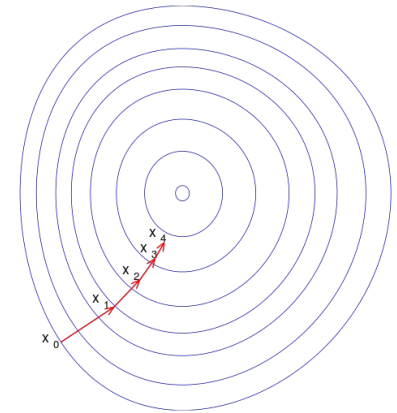
reward function

- They all mean the same thing (up to the difference in sign: a loss is a negative reward)

Learning := gradient descent

Training: find the weights that minimize the loss function → gradient descent!

$$\operatorname{argmin}_{\vec{w}} L(\vec{w})$$

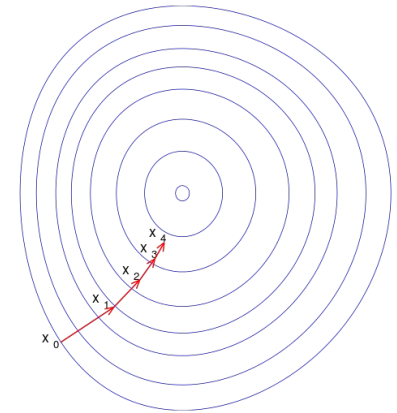


Schematic of a logistic regression classifier.

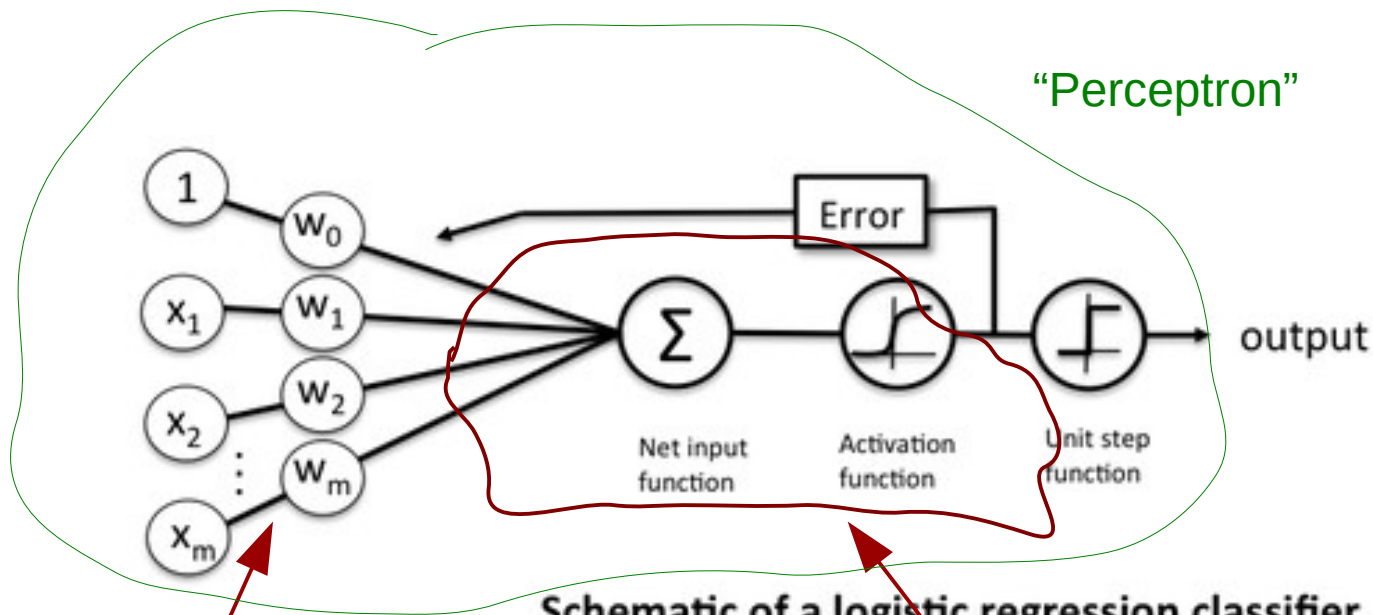
Learning := gradient descent

Training: find the weights that minimize the loss function → gradient descent!

$$\operatorname{argmin}_{\vec{w}} L(\vec{w})$$



“Perceptron”



Schematic of a logistic regression classifier.

“Synapse”

“Neuron”

Gradient descent: methods

We have a lot of freedom in the particular choice of the learning method.

- **Batch gradient descent**

computes the gradient of the loss function for the entire dataset

$$\vec{w} = \vec{w} - \eta \nabla_{\vec{w}} L(\vec{w})$$

where η is the learning rate

- **Stochastic gradient descent**

updates the weights for each data point separately

$$\vec{w} = \vec{w} - \eta \nabla_{\vec{w}} L(\vec{w}; \vec{x}^{(i)}; \vec{y}^{(i)})$$

- **Mini-batch gradient descent**

updates the weights for small “batches” of the data

$$\vec{w} = \vec{w} - \eta \nabla_{\vec{w}} L(\vec{w}; \vec{x}^{(i:j)}; \vec{y}^{(i:j)})$$

Gradient descent: methods

We have a lot of freedom in the particular choice of the learning method.

- “Bare gradient descent”

$$\vec{w} = \vec{w} - \eta \nabla_{\vec{w}} L(\vec{w})$$

where η is the learning rate.

- Momentum

Take into account the most recent past updates, similar to a physical ball with conserved momentum

$$\Delta \vec{w}_t = -\gamma \Delta \vec{w}_{t-1} - \eta \nabla_{\vec{w}} L(\vec{w})$$

where γ is a dampening term ($0 < \gamma < 1$)

Gradient descent: methods

We have a lot of freedom in the particular choice of the learning method.

- **RMSProp**

”Root Mean Square” propagation. Learning rate weighted (“adaptive learning”) by exponentially weighted average of the squares of past gradients.

$$s_t = \beta s_{t-1} + (1 - \beta) (\nabla_{\vec{w}} L(\vec{w}))^2$$

s : normalization of learning rate

β : “memory”

$$\Delta \vec{w}_t = - \frac{1}{\sqrt{s_t} + \epsilon} \eta \nabla_{\vec{w}} L(\vec{w})$$

ϵ : small number to protect against “division by zero”

Gradient descent: methods

We have a lot of freedom in the particular choice of the learning method.

- Adam

Combination of “momentum” and RMSprop. Works very well in many practical applications.

- Adagrad / adadelta

Other, similar heuristics at adapting the learning rates

Gradient descent: methods

We have a lot of freedom in the particular choice of the learning method.

- **Nesterov-accelerated gradient**

Like momentum, but with a “prescient” ball that looks at where the ball will roll, and corrects for an uncorrected “future” gradient

$$\Delta \vec{w}_t = -\gamma \Delta \vec{w}_{t-1} - \eta \nabla_{\vec{w}} L(\vec{w} - \gamma \Delta \vec{w}_{t-1})$$

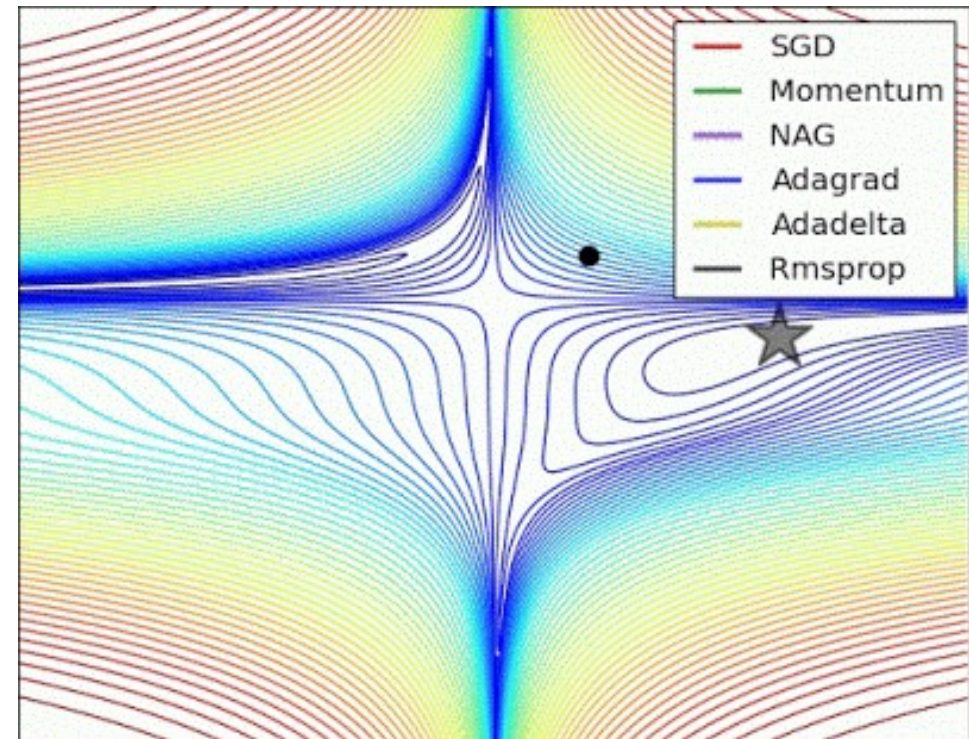
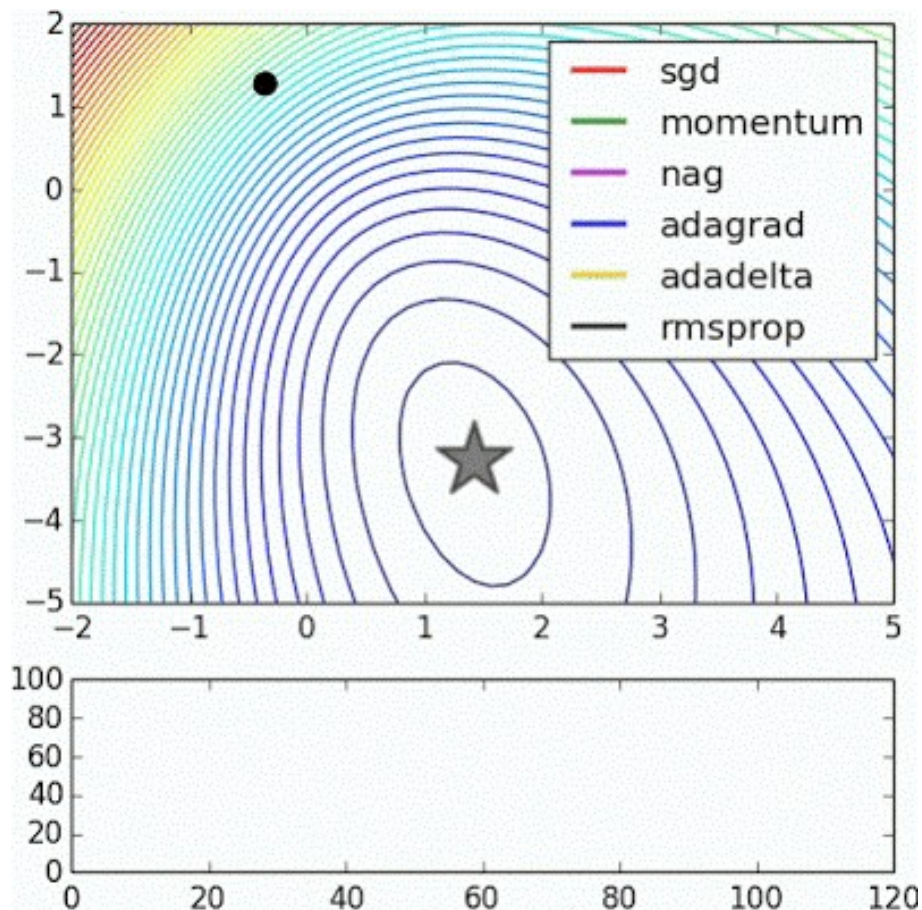
- **Nadam**

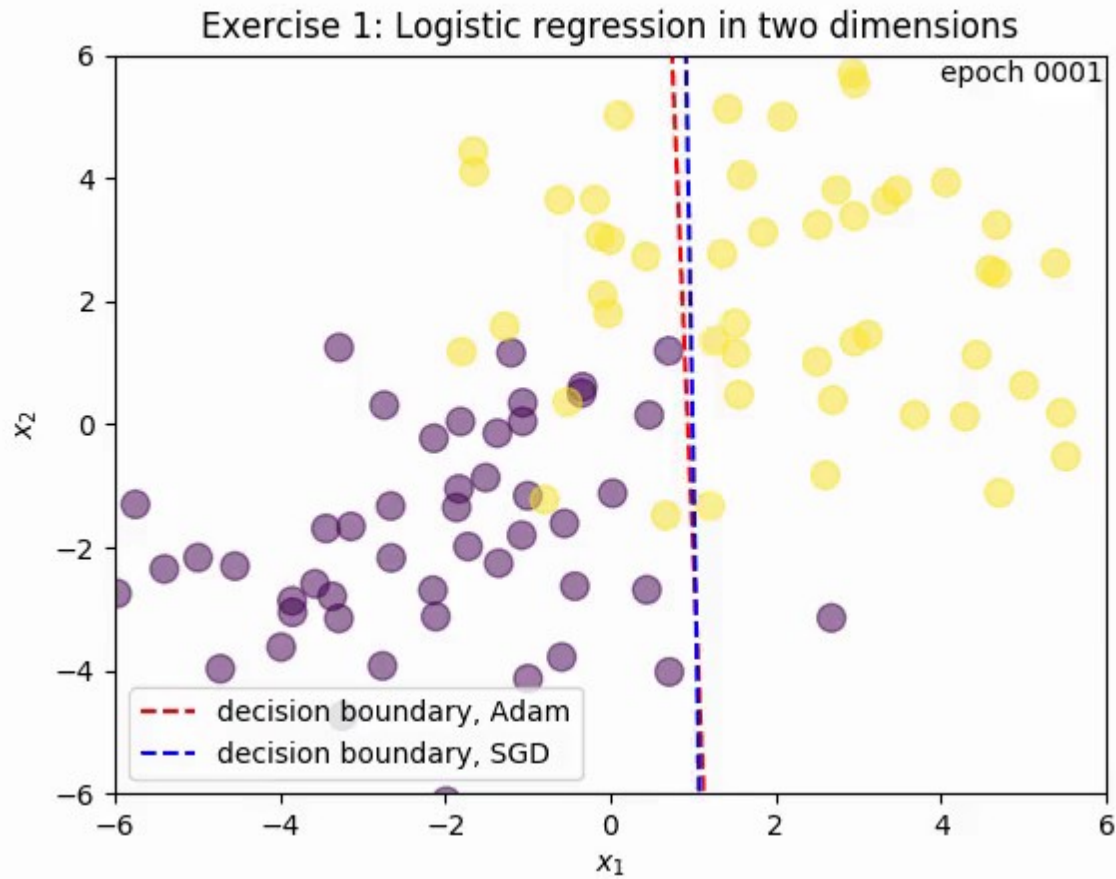
Nesterov-accelerated Adam

- **Second order optimizations (Newton, natural gradients)**

We can also add second order derivatives of the loss function to our updates. Interesting, also theoretically (information geometry), but often computationally too expensive.

Gradient descent: methods

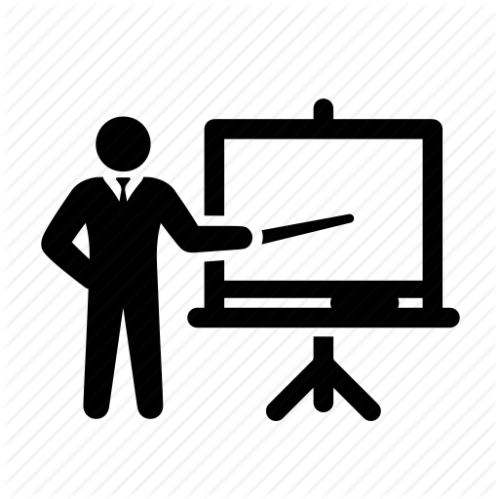






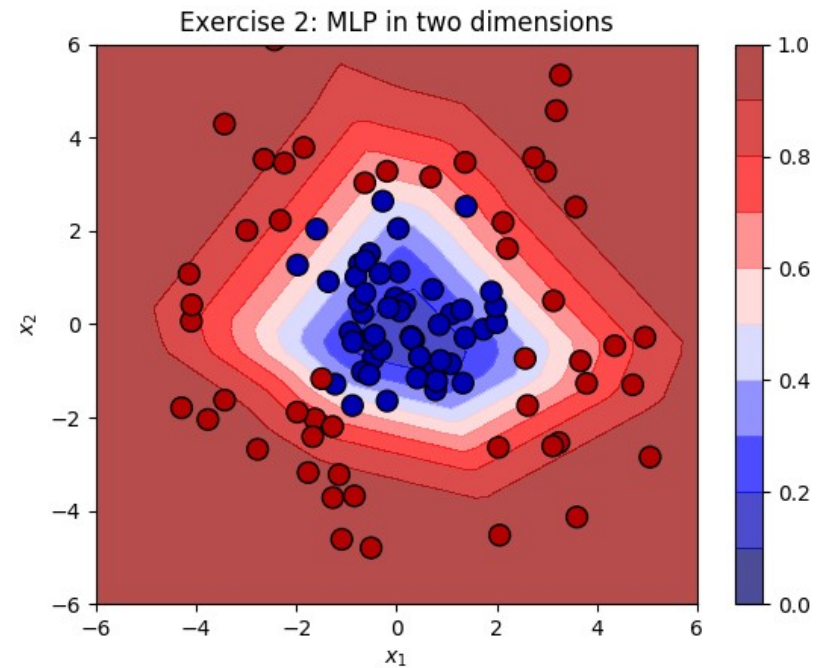
Exercise 1 – logit

- Go to <https://github.com/WolfgangWaltenberger/vietnam2022>
See explanations on that webpage
- Start with [01_logistic.ipynb](#), fill in the ellipses (“...”)
- Goal is to find the best linear decision boundary between yellow and magenta points.
- Play around with learning rate, loss function, optimizer.
- Which optimizer converges the fastest?

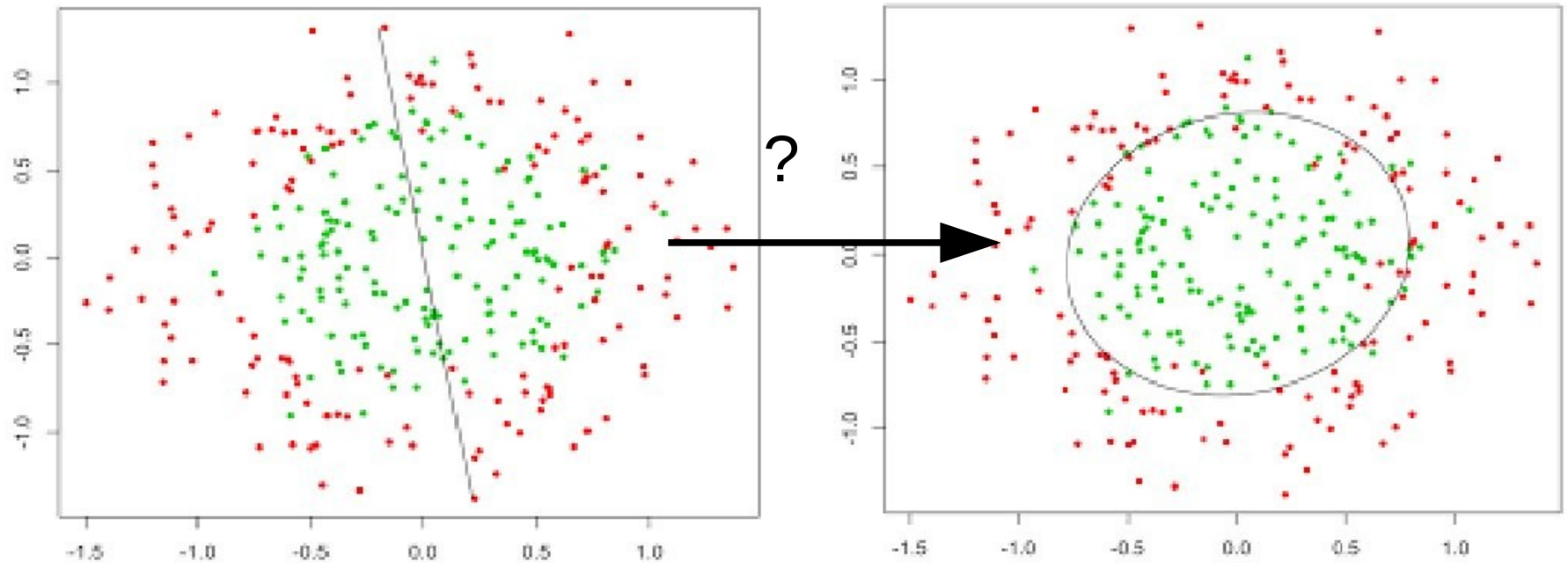


Lesson 2

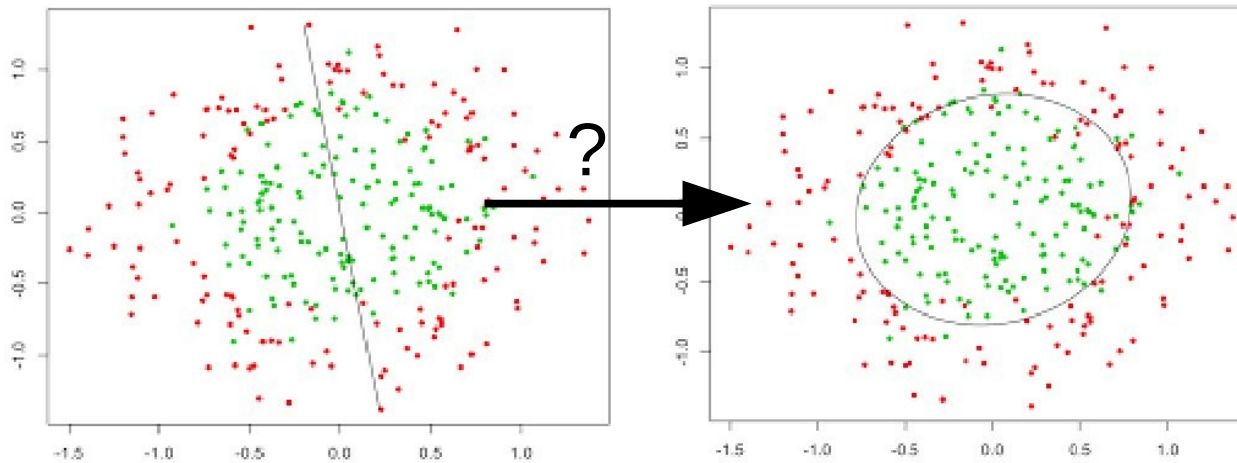
Multi-layer perceptron
and non-linear
decision boundaries



How can we go from a linear to a non-linear decision rule?

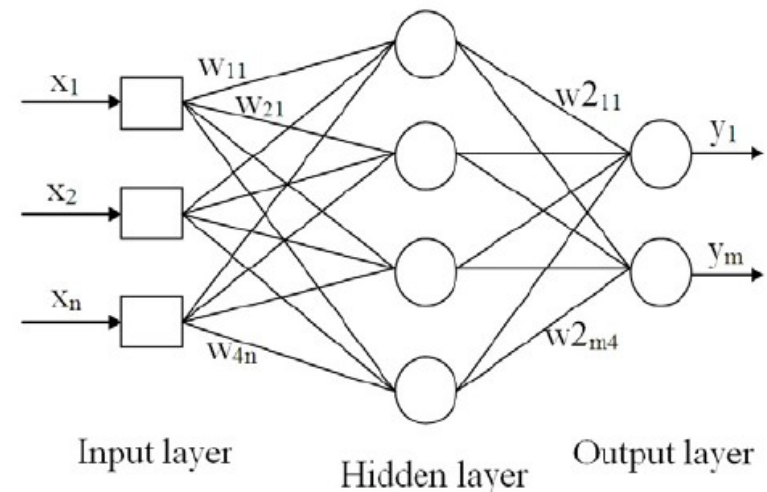


How can we go from a linear to a non-linear decision rule?

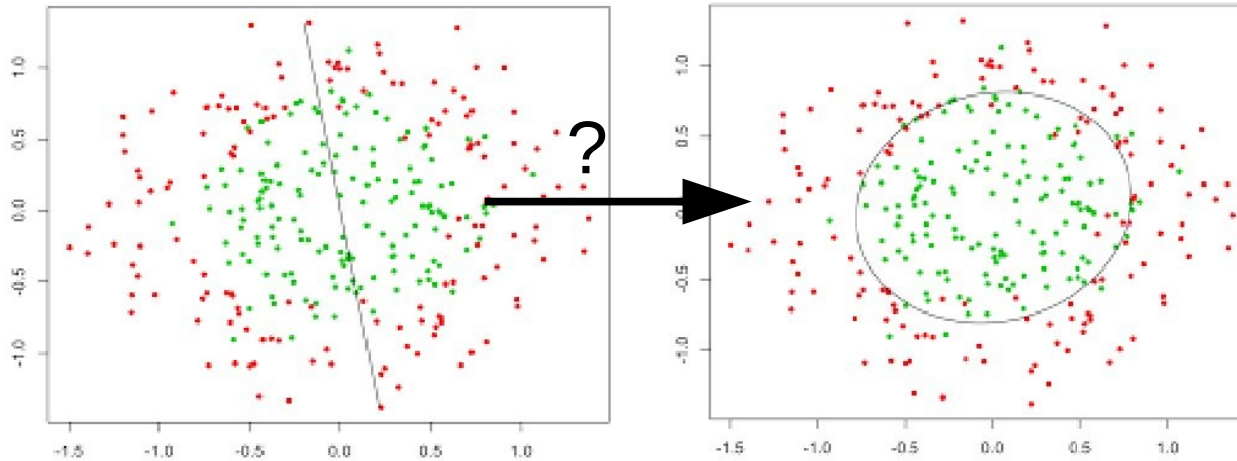


We introduce “hidden” layers of neurons!

$$p(\vec{x}; y) = \sigma_1(\text{net}_1(\sigma_2(\text{net}_2(\vec{x}))))$$



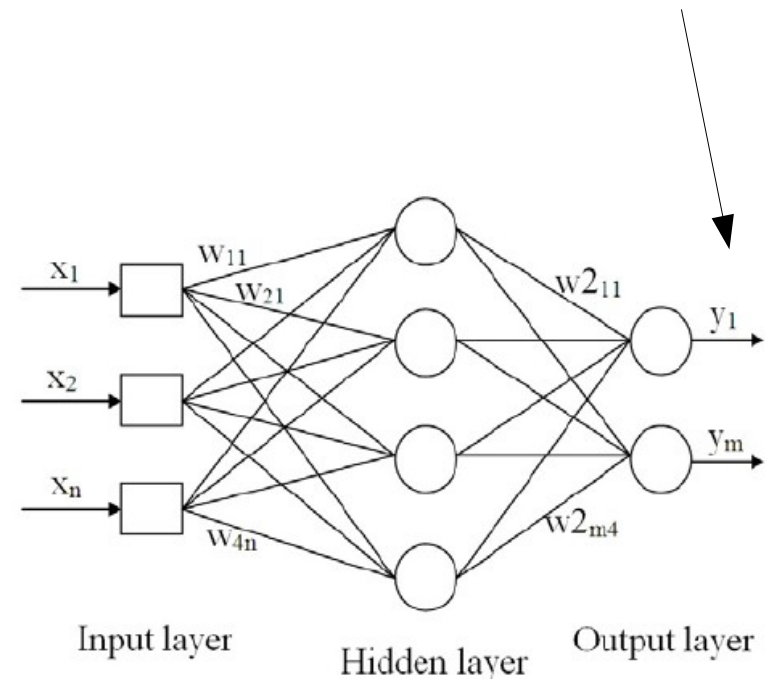
How can we go from a linear to a non-linear decision rule?



Can simultaneously regress / classify multiple targets

We introduce “hidden” layers of neurons!

$$p(\vec{x}; y) = \sigma_1(\text{net}_1(\sigma_2(\text{net}_2(\vec{x}))))$$



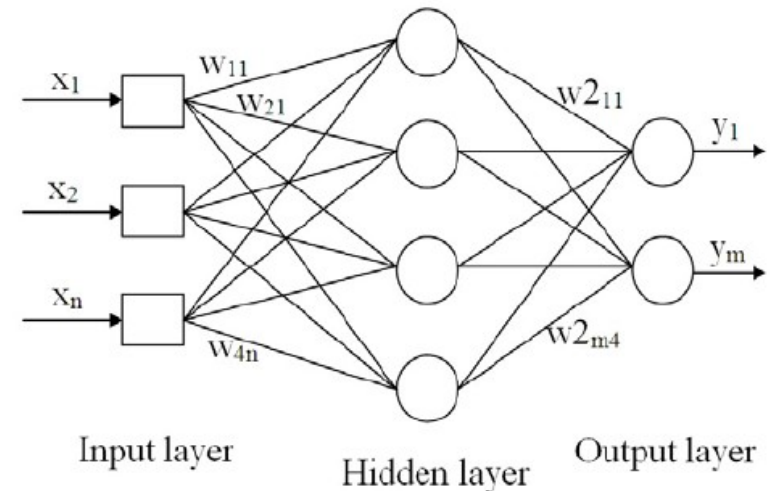
Deep learning

How do we train a deep network?

By **gradient descent + chain rule := backpropagation!**

Chain rule:
$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}}$$

where
$$o_j = \sigma(\text{net}_j) = \sigma\left(\sum_k w_{jk} o_k\right)$$



A weight update (with simple gradient descent) is then simply

$$\Delta w_{ij} = -\eta \frac{\partial L}{\partial w_{ij}}$$

(η is the learning rate)

Example: backprop for MLP

Example for multi layer perceptron (MLP) for simple regression example.

Loss function: sum of squared errors (L2) $L \propto (y - t)^2$

where (changed naming convention!)

t is the “target” output of training data
 y is the predicted output of neural network.

Network minimizes the squared error.

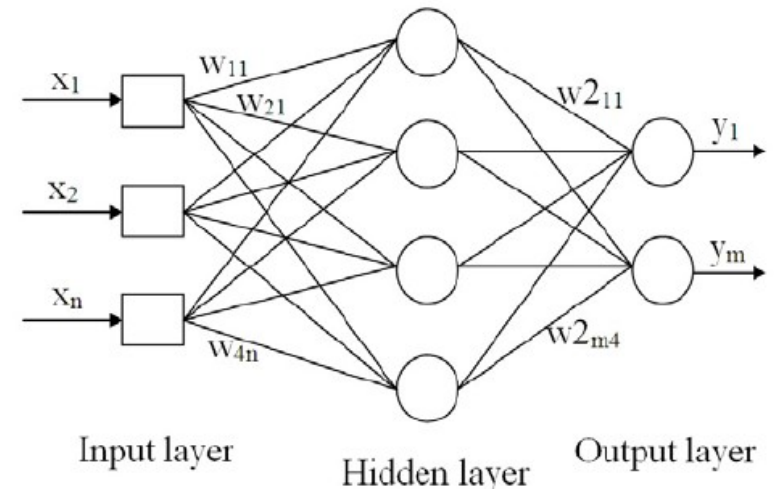
Chain rule:
$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}}$$

where

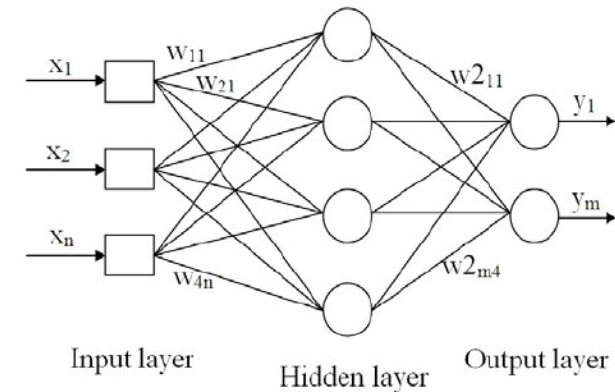
$$o_j = \sigma(\text{net}_j) = \sigma \left(\sum_k w_{jk} o_k \right)$$

A weight update is then simply

$$\Delta w_{ij} = -\eta \frac{\partial L}{\partial w_{ij}}$$



Example: backprop for MLP



$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}}$$

$$o_j = \sigma(\text{net}_j) = \sigma\left(\sum_k w_{jk} o_k\right)$$

$$\frac{\partial \text{net}_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \left(\sum_k w_{jk} o_k \right) = \delta_{ik} o_k = o_i$$

For a logistic activation function:

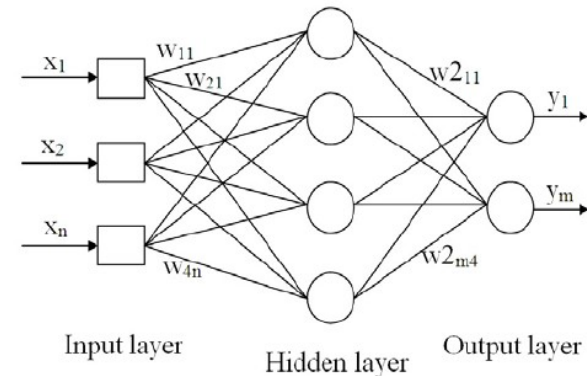
$$\frac{\partial o_j}{\partial \text{net}_j} = \sigma(\text{net}_j)(1 - \sigma(\text{net}_j)) = o_j(1 - o_j)$$

Finally, for mean squared error loss function:

$$\frac{\partial L}{\partial o_j} = y - t$$

If o_j is the last layer (with a single neuron, $o_j = y$, $t =$ target label)

Example: backprop for MLP



Finally, for mean squared error loss function:

$$\frac{\partial L}{\partial o_j} = y - t$$

If o_j is the output layer that contains a single neuron ($o_j = y$)

For layers other than the output layer, we can introduce a recursion:

$$\frac{\partial L}{\partial o_j} = \sum_l \left(\frac{\partial L}{\partial o_l} \frac{\partial o_l}{\partial net_l} w_{jl} \right)$$

where the index l sums over all neurons that receive input from neuron j .

Let's put it all together:

$$\Delta w_{ij} = -\eta \frac{\partial L}{\partial w_{ij}} = -\eta P_j o_i$$

With
$$P_j = (o_j - t_j) o_j (1 - o_j)$$

$$P_j = \left(\sum_l w_{jl} P_l \right) o_j (1 - o_j)$$

For the output layer (top) and all other layers (bottom).

Differentiable programming

Luckily, with software frameworks like tensorflow, pytorch we do not have to compute the derivatives ourselves! All we have to do is define the network architecture, the frameworks compute the gradients.

→ automatic differentiation, differentiable programming

```
self.layer1 = nn.Sequential(  
    nn.Conv2d(1, 16, kernel_size=5, stride=1, padding=2),  
    nn.BatchNorm2d(16),  
    nn.ReLU(),  
    nn.MaxPool2d(kernel_size=2, stride=2))  
self.rnn = nn.LSTM(hidden_size, hidden_size, 2, dropout=0.05)  
self.fc = nn.Linear(7*7*32, num_classes)
```

Universal approximation theorem

The “universal approximation theorem” (Cybenko, 1989, Hornik, 1991) proves, that – under a few mild assumptions -- already with a single hidden layer, we can essentially learn every continuous function.

So why introduce more than one hidden layer?

Universal approximation theorem

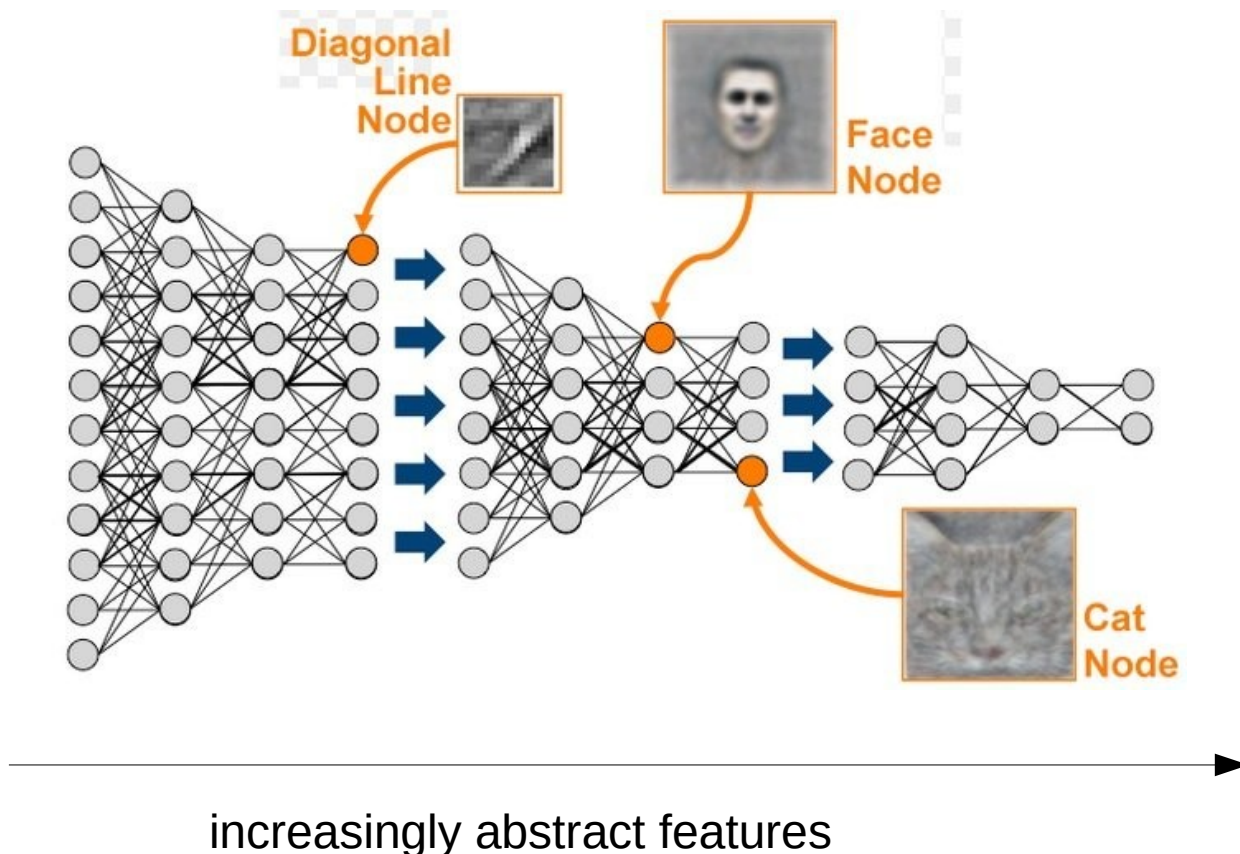
The “universal approximation theorem” (Cybenko, 1989, Hornik, 1991) proves, that already with a single hidden layer, we can essentially learn every continuous function (with a few mild assumptions).

So why introduce more than one hidden layer?

”Practicalities”: the single hidden layer may have to be infeasibly large. The learning might take too long, the learning sample might have to be too large, and generalization will not take place.

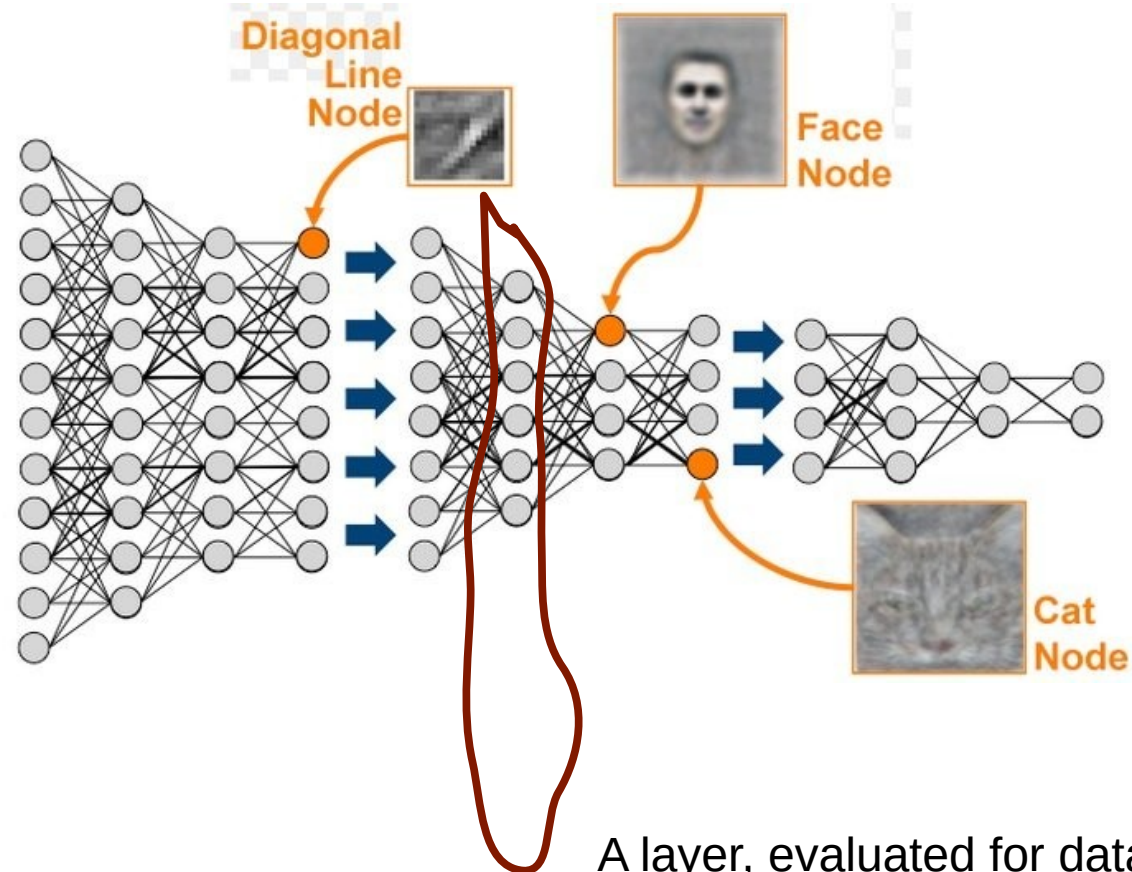
Deep learning = learning abstractions

When done right, a deep network can learn “abstract features” of the input data, and thus overcome the practical restrictions of an MLP with a single hidden layer:



Deep learning = hierarchical representation learning

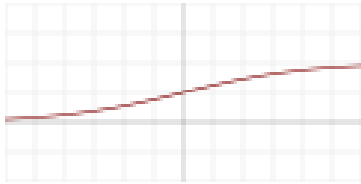
Because when done right, a deep network can learn “abstract features” of the input data:



A layer, evaluated for data = a “representation” of the data

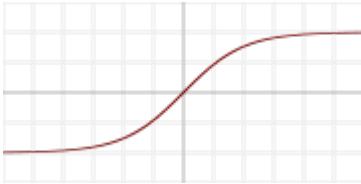
Activation functions

We have a lot of freedom in the choice of our activation functions

- sigmoid (logistic)  $f(x) = \frac{1}{1 + e^{-x}}$

pro: non-linear, differentiable, confined range [0,1]

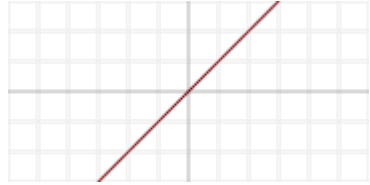
con: zero at large values of $|x|$ (vanishing gradient problem)

- tanh  $f(x) = \tanh(x)$

scaled version of sigmoid: $\tanh(x) = 2 \sigma(2x) - 1$

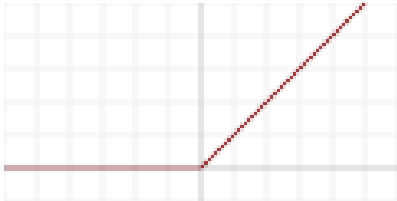
Activation functions

- We have a lot of freedom in the choice of our activation functions

- Linear  $f(x) = x$

Pro: simple, non-zero derivative

Con: constant gradient, cannot introduce non-linearity

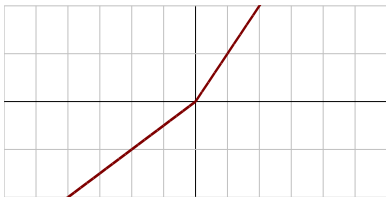
- ReLU rectified linear unit  $f(x) = \max(0, x)$

Pro: simple, computationally fast, non-linear, non-zero gradient for any large x

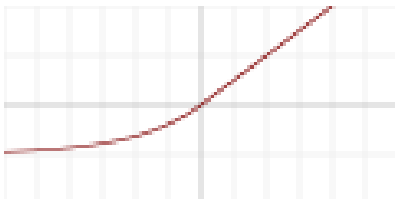
Con: not bound, neurons may “die”, get stuck at zero.

Activation functions

- We have a lot of freedom in the choice of our activation functions

- Leaky ReLU  $f(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$

Advantages of ReLU, plus it doesn't "die" for $x < 0$.

- SeLU scaled exponential linear units  $f(x) = \lambda \begin{cases} \alpha e^x - \alpha & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$

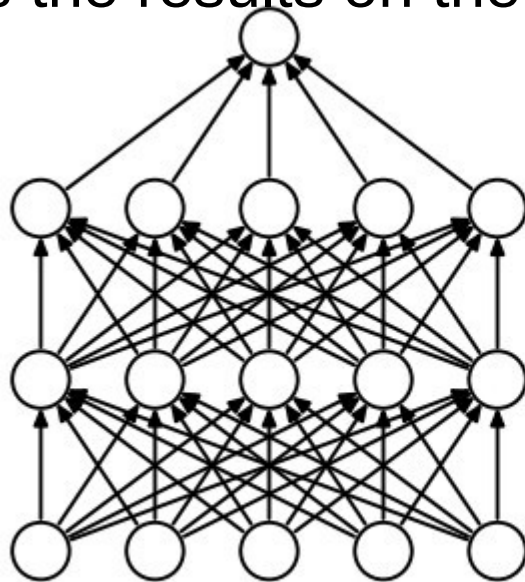
Used in "self-normalizing neural networks", where the output of the neurons in each layer is distributed with zero mean and unit variance if the input is also distributed with zero mean and unit variance

Dropout layers

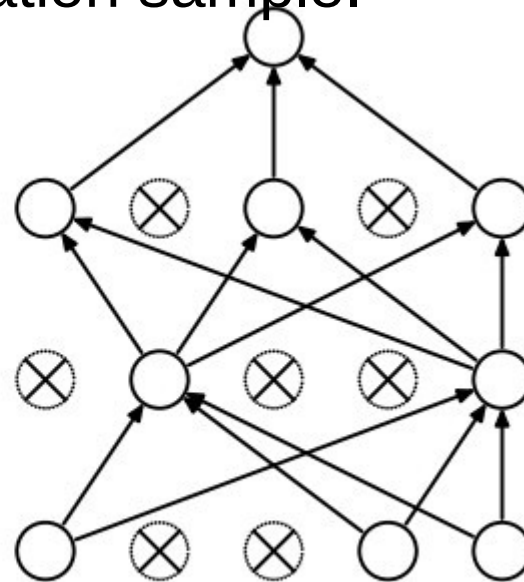
During training: drop a certain fraction of the neurons, to prevent the network from “overfitting”.

During prediction: dropout is deactivated.

Usually worsens the results on the training sample, but improves the results on the validation sample.



(a) Standard Neural Net



(b) After applying dropout.

Regularization

“Occam’s razor for neural networks”:

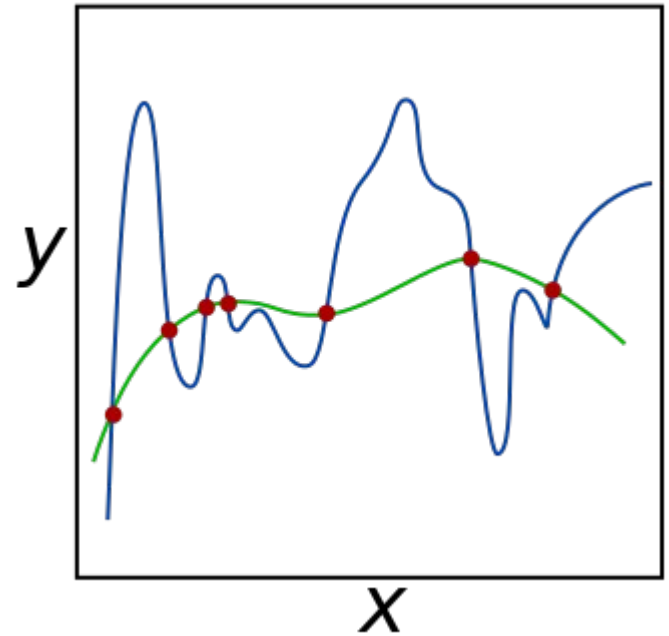
Of two models with the same predictive power, the “simpler” one is to be preferred.

→ Prefer networks with many weights of zero

“Penalty term” for weights in the loss function, e.g. L2 regularization:

$$L \rightarrow L + \lambda \sum_{ij} w_{ij}^2$$

Implemented in many of pytorch’s optimizers, lambda is often called “weight decay”.



Hyperparameter optimization

Finding the optimal network architecture and hyperparameters is yet another optimization task →

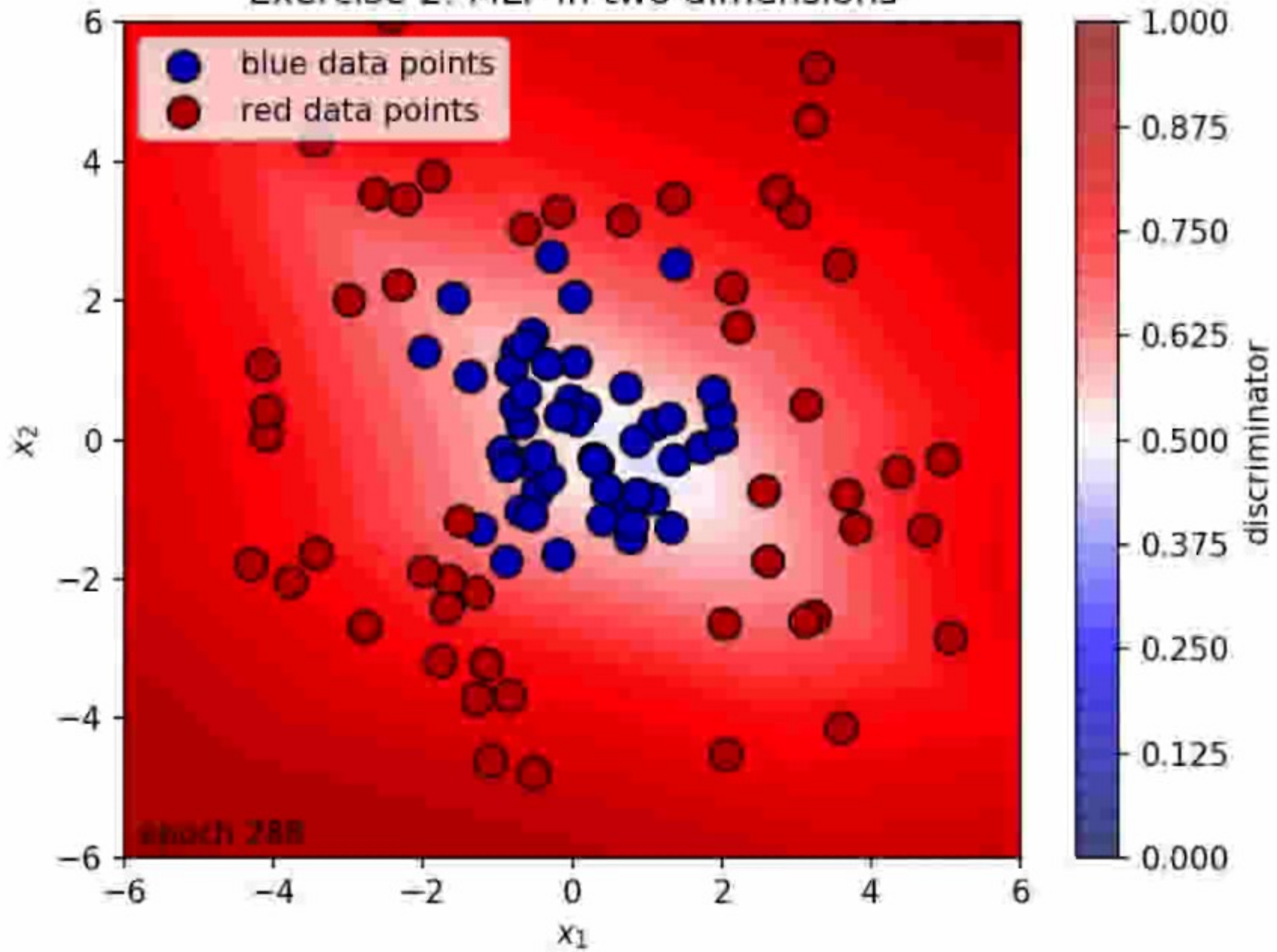
hyperparameter optimization, network optimization

Many simple algorithms:

- **grid search**: systematically try out many configurations
- **random search**: try out configurations randomly
- **gradient based optimization**: compute gradients in the space of hyperparameters
- **evolutionary algorithms**

....

Exercise 2: MLP in two dimensions



THIS IS YOUR MACHINE LEARNING SYSTEM?

YUP! YOU POUR THE DATA INTO THIS BIG PILE OF LINEAR ALGEBRA, THEN COLLECT THE ANSWERS ON THE OTHER SIDE.

WHAT IF THE ANSWERS ARE WRONG?

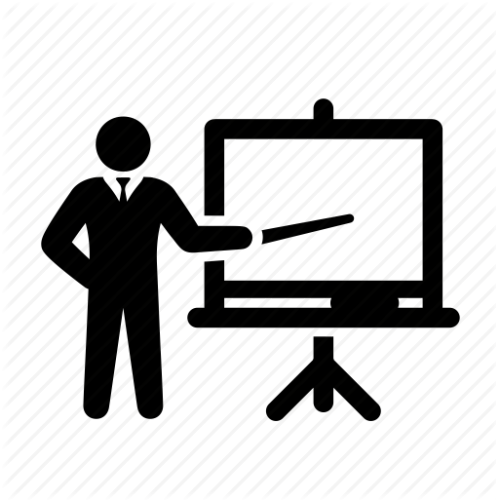
JUST STIR THE PILE UNTIL THEY START LOOKING RIGHT.





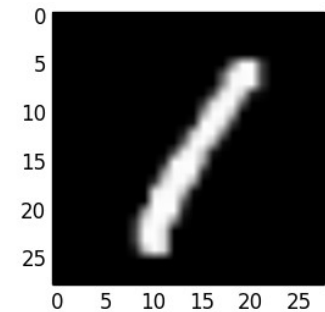
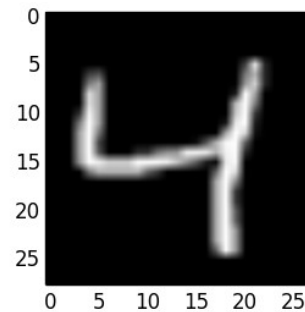
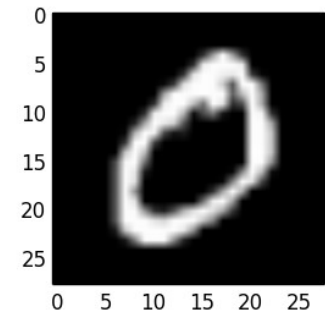
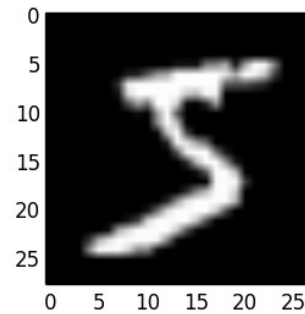
Exercise 2 – MLP

- See [02_mlp.ipynb](#) for skeleton code you may use for a quick start.
- Goal is to find the best non-linear decision boundary between blue and red points.
- Play around with learning rate, loss function, optimizer, number of layers, number of neurons, activation functions, dropout layers.
- Which network architecture worked best for you?



Lesson 3

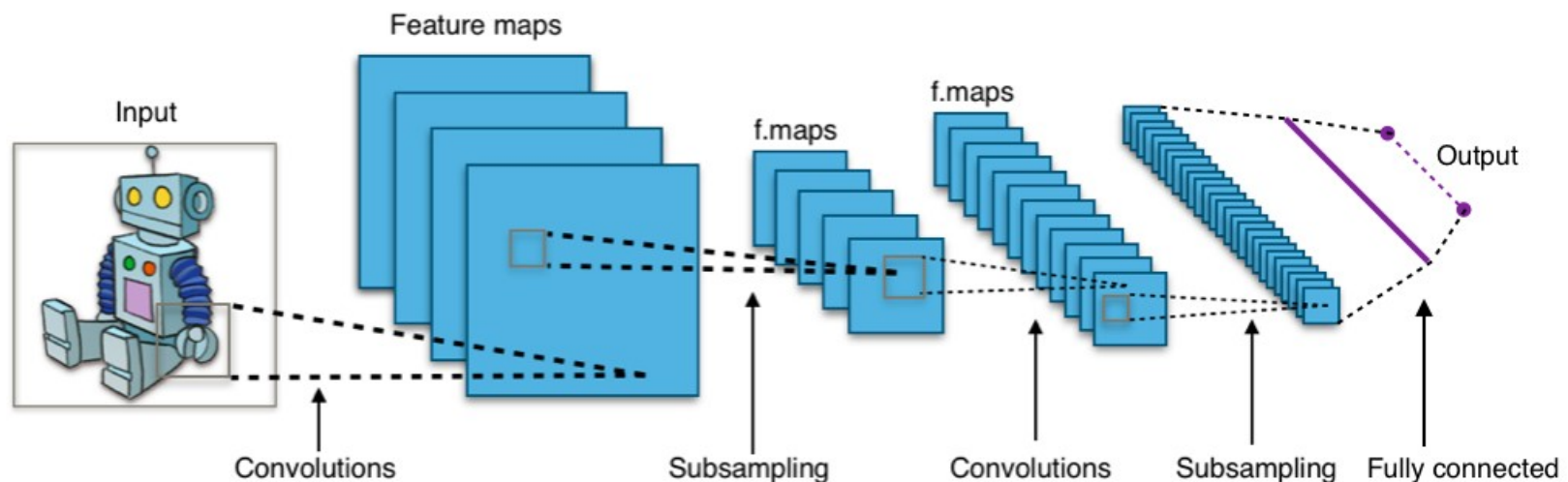
Convolutional
neural
networks



Convolutional layers

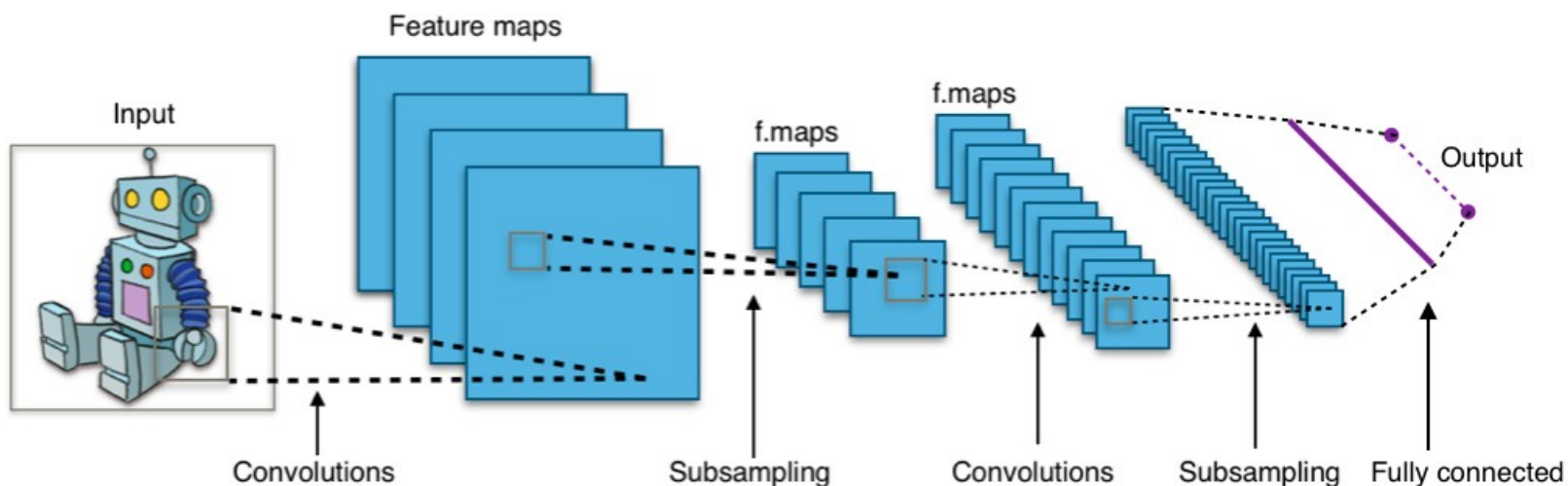
Think computer vision: we want to detect a cat, independently of where it is in the photo: translation invariance!

Answer: convolutional layers: many several small convolution matrices (“kernels”)



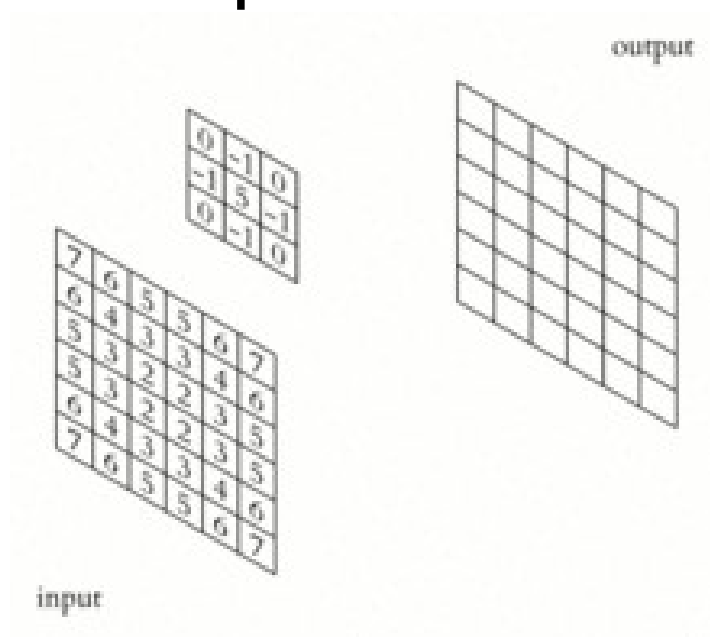
Convolutional layers

Answer: convolutional layers: many several small convolution matrices (“kernels”) that “scan” over the input image, “sharing” the learned weights over the scan and thus detect features independent of their position

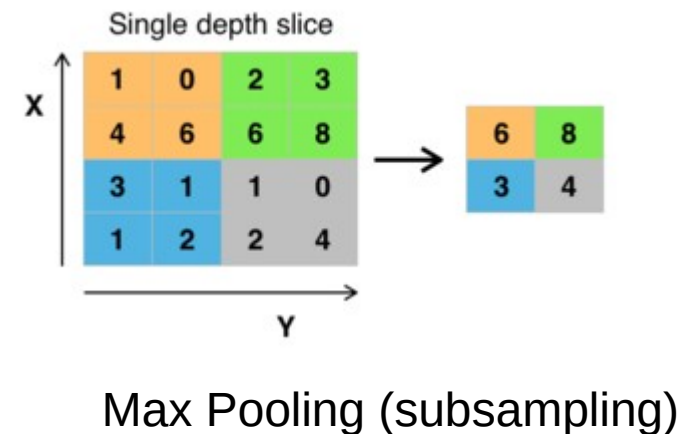


Convolutional layers

Answer: convolutional layers. Many several small convolution matrices (“kernels”) that “scan” over the input image, “sharing” the learned weights and thus detect features independent of their position



Convolutions



Inspired by the “receptive field” in biology

Side remark on convolutions

- Convolutional networks owe their success to exploiting the translational equivariance.

(equi-/invariance → symmetries → group theories is something we have some familiarity with from gauge theories)

- Can symmetries be exploited more systematically in neural networks?

<https://zenodo.org/record/6459381>

<https://arxiv.org/abs/2101.03164>

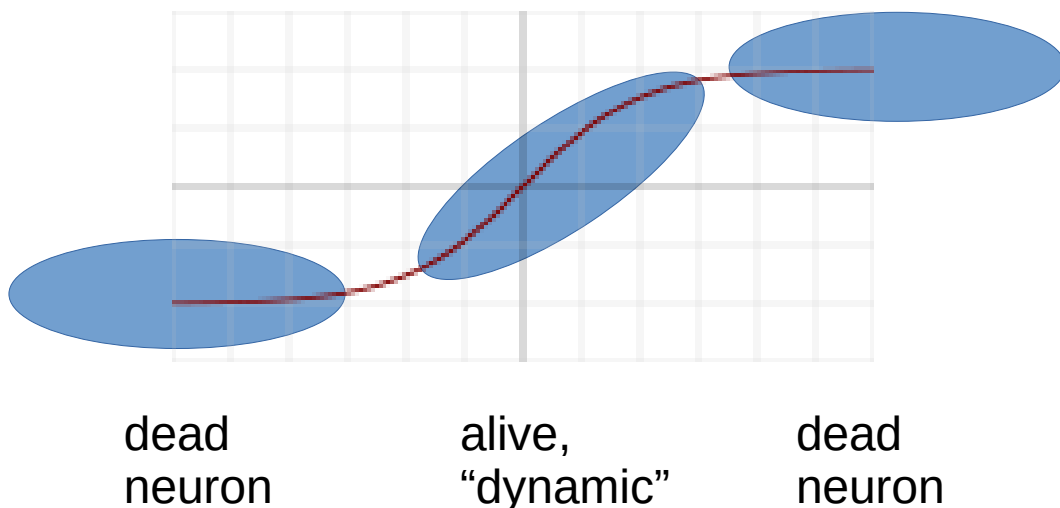
<https://arxiv.org/abs/2110.02905>

<http://proceedings.mlr.press/v76/ensign17a/ensign17a.pdf>

Batch normalization

It is usually a good trick to “standardize” the inputs: a fine transformation so they have zero mean and unit variance on a given data sample.

This is also true for hidden neurons → batch normalization.



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1..m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{ mini-batch mean}$$
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{ mini-batch variance}$$
$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{ normalize}$$
$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{ scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Softmaxing

For a classification problem with K different classes (e.g. “building”, “animal”, “object”), you want the output neurons to encode a “probability” of the input belonging to class j .
(“One-hot encoding” – one neuron per class)

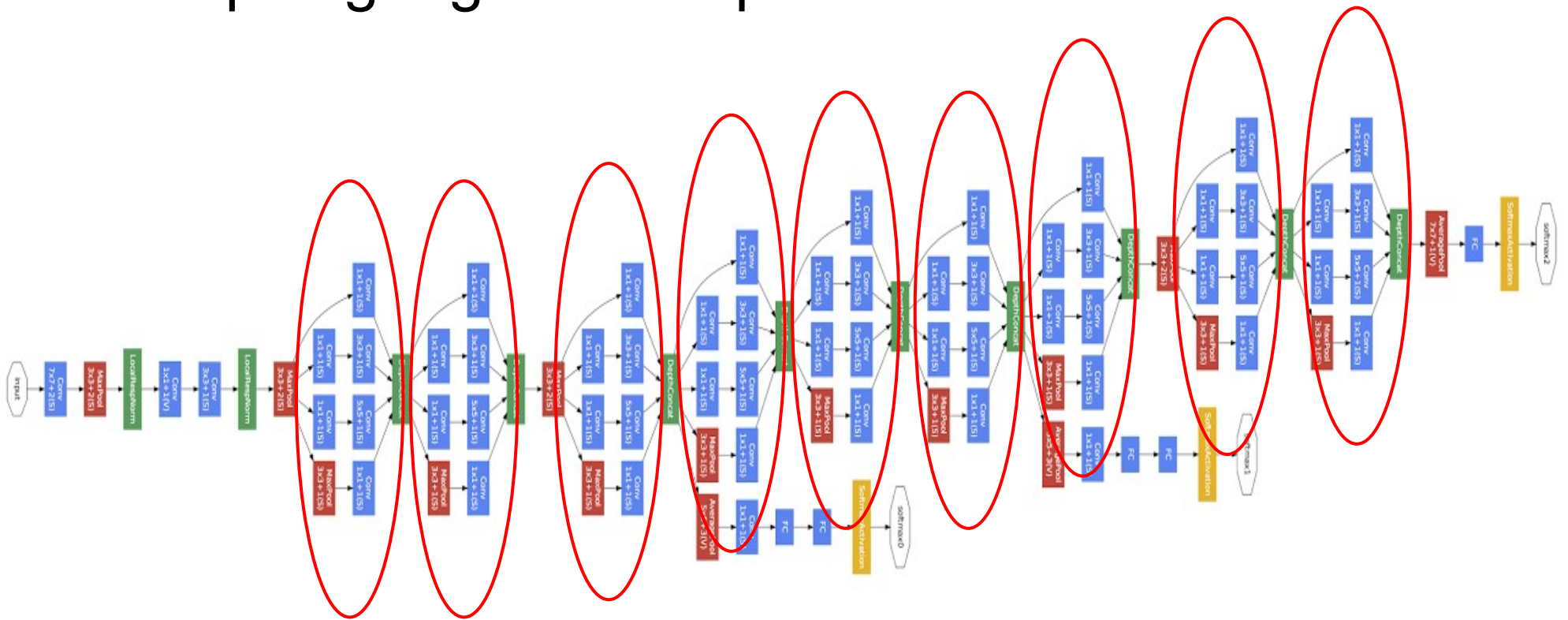
The K output neurons should therefore acquire values between 0 and 1, and sum up to 1.

Softmax function achieves this:

$$p(j|\vec{y}) = \frac{e^{y_j}}{\sum_{k=1}^K e^{y_k}} \text{ for } j = 1, \dots, K$$

As physicists we note that this is equivalent to the Boltzmann distribution: if y_j denotes the negative energy of a quantum state j divided by $k_B T$, then $p(j)$ is the probability of the quantum object being found in that state.

Example: google's "Inception" network



9 **Inception** modules

Convolution
Pooling
Softmax
Other

Network in a network in a network...

Application in particle physics: jet images

Task: learn a jet classifier that identifies highly boosted W jets

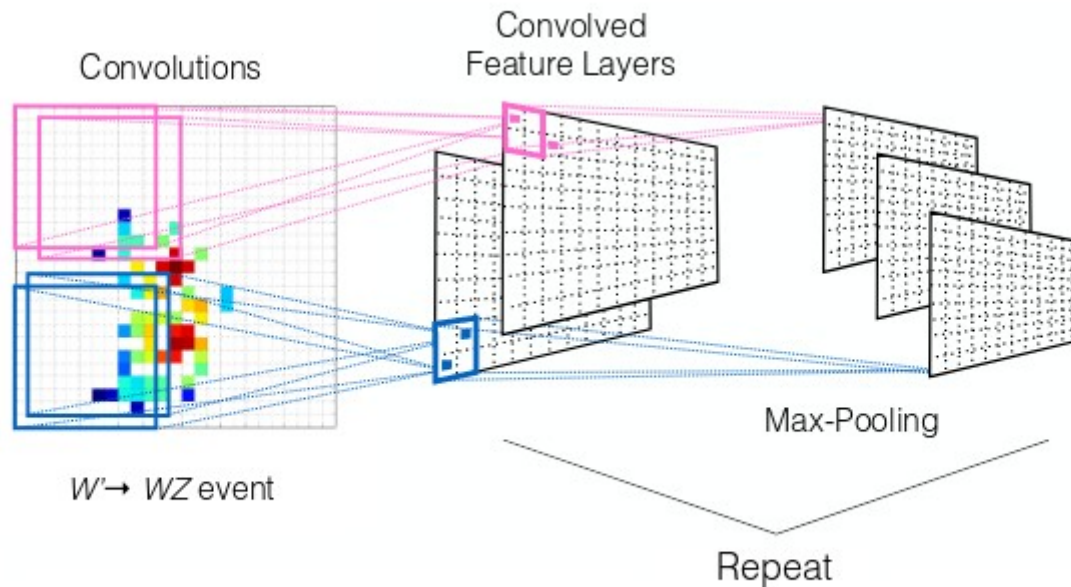
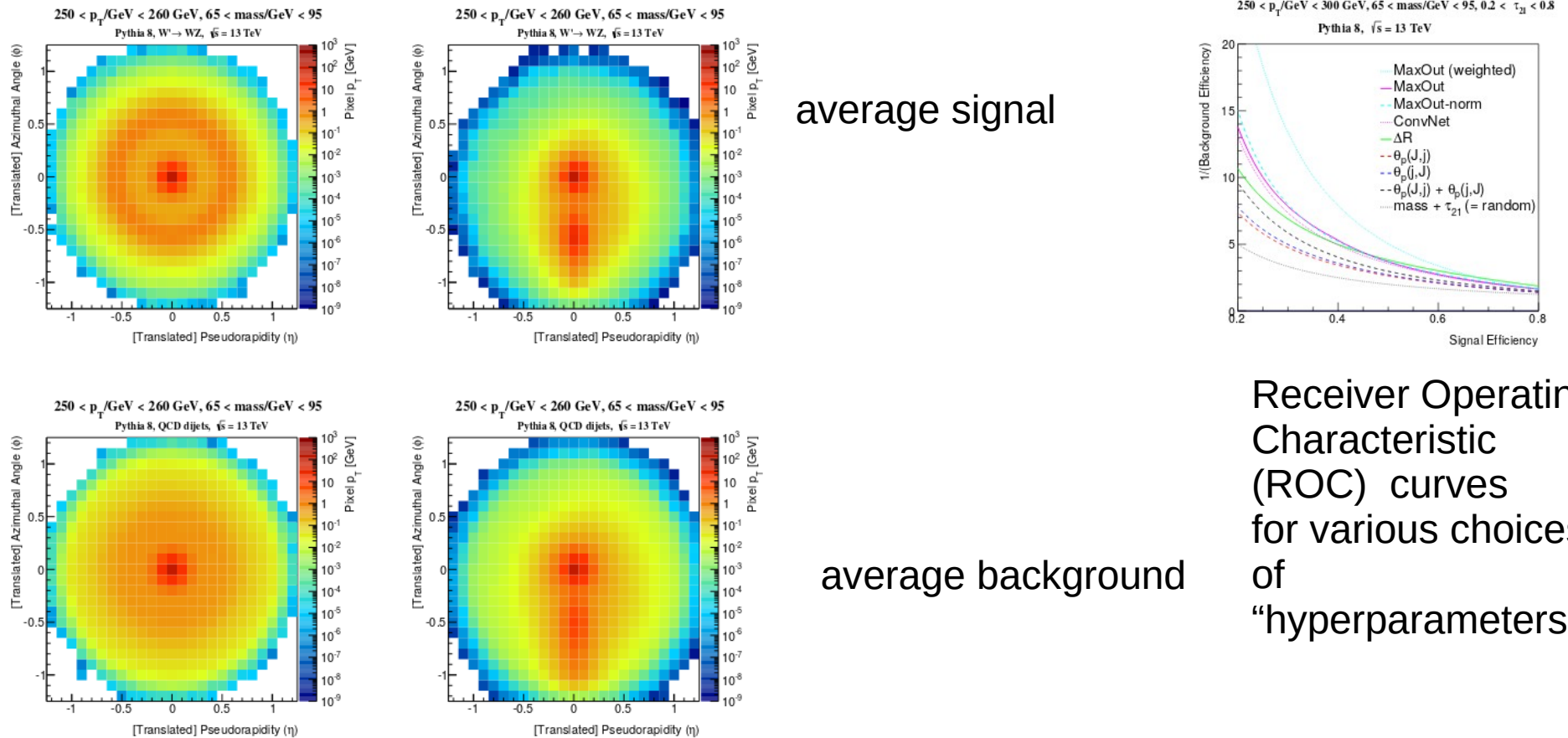


Figure 5: The convolution neural network concept as applied to jet-images.

Application in particle physics: jet images



average signal

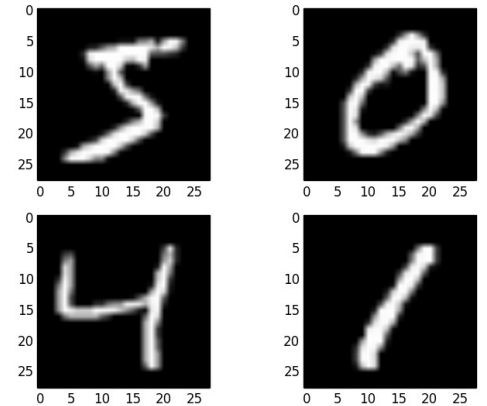
average background

Receiver Operating Characteristic (ROC) curves for various choices of “hyperparameters”

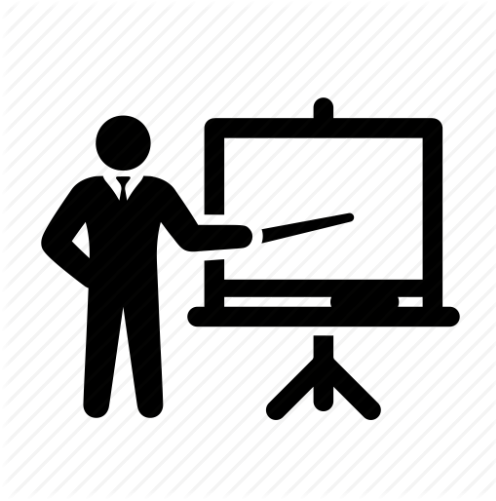
Figure 2: The average jet image for signal W jets (top) and background QCD jets (bottom) before (left) and after (right) applying the rotation, re-pixelation, and inversion steps of the pre-processing. The average is taken over images of jets with $240 \text{ GeV} < p_T < 260 \text{ GeV}$ and $65 \text{ GeV} < \text{mass} < 95 \text{ GeV}$.



Exercise 3 – CNN



- See [03_cnn.ipynb](#) for skeleton code you may use for a quick start.
- Run the skeleton code on the MNIST dataset of handwritten digits. It has one convolutional layer, which should already give you good results (accuracy around 98%).
- Add another convolutional layer. With a little tweaking you should get the accuracy around 99%.
- Play around with all hyperparameters. Can you get an accuracy above 99.25%?

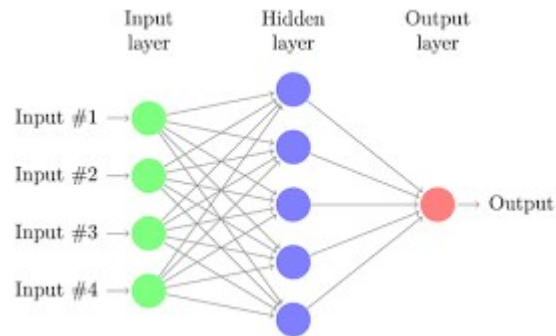


Lesson 4



- Generative models

Architectures we defined so far implemented classifiers and regressors: **Discriminative models!**



Loosely speaking, they compute [aspects of]: $p(y|x)$

Can we somehow turn the information flow around, from the output to the input, to synthesize input, computing $p(x|y)$?

Yes!

Option #1: adversarial setups

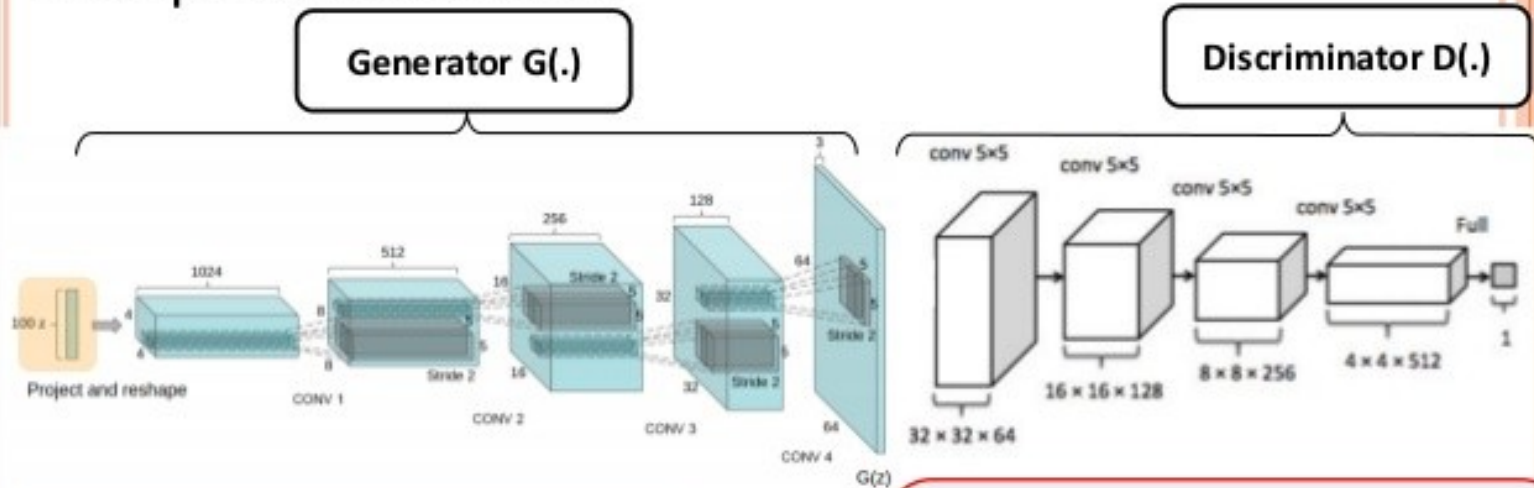
Networks can be set up to compete against each other in an adversarial setup: generative adversarial networks (GANs)

One machine is set up as a generator, it learns to generate fake data, e.g. images that resemble Picasso paintings.

Another machine learns to discriminate the real Picassos from fakes.



Example Architecture:



Generator Goal: Fool $D(G(z))$
i.e., generate an image $G(z)$ such that
 $D(G(z))$ is wrong, i.e., $D(G(z)) = 1$.

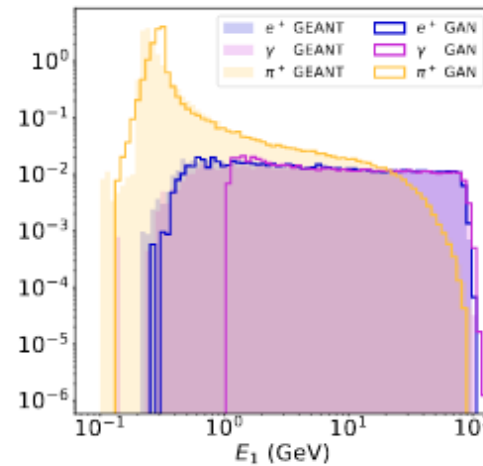
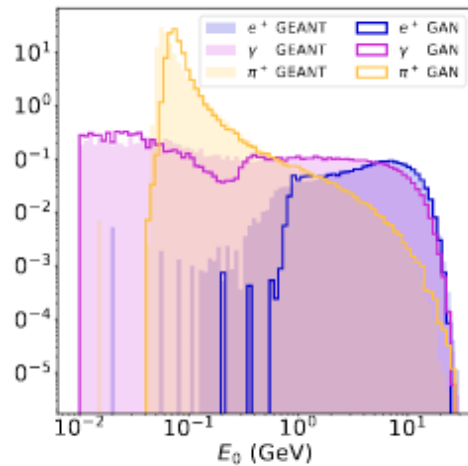
Discriminator Goal: discriminate
between **real** and **generated** images
i.e., $D(x) = 1$, where x is a **real** image
 $D(G(z)) = 0$, where $G(z)$ is a **generated**
image.

- **Conflicting goals.**
- Both goals are **unsupervised**.
- Optimal when $D(\cdot) = 0.5$ (i.e., cannot tell the difference between real and generated images) and $G(z)$ learns the training images distribution.

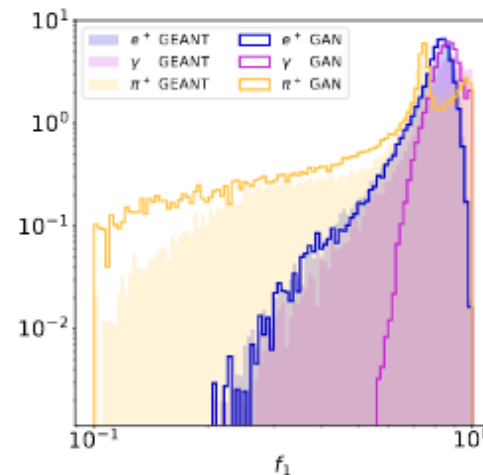
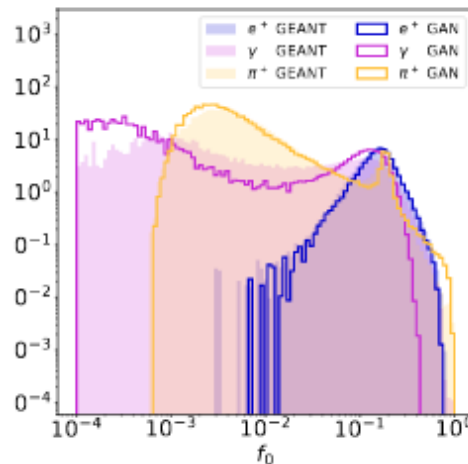


Example: the “CycleGAN”, a type of a conditional GAN, changing the condition horse → zebra

Application in Particle Physics: Fast simulation of ATLAS calorimetry (CALO GAN)

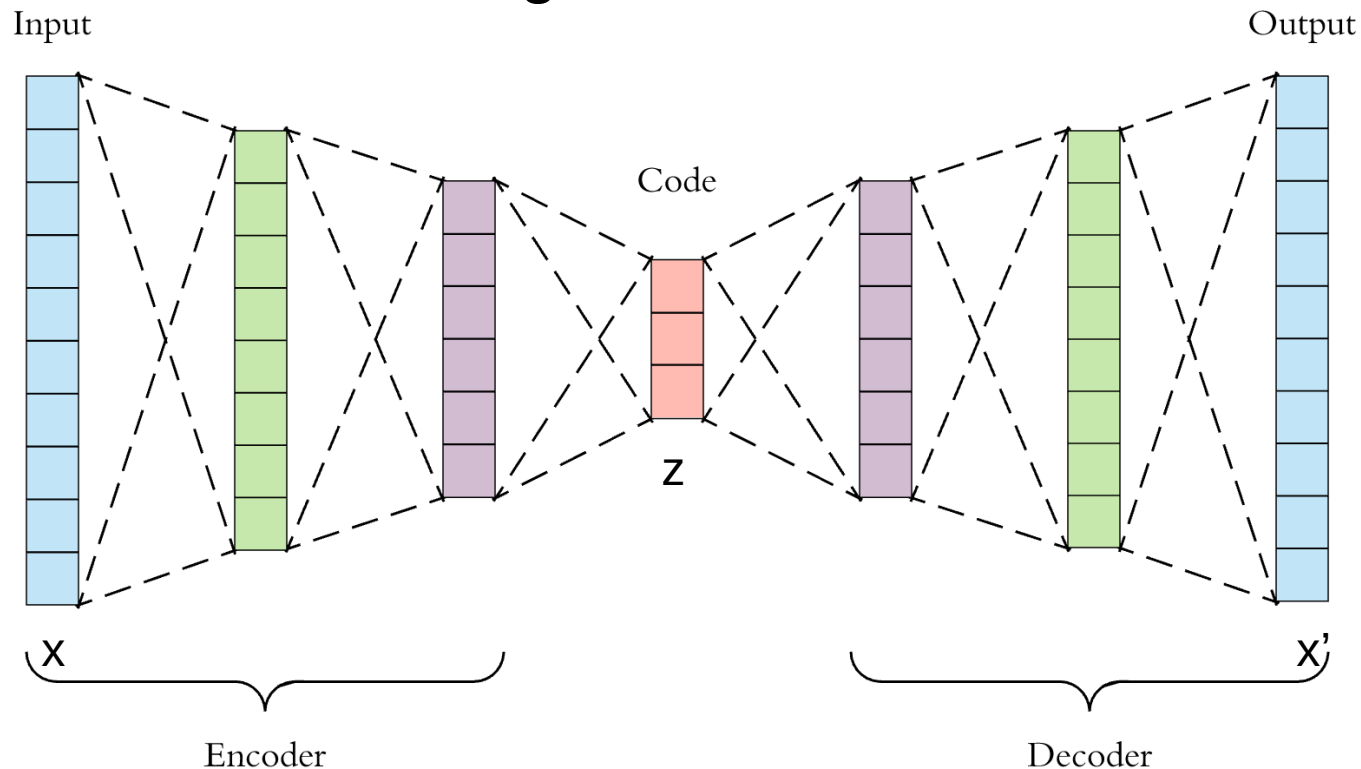


Comparison of shower
shape variables, Calo
GAN versus full Geant 4
simulation



Option #2: Autoencoders – a.k.a. “understanding is compression”

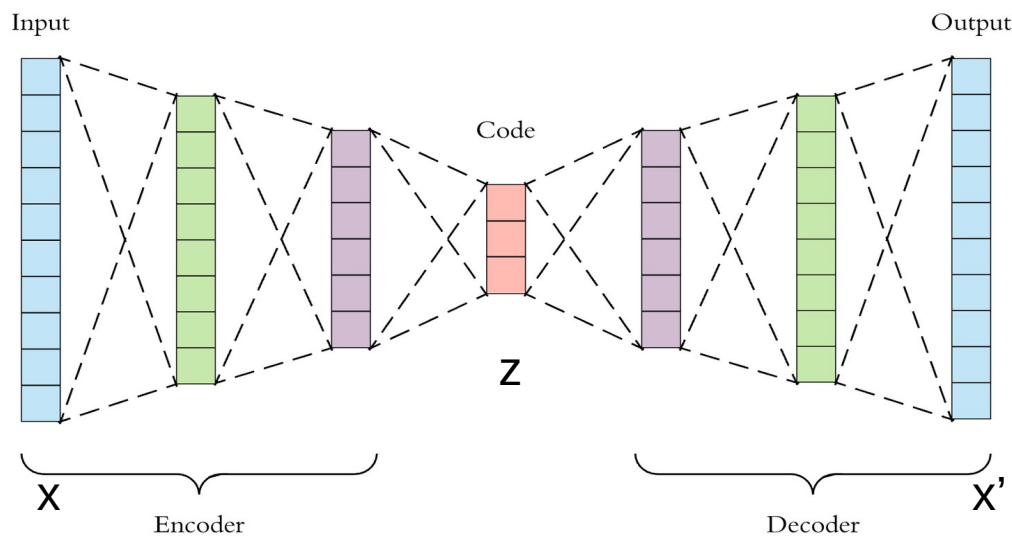
Consider the following network architecture:



With a loss function: $L = (\mathbf{x} - \mathbf{x}')^2$

Autoencoders – a.k.a. “understanding is compression”

Consider the following network architecture:



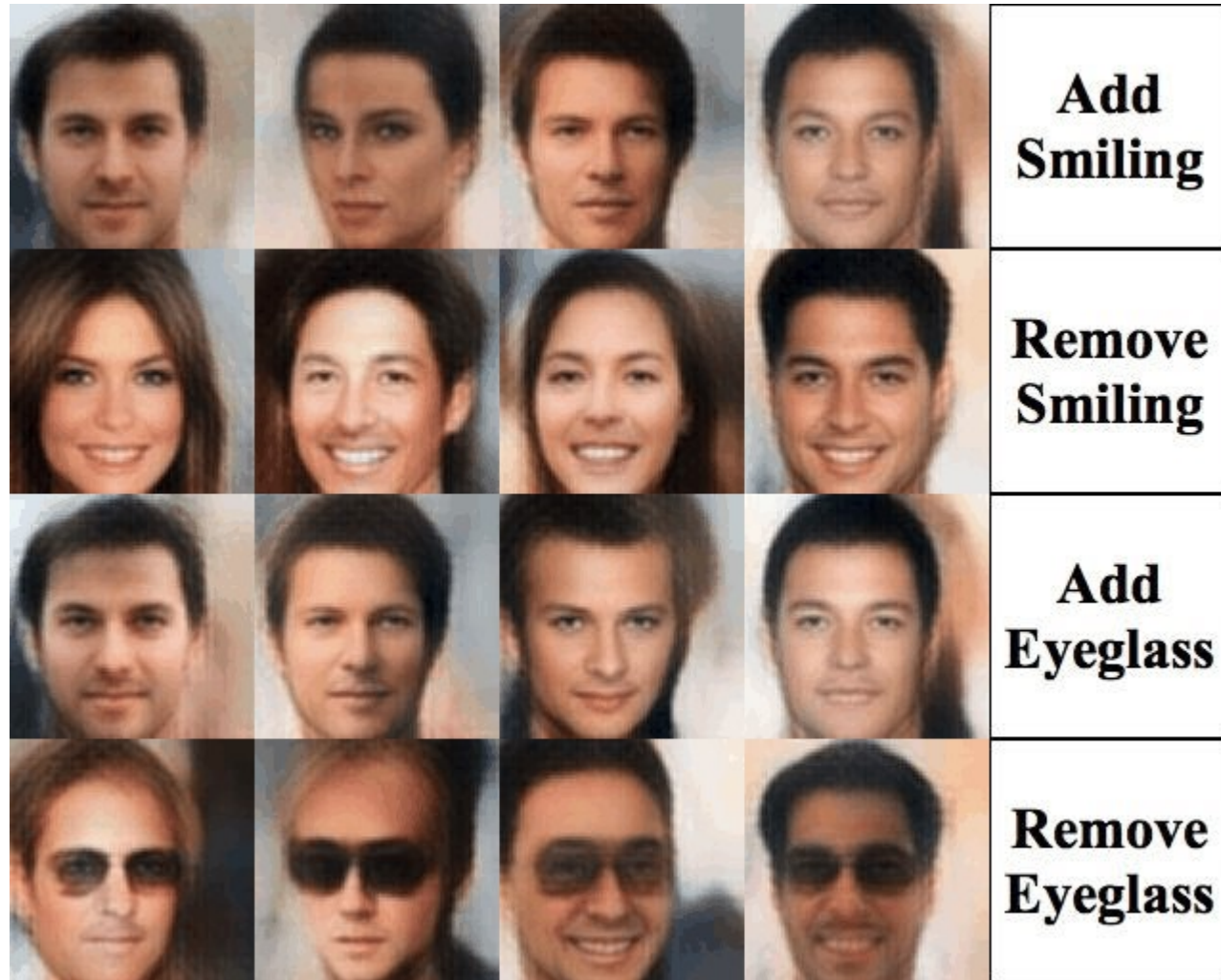
With a loss function: $L \propto (\mathbf{x} - \mathbf{x}')^2$

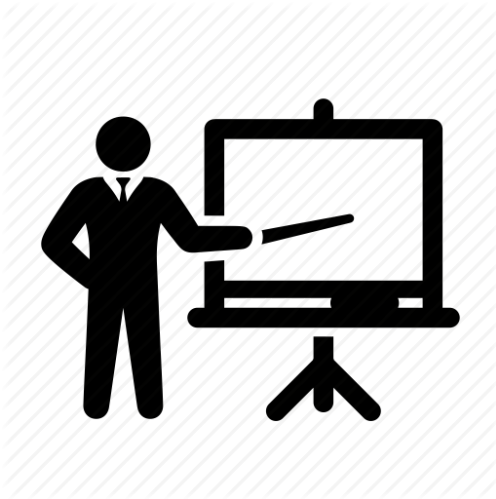
The network learns to reproduce its own input!

\mathbf{z} becomes a lower-dimensional representation (a “code”) of a higher-dimensional \mathbf{x}

Applications: Lossy compression, denoising data. But cannot yet produce fake, synthetic data. Why? Because the stochastic noise is not yet part of the model!

Example: Conditional Variational Autoencoder applied to “CelebA” dataset





Lesson 5

- Recursive and recurrent neural networks
- explanation methods
- Bayes networks and errors

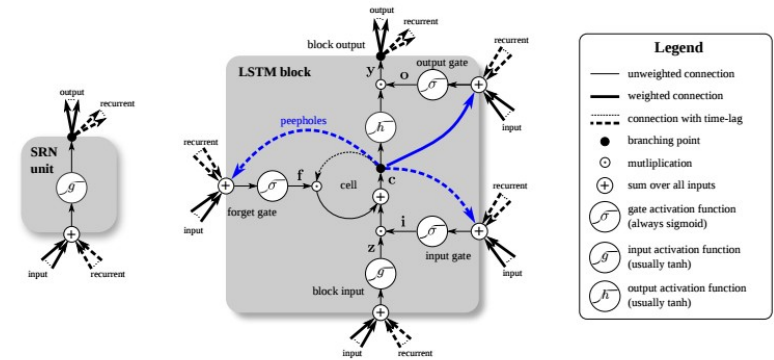
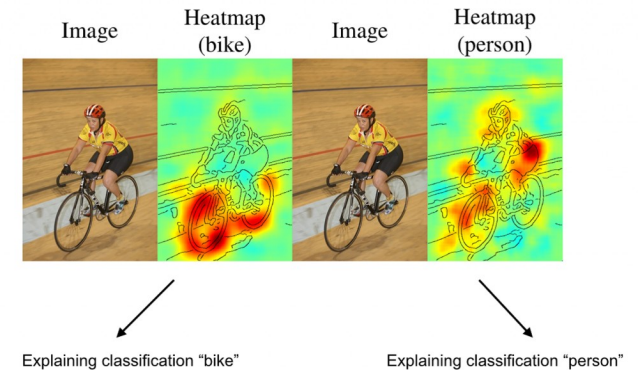


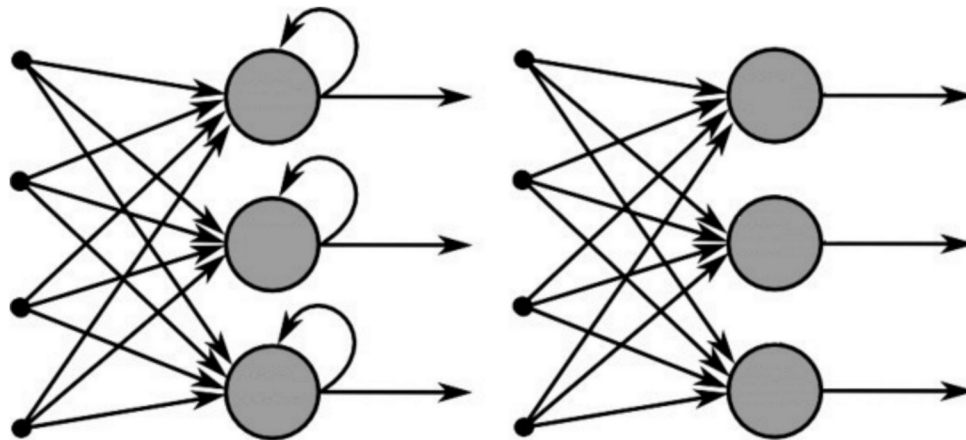
Figure 1. Detailed schematic of the Simple Recurrent Network (SRN) unit (left) and a Long Short-Term Memory block (right) as used in the hidden layers of a recurrent neural network.



Recurrent networks

How can we apply neural networks on variable-sized time series like text or speech, where the prediction of the current input must depend on the past “events”?

By making the network “recurrent”:

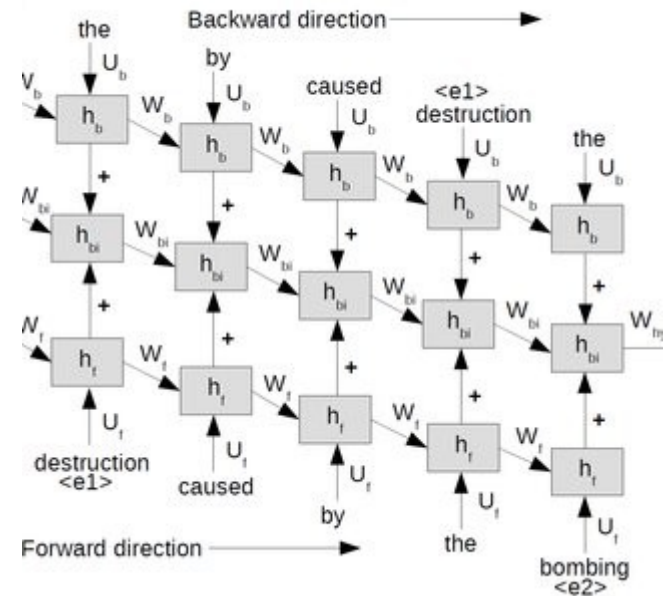
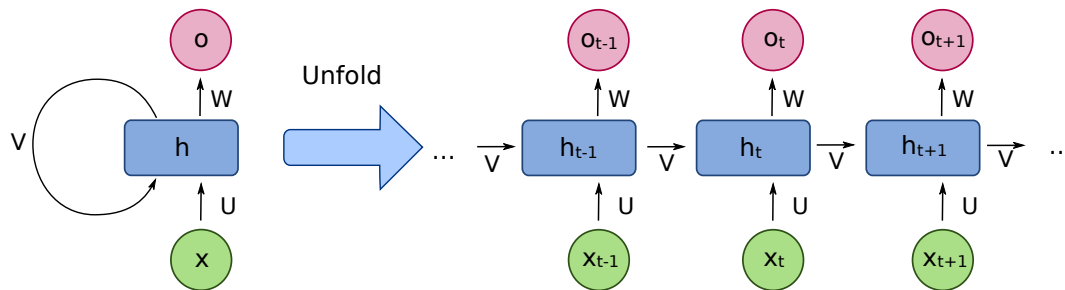


Recurrent Neural Network

Feed-Forward Neural Network

Recurrent networks

Note that one can always “unfold” recurrent networks to turn them into (complicated) feed-forward networks with fixed record lengths:



Recurrent networks

These types of networks had the problem, that the gradient quickly vanished when “propagated back through time”. The Long Short Term Memory (LSTM) solved that.

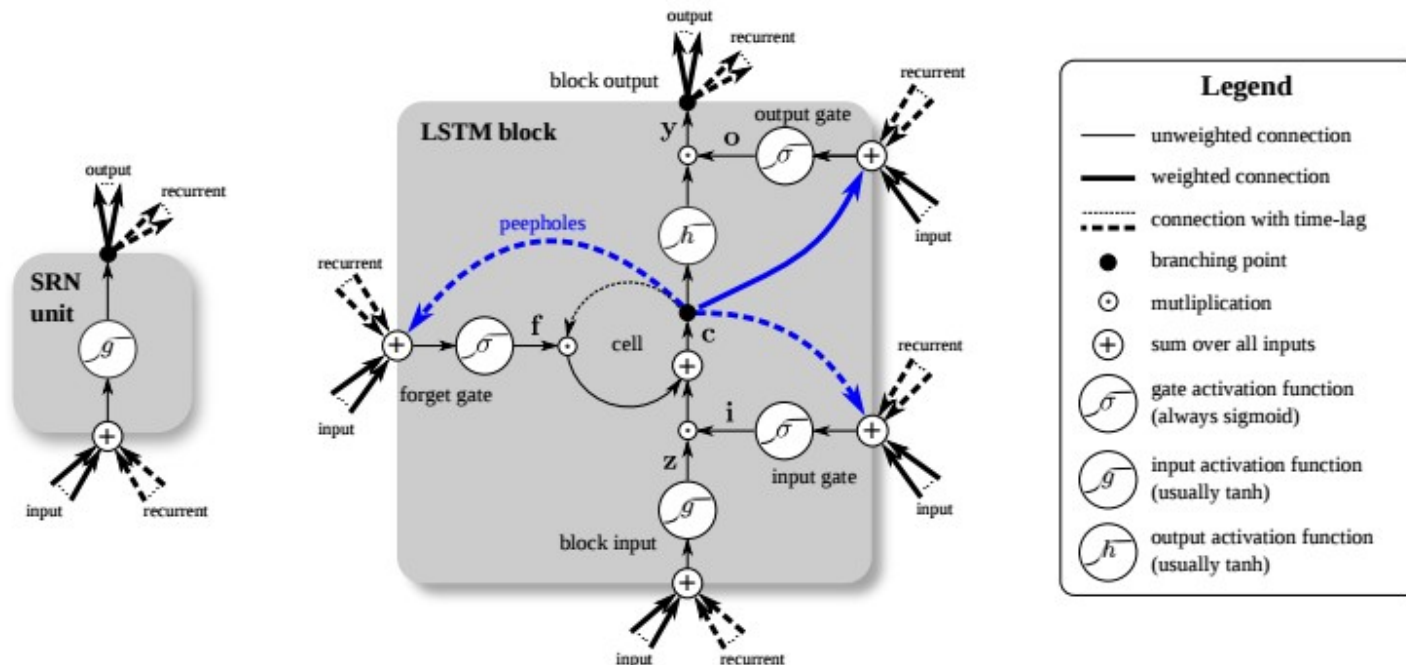
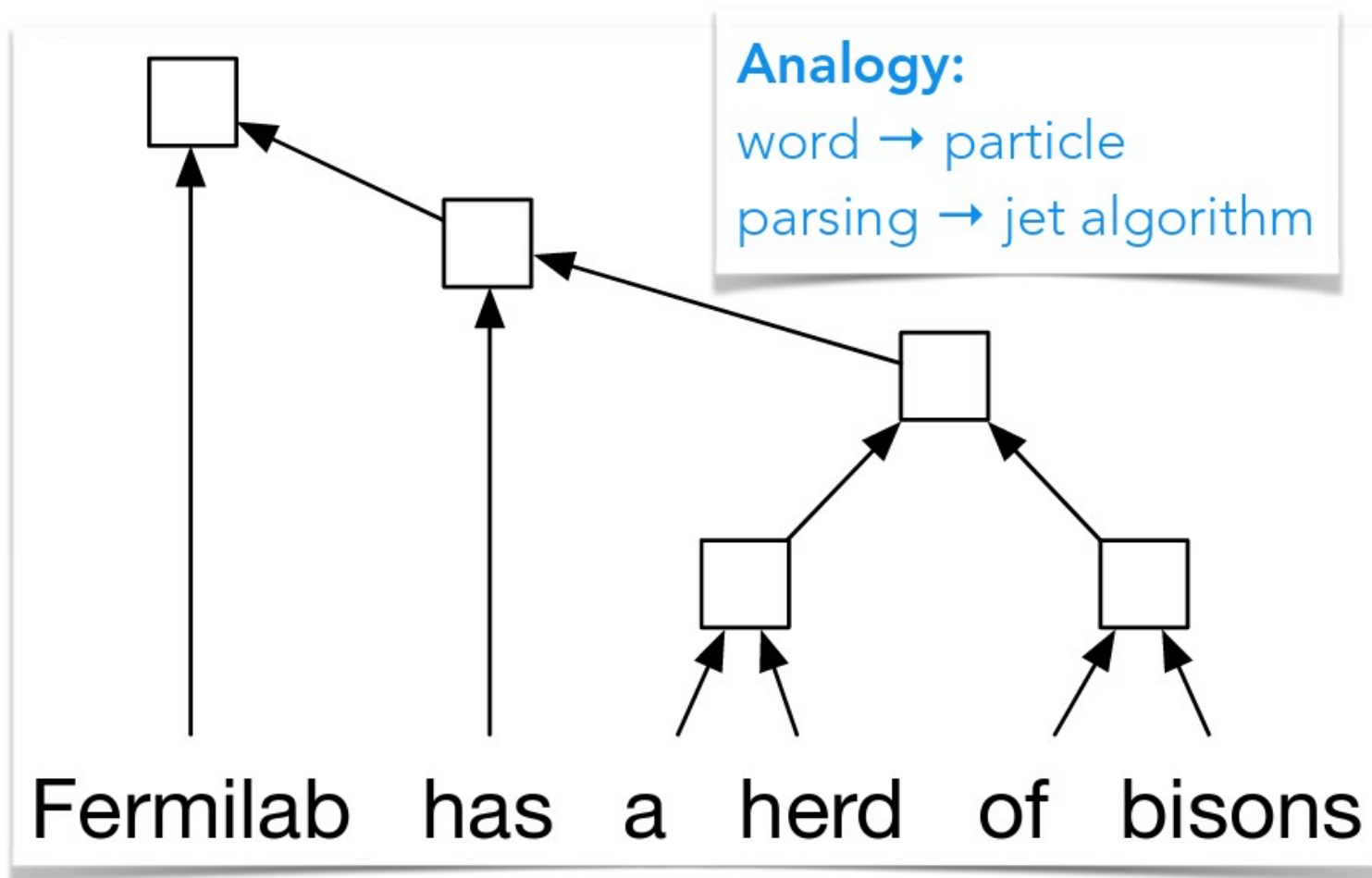


Figure 1. Detailed schematic of the Simple Recurrent Network (SRN) unit (left) and a Long Short-Term Memory block (right) as used in the hidden layers of a recurrent neural network.

Application (of a similar class of algorithms) in particle physics: jet classification

QCD-aware recursive neural networks for jet physics. (Recurrent = special version of recursive)

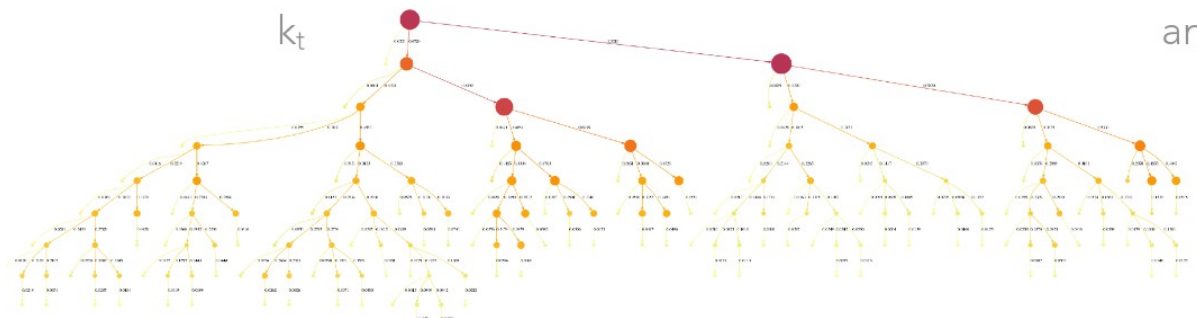


See also talk at [DS@HEP 2017 workshop](#)

<https://arxiv.org/pdf/1702.00748.pdf>⁷⁷

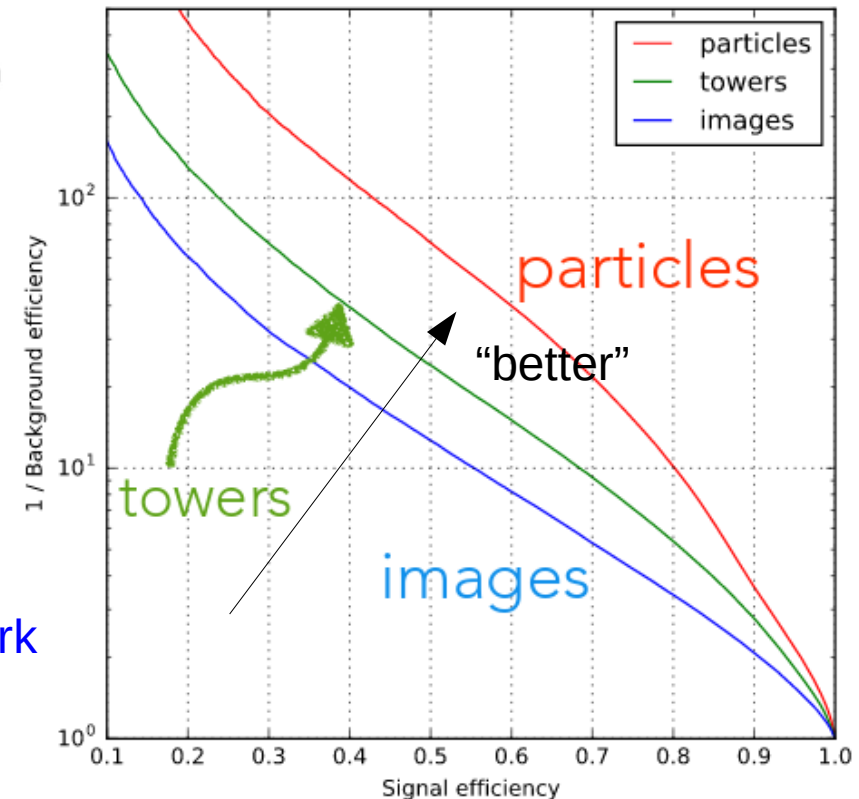
Application (of a similar class of algorithms) in particle physics: jet classification

“QCD-aware recursive neural networks for jet physics”



Visualisation of k_t jet algorithm.

“particles”: recursive network
“images”: convolutional network



ROC curve of W jet tagging example, comparison with jet images algorithm

See also talk at [DS@HEP 2017 workshop](#)

<https://arxiv.org/pdf/1702.00748.pdf>⁷⁸

Explanation methods

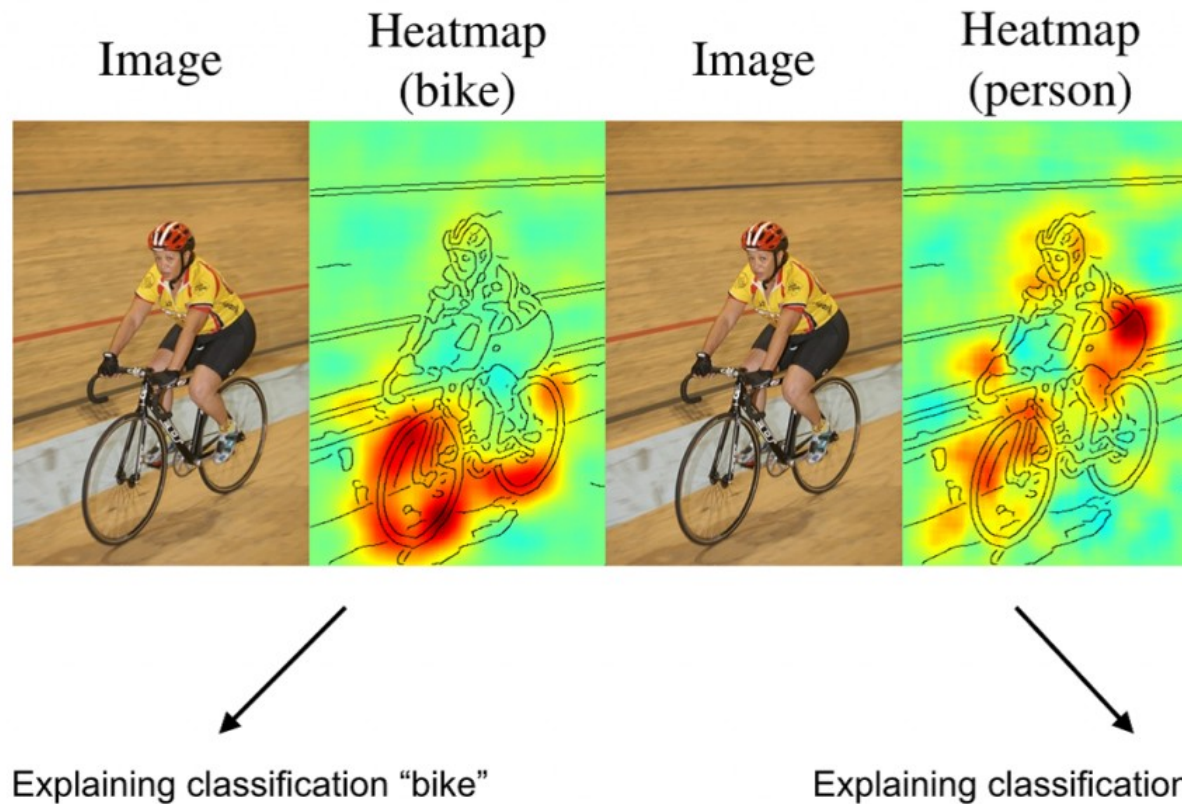
What is the machine learning? How can we “debug” deep networks?

- There are several methods on the market.

Simple class of algorithms for convolutional networks: which pixels of a picture most contributed to a prediction? Which pixels were in contradiction to prediction? “Sensitivity analysis”, or (for pictures) heatmaps (based on “input gradient” df/dx_i).

Explanation methods

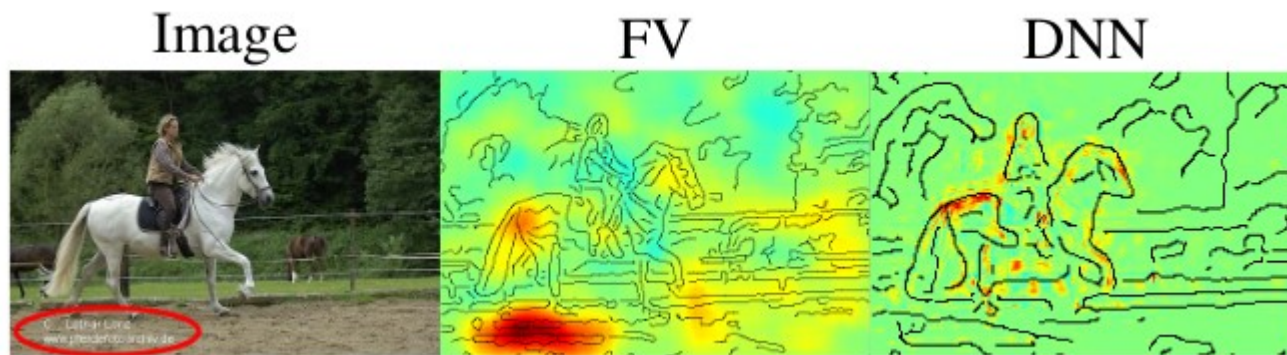
What is the machine learning? How can we “debug” deep networks? Heatmaps:



Explanation methods

What is the machine learning? How can we “debug” deep networks? Heatmaps may sometimes help discover learned “artifacts”.

The networks learns to exploit undesirable correlations



Machine learned to identify horses by the imprint.

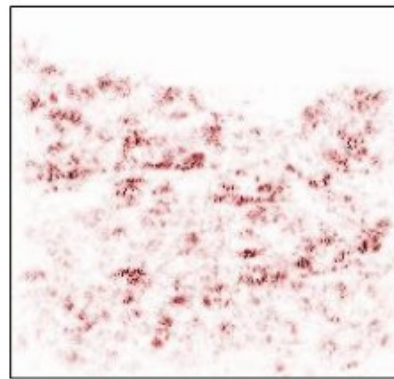
Explanation methods

But sensitivity analyses fail at grasping “the bigger picture”.

What makes the picture below a road with cars?



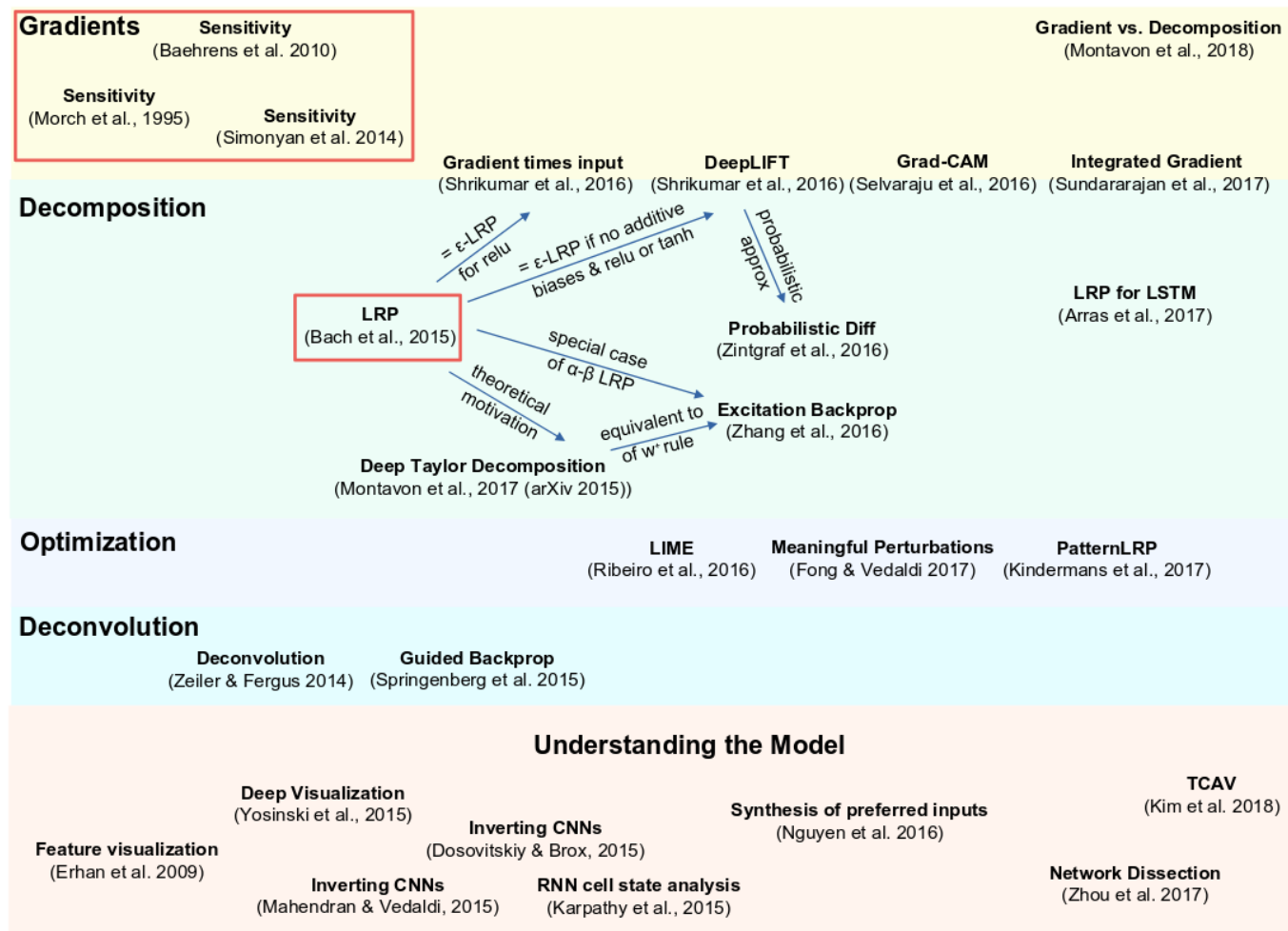
$$R_i = \left(\frac{\partial f}{\partial x_i} \right)^2$$



Problem: sensitivity analysis does not highlight cars

Explanation methods

Historical remarks on Explaining Predictors



An entire industry of explanation methods has emerged in only 3 years!!
 Very hard to keep up and identify the algorithms that are interesting for us.

See e.g. <http://heatmapping.org>,
http://heatmapping.org/slides/2018_CVPR_1.pdf

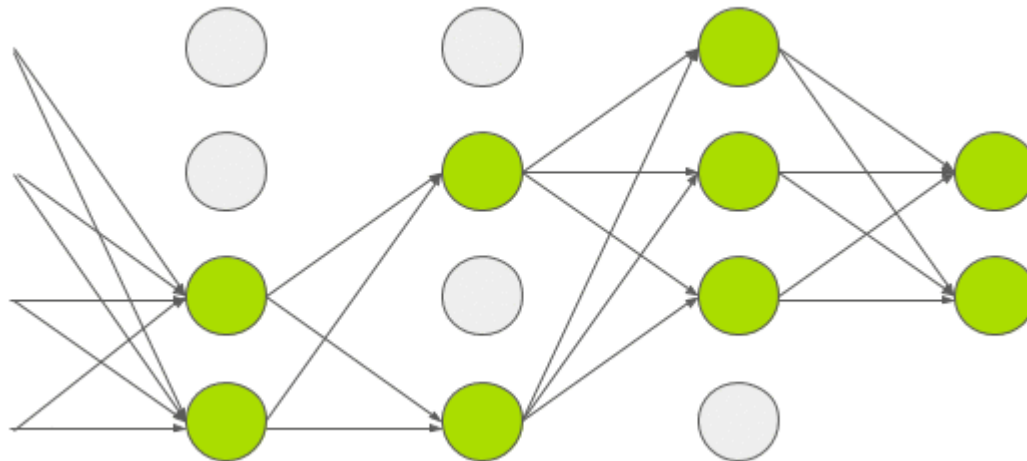
Bayesian Neural Networks and Monte Carlo dropout

Uncertainties: ML jargon / physics jargon

- **Aleatoric uncertainty:** (“alea” in latin is “dice”)
All uncertainties related to the input data, e.g. sensor noise. In particle physics we often have a good understanding of these.
- **Epistemic uncertainty:** (epistemic = “related to knowledge”)
Uncertainties on the network weights, i.e. uncertainties that are introduced with the network – and the fact that it is trained only with finite data.
- How can we **estimate epistemic uncertainties?**

Tool #5: Bayesian neural networks and error estimation

Error estimation, the easy way: dropout!!



arXiv:1506.02142 showed that randomly applying dropout in the prediction (as opposed to dropout in training) is an approximator for the epistemic error of the prediction.

Ensembling: predict several times with dropout applied to get an estimate of the epistemic error of the network.

<https://arxiv.org/abs/1506.02142>

challenge!

Backup

Recommended literature

- [Coursera course](#) by Andrew Ng
- [Deep learning](#), book by Ian Goodfellow et al.
see also <http://www.deeplearningbook.org/>
- Tensorflow [tutorials](#)
- Pytorch [tutorials](#)
- [Deep learning](#) course at univie by Philipp Grohs