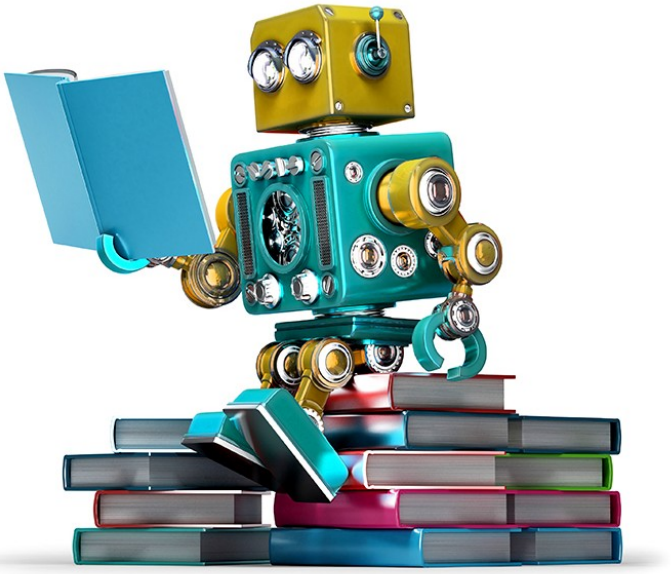
Doktoratskolleg Particles and Interactions

Machine Learning for Particle Physics –

A Hands-On Introduction

Wolfgang Waltenberger

DKPI Summer School,
Sept 17 - 21, 2018



HEPHY
INSTITUT FÜR HOCHENERGIEPHYSIK

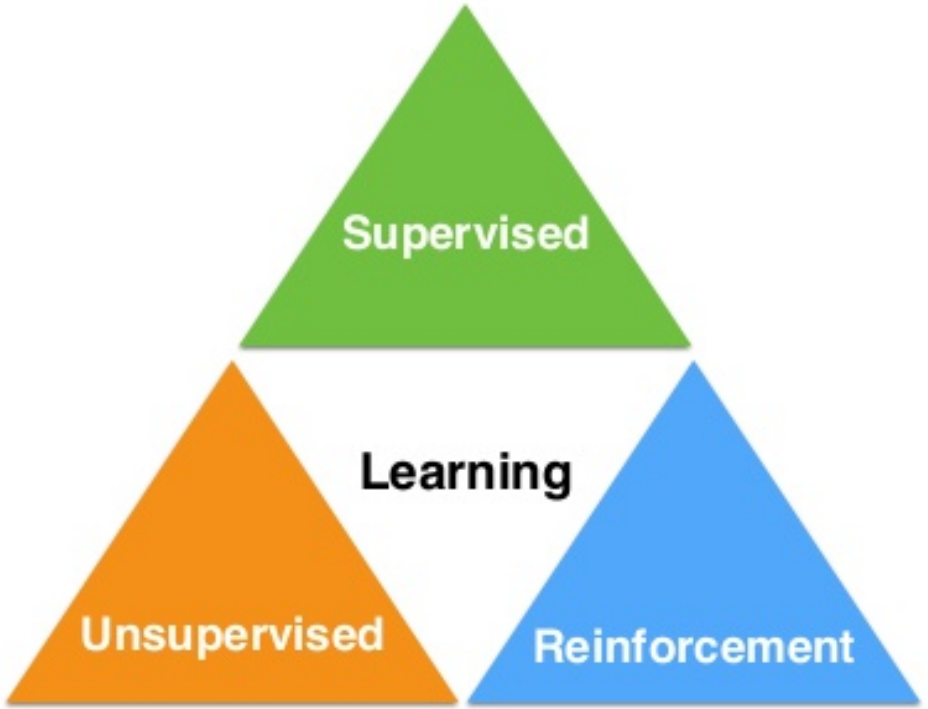
Syllabus

- Introduction
- Lesson 1: Logistic Regression
- Exercise 1
- Lesson 2: Multi-Layer Perceptron
- Exercise 2
- Lesson 3: Convolutional Neural Networks
- Exercise 3
- Lesson 4: Beyond
  (recurrent/recursive neural networks, auto encoders, generative adversarial networks, explanation methods)
- Workshop Challenge

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 pairs of two. 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 “summer school challenge”. The winning team gets a certificate and a small prize.

- Labeled data
- Direct feedback
- Predict outcome/future



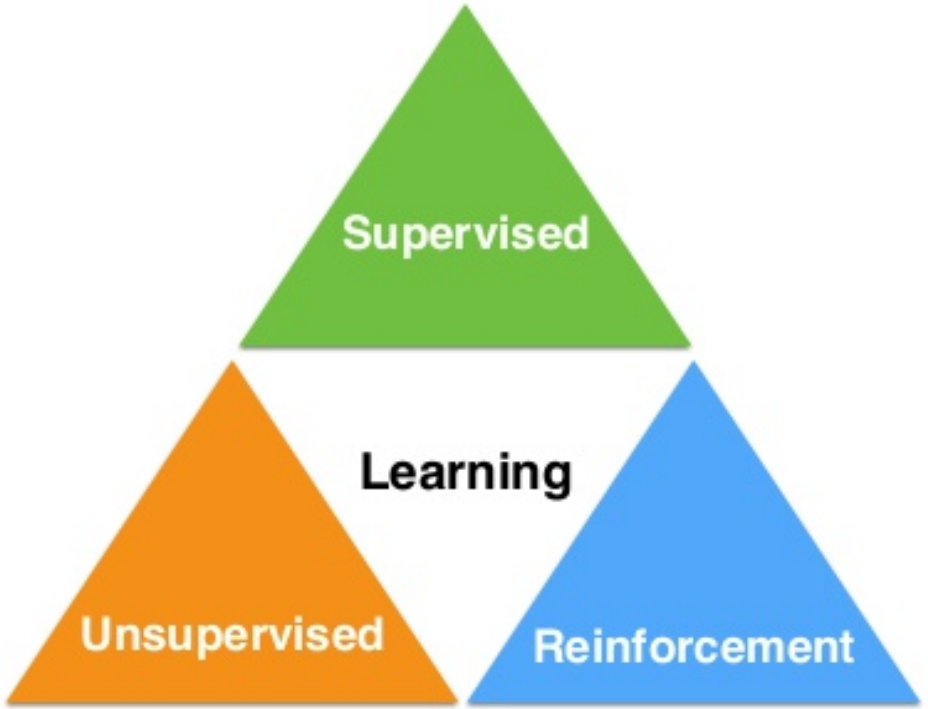
- No labels
- No feedback
- "Find hidden structure"

- Decision process
- Reward system
- Learn series of actions

In this lecture series:

mostly this

- Labeled data
- Direct feedback
- Predict outcome/future



a bit of this

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

none of this

- Decision process
- Reward system
- Learn series of actions

Supervised versus unsupervised

Naming convention:

x = data (a.k.a. “features” 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 → learn p(y|x)

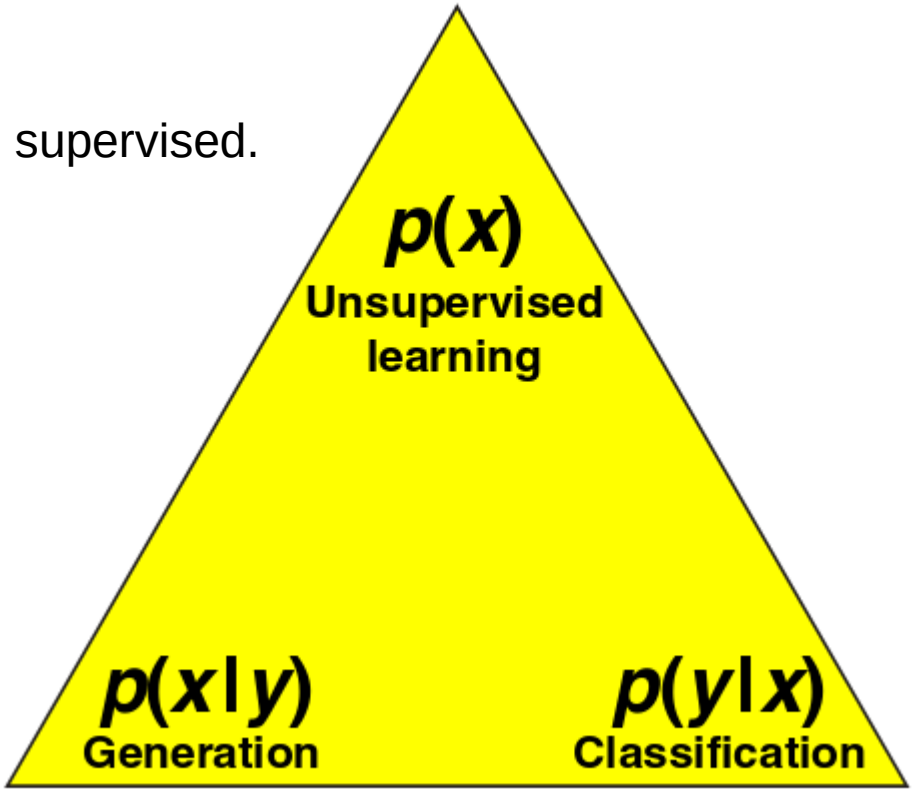
If all possible y labels form a finite set: classification
If all possible y labels are form an infinite set: regression

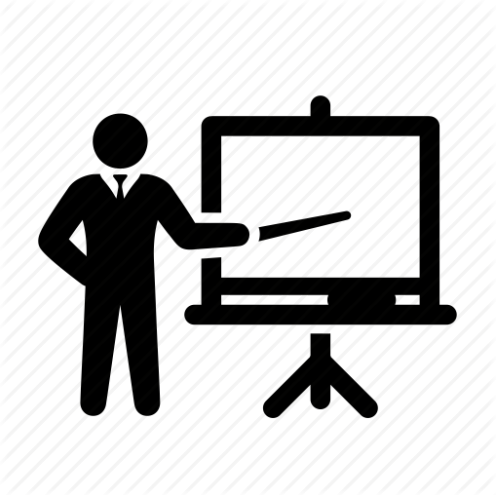
Unsupervised:
Usually more difficult, more vague, than supervised.

Find structure behind data: p(x).

Generate “fake” data belonging to category y: p(x|y).

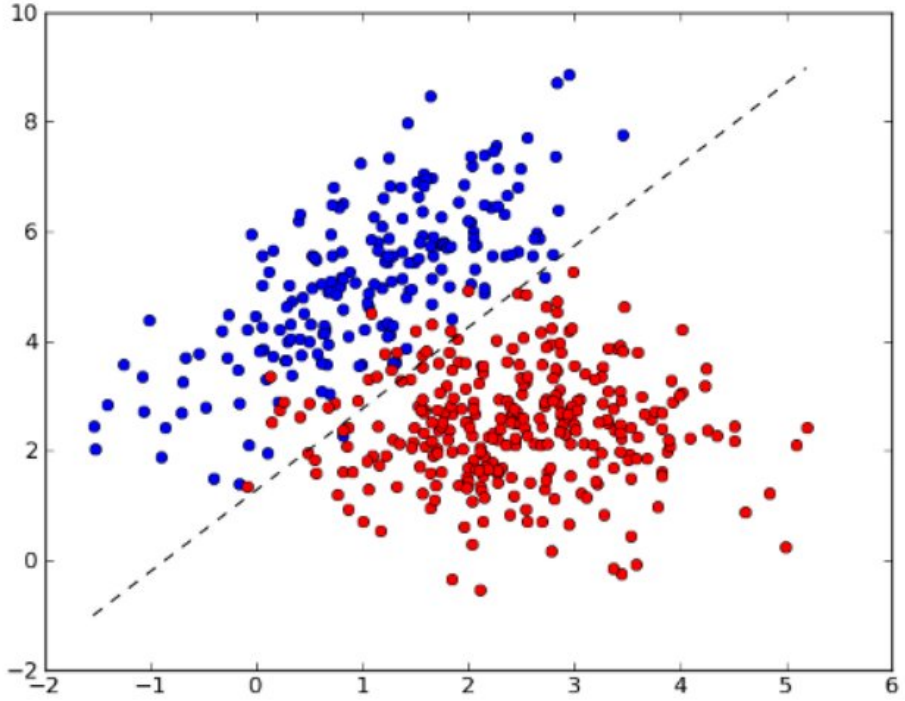
Plot from paper (written by physicists), “why does deep and cheap learning work so well”, arXiv:1608.08225





Lesson 1

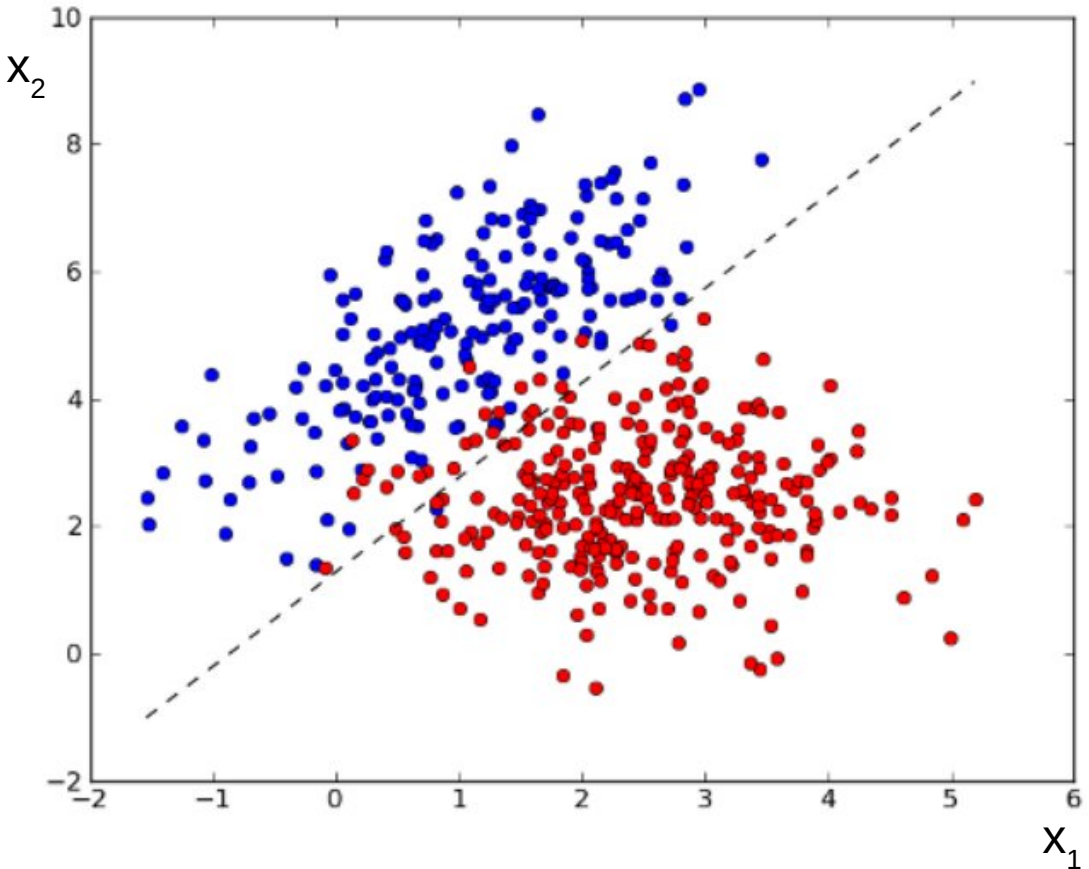
Logistic regression and linear decision boundaries



Logistic regression

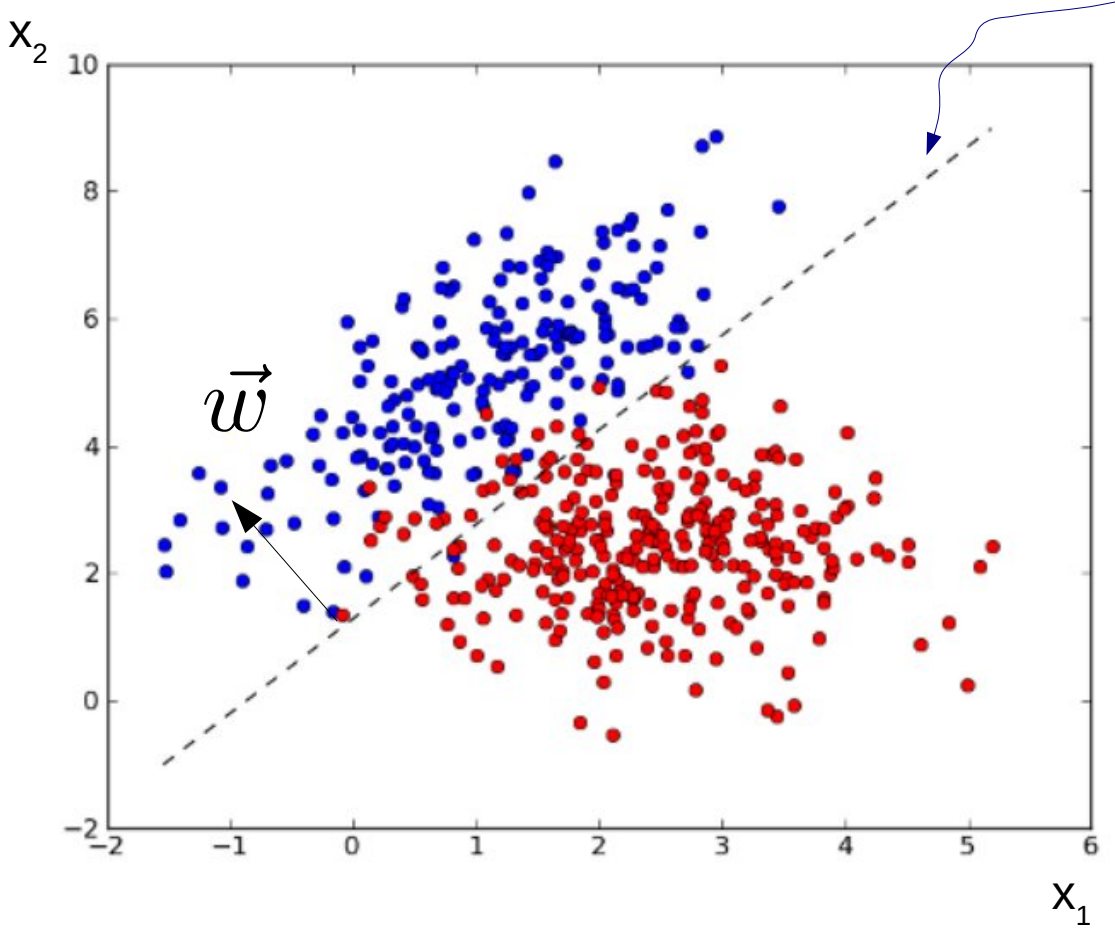
The (pedagogically) easiest way to understand neural networks is via “logistic regression”

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

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

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

Logistic regression

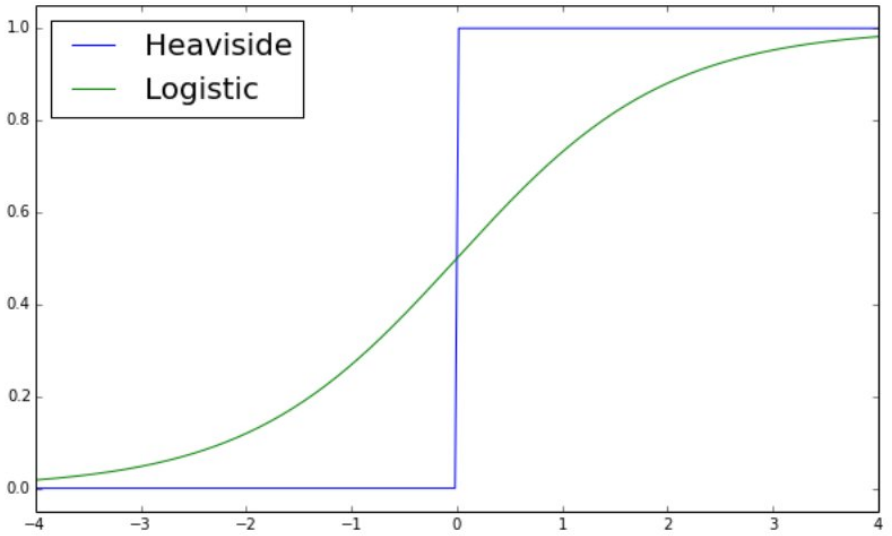
But $D(\vec{x})$ can take on any value between -∞ and +∞.
Lets map [-∞,∞] to [0,1]. Makes the handling easier. And it makes it look like a probability. One possible mapping:

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. $2\,\sigma(x) = 1 + \tanh(x/2)$



As physicists we take note that this looks a lot like the Fermi function (which also returns a probability).

Let's construct a loss function

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

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

$$p_{(blue)}(\vec{x}) = \sigma(D(\vec{x}))$$

The point is blue.
How wrong am I?

$$p_{(red)}(\vec{x}) = 1 - \sigma(D(\vec{x}))$$

The point is red.
How wrong am I?

$$L(\vec{w}) = \sum_{\text{i:points}} p_{(\text{red;blue})}(\vec{x}_i)$$

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**.

- Many other sensible choices for a loss function:

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

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

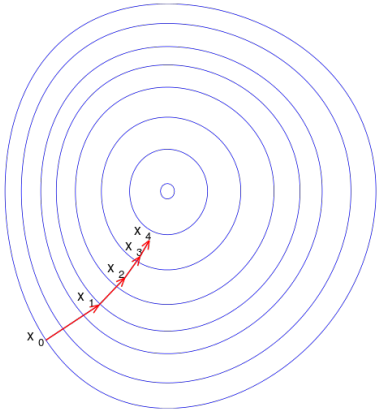
  (Binary) Cross Entropy: the expected negative log likelihood of misclassification

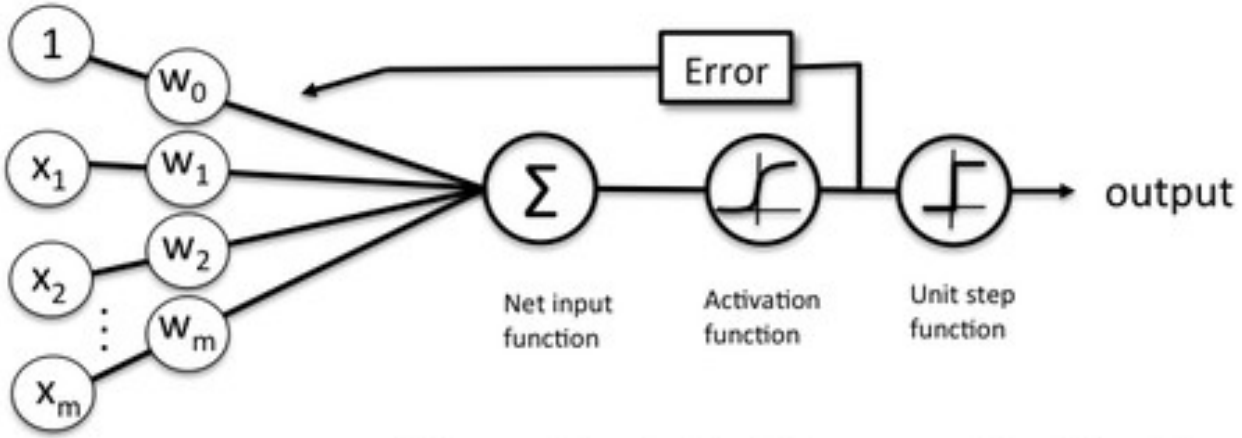
$$L(\vec{w}) = -\sum_{\text{i:points}} \ln(1 - p_{(\text{red;blue})})(\vec{x}_i)$$

Learning := gradient descent

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

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





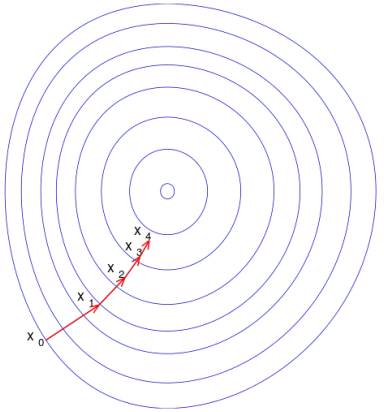
Schematic of a logistic regression classifier.

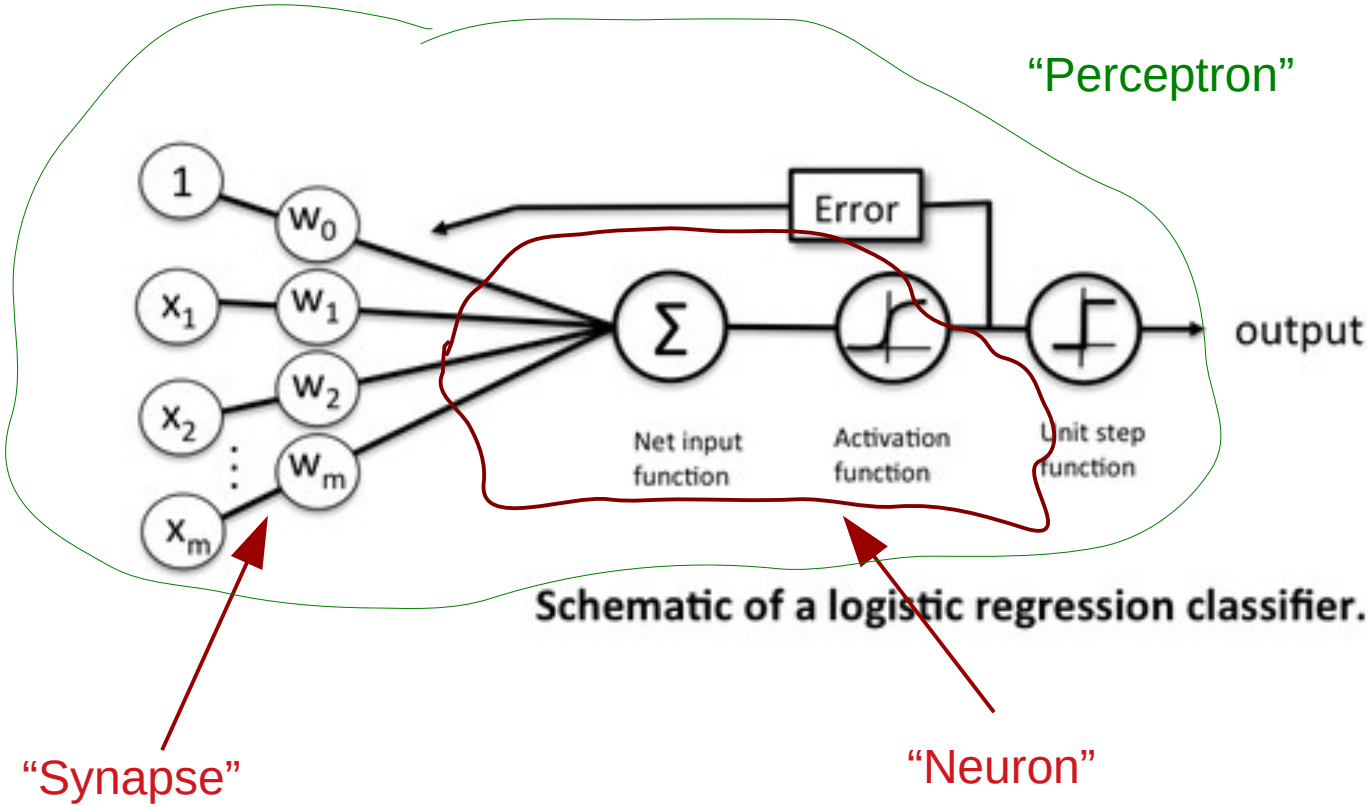
https://sebastianraschka.com/images/faq/logisticregr-neuralnet/

Learning := gradient descent

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

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





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 turn (0 < γ < 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.

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

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

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

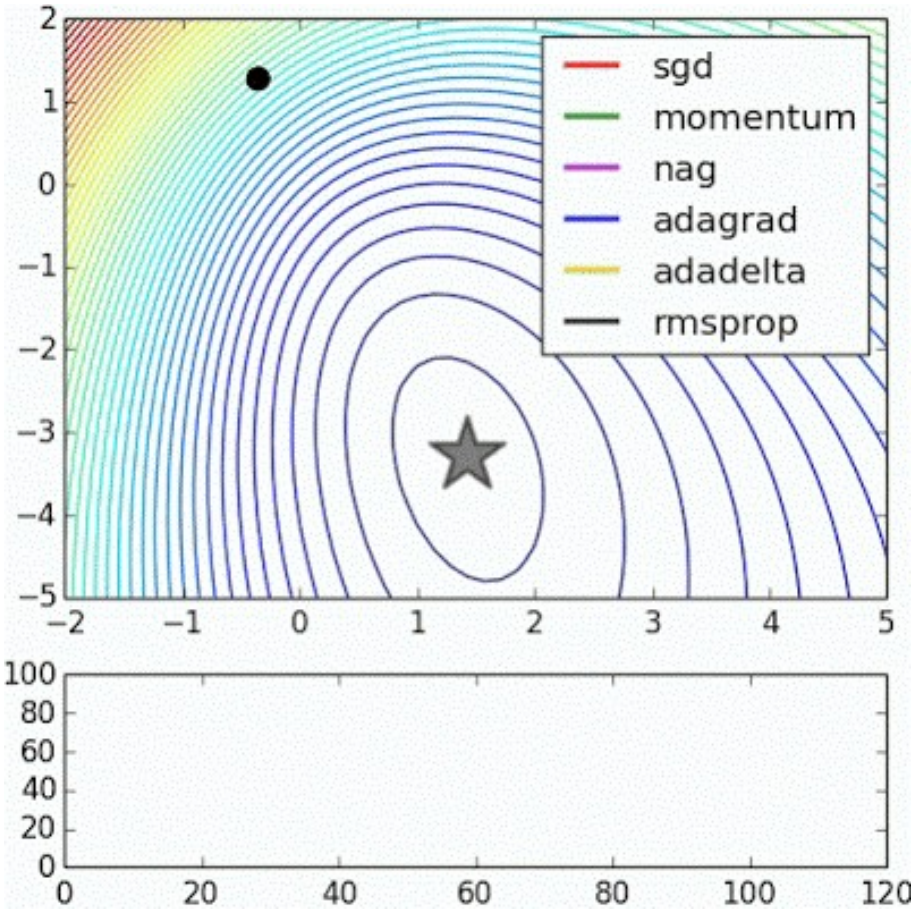
- Nadam

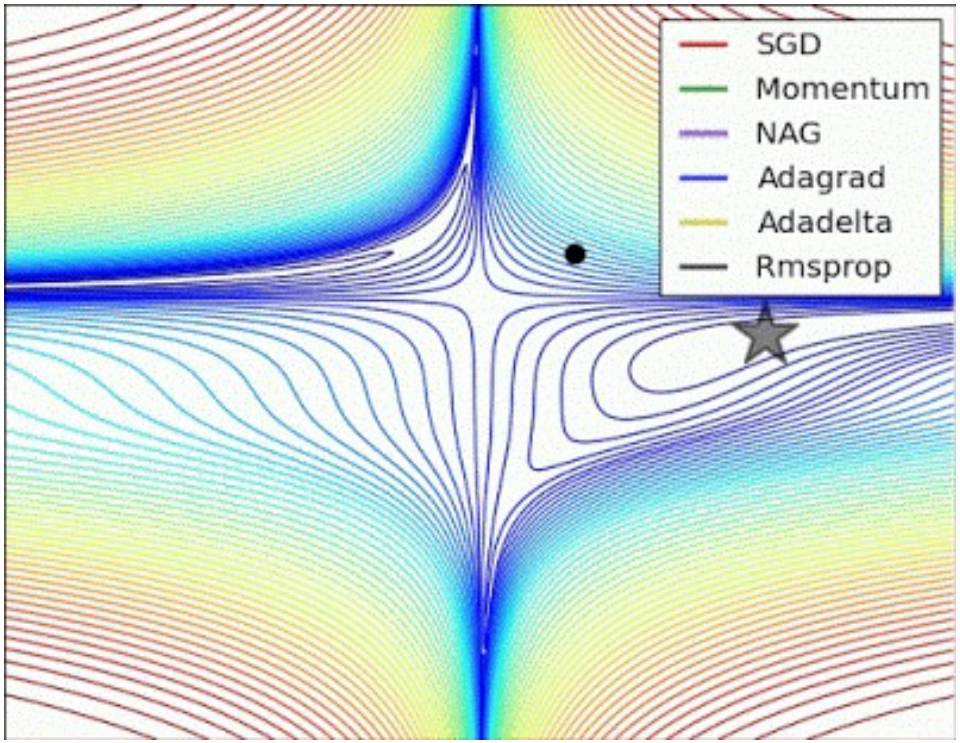
  Nesterov-accelerated Adam

- Second order optimizations (Newton, natural gradients)

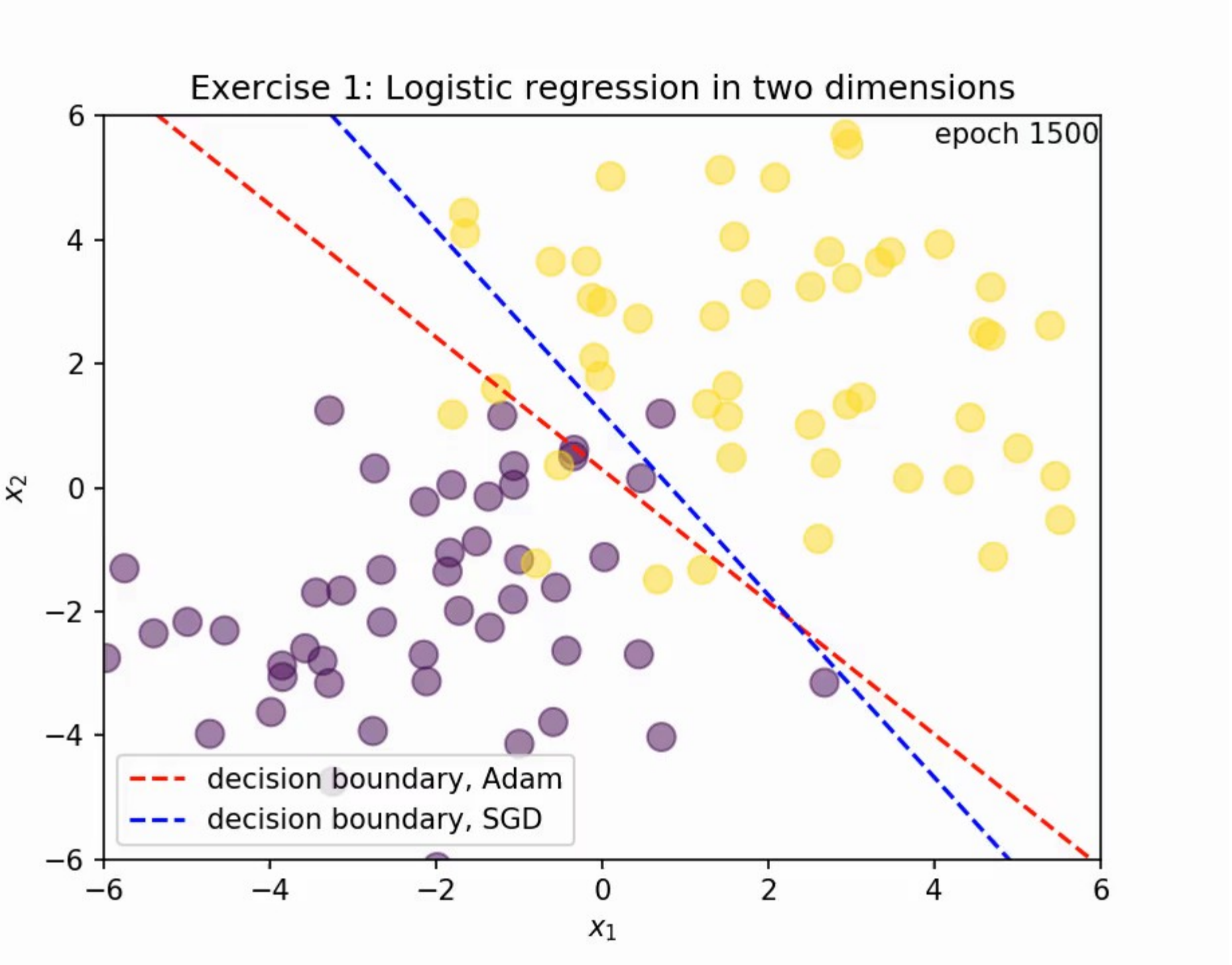
  We can also add second order derivatives of the loss function to our updates. Interesting, also theoretically (information geometry), but often computationally too expensive.

Gradient descent: methods



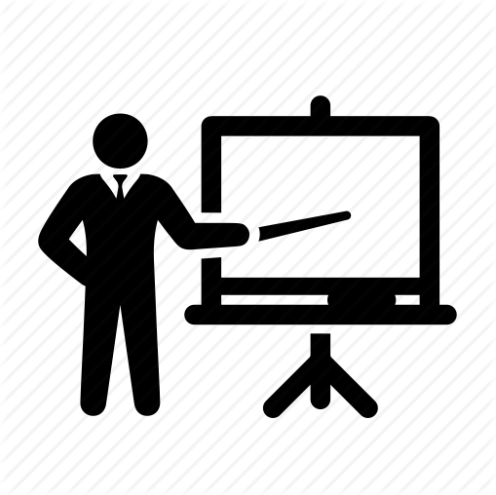


Plots taken from: http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html


Exercise 1: Logistic regression in two dimensions
epoch 1500
decision boundary, Adam
decision boundary, SGD
x_1
x_2
6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

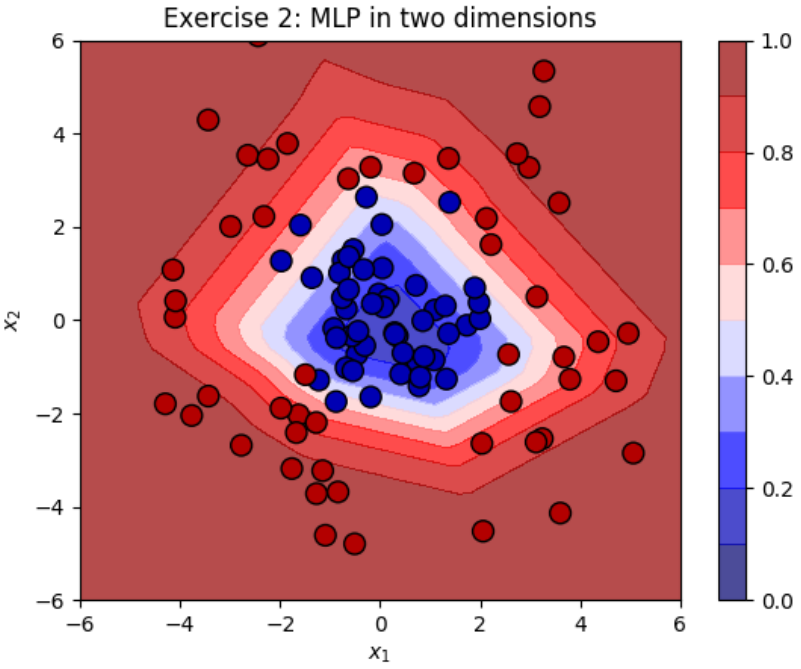
Exercise 1 – logit

- Start with 01_skeleton.py, 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.
- Download everything from http://smodels.hephy.at/dkpi/ML.zip
- All python requirements you will need are described in requirements.txt. Try ./install.sh
- Which optimizer converges the fastest?

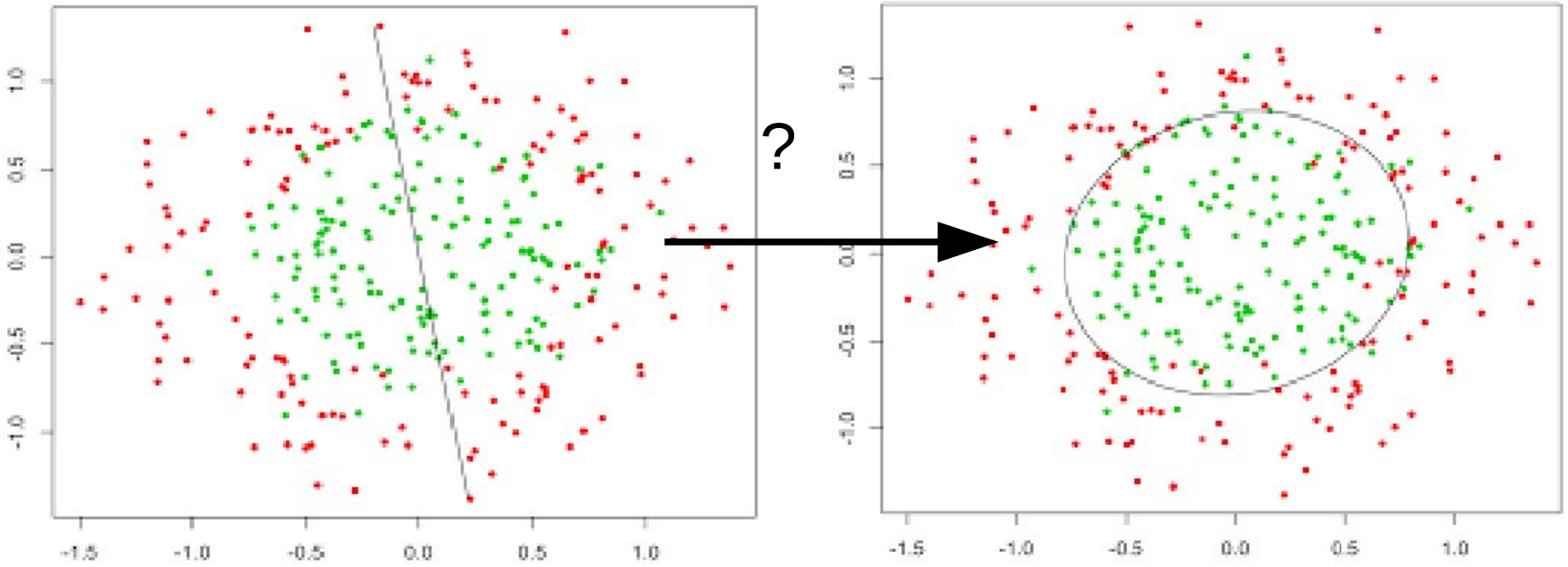


Lesson 2

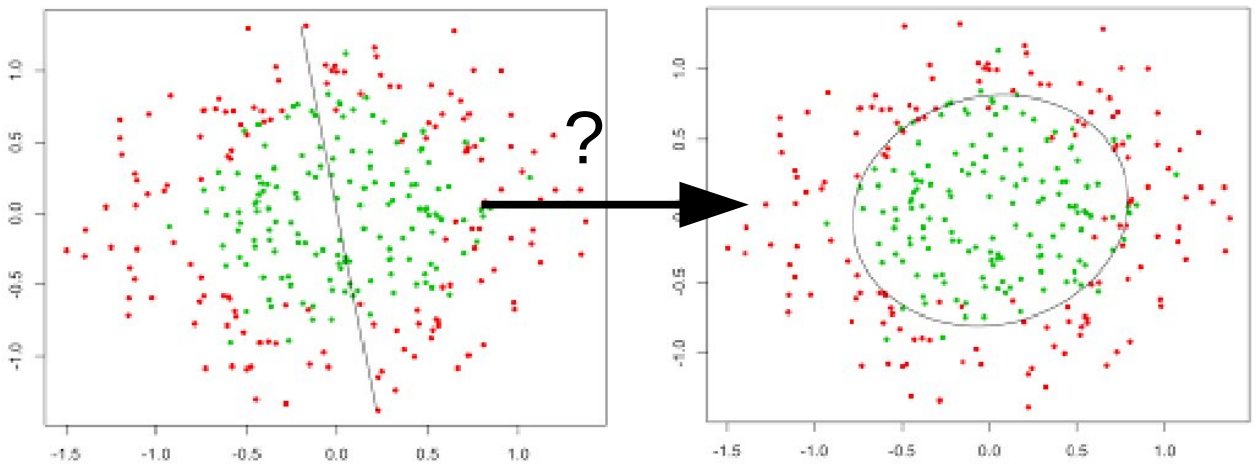
Multi-layer perceptron and non-linear decision boundaries



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

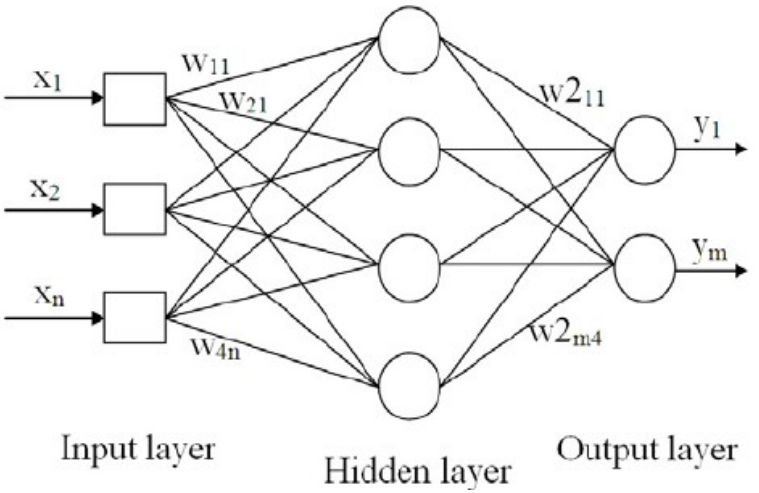


How can we go from a linear to a non-linear decision rule?



We introduce “hidden” layers of neurons!

$$p_{(blue)}(\vec{x}) = \sigma_1(D_1(\sigma_2(D_2(\vec{x}))))$$



Deep learning

How do we train a deep network?

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

Chain rule:

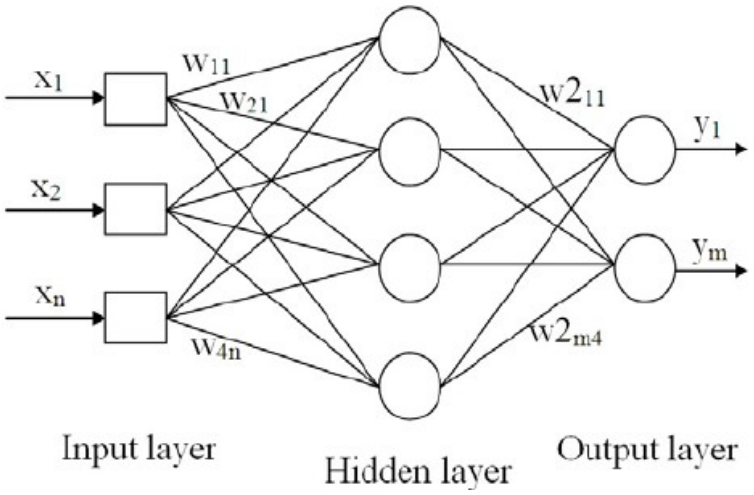
$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial o_j} \frac{\partial o_j}{\partial \mathrm{net}_j} \frac{\partial \mathrm{net}_j}{\partial w_{ij}}$$

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

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)



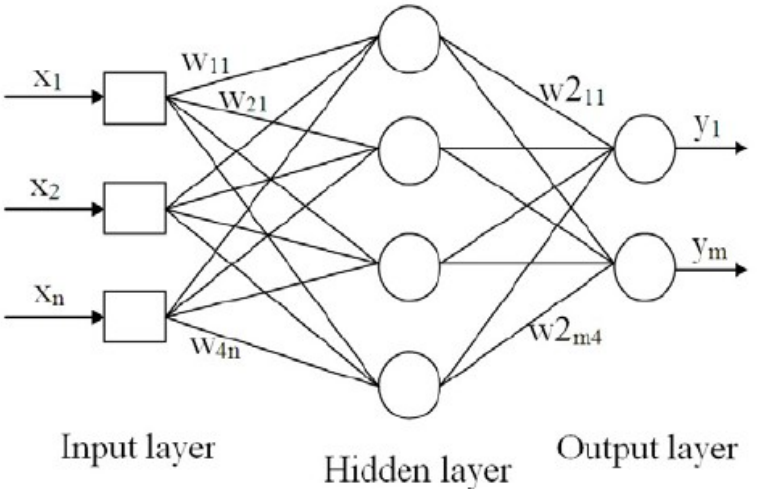
Example: backprop for MLP

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

Loss function: (mean) squared error (L2) $L(w_{ij}) \propto (t-y)^2$

where

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



Network minimizes the squared error.

Chain rule:

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

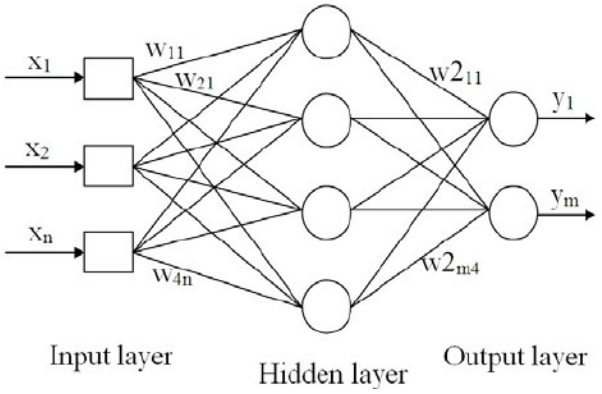
where

$$o_j = \sigma(\mathrm{net}_j) = \sigma(\sum_k w_{jk} o_k)$$

A weight update is then simply

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

Example: backprop for MLP



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

$$o_j = \sigma(\mathrm{net}_j) = \sigma(\sum_k w_{jk} o_k)$$

$$\frac{\partial net_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\left(\sum_j w_{jk} o_k\right) = \delta_{ik} o_k = o_i$$

For a logistic activation function:

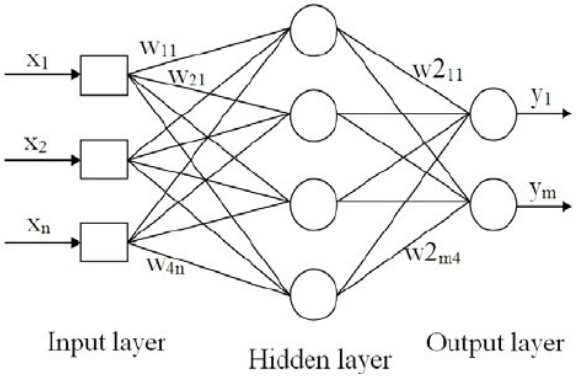
$$\frac{\partial o_j}{\partial net_j} = \sigma(net_j)(1 - \sigma(net_j)) = o_j(1 - o_j)$$

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 perform the differentiations 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 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?

Universal approximation theorem

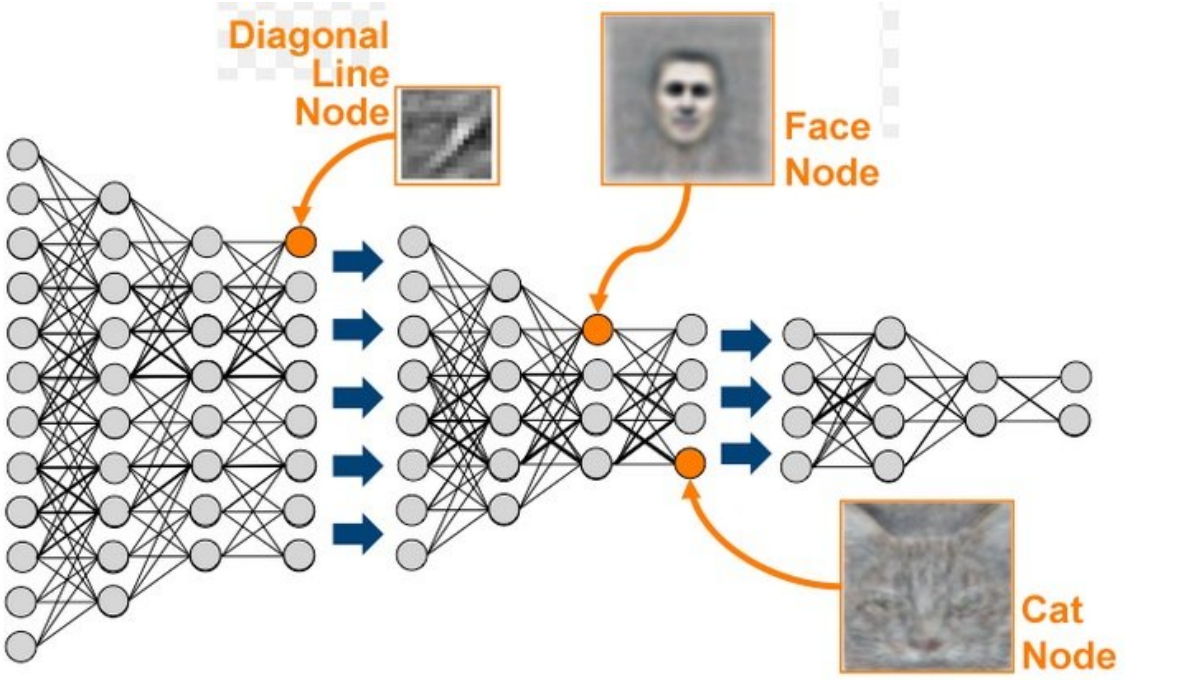
The “universal approximation theorem” (Cybenko, 1989, Hornik, 1991) proves, that already with a single hidden layer, we can essentially learn every continuous function (with a few mild assumptions).

So why introduce more than one hidden layer?

Because the hidden layer may have to be infeasibly large. Also, the learning might take too long, the learning sample might have to be too large, and generalization might not take place.

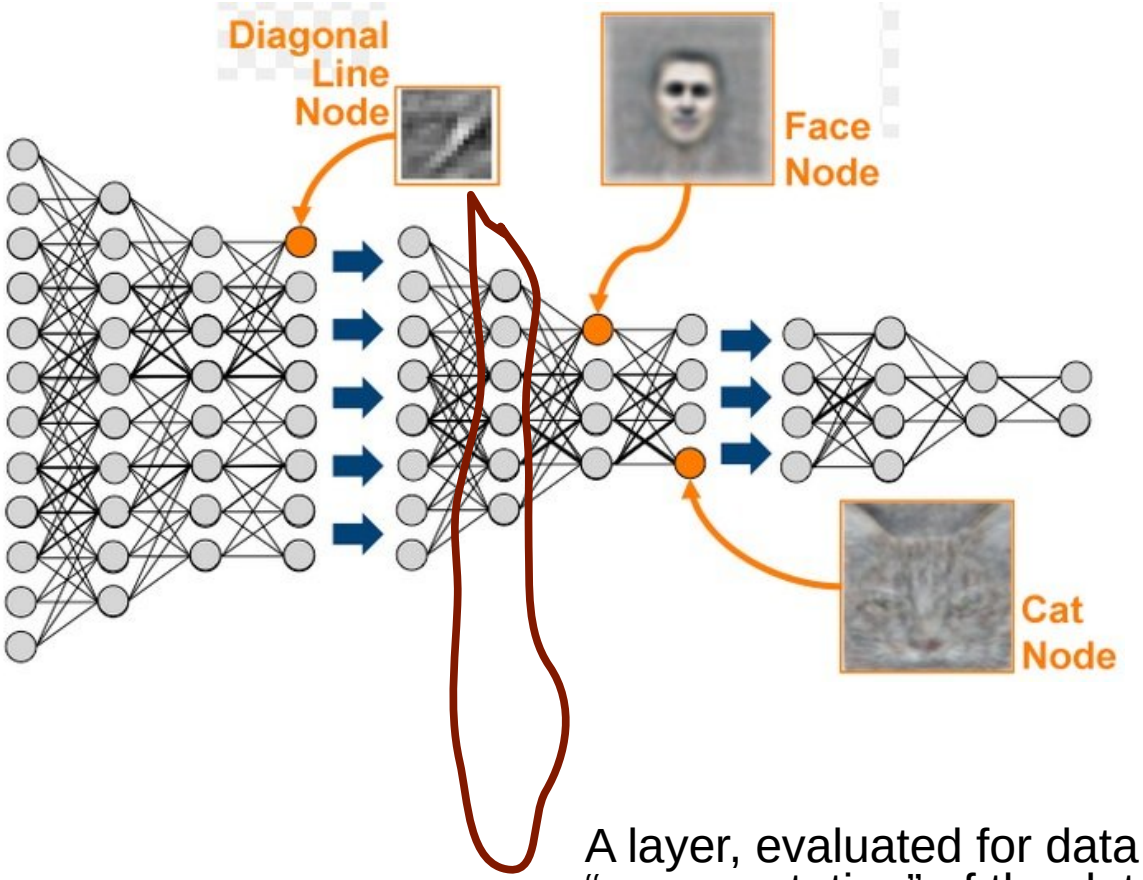
Deep learning = learning abstractions

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

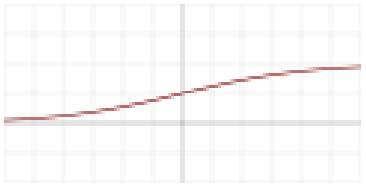
Because when done right, a deep network can learn “abstract features” of the input data:



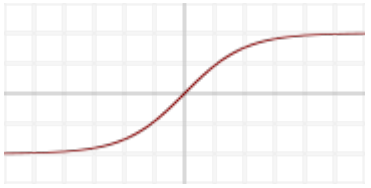
A layer, evaluated for data = a “representation” of the data

Activation functions

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

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

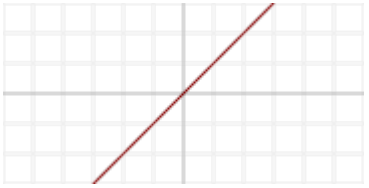
pro: non-linear, differentiable, confined range [0,1]
con: zero at large values of |x| (vanishing gradient problem)

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

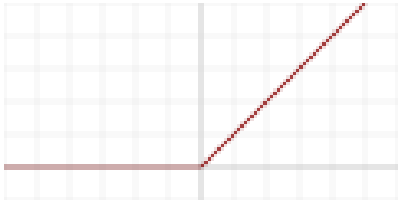
scaled version of sigmoid: tanh(x) = 2 σ(2x) - 1

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 recified linear unit  $f(x) = \max(0, x)$

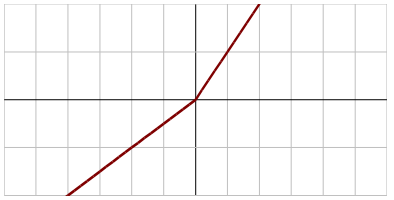
  Pro: simple, computationally fast, non-linear, non-zero gradient for any large x

  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

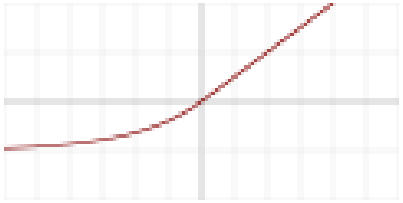


$$f(x) = \begin{cases} 0.01x \text{ for } x < 0 \\ x \text{ for } x \geq 0 \end{cases}$$

Advantages of ReLU, plus it doesnt “die” for x < 0.

- SeLU

scaled exponential linear units



$$f(x) = \lambda \begin{cases} \alpha e^x - \alpha \text{ for } x < 0 \\ x \text{ for } x \geq 0 \end{cases}$$

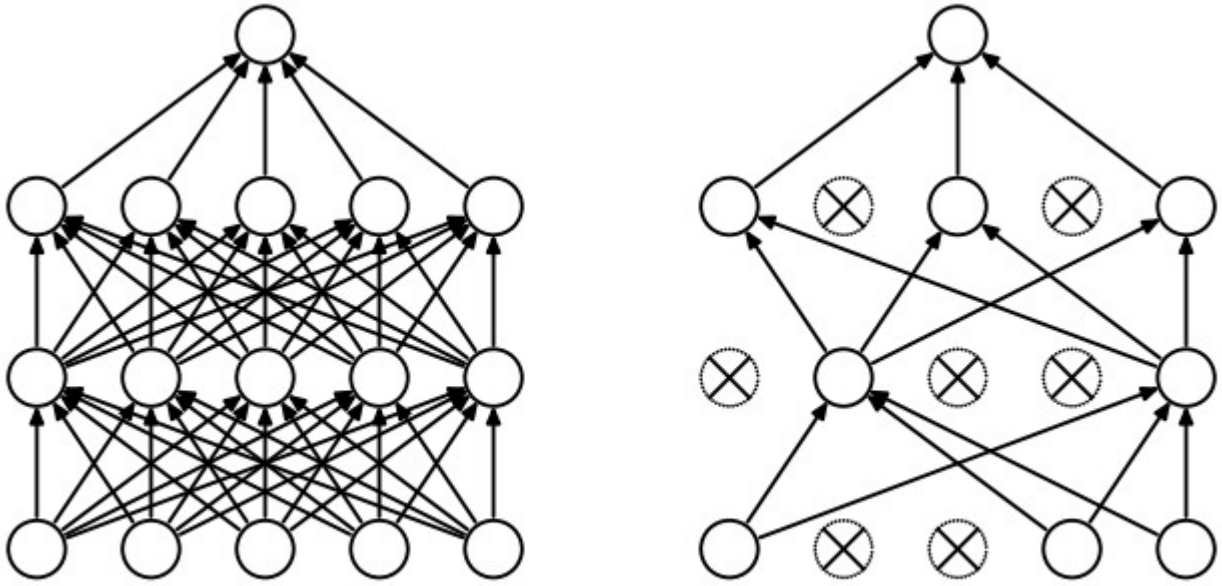
Used in “self-normalizing neural networks”, where the output of the neurons in each layer is distributed with zero mean and unity variance if the input is also distributed with zero mean and unity variance

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.

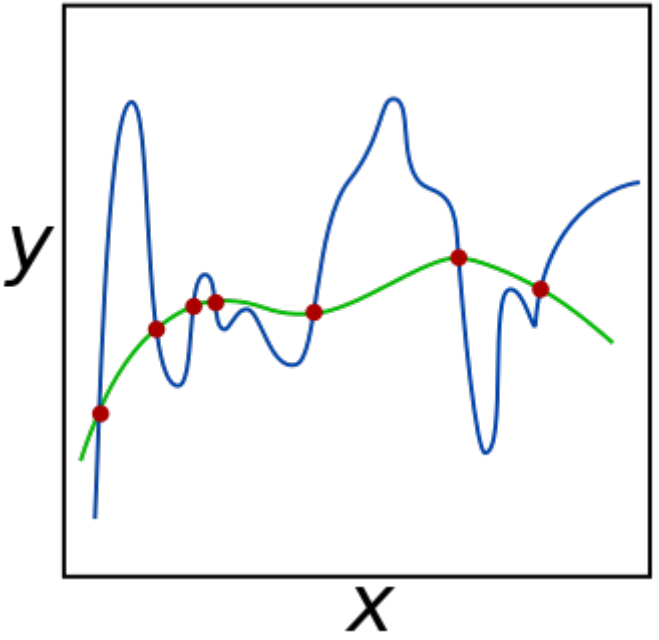
Regularization

"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 at zero

"Punishing 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:

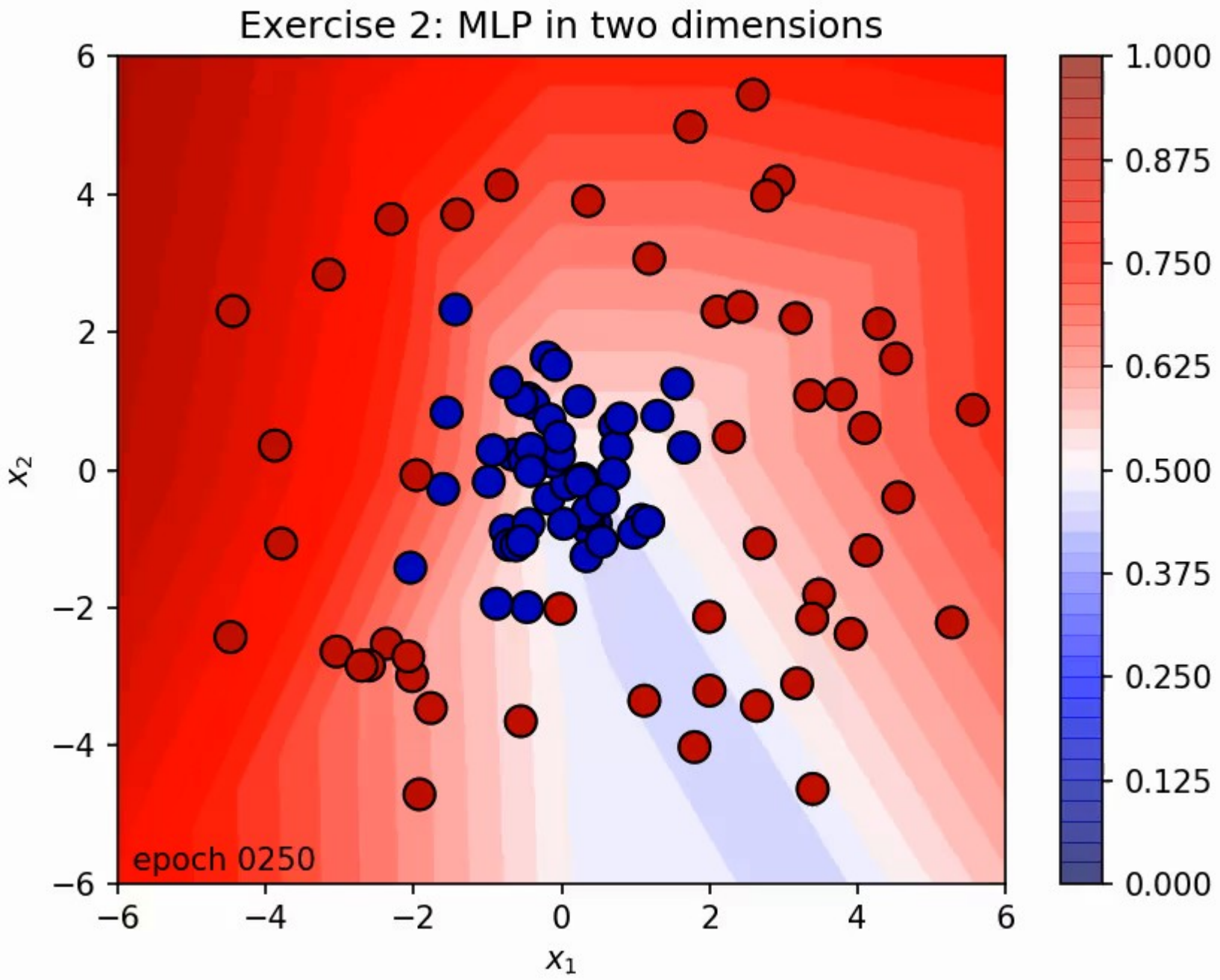
→ grid search: systematically try out many configurations

→ random search: try out configurations randomly

→ gradient based optimization: compute gradients in the space of hyperparameters

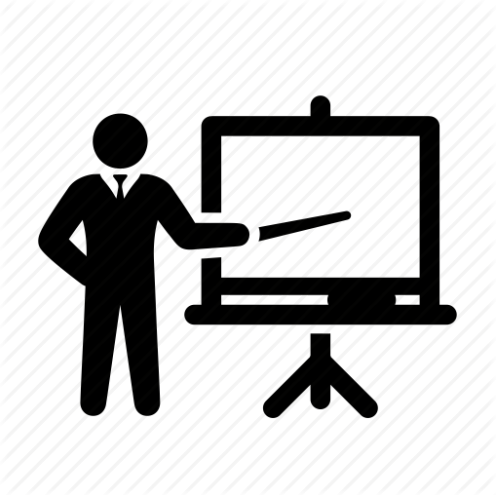
→ evolutionary algorithms

….


Exercise 2: MLP in two dimensions
epoch 0250
x_1
x_2
6
4
2
0
−2
−4
−6
1.000
0.875
0.750
0.625
0.500
0.375
0.250
0.125
0.000

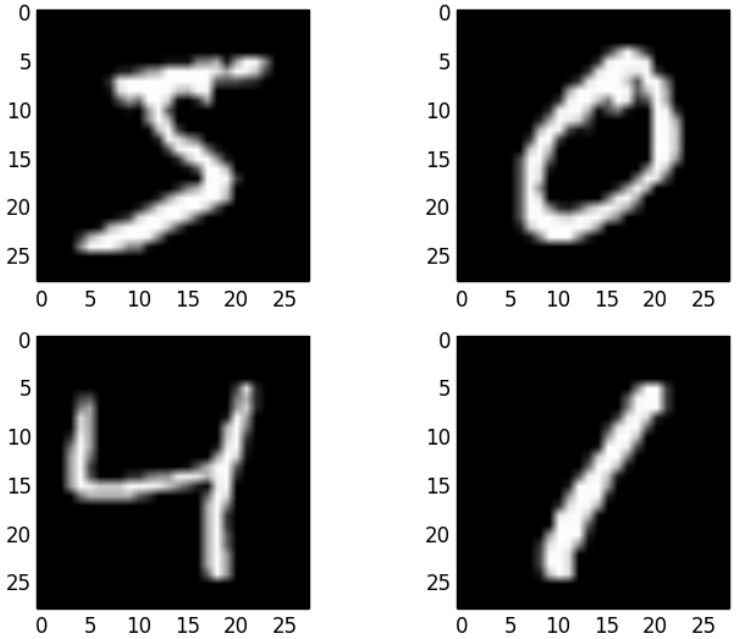
Exercise 2 – MLP

- See 02_skeleton.py 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

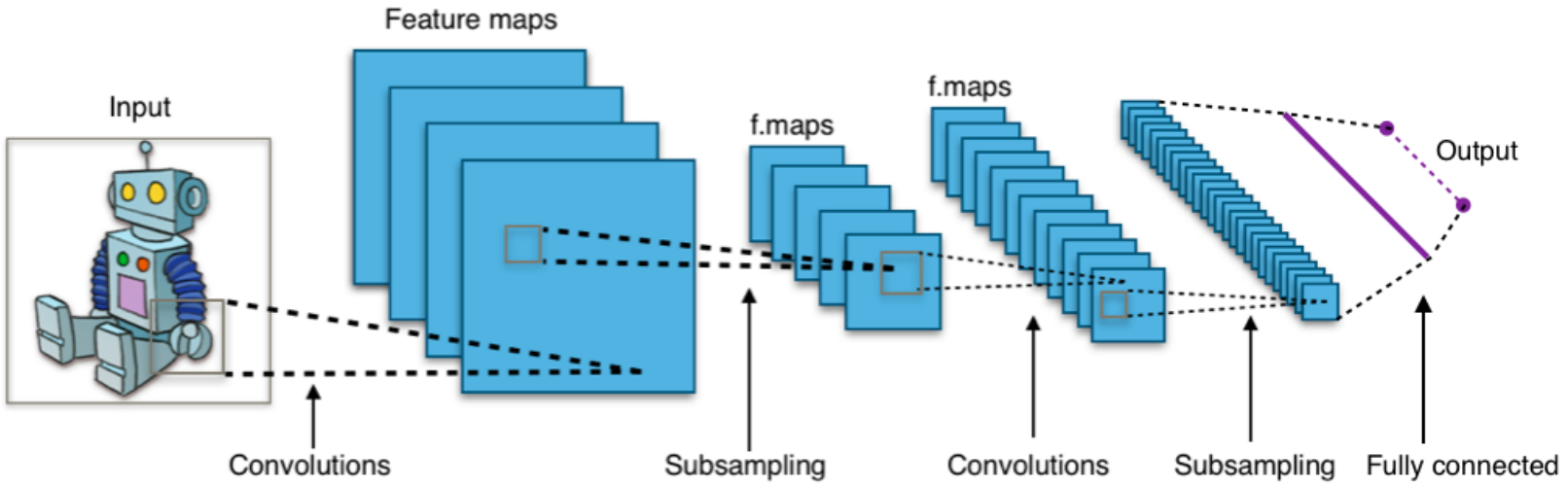


Convolutional layers

Think computer vision: we want to detect a cat,

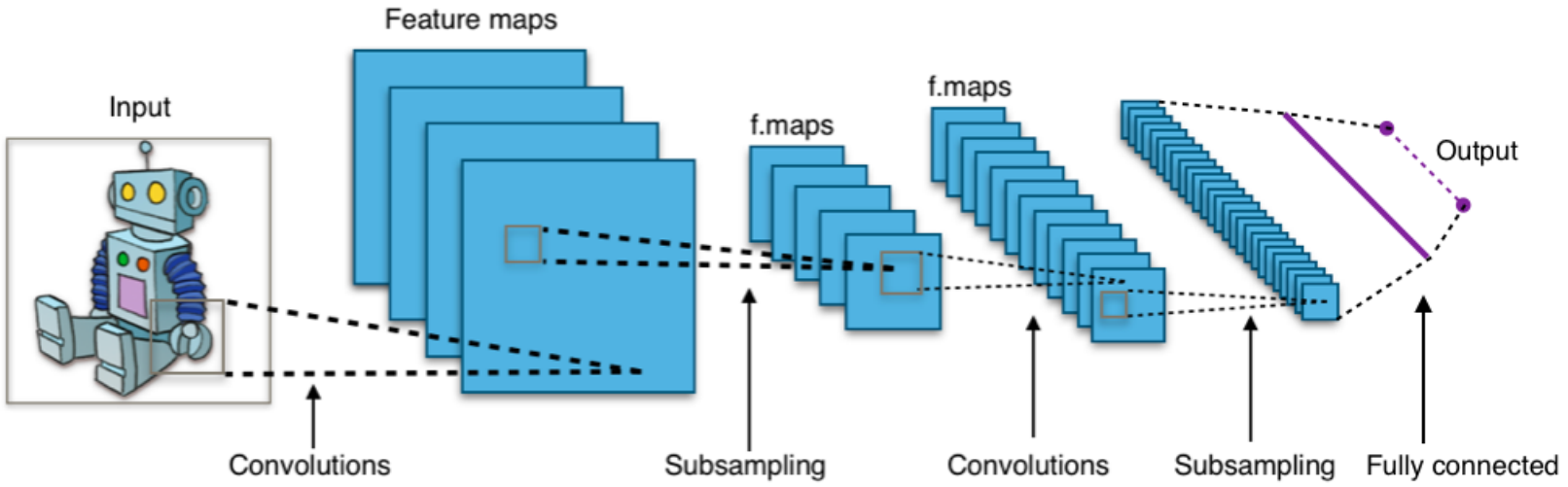
independent of where it is in the photo: translation invariance!

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



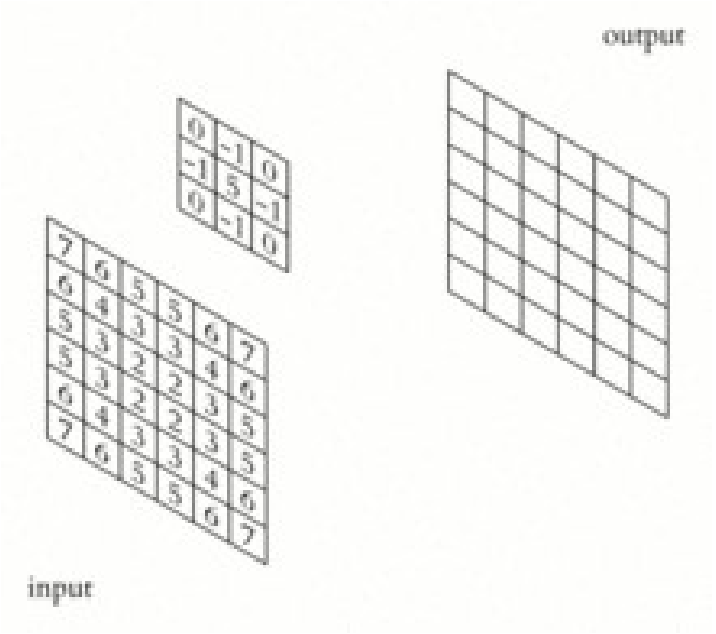
Convolutional layers

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

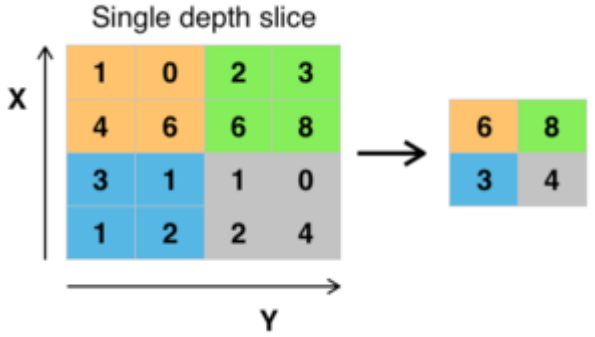


Convolutional layers

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



Max Pooling (subsampling)

Inspired by the “receptive field” in biology

Batch normalization

It is usually a good trick to “whiten” / standardize the inputs: transform them linearly so they have zero mean and unity 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: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

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)

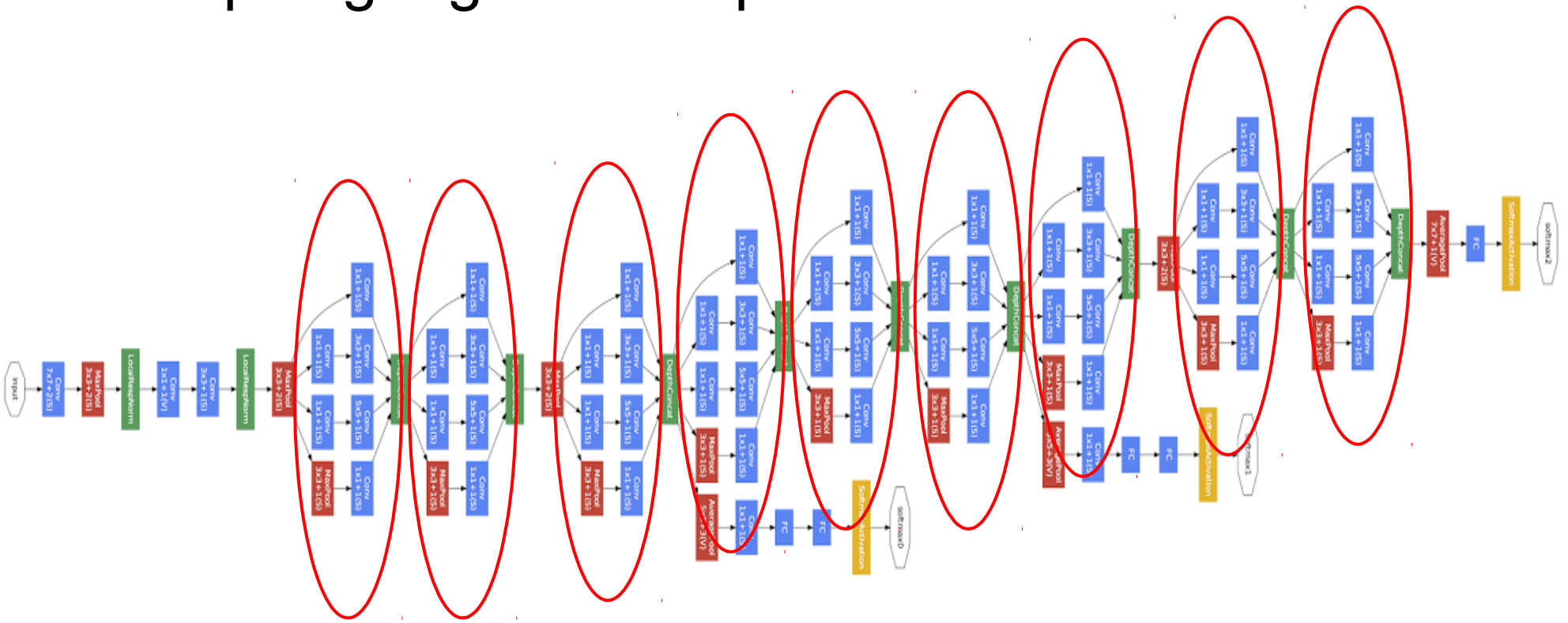
The K output neurons $\vec{y}$ should therefore acquire values between 0 and 1, and sum up to 1.

Softmax function achieves this:

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

As physicists we note that this is equivalent to the Boltzmann distribution: if y_j denotes the nevative 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...

https://storage.googleapis.com/pub-tools-public-publication-data/pdf/43022.pdf

Application in particle physics: jet images

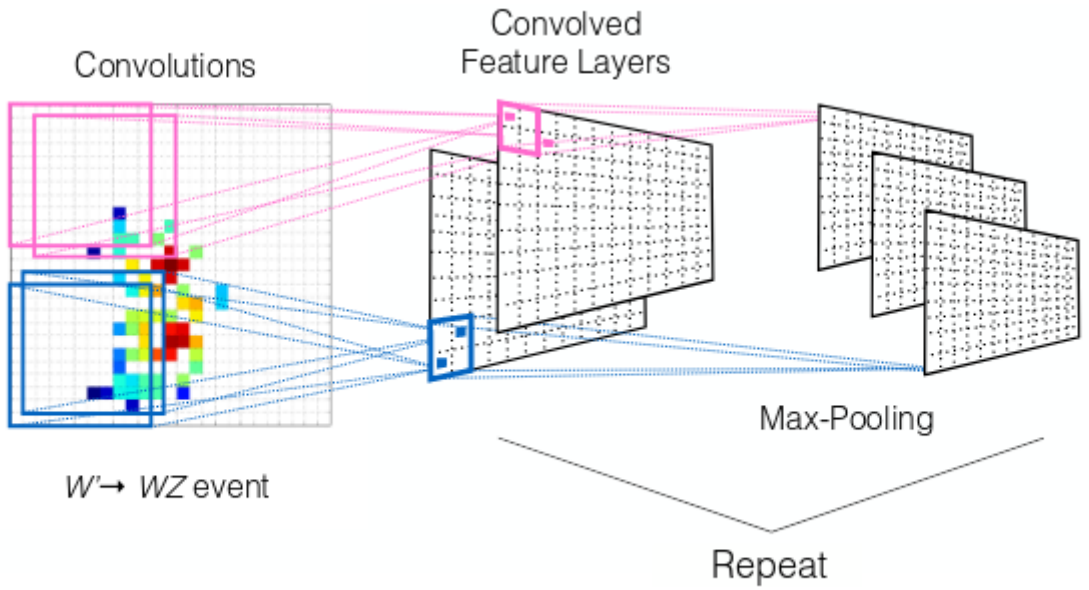
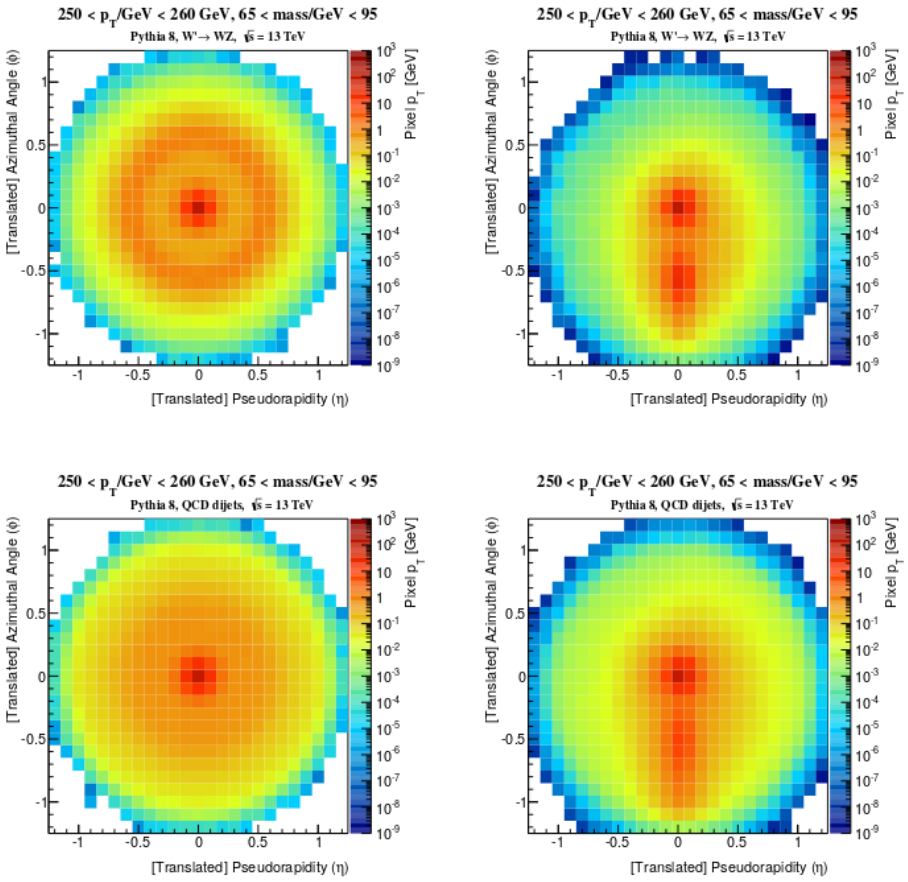


Figure 5: The convolution neural network concept as applied to jet-images.

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

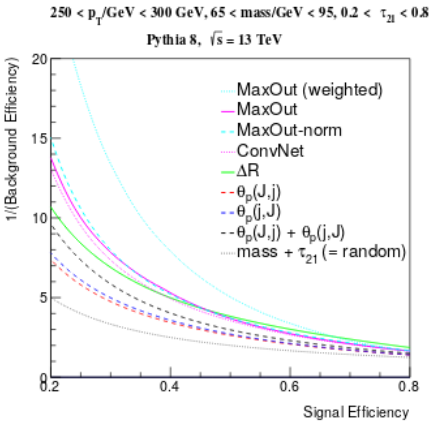
https://arxiv.org/pdf/1511.05190.pdf

Application in particle physics: jet images



average signal

average background

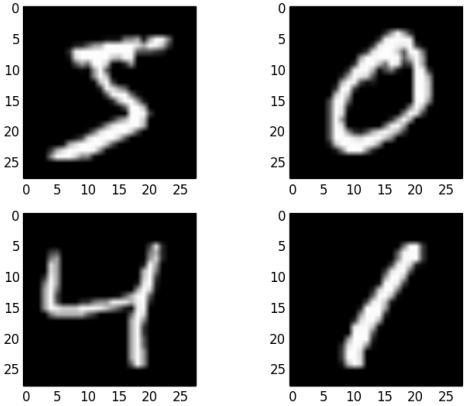


Receiver Operating Characteristic (ROC) curves for various choices of "hyperparameters"

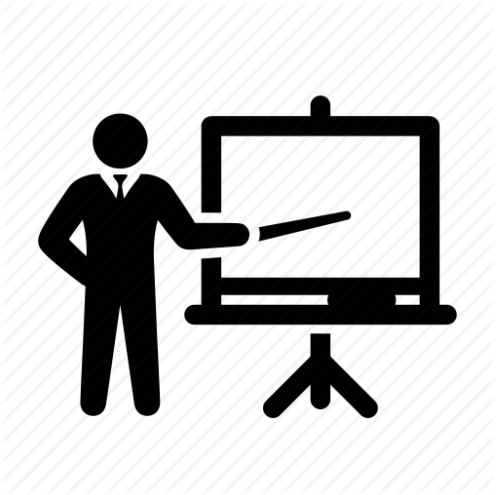
Figure 2: The average jet image for signal W jets (top) and background QCD jets (bottom) before (left) and after (right) applying the rotation, re-pixelation, and inversion steps of the pre-processing. The average is taken over images of jets with 240 GeV $< p_T <$ 260 GeV and 65 GeV $<$ mass $<$ 95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

https://arxiv.org/pdf/1511.05190.pdf

Exercise 3 – CNN



- See 03_skeleton.py 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%?



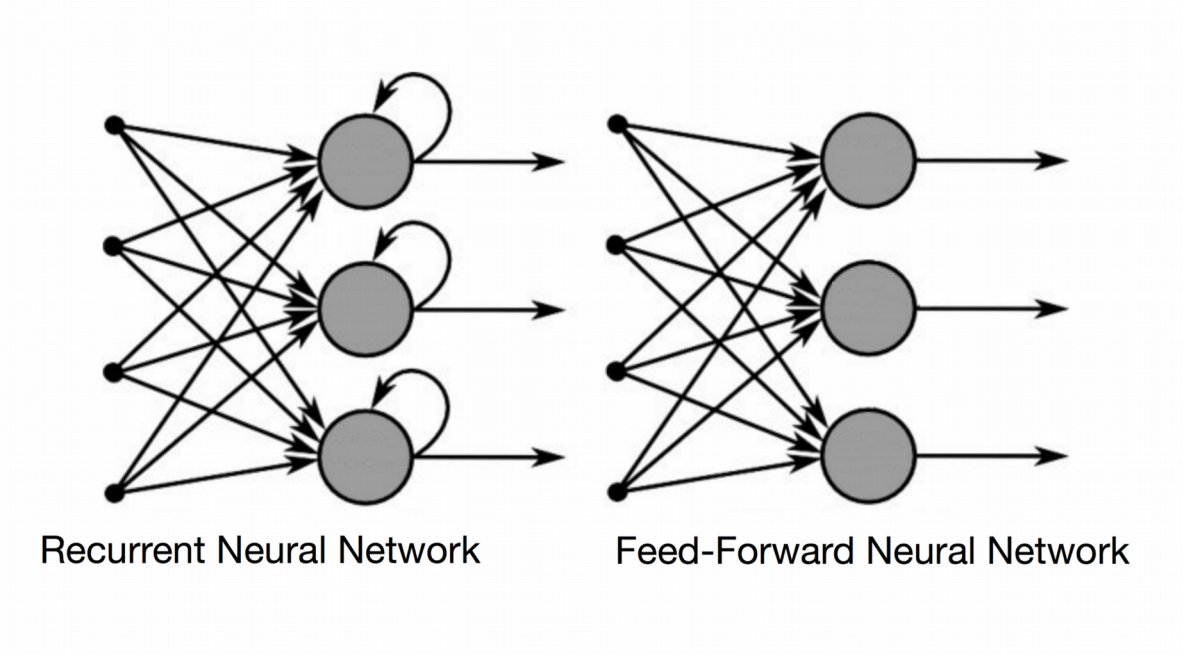
Lesson 4

- Recursive and recurrent neural networks,
- auto encoders
- generative adversarial networks,
- explanation methods

Recurrent networks

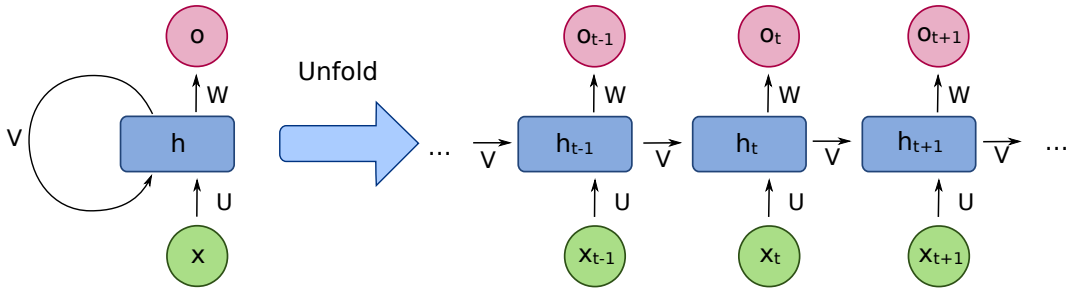
How can we apply neural networks on variable-sized time series like text or speech, where the prediction of the current input must depend on the past “events”.

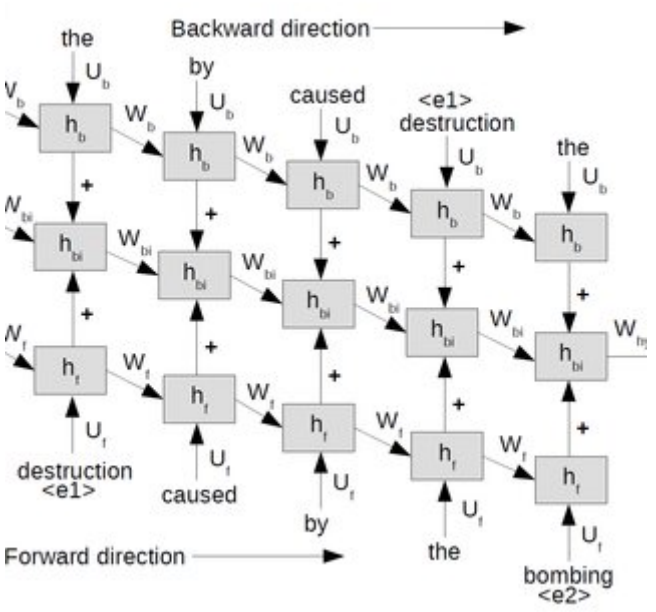
By making the network “recurrent”:



Recurrent 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.

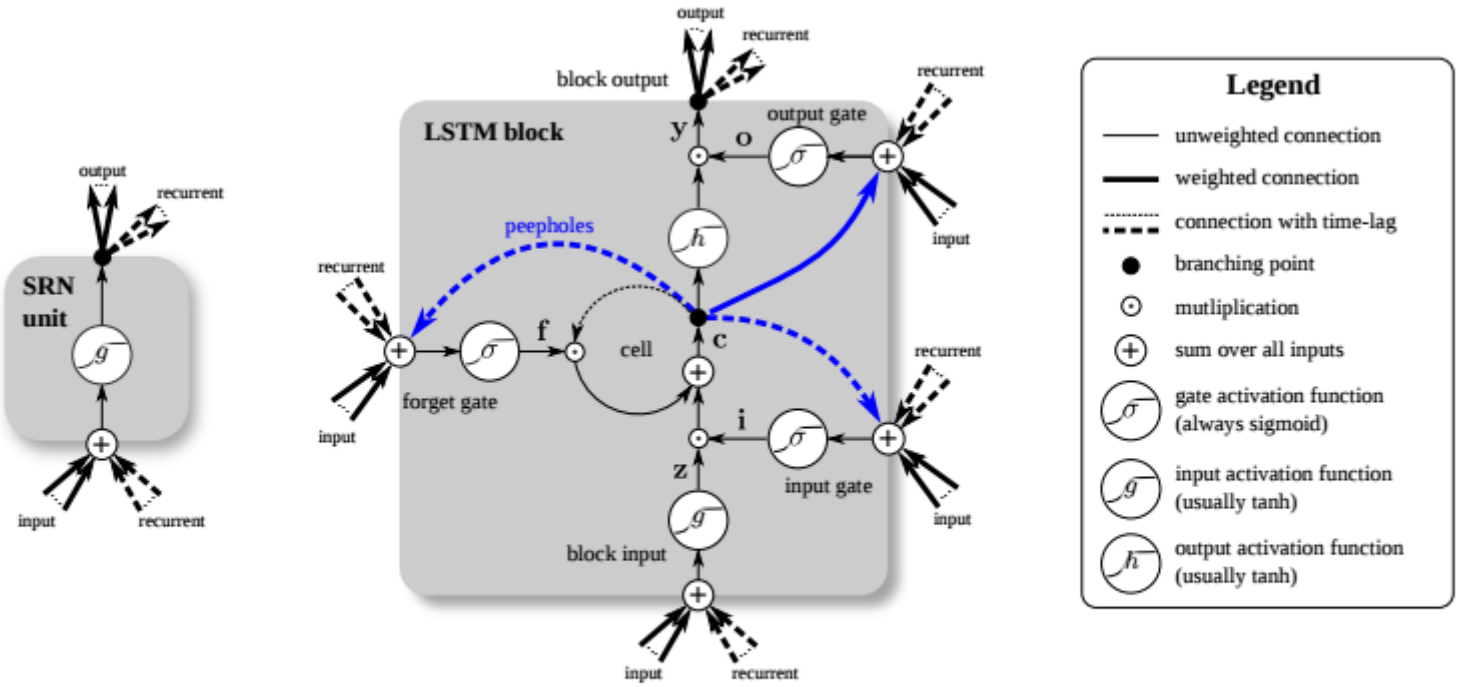
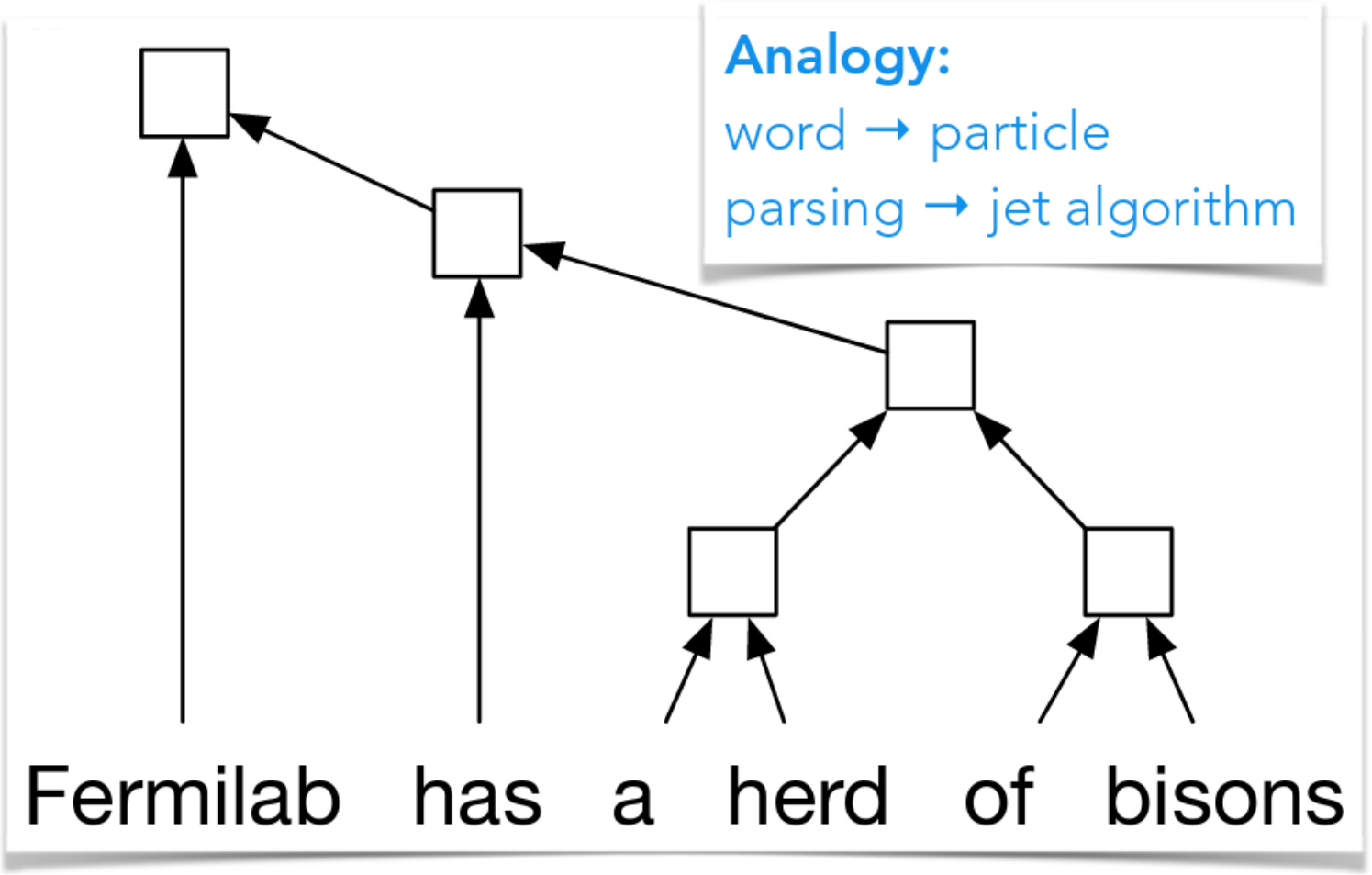


Figure 1. Detailed schematic of the Simple Recurrent Network (SRN) unit (left) and a Long Short-Term Memory block (right) as used in the hidden layers of a recurrent neural network.

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)

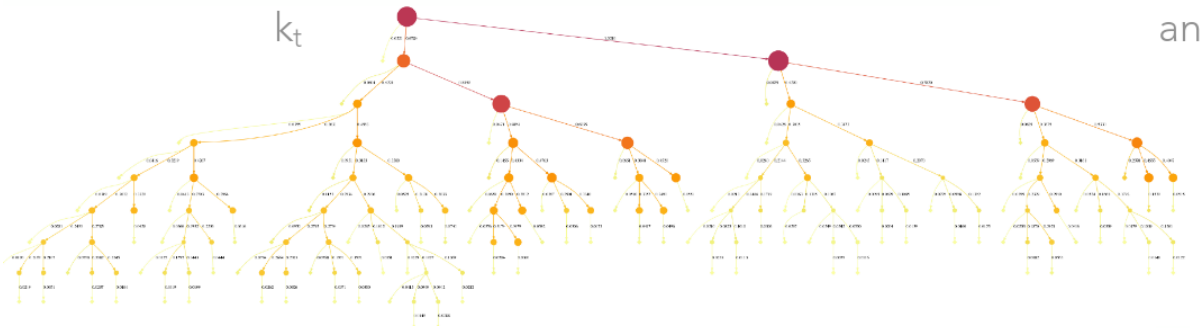


See also talk at DS@HEP 2017 workshop

https://arxiv.org/pdf/1702.00748.pdf

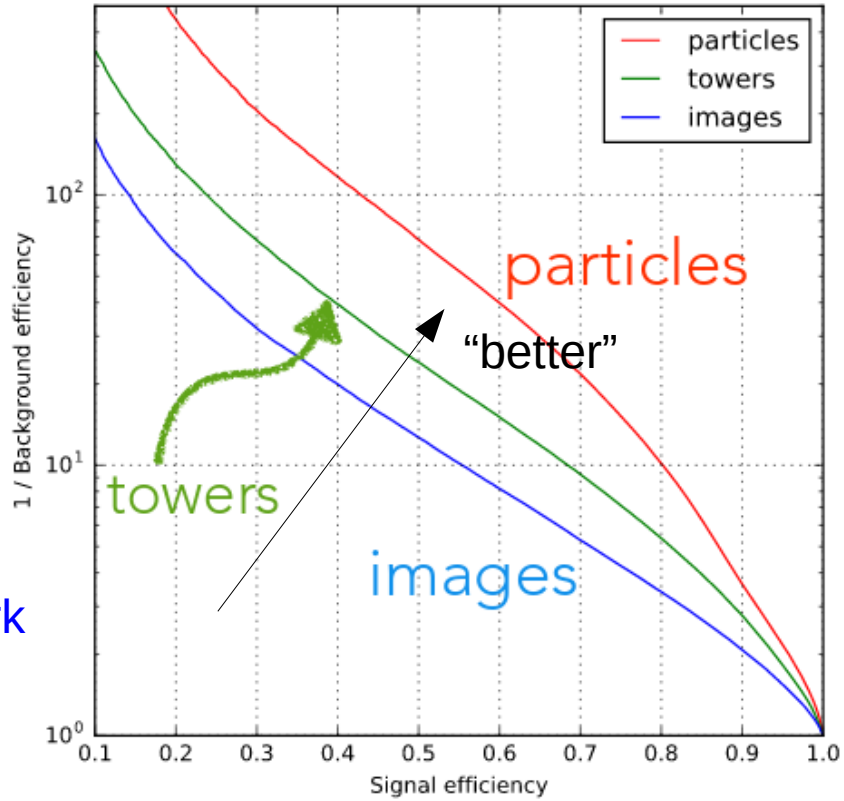
Application (of a similar class of algorithms) in particle physics: jet classification

"QCD-aware recursive neural networks for jet physics"



Visualisation of k_t jet algorithm.

"particles": recursive network
"images": convolutional network



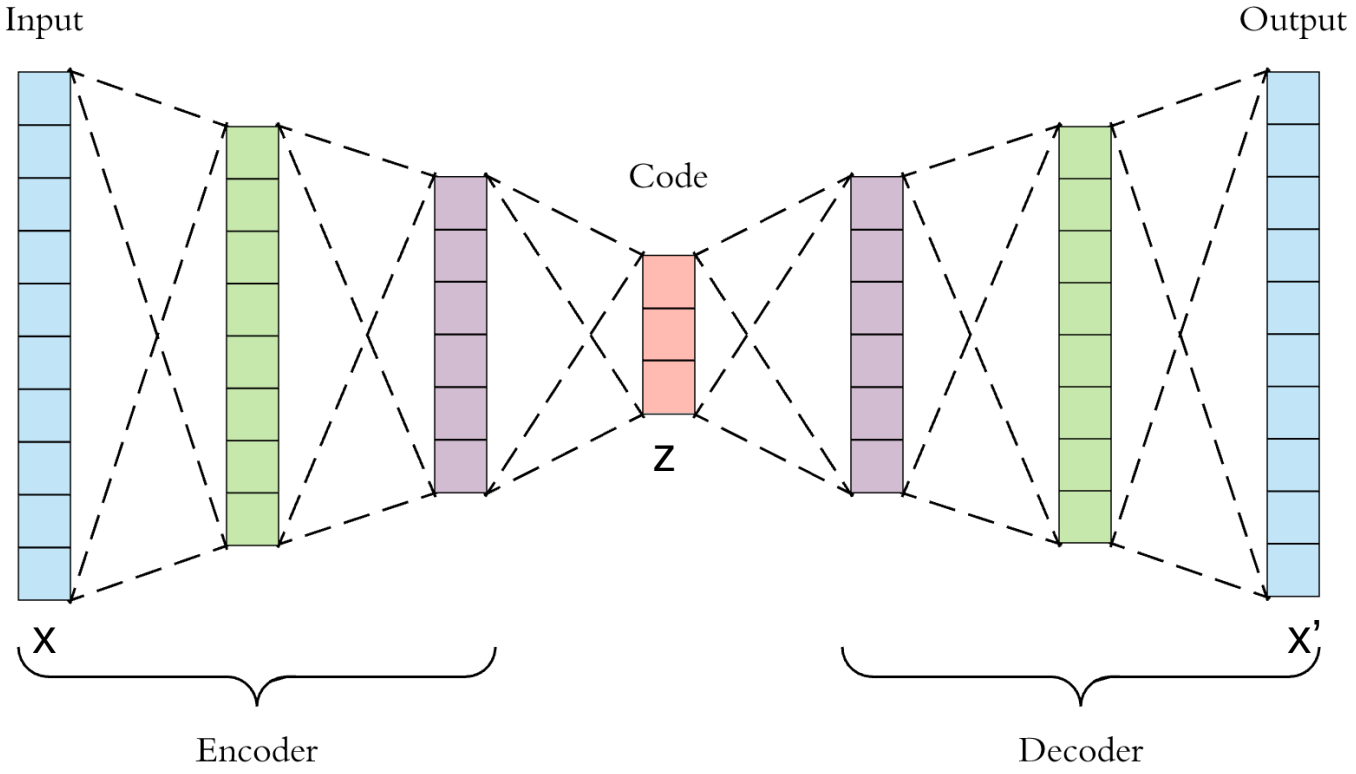
ROC curve of W jet tagging example, comparison with jet images algorithm

See also talk at DS@HEP 2017 workshop

https://arxiv.org/pdf/1702.00748.pdf

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

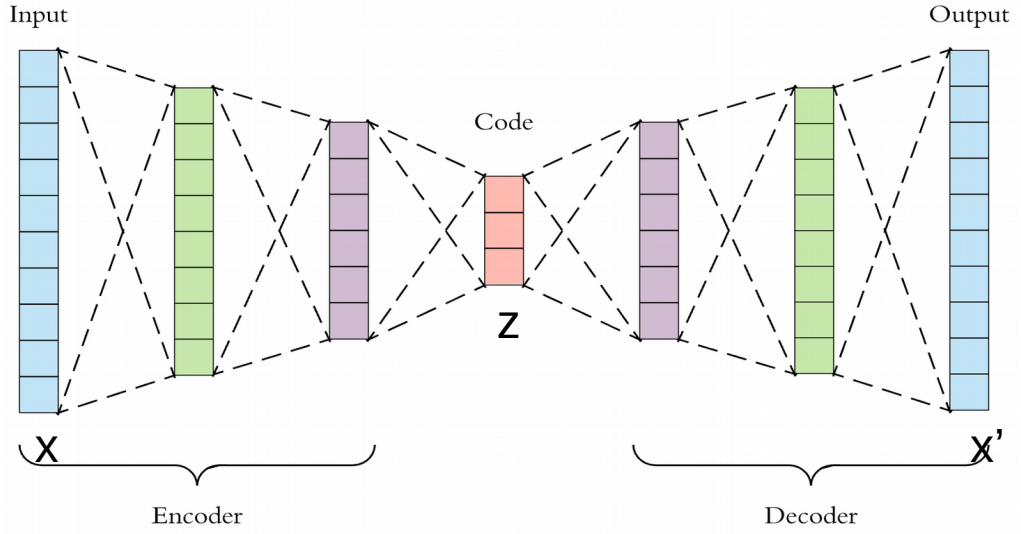
Consider the following network architecture:



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

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

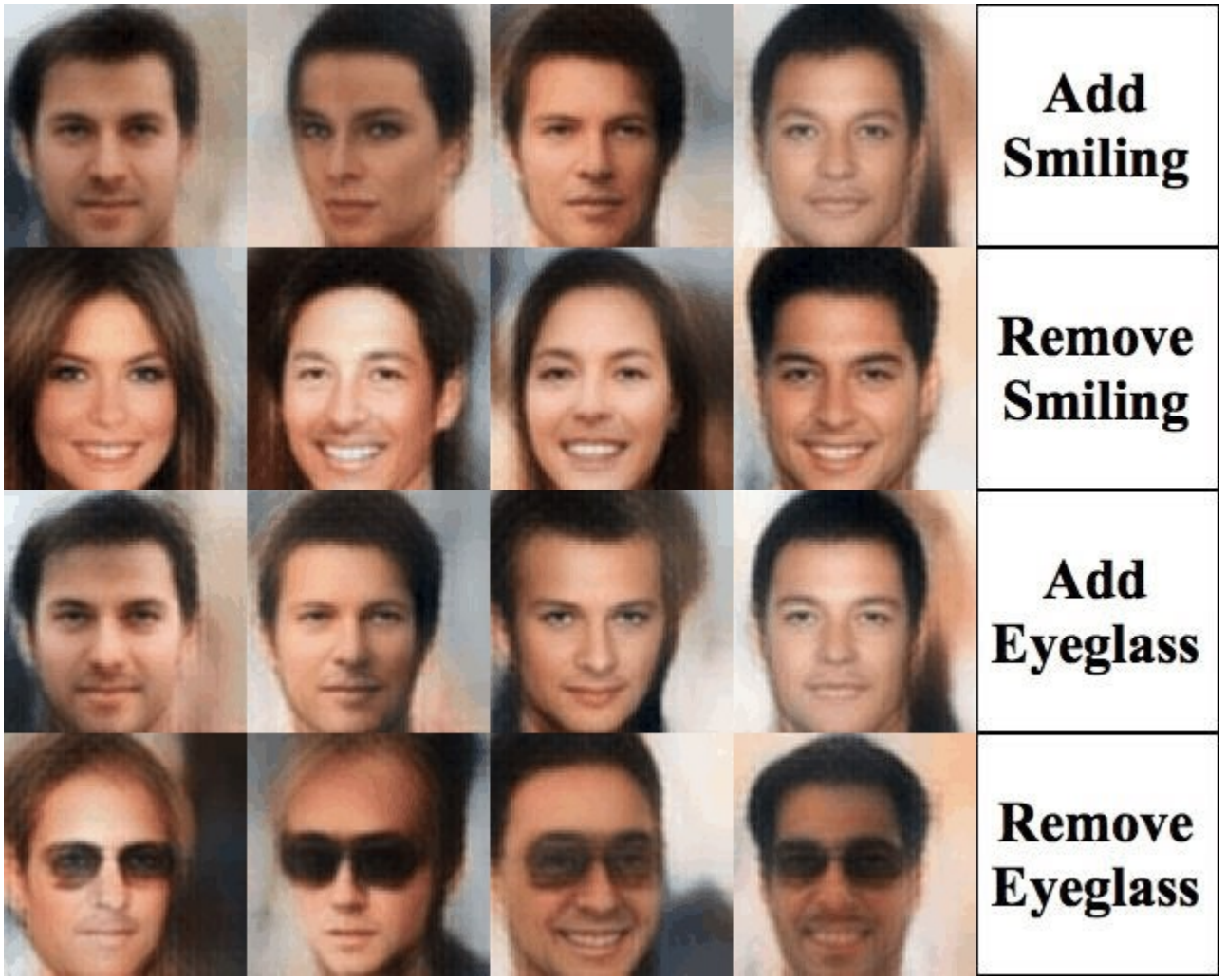
Consider the following network architecture:



With a loss function: $L \propto (\mathbf{x} - \mathbf{x}')^2$
The network learns to reproduce its own input!

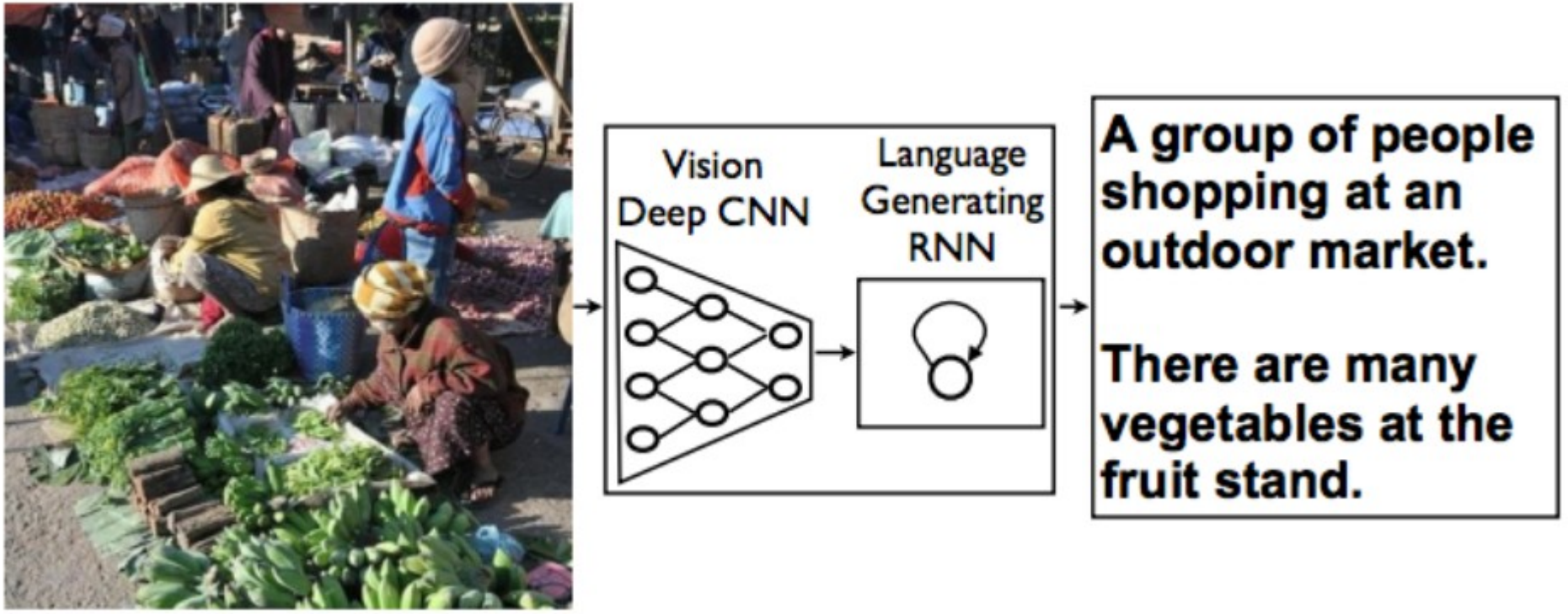
z becomes a lower-dimensional representation (a “code”) of a higher-dimensional **x**
Applications: Lossy compression, denoising data. With another trick (by imposing constraints on **z**): generative model (variational autoencoder)

Example: Variational Autoencoder applied to “CelebA” dataset



Taken from: https://houxianxu.github.io/assets/project/dfcvae

Example: Combination of CNN (encoder) and LSTM (decoder)



http://cvlab.postech.ac.kr/~bhhan/class/cse703r_2016s/csed703r_lecture11.pdf

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.

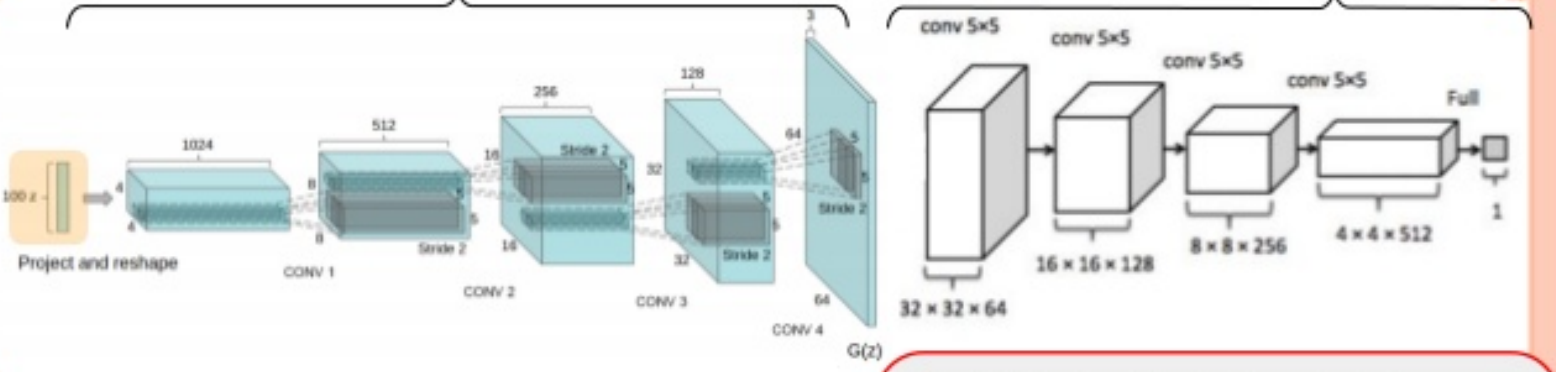
Another machine learns to discriminate the real Picassos from fakes.

Training data B

Example Architecture:

Generator G(.)

Discriminator D(.)



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(.)=0.5 (i.e., cannot tell the difference between real and generated images) and G(z)=learns the training images distribution.

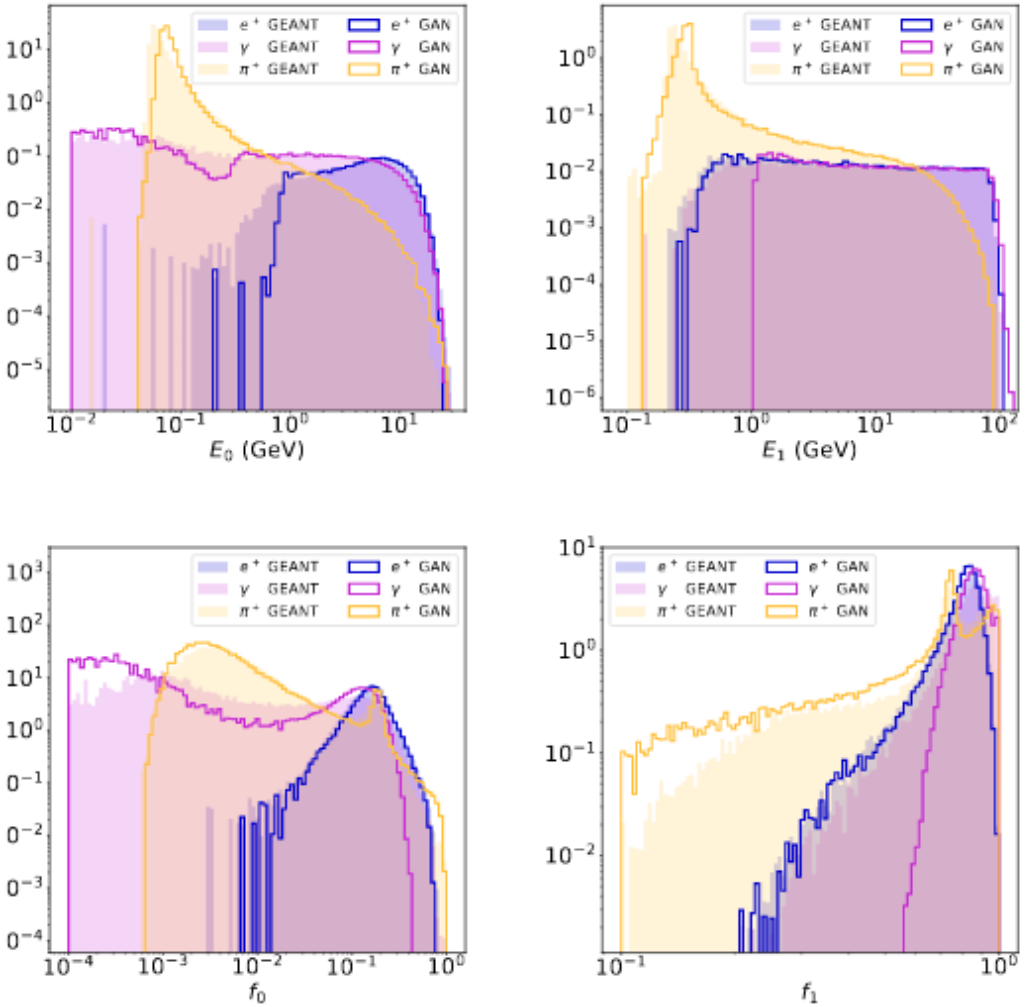
https://www.slideshare.net/shyamkkhadka/unsupervised-learning-represenation-with-dcgan

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

A “failure case”

https://github.com/junyanz/CycleGAN

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



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

See also: talk in ACAT 2017

https://arxiv.org/pdf/1705.02355.pdf

Explanation methods

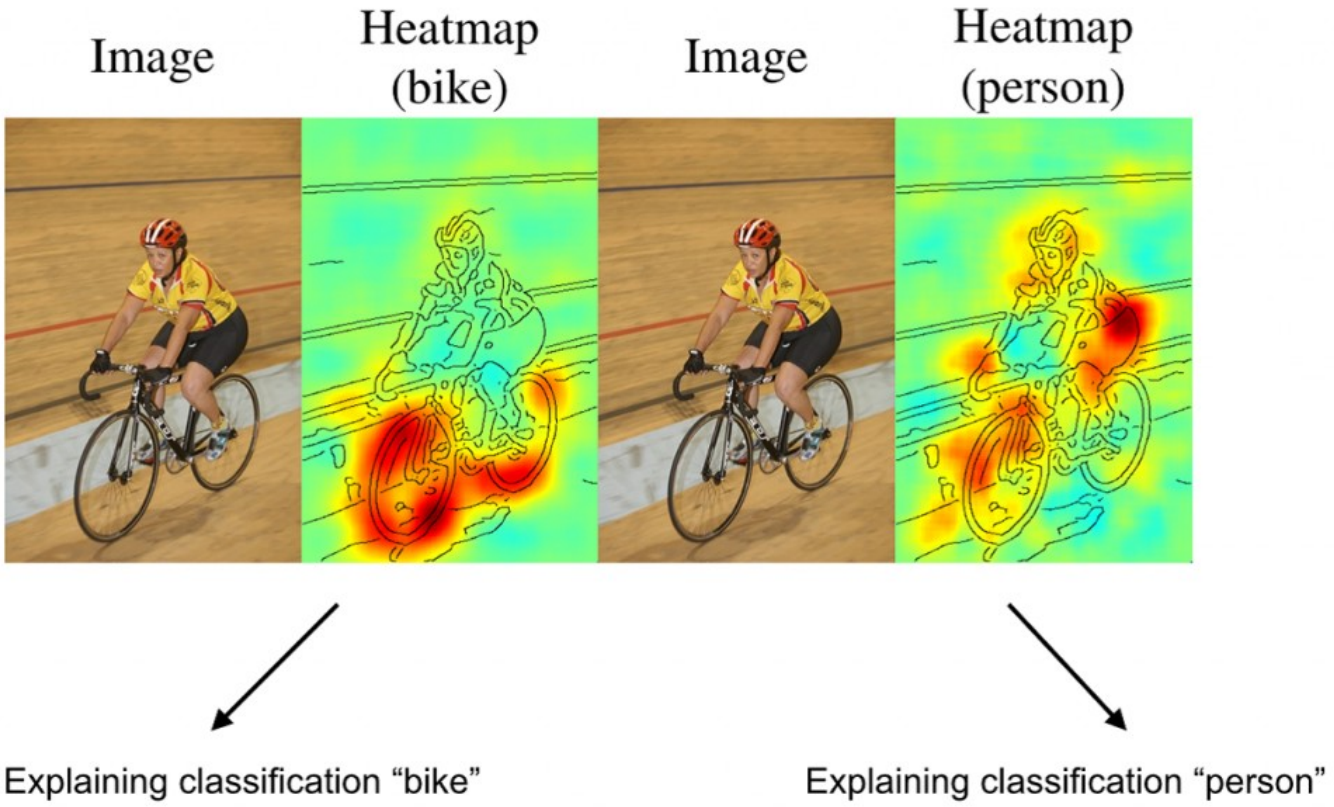
What is the machine learning? How can we “debug” deep networks?

- 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:

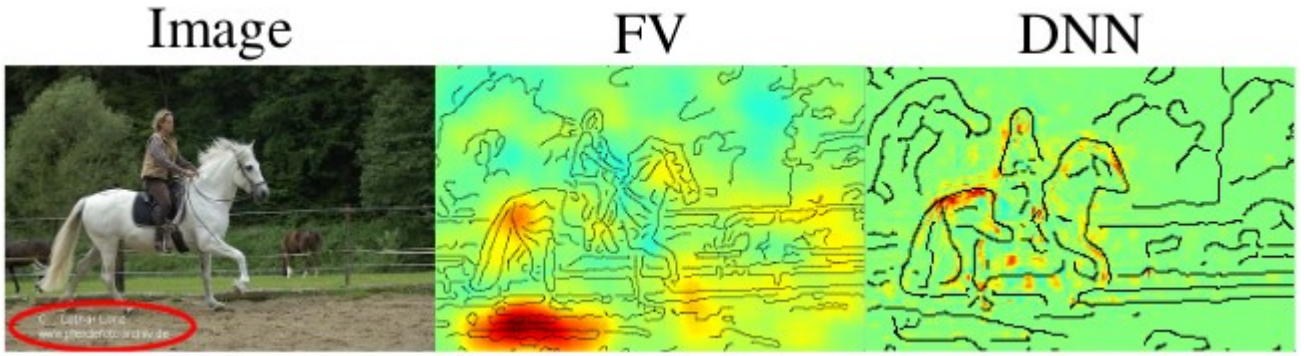


https://arxiv.org/pdf/1512.00172.pdf

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.

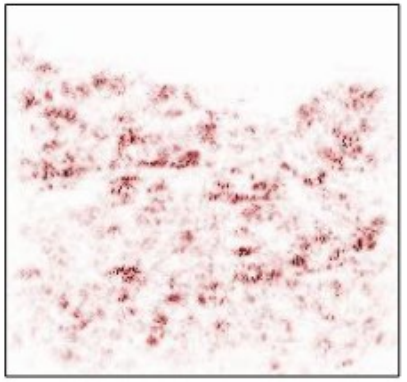
https://arxiv.org/pdf/1512.00172.pdf

Explanation methods

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

What makes the picture below a road with cars?

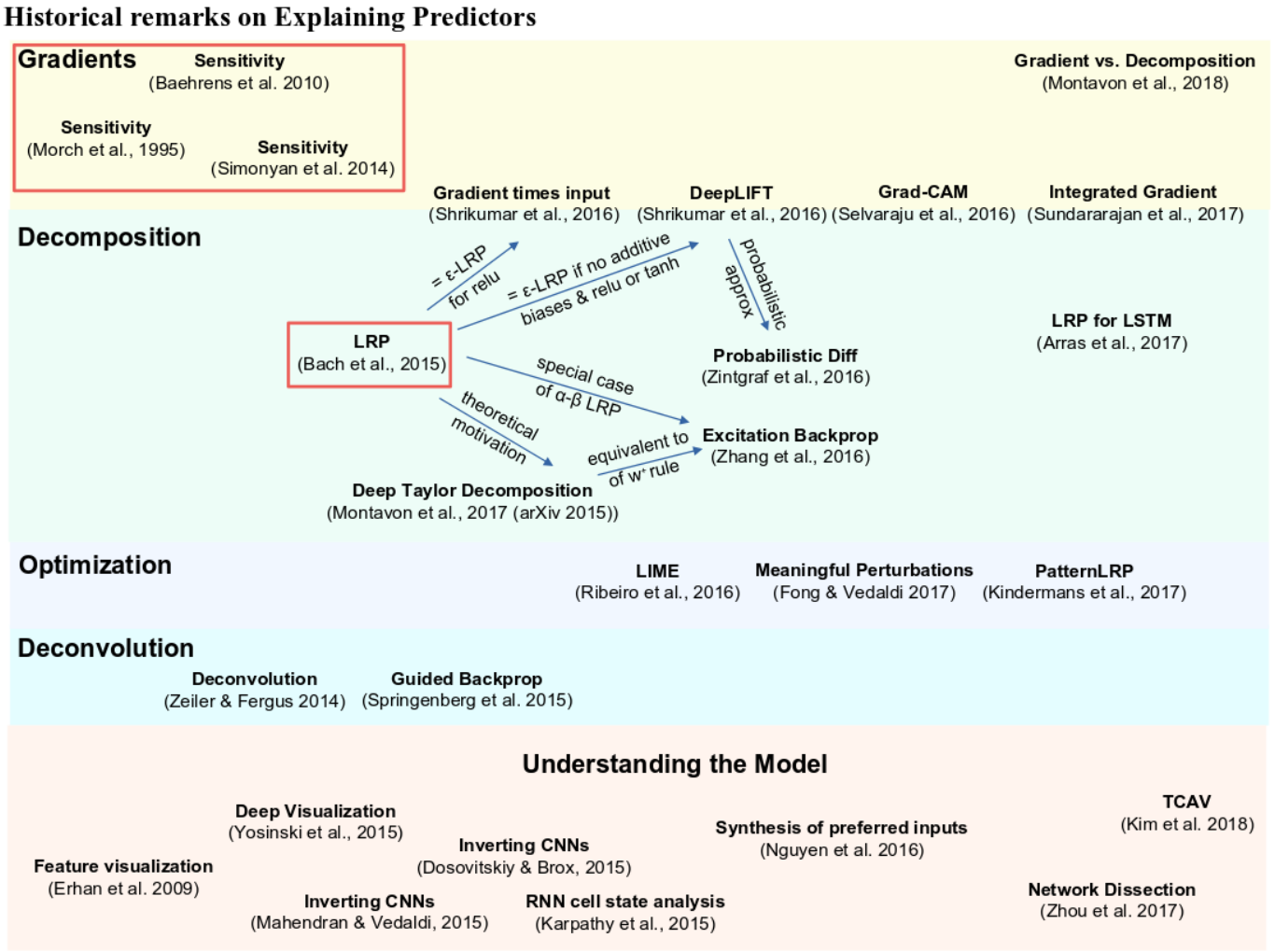
$$R_i = \left(\frac{\partial f}{\partial x_i}\right)^2$$



Problem: sensitivity analysis does not highlight cars

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

Explanation methods



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

Workshop 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