Introduction to Data Analysis and Machine Learning in Physics: 5. Neural networks

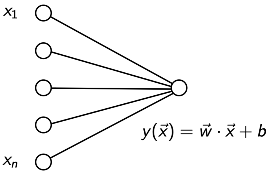
Martino Borsato, Jörg Marks, Klaus Reygers

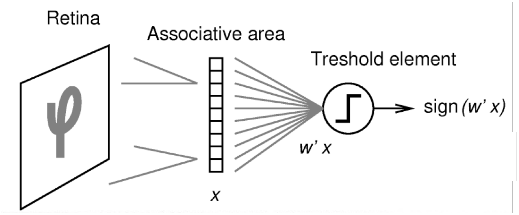
Studierendentage, 6-9 April 2021

Exercises

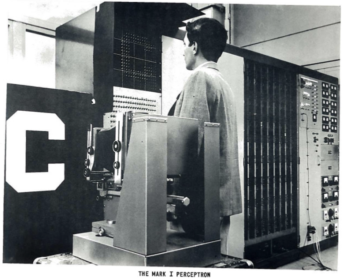
- Exercise 1: Learn XOR with a MLP
  - `05_neural_networks_ex_1_xor.ipynb`
- Exercise 2: Visualising decision boundaries of classifiers
  - `05_neural_networks_ex_2_decision_boundaries.ipynb`
- Exercise 3: Boston house prices (MLP regression)
  - `05_neural_networks_ex_3_boston_house_prices.ipynb`
- Exercise 4: Training a digit-classification neural network on the MNIST dataset using Keras
  - `05_neural_networks_ex_4_mnist_keras_train.ipynb`
- Exercise 5: Higgs data set

Perceptron (1)





$$h(\vec{x}) = \begin{cases} 1 & \text{if} \quad \vec{w} \cdot \vec{x} + b > 0, \\ 0 & \text{otherwise} \end{cases}$$



Mark 1 Perceptron. Frank Rosenblatt (1961)

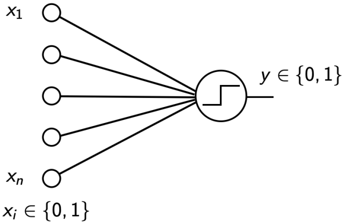
The perceptron was designed for image recognition. It was first implemented in hardware (400 photocells, weights = potentiometer settings).

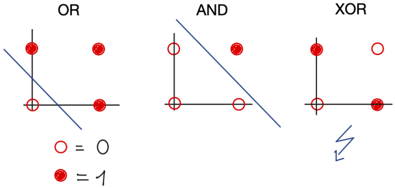
Perceptron (2)

- McCulloch–Pitts (MCP) neuron (1943)
  - First mathematical model of a biological neuron
  - Boolean input
  - Equal weights for all inputs
  - Threshold hardcoded
- Improvements by Rosenblatt
  - Different weights for inputs
  - Algorithm to update weights and threshold given labeled training data

Shortcoming of the perceptron:
it cannot learn the XOR function

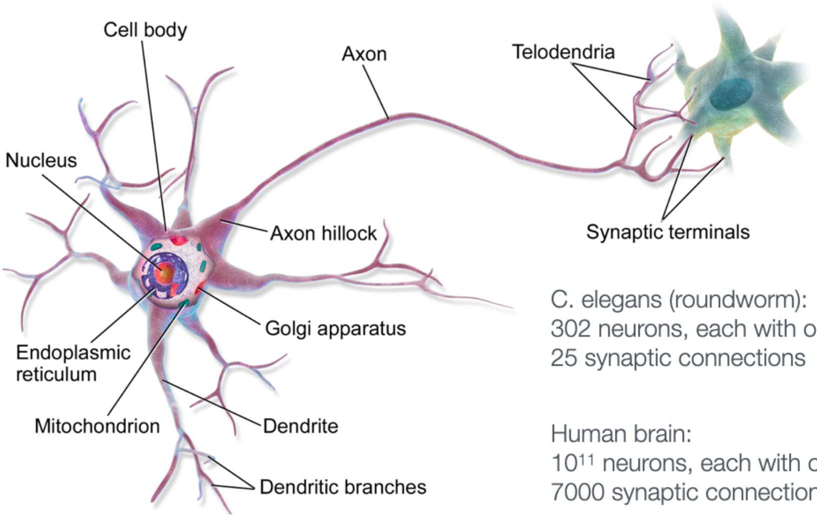
Minsky, Papert, 1969





XOR: not linearly separable

The biological inspiration: the neuron



C. elegans (roundworm):
302 neurons, each with on average
25 synaptic connections

Human brain:
10^{11} neurons, each with on average
7000 synaptic connections

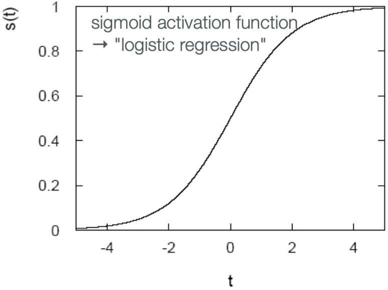
Non-linear transfer / activation function

Discriminant:

$$y(\vec{x}) = h\left(w_0 + \sum_{i=1}^{n} w_i x_i\right)$$

Examples for function h:

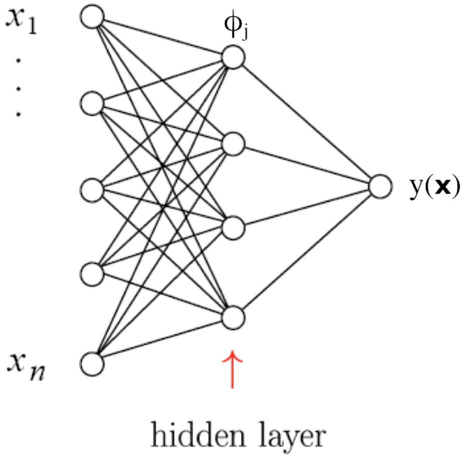
$$\frac{1}{1+e^{-x}} \text{ ("sigmoid" or "logistic" function)}, \quad \tanh x$$



Non-linear activation function needed in neural networks when feature space is not linearly separable.

Neural net with linear activation functions is just a perceptron

Feedforward neural network with one hidden layer



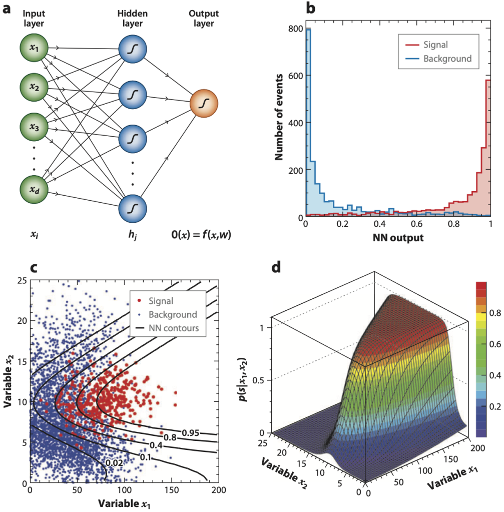
$$\phi_i(\vec{x}) = h\left(w_{i0}^{(1)} + \sum_{j=1}^{n} w_{ij}^{(1)} x_j\right)$$

$$y(\vec{x}) = h\left(w_{10}^{(2)} + \sum_{j=1}^{m} w_{1j}^{(2)} \phi_j(\vec{x})\right)$$

superscripts indicates layer number, i.e., $w_{ij}^{(1)}$ refers to the input weights of neuron i in the hidden layer ($=$ layer 1).

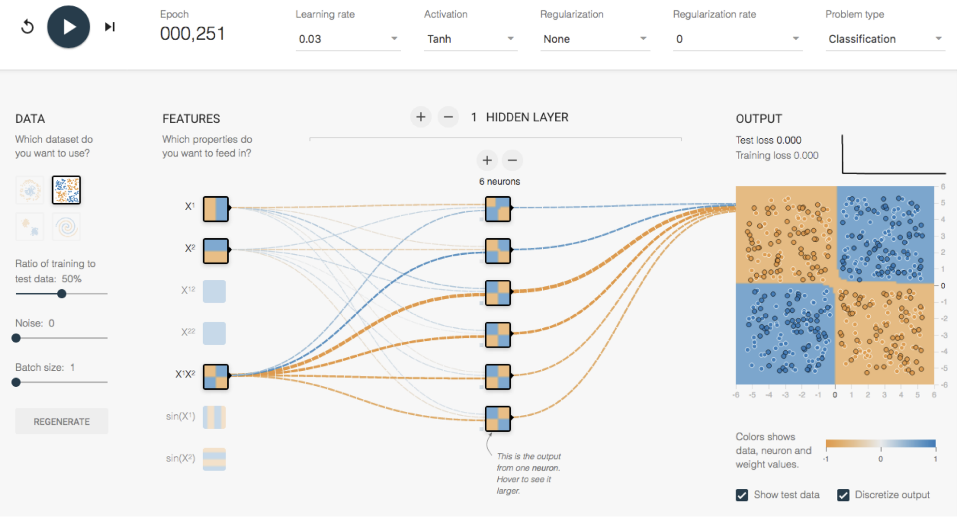
Straightforward to generalize to multiple hidden layers

Neural network output and decision boundaries



P. Bhat, Multivariate Analysis Methods in Particle Physics, inspire-hep.net/record/879273

Fun with neural nets in the browser



Epoch 000,251

Learning rate 0.03

Activation Tanh

Regularization None

Regularization rate 0

Problem type Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 1

FEATURES

Which properties do you want to feed in?

1 HIDDEN LAYER

6 neurons

This is the output from one neuron. Hover to see it larger.

OUTPUT

Test loss 0.000

Training loss 0.000

Colors shows data, neuron and weight values.

☑ Show test data ☑ Discretize output

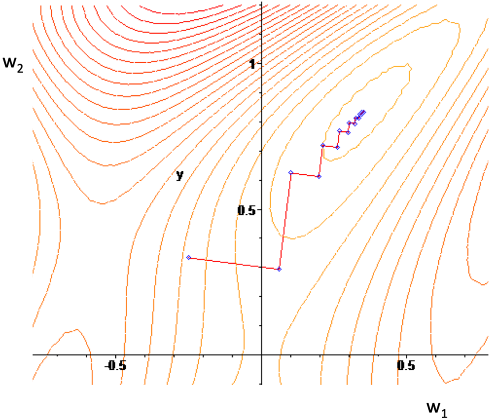
http://playground.tensorflow.org

Backpropagation (1)

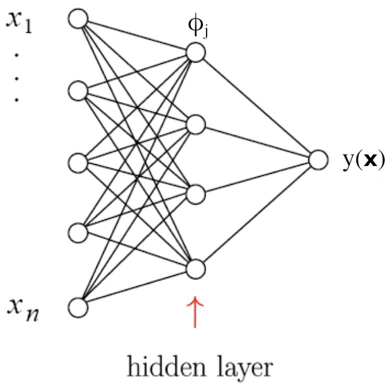
Start with an initial guess $\vec{w}_0$ for the weights an then update weights after each training event:

$$\vec{w}^{(\tau+1)} = \vec{w}^{(\tau)} - \eta \nabla E_a(\vec{w}^{(\tau)}), \quad \eta = \text{learning rate}$$

Gradient descent:



Backpropagation (2)



Let's write network output as follows:

$$y(\vec{x}) = h(u(\vec{x})); \quad u(\vec{x}) = \sum_{j=0}^{m} w_{1j}^{(2)} \phi_j(\vec{x})$$

$$\phi_j(\vec{x}) = h\left(\sum_{k=0}^{n} w_{jk}^{(1)} x_k\right) \equiv h\left(v_j(\vec{x})\right)$$

For $E_a = \frac{1}{2}(y_a - t_a)^2$ one obtains for the weights from hidden layer to output:

$$\begin{aligned}\frac{\partial E_a}{\partial w_{1j}^{(2)}} &= (y_a - t_a) h'(u(\vec{x}_a)) \frac{\partial u}{\partial w_{1j}^{(2)}} \\ &= (y_a - t_a) h'(u(\vec{x}_a)) \phi_j(\vec{x}_a)\end{aligned}$$

Further application of the chain rule gives weights from input to hidden layer.

Backpropagation (3)

Backpropagation summary

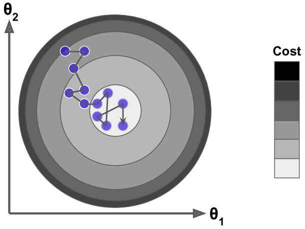
- Make prediction for a given training instance (forward pass)
- Calculate error (value of loss function)
- Go backwards and determine the contribution of each weight (reverse pass)
- Adjust the weights to reduce the error

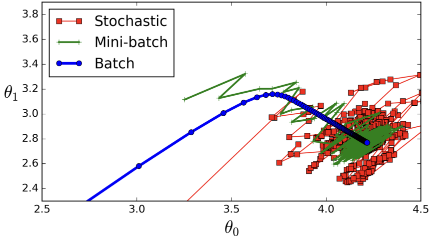
Practical considerations:

- Nowadays, people will implements neural networks with frameworks like Keras or TensorFlow
- No need to implement backpropagation yourself
- TensorFlow efficiently calculates gradient function based on a kind of symbolic differentiation

More on gradient descent

- Stochastic gradient descent
  - just uses one training event at a time
  - fast, but quite irregular approach to the minimum
  - can help escape local minima
  - one can decrease learning rate to settle at the minimum (“simulated annealing”)
- Batch gradient descent
  - use entire training sample to calculate gradient of loss function
  - computationally expensive
- Mini-batch gradient descent
  - calculate gradient for a random sub-sample of the training set



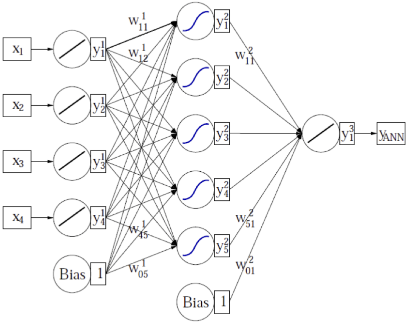


Universal approximation theorem

“A feed-forward network with a single hidden layer containing a finite number of neurons (i.e., a multilayer perceptron), can approximate continuous functions on compact subsets of $\mathbb{R}^n$.”

One of the first versions of the theorem was proved by George Cybenko in 1989 for sigmoid activation functions

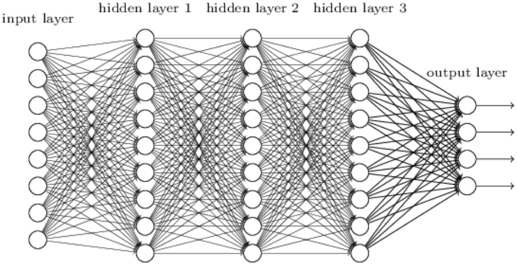
The theorem does not touch upon the algorithmic learnability of those parameters



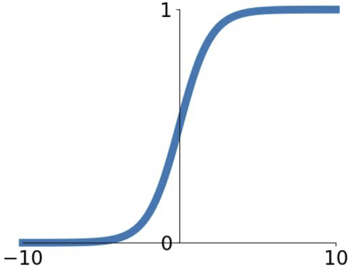
Deep neural networks

Deep networks: many hidden layers with large number of neurons

- Challenges
  - Hard too train ("vanishing gradient problem")
  - Training slow
  - Risk of overtraining
- Big progress in recent years
  - Interest in NN waned before ca. 2006
  - Milestone: paper by G. Hinton (2006): "learning for deep belief nets"
  - Image recognition, AlphaGo, . . .
  - Soon: self-driving cars, . . .



Drawbacks of the sigmoid activation function



Sigmoid

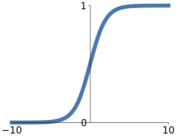
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- Saturated neurons “kill” the gradients
- Sigmoid outputs are not zero-centered
- exp() is a bit compute expensive

Activation functions

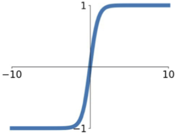
Sigmoid

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



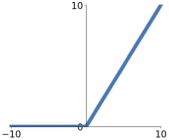
tanh

$\tanh(x)$



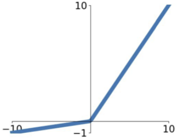
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

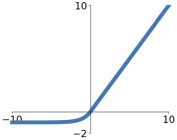


Maxout

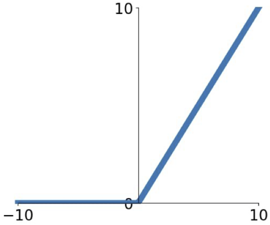
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



ReLU



ReLU
(Rectified Linear Unit)

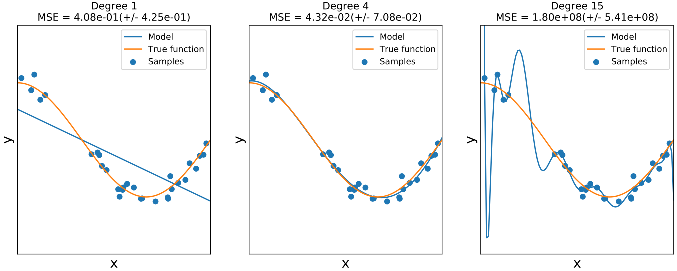
$$f(x) = \max(0, x)$$

- Does not saturate (in $+$region)
- Very computationally efficient
- Converges much faster than sigmoid tanh in practice
- Actually more biologically plausible than sigmoid
- But: gradient vanishes for $x < 0$

Bias-variance tradeoff

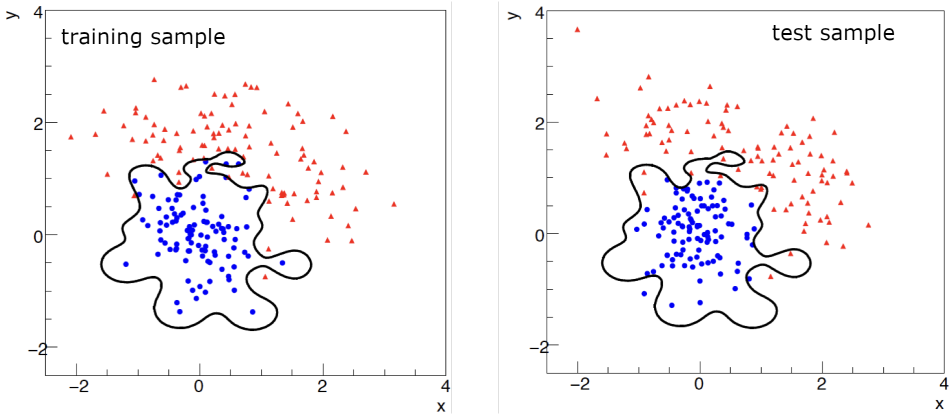
Goal: generalization of training data

- Simple models (few parameters): danger of bias
  - Classifiers with a small number of degrees of freedom are less prone to statistical fluctuations: different training samples would result in similar classification boundaries ("small variance")
- Complex models (many parameters): danger of overfitting
  - large variance of decision boundaries for different training samples



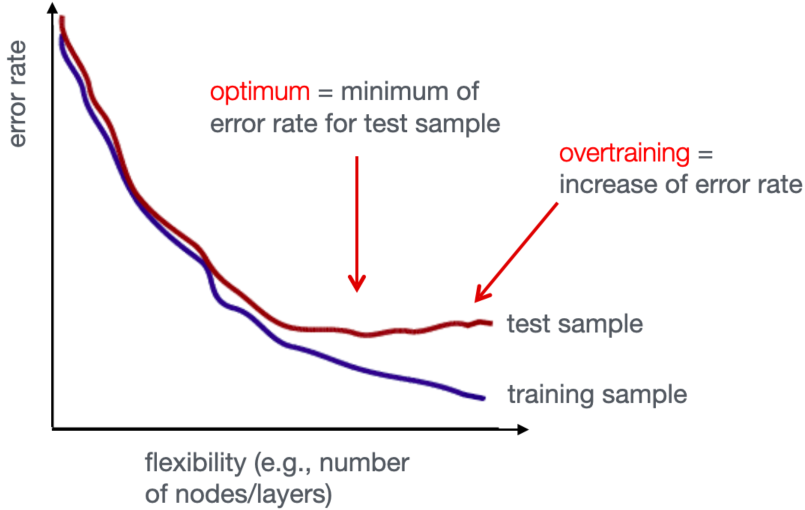
Example of overtraining

Too many neurons/layers make a neural network too flexible
→ overtraining



Monitoring overtraining

Monitor fraction of misclassified events (or loss function:)



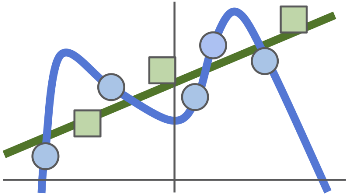
Regularization: Avoid overfitting

http://cs231n.stanford.edu/slides

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)} + \underbrace{\lambda R(W)}$$

Data loss: Model predictions should match training data

Regularization: Model should be "simple", so it works on test data

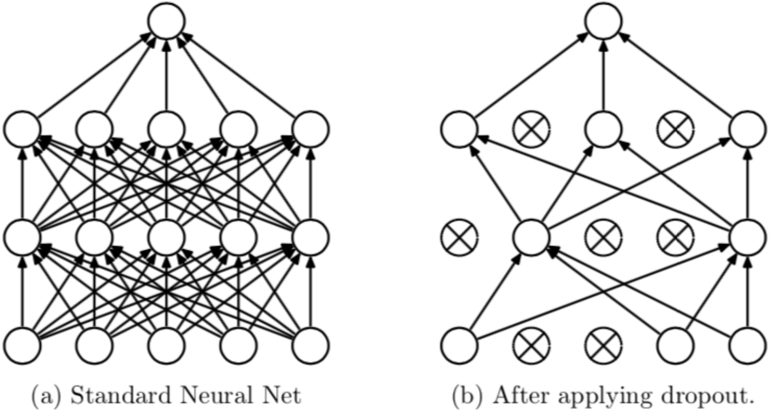


Occam's Razor:
"Among competing hypotheses, the simplest is the best"
William of Ockham, 1285 - 1347

L_1 regularization: $R(W) = \sum_k |W_k|$, L_2 regularization: $R(W) = \sum_k W_k^2$

Another approach to prevent overfitting: Dropout

- Randomly remove nodes during training
- Avoid co-adaptation of nodes



Srivastava et al., "Dropout: A Simple Way to Prevent Neural Networks from Overfitting"

Pros and cons of multi-layer perceptrons

Pros

- Capability to learn non-linear models

Cons

- Loss function can have several local minima
- Hyperparameters need to be tuned
  - number of layers, neurons per layer, and training iterations
- Sensitive to feature scaling
  - preprocessing needed (e.g., scaling of all feature to range [0,1])

Example 1: Boston house prices (MLP regression) (1)

- Objective: predict house prices in Boston suburbs in the mid-1970s
- Boston house data set: 506 instances, 13 features

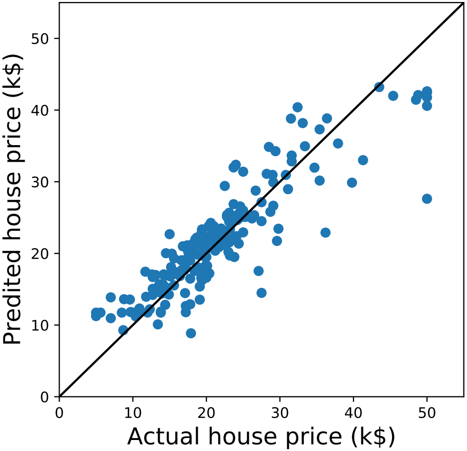
```
- CRIM     per capita crime rate by town
- ZN       proportion of residential land zoned for lots over 25,000 sq.ft.
- INDUS    proportion of non-retail business acres per town
- CHAS     Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
- NOX      nitric oxides concentration (parts per 10 million)
- RM       average number of rooms per dwelling
- AGE      proportion of owner-occupied units built prior to 1940
- DIS      weighted distances to five Boston employment centres
- RAD      index of accessibility to radial highways
- TAX      full-value property-tax rate per $10,000
- PTRATIO  pupil-teacher ratio by town
- B        1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
- LSTAT    % lower status of the population
- MEDV     Median value of owner-occupied homes in $1000's
```

05_neural_networks_boston_house_prices.ipynb

Example 1: Boston house prices (MLP regression) (2)

```
boston = datasets.load_boston()
X = boston.data
y = boston.target

from sklearn.neural_network import MLPRegressor
mlp = MLPRegressor(hidden_layer_sizes=(100),
    activation='logistic', random_state=1, max_iter=5000)
mlp.fit(X_train, y_train)

y_pred_mlp = mlp.predict(X_test)

rms = np.sqrt(mean_squared_error(y_test, y_pred_mlp))
print(f"root mean square error {rms:.2f}")
```

Example 1: Boston house prices (MLP regression) (3)

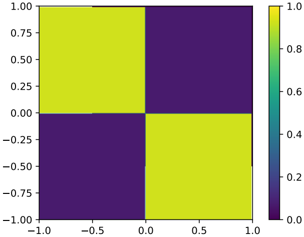


Exercise 1: XOR

05_neural_networks_ex_1_xor.ipynb

a) Define a multi-layer perceptron classifier that learns the XOR problem.

```
from sklearn.neural_network import MLPClassifier

X = [[0, 0], [0, 1], [1, 0], [1, 1]]
y = [0, 1, 1, 0]
```



b) Define a multi-layer perceptron regressor that fits the depicted 2d data (see notebook).

c) Plot the mean square error vs. the number of number of training epochs for b).

Exercise 2: Visualising decision boundaries of classifiers

05_neural_networks_ex_2_decision_boundaries.ipynb

Visualize the decision boundaries of a scikit-learn decision tree, a scikit-learn multi-layer perceptron, and XGBoost for different toy data sets.

Exercise 3: Boston house prices (hyperparameter optimization)

05_neural_networks_ex_3_boston_house_prices.ipynb

a) Can you find better hyperparameters (number of hidden layers, neurons per layer, loss function, . . .)? Try this first by hand.

b) Now use sklearn.model_selection.GridSearchCV to find optimal parameters.

TensorFlow

- Powerful open source library with a focus on deep neural networks
- Performs computations of data flow graphs
- Takes care of computing gradients of the defined functions (*automatic differentiation*)
- Computations in parallel on multiple CPUs or GPUs
- Developed by the Google Brain team
- Initial release in 2015
- `https://www.tensorflow.org/`

Keras

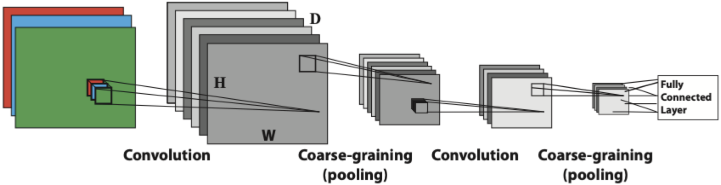
- Open-source library providing high-level building blocks for developing deep-learning models
- Uses TensorFlow as *backend engine* for low-level tensor manipulation (version 2.4)
- Part of TensorFlow core API since TensorFlow 1.4 release
- Over 375,000 individual users as of early-2020
- Primary author: François Chollet (Google engineer)
- `https://keras.io/`

Example 2: Boston house prices with Keras

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(64, activation='relu',
                       input_shape=(train_data.shape[1],)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(1))
model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])

model.fit(partial_train_data, partial_train_targets,
              epochs=num_epochs, batch_size=1, verbose=0)

# Evaluate the model on the validation data
val_mse, val_mae = model.evaluate(val_data, val_targets, verbose=0)
```

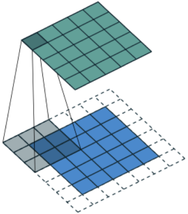
05_neural_networks_boston_keras.ipynb

Convolutional neutral networks (CNNs)



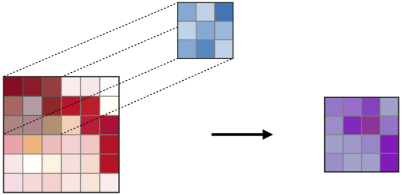
- CNNs emerged from the study of the visual cortex
- Behind many deep learning successes (e.g. in image recognition)
- Partially connected layers
  - Fully connected layers impractical for large images (too many neurons, overfitting)
- Key component: Convolutional layers
  - Set of learnable filters
  - Low-level features at the first layers; high-level features a the end

Sliding 3×3 filter

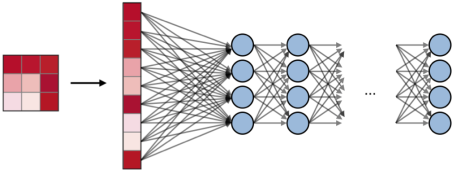


Different types of layers in a CNN

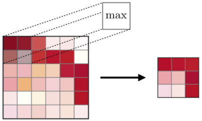
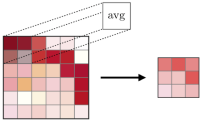
1. Convolutional layers



3. Fully connected layers



2. Pooling layers

	Max pooling	**Average pooling**
Purpose	Each pooling operation selects the maximum value of the current view	Each pooling operation averages the values of the current view
Illustration		
Comments	- Preserves detected features - Most commonly used	- Downsamples feature map - Used in LeNet

Afshine Amidi, Shervine Amidi

Convolutional Neural Networks cheatsheet

MNIST classification with a CNN in Keras

```
from keras.models import Sequential
from keras.layers import Dense, Flatten, MaxPooling2D, Conv2D, Input, Dropout

# conv layer with 8 3x3 filters

model = Sequential(
    [
        Input(shape=input_shape),
        Conv2D(8, kernel_size=(3, 3), activation="relu"),
        MaxPooling2D(pool_size=(2, 2)),
        Flatten(),
        Dense(16, activation="relu"),
        Dense(num_classes, activation="softmax"),
    ]
)

model.summary()
```

Defining the CNN in Keras (2)

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 8)         80
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 13, 13, 8)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 1352)              0
_________________________________________________________________
dense_2 (Dense)              (None, 16)                21648
_________________________________________________________________
dense_3 (Dense)              (None, 10)                170
=================================================================
Total params: 21,898
Trainable params: 21,898
Non-trainable params: 0
```

Model definition

Using Keras, you have to `compile` a model, which means adding the loss function, the optimizer algorithm and validation metrics to your training setup.

```
model.compile(loss="categorical_crossentropy",
        optimizer="adam",
        metrics=["accuracy"])
```

Model training

```python
from keras.callbacks import ModelCheckpoint, EarlyStopping

checkpoint = ModelCheckpoint(
            filepath="mnist_keras_model.h5",
            save_best_only=True,
            verbose=1)
early_stopping = EarlyStopping(patience=2)

history = model.fit(x_train, y_train, # Training data
            batch_size=200, # Batch size
            epochs=50, # Maximum number of training epochs
            validation_split=0.5, # Use 50% of the train dataset for validation
            callbacks=[checkpoint, early_stopping]) # Register callbacks
```

Exercise 4: Training a digit-classification neural network on the MNIST dataset using Keras

05_neural_networks_ex_4_mnist_keras_train.ipynb

a) Plot training and validation loss as well as training and validation accuracy as a function of the number of epochs

b) Determine the accuracy of the fully trained model.

c) Create a second notebook that reads the trained model (`mnist_keras_model.h5`). Read `your_own_digit.png` and classify it. Create your own 28×28 pixel digits with a program like gimp and check how the model performs.

Exercise 5: Higgs data set (1)

Application of deep neural networks for separation of signal and background in an exotic Higgs scenario

In this exercise we want to explore various techniques to optimize the event selection in the search for supersymmetric Higgs bosons at the LHC. In supersymmetry the Higgs sector constitutes of five Higgs bosons in contrast to the single Higgs in the standard model. Here we deal with a heavy Higgs boson which decays into two W-bosons and a standard Higgs boson ($H^0 \to W^+W^-h$) which decay further into leptons ($W^\pm \to l^\pm\nu$) and b-quarks ($h \to b\bar{b}$) respectively.
This exercise is based on a Nature paper (Pierre Baldi, Peter Sadowski, Daniel Whiteson) which contains much more information like general background information, details about the selection variables and links to large sets of simulated events. You might also use the paper as inspiration for the solution of this exercise.

Exercise 5: Higgs data set (2)

The two dataset consists of 10k and 100k events respectively. For each event 29 variables are stored:

```
0: classification (1 = signal, 0 = background)
1 - 21 : low level quantities (var1 - var21)
22 -28 : high level quantities (var22 - var28)
```

You can read the data as follows:

```
#filename = "https://www.physi.uni-heidelberg.de/~reygers/lectures/2021/ml/data/HIGGS_10k.csv"
filename = "https://www.physi.uni-heidelberg.de/~reygers/lectures/2021/ml/data/HIGGS_100k.csv"

df = pd.read_csv(filename, engine='python')
```

a) Use a classifier of you choice to separate signal and background events. Determine the accuracy score.

b) Compare the results when using i) the low level quantities and ii) the high level quantities

Practical advice – Which algorithm to choose?

From Kaggle competitions:

Structured data: “High level” features that have meaning:

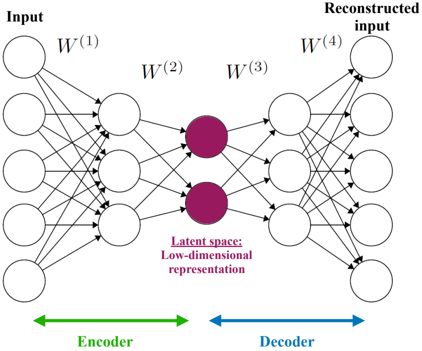
- feature engineering + decision trees
- Random forests
- XGBoost

Unstructured data: “Low level” features, no individual meaning:

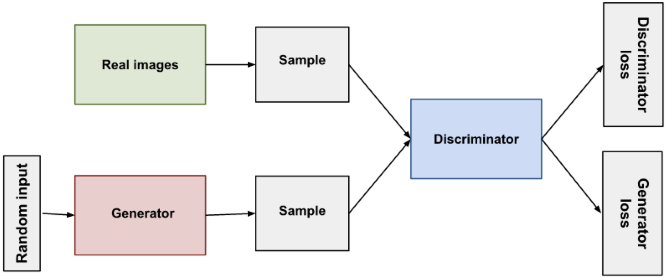
- deep neural networks
- e.g. image classification: convolutional NN

Outlook: Autoencoders

- Unsupervised method based on neural networks to learn a representation of the input data
- Autoencoders learn to copy the input to the output layer
  - low dimensional coding of the input in the central layer
- The decoder generates data based on the coding (*generative model*)
- Applications
  - Dimensionality reduction
  - Denoising of data
  - Machine translation



Outlook: Generative adversarial network (GANs)



https://developers.google.com/machine-learning/gan/gan_structure

- Discriminator's classification provides a signal that the generator uses to update its weights
- Application in particle physics: fast detector simulation
- Full GEANT simulation usually very CPU intensive

The future

“Das Interessante an unserer Intelligenz ist, dass wir Go spielen können und dann vom Tisch aufstehen und Essen machen können, was eine Maschine nicht kann.”

Bernhard Schölkopf, Max-Planck-Institut für intelligente Systeme (Interview FAZ)

“My view is throw it all away and start again”

Geoffrey Hinton (DNN pioneer) on deep neural networks and backpropagation (Interview, 2017)