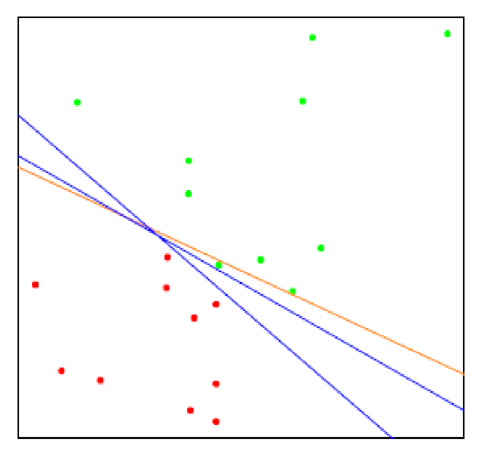


Parallel & Scalable Machine Learning

Introduction to Machine Learning Algorithms

Prof. Dr. – Ing. Morris Riedel

Adjunct Associated Professor

School of Engineering and Natural Sciences, University of Iceland

Research Group Leader, Juelich Supercomputing Centre, Germany

LECTURE 6

Validation & Regularization

February 26th, 2019

Juelich Supercomputing Centre, Juelich, Germany

PRACE PARTNERSHIP FOR ADVANCED COMPUTING IN EUROPE TRAINING PORTAL

UNIVERSITY OF ICELAND
SCHOOL OF ENGINEERING AND NATURAL SCIENCES
FACULTY OF INDUSTRIAL ENGINEERING, MECHANICAL ENGINEERING AND COMPUTER SCIENCE

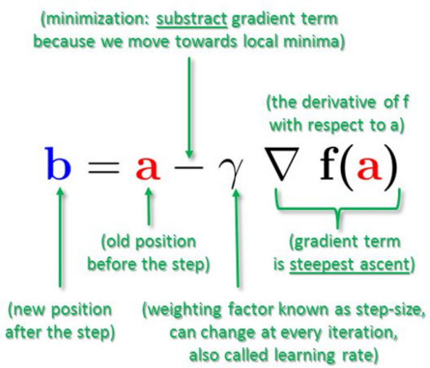
JÜLICH Forschungszentrum

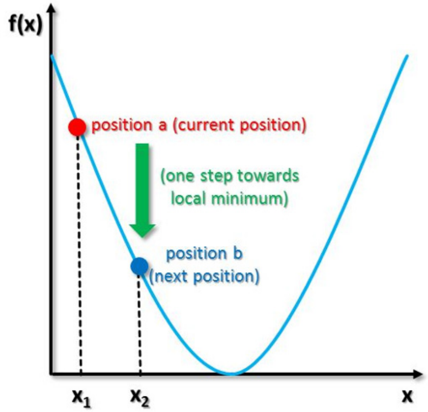
HELMHOLTZ RESEARCH FOR GRAND CHALLENGES

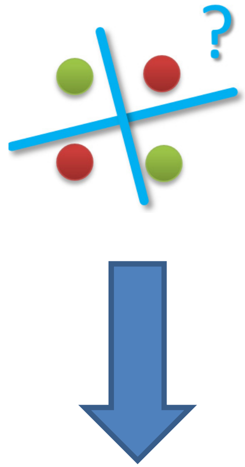
DEEP Projects

Review of Lecture 5 – Feed Forward Neural Networks

1952 Stochastic Gradient Descent
- Optimization

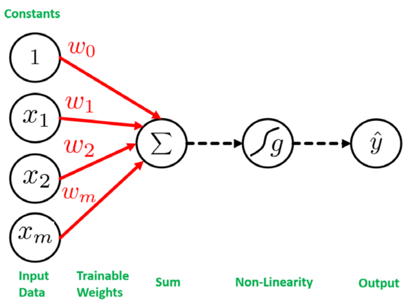


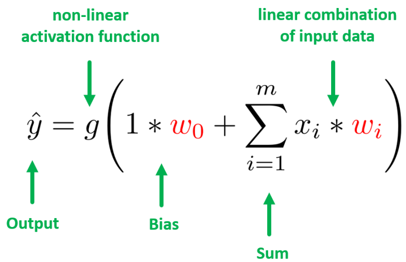


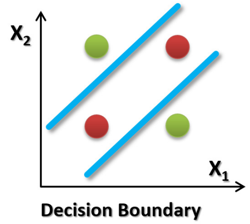


1958 Perceptron Linear Learning Model
- Learnable Weights

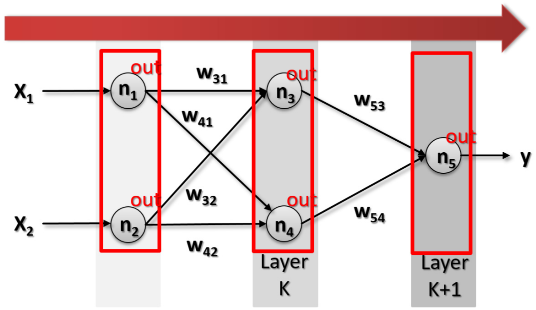
⋮



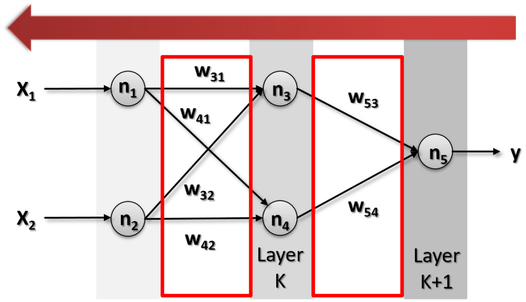




1986 Backpropagation of Error in Learning through Layers
- Multi-Layer Perceptron



(forward pass)



(backward pass)

1995 Deep Convolutional Neural Networks
- Image Recognition

⋮

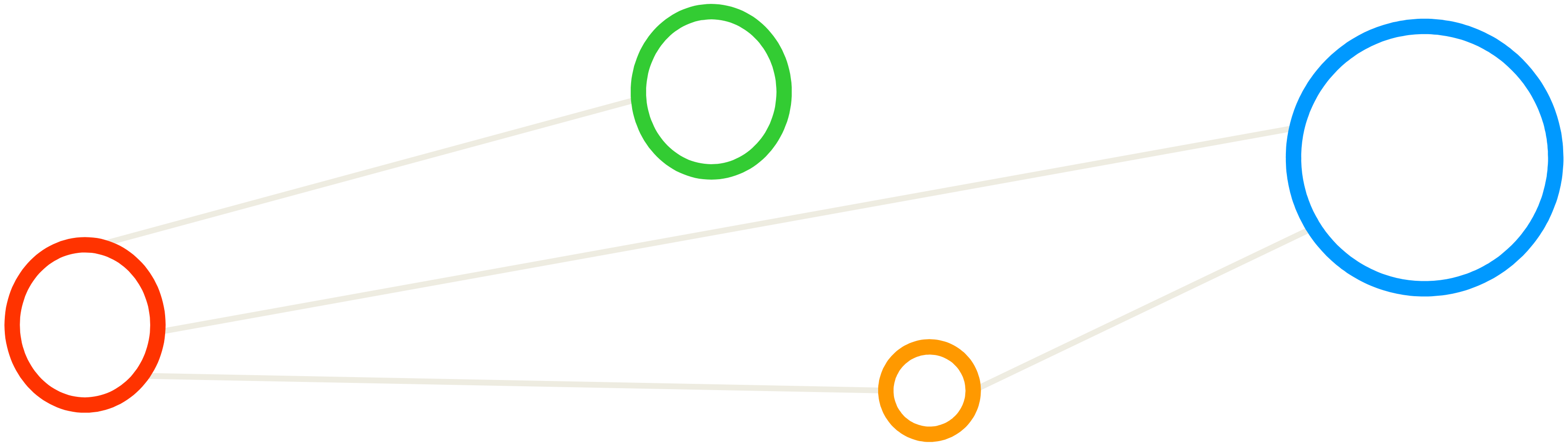
Outline of the Course

1. Parallel & Scalable Machine Learning driven by HPC
2. Introduction to Machine Learning Fundamentals
3. Introduction to Machine Learning Fundamentals
4. Feed Forward Neural Networks
5. Feed Forward Neural Networks
6. Validation and Regularization
7. Validation and Regularization
8. Data Preparation and Performance Evaluation
9. Data Preparation and Performance Evaluation
10. Theory of Generalization
11. Unsupervised Clustering and Applications
12. Unsupervised Clustering and Applications
13. Deep Learning Introduction

Theoretical Lectures

Practical Lectures

Outline

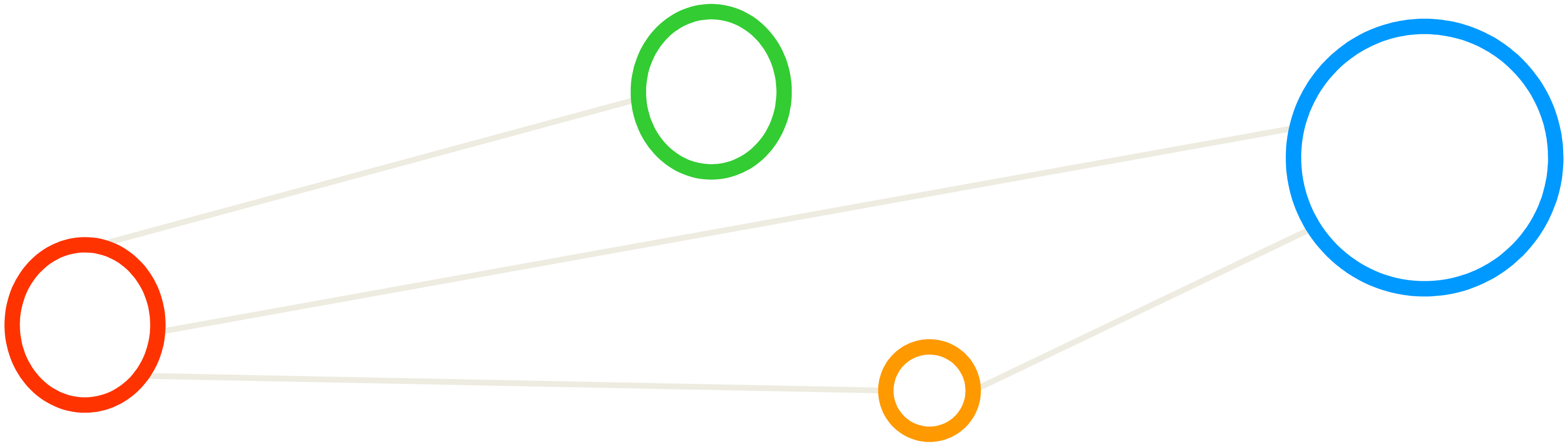


Outline

- Validation for Model Selection
  - Problem of Overfitting
  - Overfitting Reasoning & Validation
  - Creating ANN Network Topologies
  - Many Parameters & Hidden Layers
  - Validation Datasets & Splits

- Regularization
  - Overfitting Reasoning
  - Regularization & Validation Counter Approach
  - Regularization Techniques
  - Dropout Regularizer
  - Optimizers RMSprop & Adam

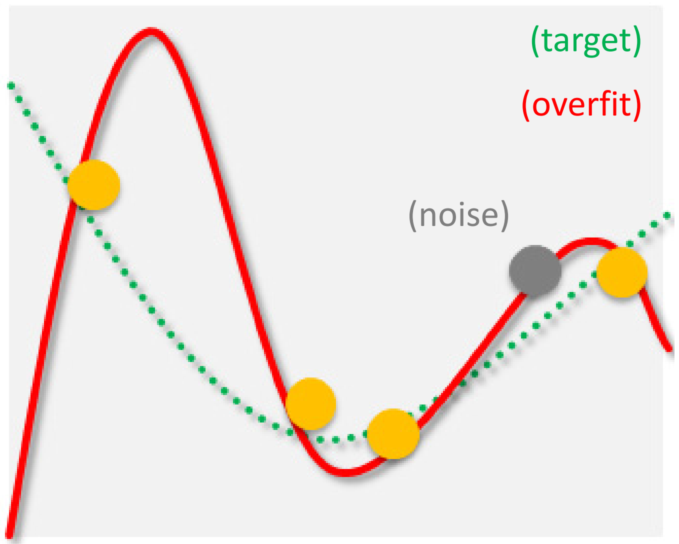
Validation for Model Selection



Machine Learning Challenges – Problem of Overfitting

- Overfitting refers to fit the data too well – more than is warranted – thus may misguide the learning
- Overfitting is not just 'bad generalization' - e.g. the VC dimension covers noiseless & noise targets
- Theory of Regularization are approaches against overfitting and prevent it using different methods

- Key problem: noise in the target function leads to overfitting
  - Effect: 'noisy target function' and its noise misguides the fit in learning
  - There is always 'some noise' in the data
  - Consequence: poor target function ('distribution') approximation
- Example: Target functions is second order polynomial (i.e. parabola)
  - Using a higher-order polynomial fit
  - Perfect fit: low $E_{in}(g)$, but large $E_{out}(g)$



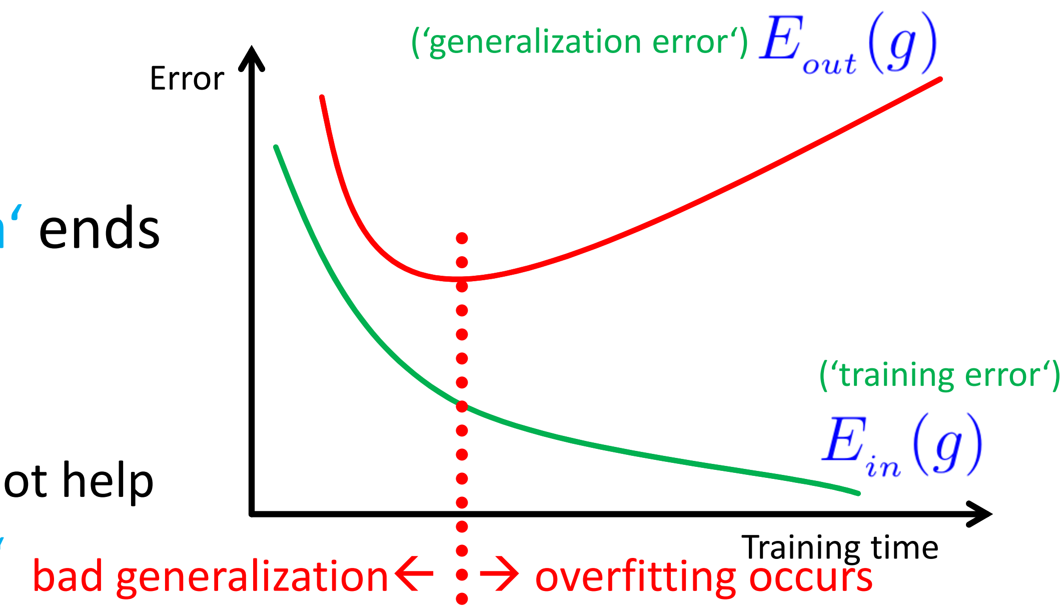
(but simple polynomial works good enough)
('over': here meant as 4th order, a 3rd order would be better, 2nd best)

Problem of Overfitting – Clarifying Terms

- **A good model must have low training error (E_{in}) and low generalization error (E_{out})**
- **Model overfitting is if a model fits the data too well (E_{in}) with a poorer generalization error (E_{out}) than another model with a higher training error (E_{in})**

[1] Introduction to Data Mining

- Overfitting & Errors
  - $E_{in}(g)$ goes down
  - $E_{out}(g)$ goes up
- 'Bad generalization area' ends
  - Good to reduce $E_{in}(g)$
- 'Overfitting area' starts
  - Reducing $E_{in}(g)$ does not help
  - Reason 'fitting the noise'



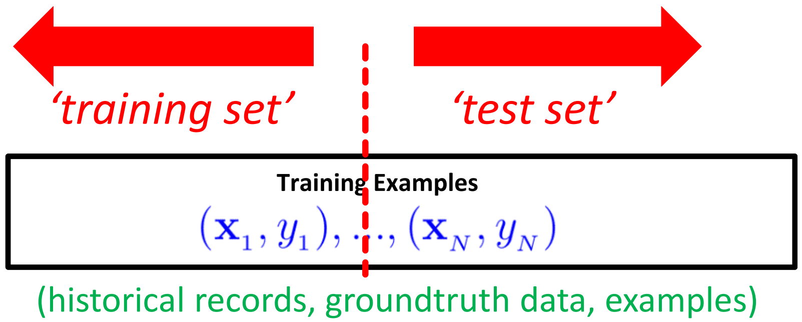
- **The two general approaches to prevent overfitting are (1) regularization and (2) validation**

Terminologies & Different Dataset Elements

- Target Function $f: X \to Y$
  - Ideal function that 'explains' the data we want to learn
- Labelled Dataset (samples)
  - 'in-sample' data given to us: $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$
- Learning vs. Memorizing
  - The goal is to create a system that works well 'out of sample'
  - In other words we want to classify 'future data' (ouf of sample) correct
- Dataset Part One: Training set
  - Used for training a machine learning algorithms
  - Result after using a training set: a trained system
- Dataset Part Two: Test set
  - Used for testing whether the trained system might work well
  - Result after using a test set: accuracy of the trained model

Model Evaluation – Training and Testing Phases

- Different Phases in Learning (cf. day one remote sensing)
  - Training phase is a hypothesis search
  - Testing phase checks if we are on right track
    (once the hypothesis clear)
- Work on 'training examples'
  - Create two disjoint datasets
  - One used for training only
    (aka training set)
  - Another used for testing only
    (aka test set)
  - Exact seperation is rule of thumb per use case (e.g. 10 % training, 90% test)
  - Practice: If you get a dataset take immediately test data away
    ('throw it into the corner and forget about it during modelling')
  - Reasoning: Once we learned from training data it has an 'optimistic bias'



Learning Approaches – Supervised Learning – Formalization

- Each observation of the predictor measurement(s) has an associated response measurement:
  - Input $\mathbf{x} = x_1, \ldots, x_d$
  - Output $y_i, i = 1, .., n$
  - Data $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$

Training Examples

$(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$

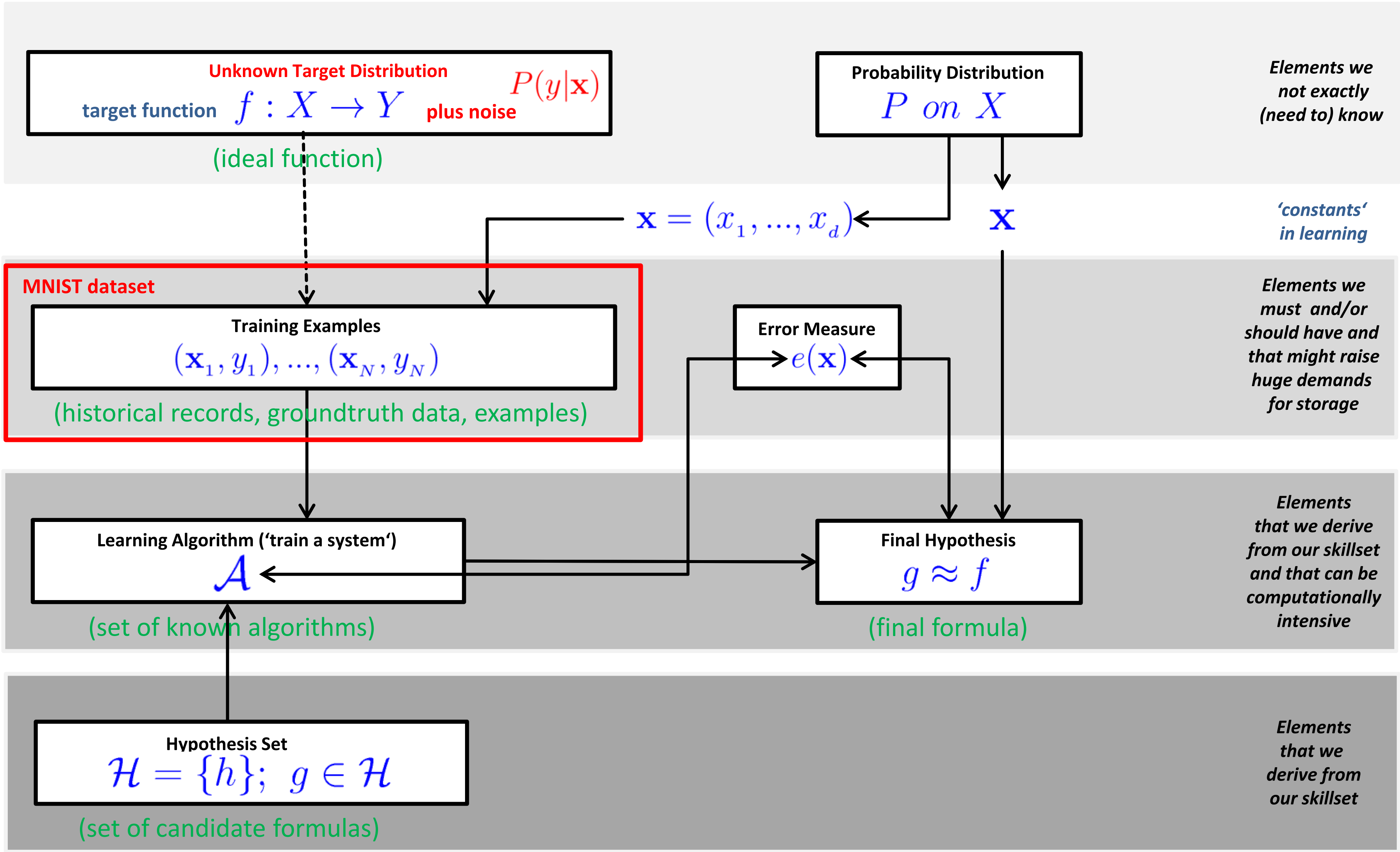
(historical records, groundtruth data, examples)

- Goal: Fit a model that relates the response to the predictors
  - Prediction: Aims of accurately predicting the response for future observations
  - Inference: Aims to better understanding the relationship between the response and the predictors

- **Supervised learning approaches fits a model that related the response to the predictors**
- **Supervised learning approaches are used in classification algorithms such as SVMs**
- **Supervised learning works with data = [input, correct output]**

[1] An Introduction to Statistical Learning

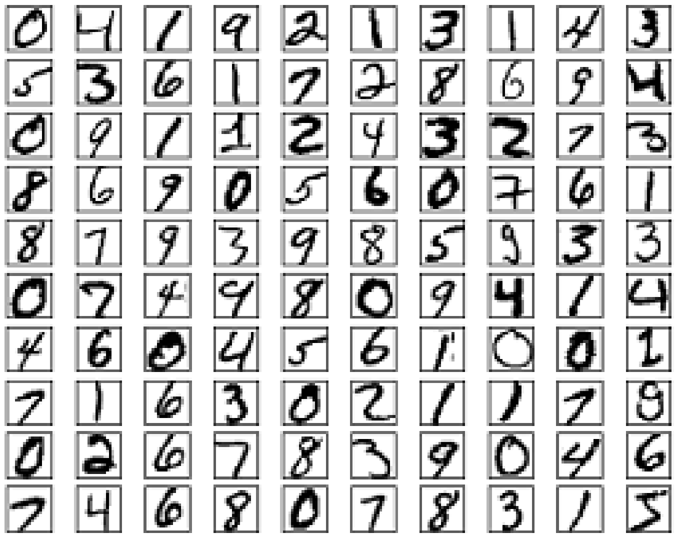
Supervised Learning – Training Examples



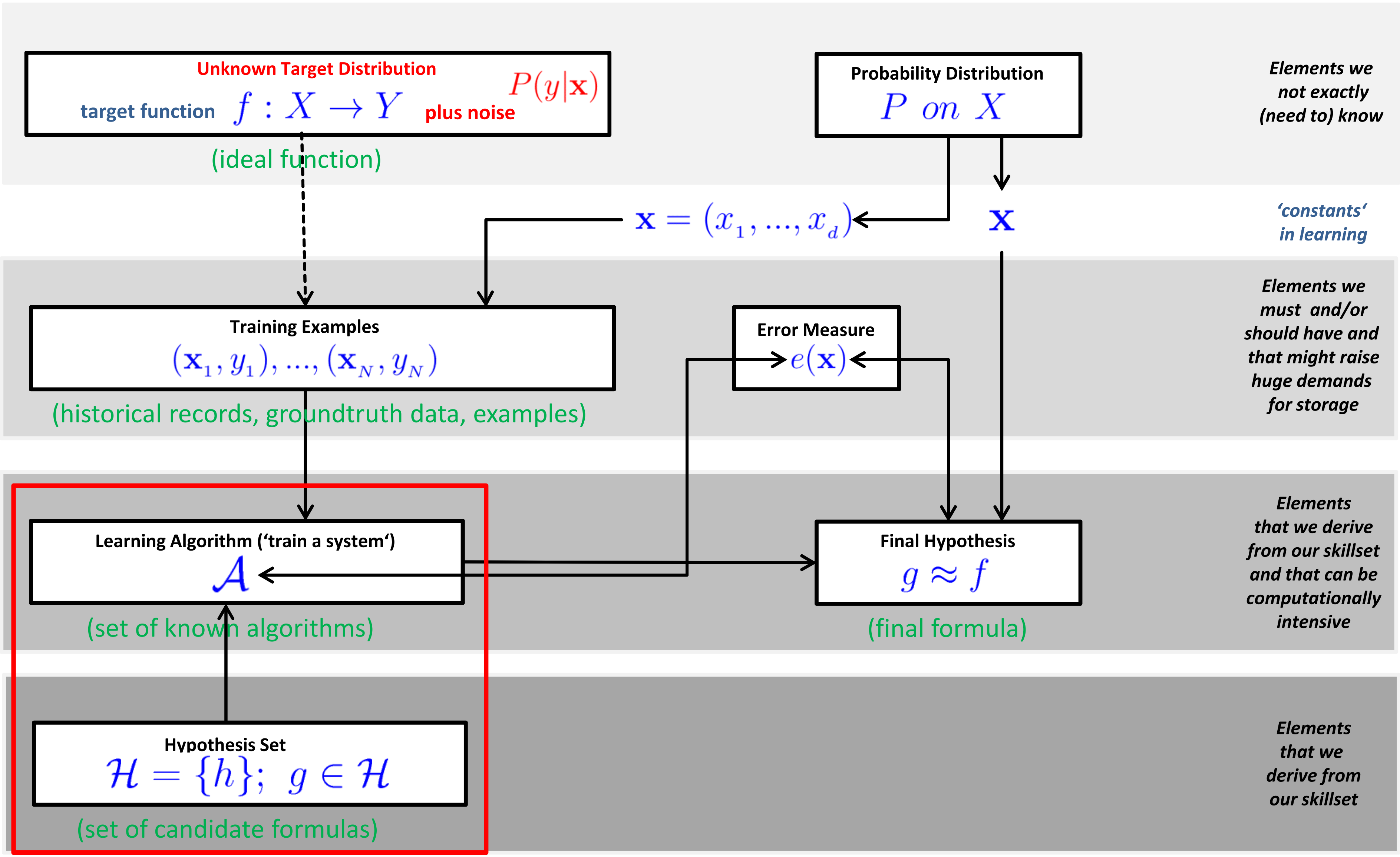
Handwritten Character Recognition MNIST Dataset

- Metadata
  - Subset of a larger dataset from US National Institute of Standards (NIST)
  - Handwritten digits including corresponding labels with values 0 to 9
  - All digits have been size-normalized to 28 * 28 pixels
    and are centered in a fixed-size image for direct processing
  - Not very challenging dataset, but good for experiments / tutorials

- Dataset Samples
  - Labelled data (10 classes)
  - Two separate files
    for training and test
  - 60000 training samples (~47 MB)
  - 10000 test samples (~7.8 MB)



Supervised Learning – Many Hypothesis to Choose



Different Models – Understanding the Hypothesis Set

Hypothesis Set

$$\mathcal{H} = \{h\};\ g \in \mathcal{H}$$

$$\mathcal{H} = \{h_1, \ldots, h_m\};$$

(all candidate functions
derived from models
and their parameters)

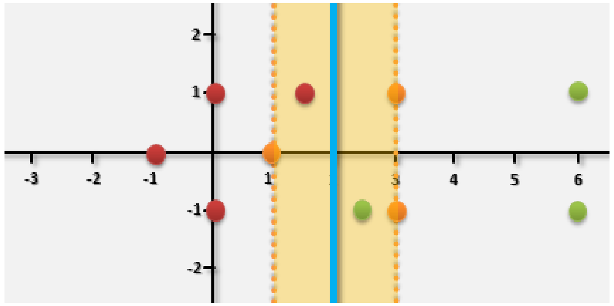
- **Choosing from various model approaches $h_1, \ldots, h_m$ is a different hypothesis**
- **Additionally a change in model parameters of $h_1, \ldots, h_m$ means a different hypothesis too**

'select one function'
that best approximates

Final Hypothesis

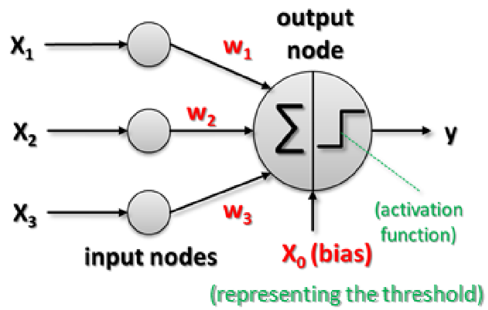
$$g \approx f$$

h_1



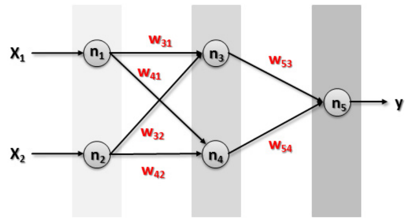
(e.g. support vector machine model)

h_2



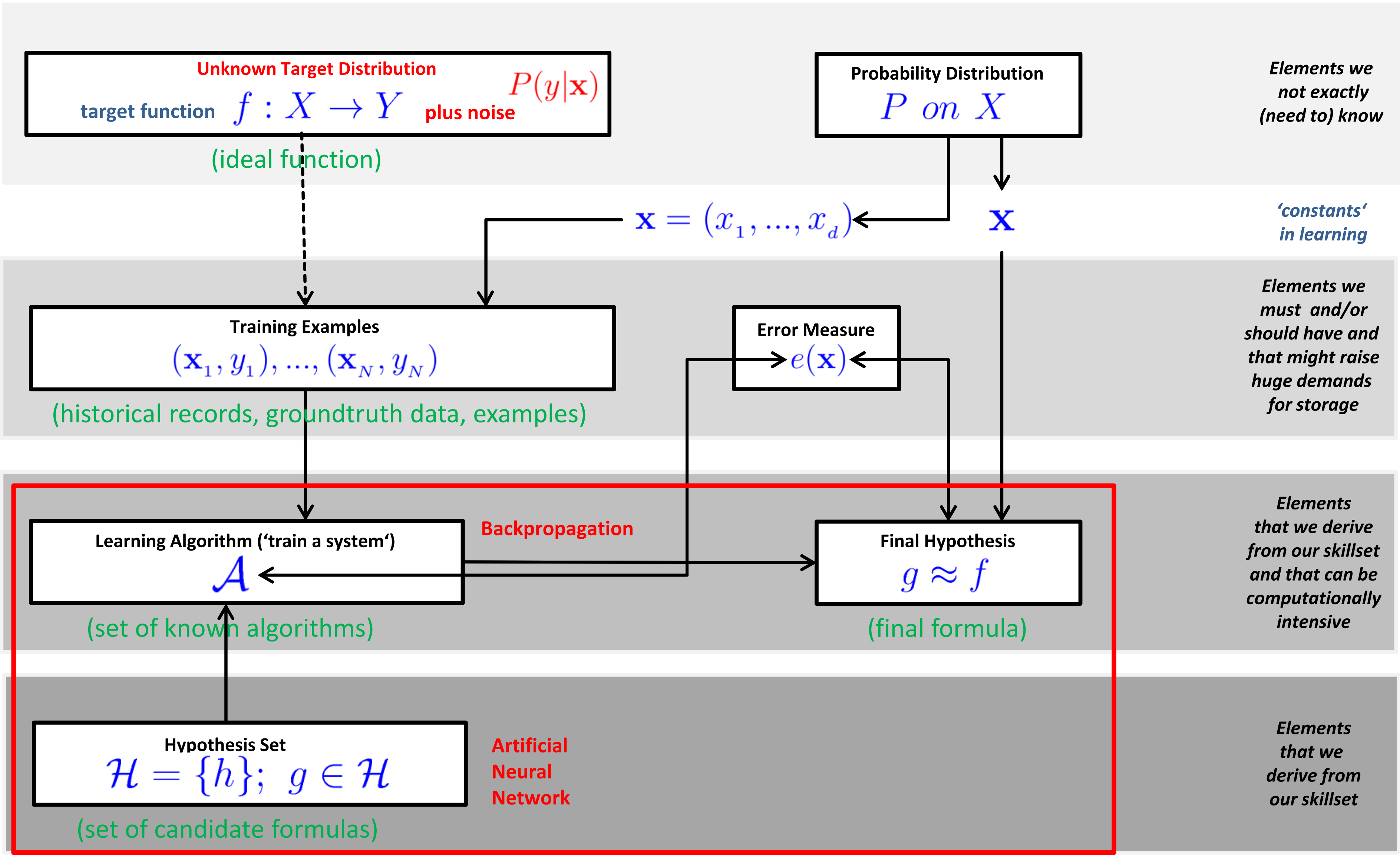
(e.g. linear perceptron model)

h_m



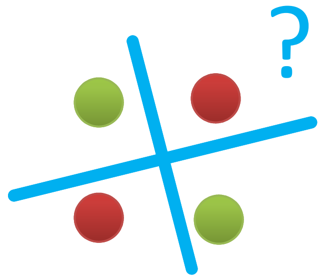
(e.g. artificial neural network model)

Supervised Learning – Training Examples



Artificial Neural Network (ANN)

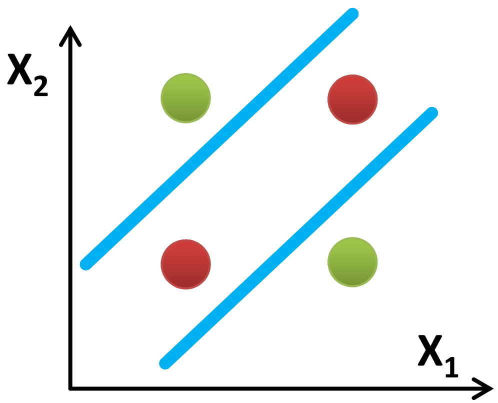
- Simple perceptrons fail: 'not linearly seperable'



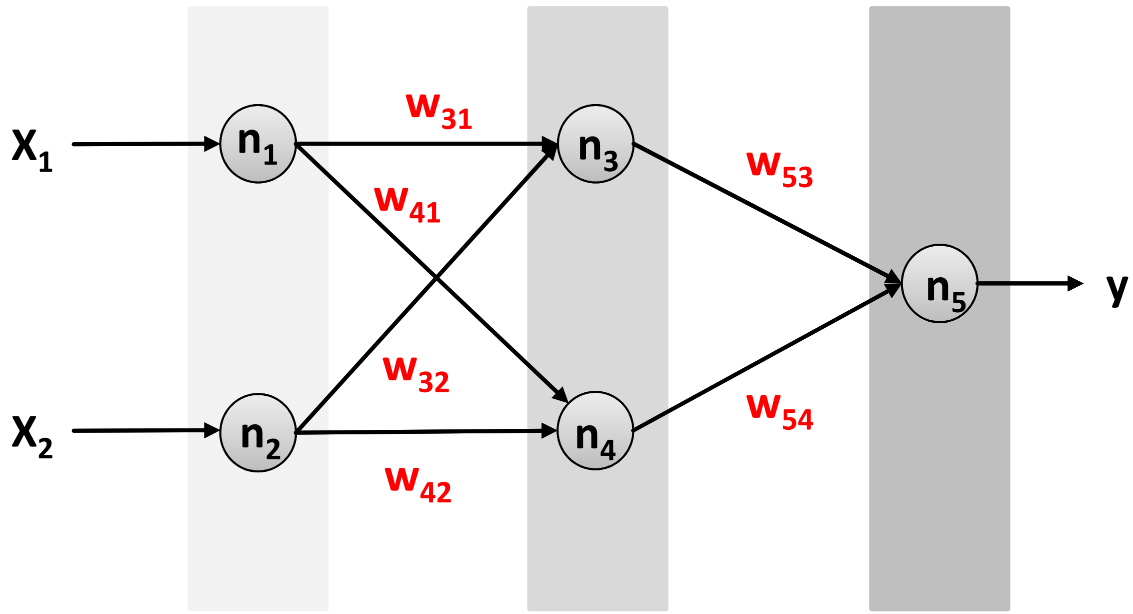
X_1	X_2	Y
0	0	-1
1	0	1
0	1	1
1	1	-1

Labelled Data Table

(Idea: instances can be classified using two lines at once to model XOR)



Decision Boundary



Two-Layer, feed-forward Artificial Neural Network topology

High-level Tools – Keras

- Keras is a high-level deep learning library implemented in Python that works on top of existing other rather low-level deep learning frameworks like Tensorflow, CNTK, or Theano
- The key idea behind the Keras tool is to enable faster experimentation with deep networks
- Created deep learning models run seamlessly on CPU and GPU via low-level frameworks

```
keras.layers.Dense(units,
                   activation=None,
                   use_bias=True,
                   kernel_initializer='glorot_uniform',
                   bias_initializer='zeros',
                   kernel_regularizer=None,
                   bias_regularizer=None,
                   activity_regularizer=None,
                   kernel_constraint=None,
                   bias_constraint=None)

keras.optimizers.SGD(lr=0.01,
                     momentum=0.0,
                     decay=0.0,
                     nesterov=False)
```

- Tool Keras supports inherently the creation of artificial neural networks using Dense layers and optimizers (e.g. SGD)
- Includes regularization (e.g. weight decay) or momentum

[2] Keras Python Deep Learning Library

ANN – MNIST Dataset – Create ANN Blueprint

✓ Data Preprocessing done (i.e. data normalization, reshape, etc.)

1. Define a neural network topology
   - Which layers are required?
   - Think about input layer need to match the data – what data we had?
   - Maybe hidden layers?
   - Think Dense layer – Keras?
   - Think about final Activation as Softmay (cf. Day One) → output probability
2. Compile the model → model representation for Tensorflow et al.
   - Think about what loss function you want to use in your problem?
   - What is your optimizer strategy, e.g. SGD (cf. Day One)
3. Fit the model → the model learning takes place
   - How long you want to train (e.g. NB_EPOCHS)
   - How much samples are involved (e.g. BATCH_SIZE)

Exercises – Create a Simple ANN Model – One Dense

MNIST Dataset – Model Parameters & Data Normalization

```
import numpy as np
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.optimizers import SGD
from keras.utils import np_utils
```

```
# parameter setup
NB_EPOCH = 20
BATCH_SIZE = 128
NB_CLASSES = 10 # number of outputs = number of digits
OPTIMIZER = SGD() # optimization technique
VERBOSE = 1
```

```
# download and shuffled as training and testing set
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

```
# X_train is 60000 rows of 28x28 values --> reshaped in 60000 x 784
RESHAPED = 784
X_train = X_train.reshape(60000, RESHAPED)
X_test = X_test.reshape(10000, RESHAPED)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
```

```
# normalize
X_train /= 255
X_test /= 255
```

```
# output number of samples
print(X_train.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')
```

- **NB_CLASSES: 10 Class Problem**
- **NB_EPOCH: number of times the model is exposed to the overall training set – at each iteration the optimizer adjusts the weights so that the objective function is minimized**
- **BATCH_SIZE: number of training instances taken into account before the optimizer performs a weight update to the model**
- **OPTIMIZER: Stochastic Gradient Descent ('SGD') – only one training sample/iteration**

- **Data load shuffled between training and testing set in files**
- **Data preparation, e.g. X_train is 60000 samples / rows of 28 x 28 pixel values that are reshaped in 60000 x 784 including type specification (i.e. float32)**
- **Data normalization: divide by 255 – the max intensity value to obtain values in range [0,1]**

MNIST Dataset – A Multi Output Perceptron Model

- The Sequential() Keras model is a linear pipeline (aka ‘a stack‘) of various neural network layers including Activation functions of different types (e.g. softmax)

- Dense() represents a fully connected layer used in ANNs that means that each neuron in a layer is connected to all neurons located in the previous layer

- The non-linear activation function ‘softmax‘ is a generalization of the sigmoid function – it squashes an n-dimensional vector of arbitrary real values into a n-dimenensional vector of real values in the range of 0 and 1 – here it aggregates 10 answers provided by the Dense layer with 10 neurons

```
# convert class label vectors using one hot encoding
Y_train = np_utils.to_categorical(y_train, NB_CLASSES)
Y_test = np_utils.to_categorical(y_test, NB_CLASSES)

# model Keras sequential
model = Sequential()

# add fully connected layer - input with output
model.add(Dense(NB_CLASSES, input_shape=(RESHAPED,)))

# add activation function layer to get class probabilities
model.add(Activation('softmax'))

# printout a summary of the model to understand model complexity
model.summary()

# specify loss, optimizer and metric
model.compile(loss='categorical_crossentropy', optimizer=OPTIMIZER, metrics=['accuracy'])

# model training
history = model.fit(X_train, Y_train, batch_size=BATCH_SIZE, epochs=NB_EPOCH, verbose=VERBOSE)

# model evaluation
score = model.evaluate(X_test, Y_test, verbose=VERBOSE)
print("Test score:", score[0])
print('Test accuracy:', score[1])
```

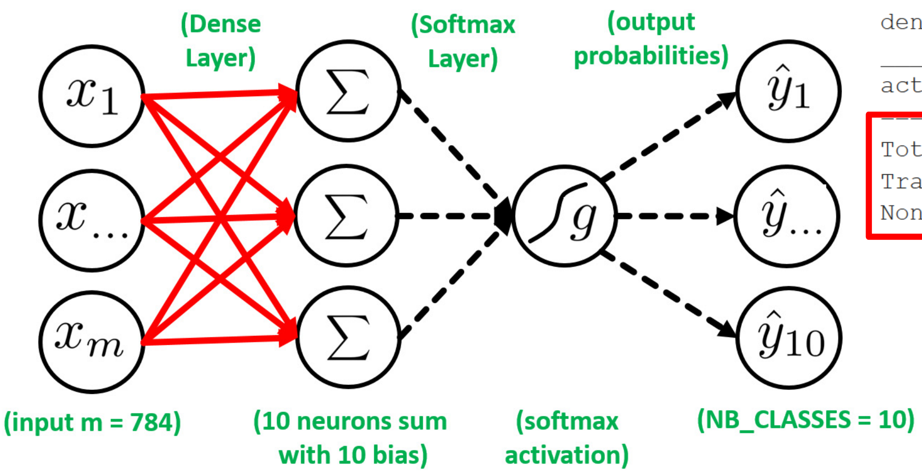
- Loss function is a multi-class logarithmic loss: target is ***ti,j*** and prediction is ***pi,j***

$$L_i = -\Sigma_j t_{i,j} \log(p_{i,j})$$

- Train the model (‘fit‘)

MNIST Dataset & Model Summary & Parameters

- Activation Function Softmax
  - Softmax enables probabilities for 10 classes




```
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 10)                7850
_________________________________________________________________
activation_1 (Activation)    (None, 10)                0
=================================================================
Total params: 7,850
Trainable params: 7,850
Non-trainable params: 0
```

(parameters = 784 * 10 + 10 bias = 7850)

- **Relevant for validation: Choosing a model with different layers is a model selection that directly also influences the number of parameters (e.g. add Dense layer from Keras means new weights)**

```
# printout a summary of the model to understand model complexity
model.summary()
```

Model Evaluation – Testing Phase & Confusion Matrix

- Model is fixed
  - Model is just used with the testset
  - Parameters are set
- Evaluation of model performance
  - Counts of test records that are incorrectly predicted
  - Counts of test records that are correctly predicted
  - E.g. create confusion matrix for a two class problem

Counting per sample		Predicted Class	
		Class = 1	Class = 0
Actual Class	Class = 1	f_{11}	f_{10}
	Class = 0	f_{01}	f_{00}

(serves as a basis for further performance metrics usually used)

Model Evaluation – Testing Phase & Performance Metrics

Counting per sample		Predicted Class	
		Class = 1	Class = 0
Actual Class	Class = 1	f_{11}	f_{10}
	Class = 0	f_{01}	f_{00}

(100% accuracy in learning often points to problems using machine learning methos in practice)

- Accuracy (usually in %)

$$Accuracy = \frac{number\ of\ correct\ predictions}{total\ number\ of\ predictions}$$

- Error rate

$$Error\ rate = \frac{number\ of\ wrong\ predictions}{total\ number\ of\ predictions}$$

Exercises – Evaluate Multi Output Perceptron Model

MNIST Dataset – A Multi Output Perceptron Model – Output

```
Epoch 7/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.4419 - acc: 0.8838
Epoch 8/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.4271 - acc: 0.8866
Epoch 9/20
60000/60000 [==============================] - 2s 25us/step - loss: 0.4151 - acc: 0.8888
Epoch 10/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.4052 - acc: 0.8910
Epoch 11/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.3968 - acc: 0.8924
Epoch 12/20
60000/60000 [==============================] - 2s 25us/step - loss: 0.3896 - acc: 0.8944
Epoch 13/20
60000/60000 [==============================] - 2s 26us/step - loss: 0.3832 - acc: 0.8956
Epoch 14/20
60000/60000 [==============================] - 2s 25us/step - loss: 0.3777 - acc: 0.8969
Epoch 15/20
60000/60000 [==============================] - 2s 25us/step - loss: 0.3727 - acc: 0.8982
Epoch 16/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.3682 - acc: 0.8989
Epoch 17/20
60000/60000 [==============================] - 1s 25us/step - loss: 0.3641 - acc: 0.9001
Epoch 18/20
60000/60000 [==============================] - 1s 25us/step - loss: 0.3604 - acc: 0.9007
Epoch 19/20
60000/60000 [==============================] - 2s 25us/step - loss: 0.3570 - acc: 0.9016
Epoch 20/20
60000/60000 [==============================] - 1s 24us/step - loss: 0.3538 - acc: 0.9023
```

```
# model evaluation
score = model.evaluate(X_test, Y_test, verbose=VERBOSE)
print("Test score:", score[0])
print('Test accuracy:', score[1])
```

```
10000/10000 [==============================] - 0s 41us/step
Test score: 0.33423959468007086
Test accuracy: 0.9101
```

✓ **Multi Output Perceptron: ~91,01% (20 Epochs)**

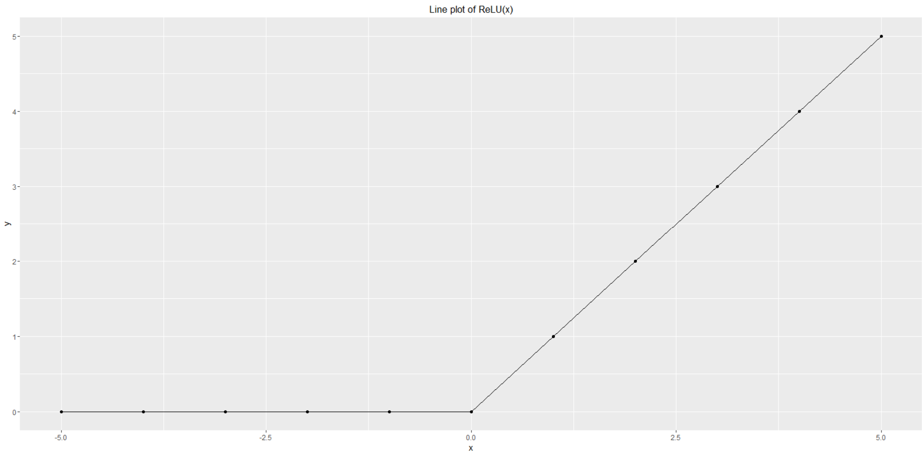
ANN – MNIST Dataset – Extend ANN Blueprint

- ✓ Data Preprocessing done (i.e. data normalization, reshape, etc.)
- ✓ Initial ANN topology existing
- ✓ Initial setup of model works (create, compile, fit)

- Extend the neural network topology
  - Which layers are required?
  - Think about input layer need to match the data – what data we had?
  - Maybe hidden layers?
  - How many hidden layers?
  - What activation function for which layer (e.g. maybe ReLU)?
  - Think Dense layer – Keras?
  - Think about final Activation as Softmay (cf. Day One) → output probability

Selected Activation Functions

- Rectified Linear Unit

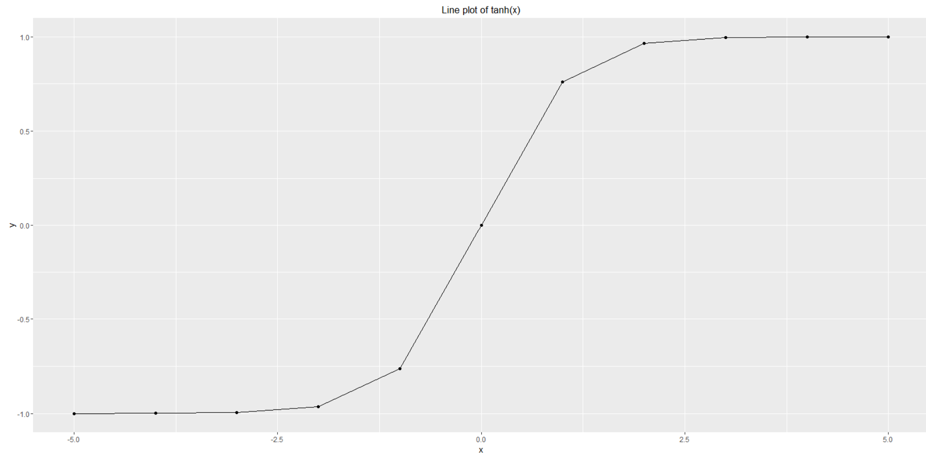


	x	y
1	-5	0
2	-4	0
3	-3	0
4	-2	0
5	-1	0
6	0	0
7	1	1
8	2	2
9	3	3
10	4	4
11	5	5

[5] big-data.tips, ‘Relu Neural Network’

```
model.add(Dense(N_HIDDEN))
model.add(Activation('relu'))
```

- Tanh



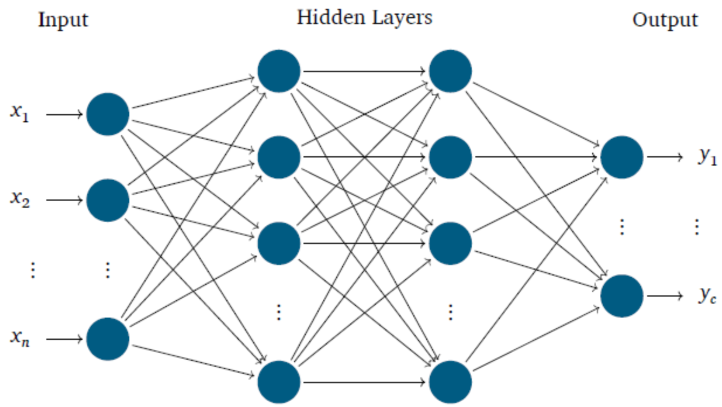
	x	y
1	-5	-0.9999092
2	-4	-0.9993293
3	-3	-0.9950548
4	-2	-0.9640276
5	-1	-0.7615942
6	0	0.0000000
7	1	0.7615942
8	2	0.9640276
9	3	0.9950548
10	4	0.9993293
11	5	0.9999092

[6] big-data.tips, ‘tanh’

```
model.add(Dense(N_HIDDEN))
model.add(Activation('tanh'))
```

Exercises – Add Two Hidden Layers

✓ **Multi Output Perceptron: ~91,01% (20 Epochs)**



ANN – MNIST Dataset – Add Two Hidden Layers

- All parameter value remain the same as before
- We add N_HIDDEN as parameter in order to set 128 neurons in one hidden layer – this number is a hyperparameter that is not directly defined and needs to be find with parameter search

```
# parameter setup
NB_EPOCH = 20
BATCH_SIZE = 128
NB_CLASSES = 10 # number of outputs = number of digits
OPTIMIZER = SGD() # optimization technique
VERBOSE = 1
N_HIDDEN = 128 # number of neurons in one hidden layer
```

```
# model Keras sequential
model = Sequential()
```

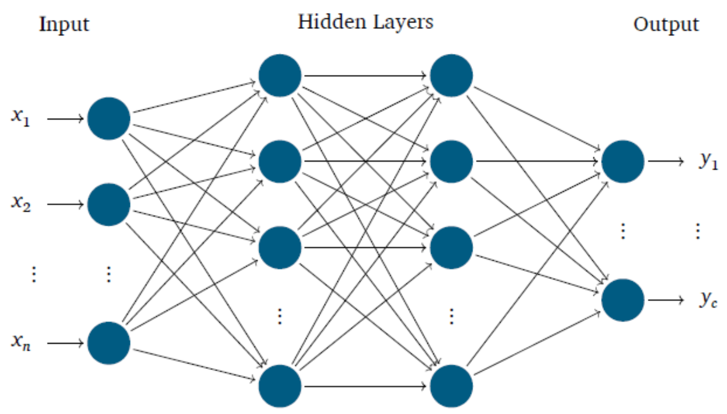
```
# modeling step
# 2 hidden layers each N_HIDDEN neurons
model.add(Dense(N_HIDDEN, input_shape=(RESHAPED,)))
model.add(Activation('relu'))
model.add(Dense(N_HIDDEN))
model.add(Activation('relu'))
model.add(Dense(NB_CLASSES))
```

```
# add activation function layer to get class probabilities
model.add(Activation('softmax'))
```

- The non-linear Activation function 'relu' represents a so-called Rectified Linear Unit (ReLU) that only recently became very popular because it generates good experimental results in ANNs and more recent deep learning models – it just returns 0 for negative values and grows linearly for only positive values
- A hidden layer in an ANN can be represented by a fully connected Dense layer in Keras by just specifying the number of hidden neurons in the hidden layer

MNIST Dataset & Model Summary & Parameters

- Added two Hidden Layers
  - Each hidden layers has 128 neurons



Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 128)	100480
activation_1 (Activation)	(None, 128)	0
dense_2 (Dense)	(None, 128)	16512
activation_2 (Activation)	(None, 128)	0
dense_3 (Dense)	(None, 10)	1290
activation_3 (Activation)	(None, 10)	0

Total params: 118,282
Trainable params: 118,282
Non-trainable params: 0

- **Relevant for validation: Choosing a model with different layers is a model selection that directly also influences the number of parameters (e.g. add Dense layer from Keras means new weights)**

```
# printout a summary of the model to understand model complexity
model.summary()
```

ANN 2 Hidden – MNIST Dataset – Output

```
Epoch 7/20
60000/60000 [==============================] - 1s 18us/step - loss: 0.2743 - acc: 0.9223
Epoch 8/20
60000/60000 [==============================] - 1s 18us/step - loss: 0.2601 - acc: 0.9266
Epoch 9/20
60000/60000 [==============================] - 1s 18us/step - loss: 0.2477 - acc: 0.9301
Epoch 10/20
60000/60000 [==============================] - 1s 18us/step - loss: 0.2365 - acc: 0.9329
Epoch 11/20
60000/60000 [==============================] - 1s 18us/step - loss: 0.2264 - acc: 0.9356
Epoch 12/20
60000/60000 [==============================] - 1s 18us/step - loss: 0.2175 - acc: 0.9386
Epoch 13/20
60000/60000 [==============================] - 1s 18us/step - loss: 0.2092 - acc: 0.9412
Epoch 14/20
60000/60000 [==============================] - 1s 18us/step - loss: 0.2013 - acc: 0.9432
Epoch 15/20
60000/60000 [==============================] - 1s 18us/step - loss: 0.1942 - acc: 0.9454
Epoch 16/20
60000/60000 [==============================] - 1s 18us/step - loss: 0.1876 - acc: 0.9472
Epoch 17/20
60000/60000 [==============================] - 1s 18us/step - loss: 0.1813 - acc: 0.9487
Epoch 18/20
60000/60000 [==============================] - 1s 18us/step - loss: 0.1754 - acc: 0.9502
Epoch 19/20
60000/60000 [==============================] - 1s 18us/step - loss: 0.1700 - acc: 0.9522
Epoch 20/20
60000/60000 [==============================] - 1s 18us/step - loss: 0.1647 - acc: 0.9536
```

```python
# model evaluation
score = model.evaluate(X_test, Y_test, verbose=VERBOSE)
print("Test score:", score[0])
print('Test accuracy:', score[1])
```

```
10000/10000 [==============================] - 0s 33us/step
Test score: 0.16286438911408185
Test accuracy: 0.9514
```

- ✓ **Multi Output Perceptron: ~91,01% (20 Epochs)**
- ✓ **ANN 2 Hidden Layers: ~95,14 % (20 Epochs)**

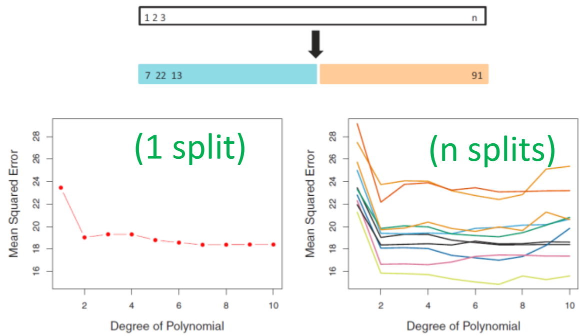
Validation & Model Selection – Terminology

- **The 'Validation technique' should be used in all machine learning or data mining approaches**
- **Model assessment is the process of evaluating a models performance**
- **Model selection is the process of selecting the proper level of flexibility for a model**

modified from [4] 'An Introduction to Statistical Learning'

- 'Training error'
  - Calculated when learning from data (i.e. dedicated training set)
- 'Test error'
  - Average error resulting from using the model with 'new/unseen data'
  - 'new/unseen data' was not used in training (i.e. dedicated test set)
  - In many practical situations, a dedicated test set is not really available
- 'Validation Set'
  - Split data into training & validation set
- 'Variance' & 'Variability'
  - Result in different random splits (right)

(split creates a two subsets of comparable size)



Validation Technique – Formalization & Goal

- **Validation is a very important technique to estimate the out-of-sample performance of a model**
- **Main utility of regularization & validation is to control or avoid overfitting via model selection**

- Regularization & Validation
  - Approach: introduce a 'overfit penalty' that relates to model complexity
  - Problem: Not accurate values: 'better smooth functions'

(regularization uses a term that captures the overfit penalty)

$$E_{out}(h) = E_{in}(h) + \textbf{overfit penalty}$$

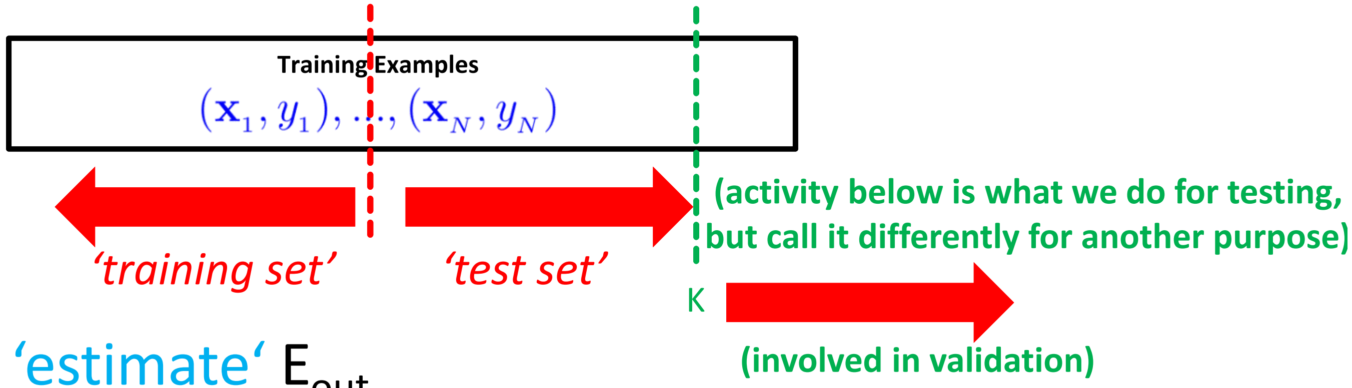
(minimize both to be better proxy for E_{out})

$E_{out}(h)$ ↑ **(validation estimates this quantity)**

overfit penalty ↑ **(regularization estimates this quantity)**

- Validation **(measuring E_{out} is not possible as this is an unknown quantity, another quantity is needed that is measurable that at least estimates it)**
  - Goal 'estimate the out-of-sample error' **(establish a quantity known as validation error)**
  - Distinct activity from training and testing **(testing also tries to estimate the E_{out})**

Validation Technique – Pick one point & Estimate E_{out}



- Understanding 'estimate' E_{out}
  - On one out-of-sample point $(\mathbf{x}, y)$ the error is $e(h(\mathbf{x}), y)$
  - E.g. use squared error: $e(h(\mathbf{x}), f(\mathbf{x})) = (h(\mathbf{x}) - f(\mathbf{x}))^2$
    $e(h(\mathbf{x}), y) = (h(\mathbf{x}) - y)^2$
  - Use this quantity as estimate for E_{out} **(poor estimate)**
  - Term 'expected value' to formalize (probability theory)

(Taking into account the theory of Lecture 1 with probability distribution on X etc.)

Probability Distribution
P on X

$\mathbf{x} = (x_1, \ldots, x_d)$

(aka 'random variable')

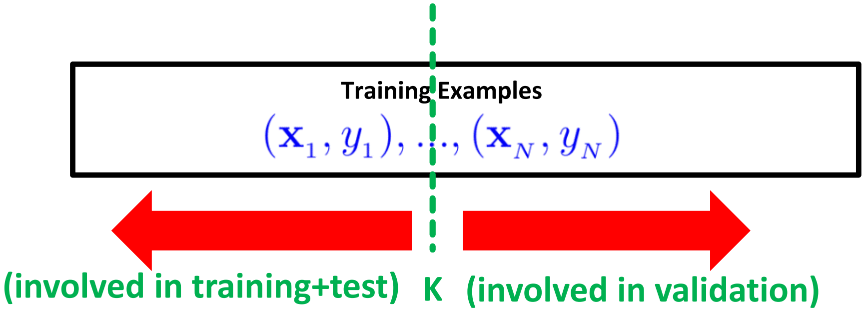
$\mathbb{E}[e(h(\mathbf{x}), y)] = E_{out}(h)$ **(aka the long-run average value of repetitions of the experiment)**

(one point as unbiased estimate of E_{out} that can have a high variance leads to bad generalization)

Validation Technique – Validation Set

- **Validation set consists of data that has been not used in training to estimate true out-of-sample**
- **Rule of thumb from practice is to take 20% (1/5) for validation of the learning model**

- Solution for high variance in expected values $\mathbb{E}[e(h(\mathbf{x}), y)] = E_{out}(h)$
  - Take a 'whole set' instead of just one point $(\mathbf{x}, y)$ for validation



(we need points not used in training to estimate the out-of-sample performance)

(we do the same approach with the testing set, but here different purpose)

  - Idea: K data points for validation

$(\mathbf{x}_1, y_1), ..., (\mathbf{x}_K, y_K)$ **(validation set)**

$$E_{val}(h) = \frac{1}{K}\sum_{k=1}^{K} e(h(\mathbf{x})_k, y_k)$$ **(validation error)**

  - Expected value to 'measure' the out-of-sample error

(expected values averaged over set)

$$\mathbb{E}[E_{val}(h)] = \frac{1}{K}\sum_{k=1}^{K} \mathbb{E}[e(h(\mathbf{x})_k, y_k)] = E_{out}$$

  - 'Reliable estimate' if K is large

(on rarely used validation set, otherwise data gets contaminated)

(this gives a much better (lower) variance than on a single point given K is large)

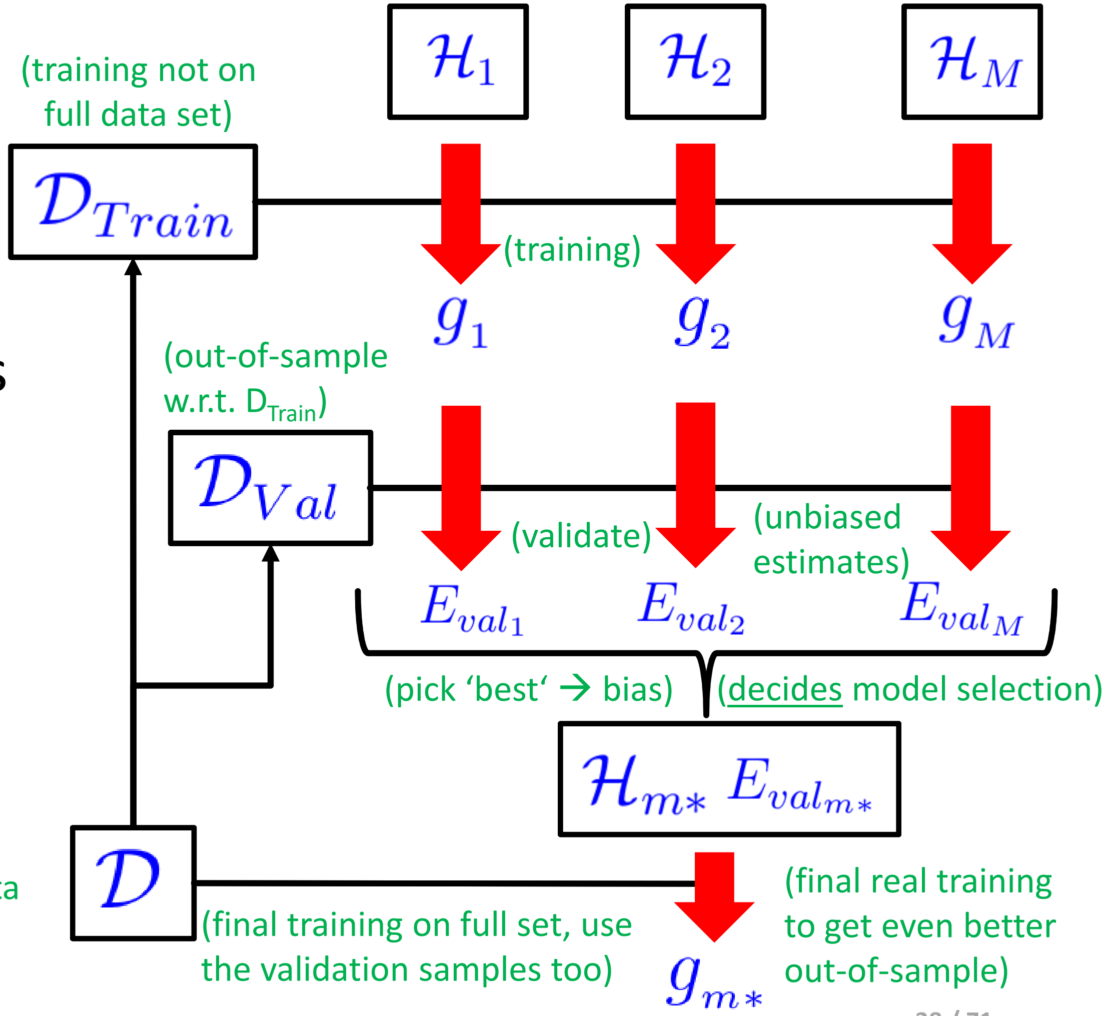
Validation Technique – Model Selection Process

- **Model selection is choosing (a) different types of models or (b) parameter values inside models**
- **Model selection takes advantage of the validation error in order to decide → 'pick the best'**

Hypothesis Set
$\mathcal{H} = \{h\}; \; g \in \mathcal{H}$

(set of candidate formulas across models)

- Many different models Use validation error to perform select decisions
- Careful consideration:
  - 'Picked means decided' hypothesis has already bias (→ contamination)
  - Using $\mathcal{D}_{Val}$ M times



Final Hypothesis
$g_{m*} \approx f$

(test this on unseen data good, but depends on availability in practice)

Exercises – Add Validation – Table & Groups

- ✓ **Multi Output Perceptron: ~91,01% (20 Epochs)**
- ✓ **ANN 2 Hidden Layers: ~95,14% (20 Epochs) – overfit?**

VAL_SPLIT	Accuracy Groups
0.0	97,79%
0.1	97,83%
0.2	97,64%
0.3	97,52 %
0.4	
0.5	97,13 %

ANN 2 Hidden 1/5 Validation – MNIST Dataset

- If there is enough data available one rule of thumb is to take 1/5 (0.2) 20% of the datasets for validation only
- Validation data is used to perform model selection (i.e. parameter / topology decisions)

```
# parameter setup
NB_EPOCH = 20
BATCH_SIZE = 128
NB_CLASSES = 10 # number of outputs = number of digits
OPTIMIZER = SGD() # optimization technique
VERBOSE = 1
N_HIDDEN = 128 # number of neurons in one hidden layer
VAL_SPLIT = 0.2 # 1/5 for validation rule of thumb
```

- The validation split parameter enables an easy validation approach during the model training (aka fit)
- Expectations should be a higher accuracy for unseen data since training data is less biased when using validation for model decisions (check statistical learning theory)
- VALIDATION_SPLIT: Float between 0 and 1
- Fraction of the training data to be used as validation data
- The model fit process will set apart this fraction of the training data and will not train on it
- Intead it will evaluate the loss and any model metrics on the validation data at the end of each epoch.

```
# model training
history = model.fit(X_train, Y_train, batch_size=BATCH_SIZE, epochs=NB_EPOCH, verbose=VERBOSE, validation_split = VAL_SPLIT)
```

```
Train on 48000 samples, validate on 12000 samples
```

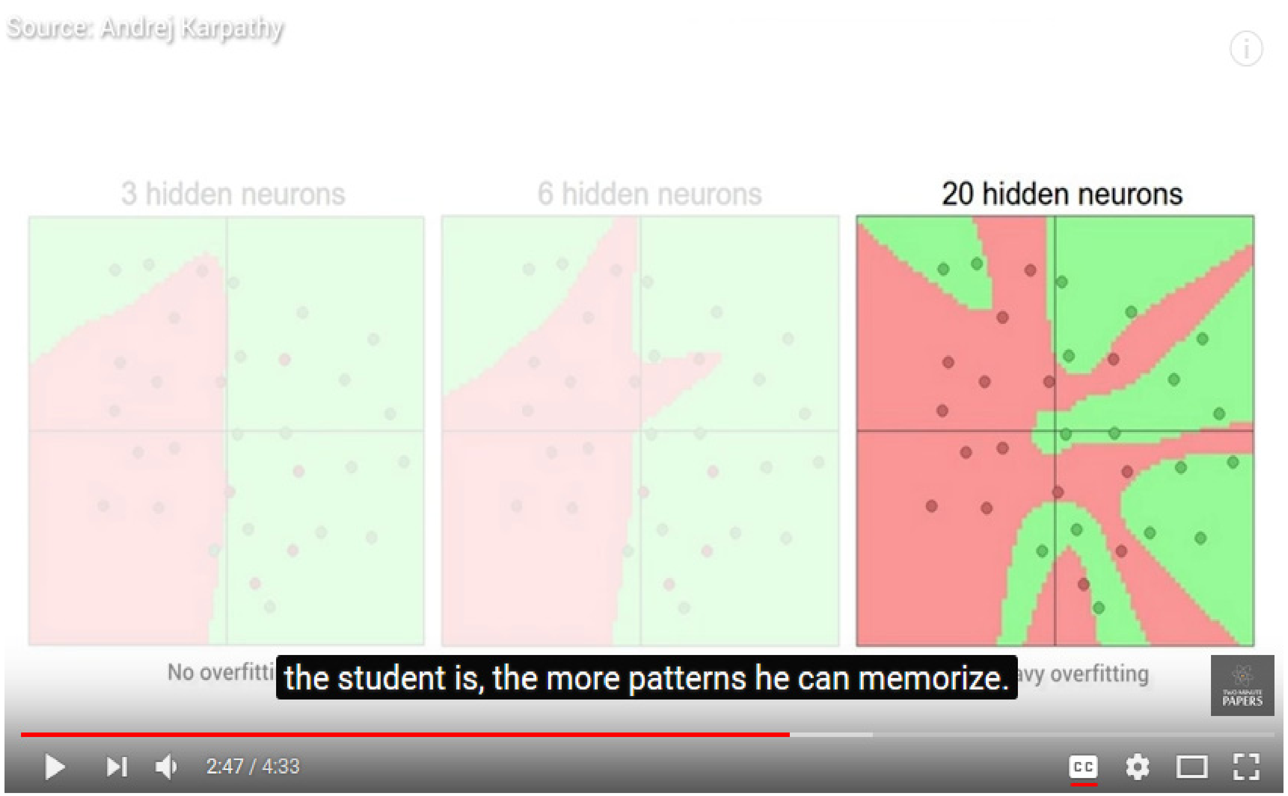
ANN 2 Hidden – 1/5 Validation – MNIST Dataset – Output

```
Epoch 7/20
48000/48000 [==============================] - 1s 18us/step - loss: 0.2967 - acc: 0.9148 - val_loss: 0.2759 - val_acc: 0.9212
Epoch 8/20
48000/48000 [==============================] - 1s 18us/step - loss: 0.2825 - acc: 0.9187 - val_loss: 0.2636 - val_acc: 0.9248
Epoch 9/20
48000/48000 [==============================] - 1s 18us/step - loss: 0.2702 - acc: 0.9222 - val_loss: 0.2550 - val_acc: 0.9272
Epoch 10/20
48000/48000 [==============================] - 1s 17us/step - loss: 0.2593 - acc: 0.9259 - val_loss: 0.2461 - val_acc: 0.9311
Epoch 11/20
48000/48000 [==============================] - 1s 18us/step - loss: 0.2494 - acc: 0.9283 - val_loss: 0.2367 - val_acc: 0.9338
Epoch 12/20
48000/48000 [==============================] - 1s 18us/step - loss: 0.2403 - acc: 0.9309 - val_loss: 0.2304 - val_acc: 0.9348
Epoch 13/20
48000/48000 [==============================] - 1s 18us/step - loss: 0.2319 - acc: 0.9334 - val_loss: 0.2228 - val_acc: 0.9392
Epoch 14/20
48000/48000 [==============================] - 1s 18us/step - loss: 0.2242 - acc: 0.9358 - val_loss: 0.2172 - val_acc: 0.9397
Epoch 15/20
48000/48000 [==============================] - 1s 17us/step - loss: 0.2172 - acc: 0.9381 - val_loss: 0.2105 - val_acc: 0.9418
Epoch 16/20
48000/48000 [==============================] - 1s 18us/step - loss: 0.2103 - acc: 0.9394 - val_loss: 0.2059 - val_acc: 0.9431
Epoch 17/20
48000/48000 [==============================] - 1s 18us/step - loss: 0.2040 - acc: 0.9417 - val_loss: 0.2007 - val_acc: 0.9447
Epoch 18/20
48000/48000 [==============================] - 1s 18us/step - loss: 0.1982 - acc: 0.9432 - val_loss: 0.1949 - val_acc: 0.9473
Epoch 19/20
48000/48000 [==============================] - 1s 18us/step - loss: 0.1926 - acc: 0.9447 - val_loss: 0.1920 - val_acc: 0.9472
Epoch 20/20
48000/48000 [==============================] - 1s 17us/step - loss: 0.1876 - acc: 0.9464 - val_loss: 0.1866 - val_acc: 0.9499
```

```python
# model evaluation
score = model.evaluate(X_test, Y_test, verbose=VERBOSE)
print("Test score:", score[0])
print('Test accuracy:', score[1])
```

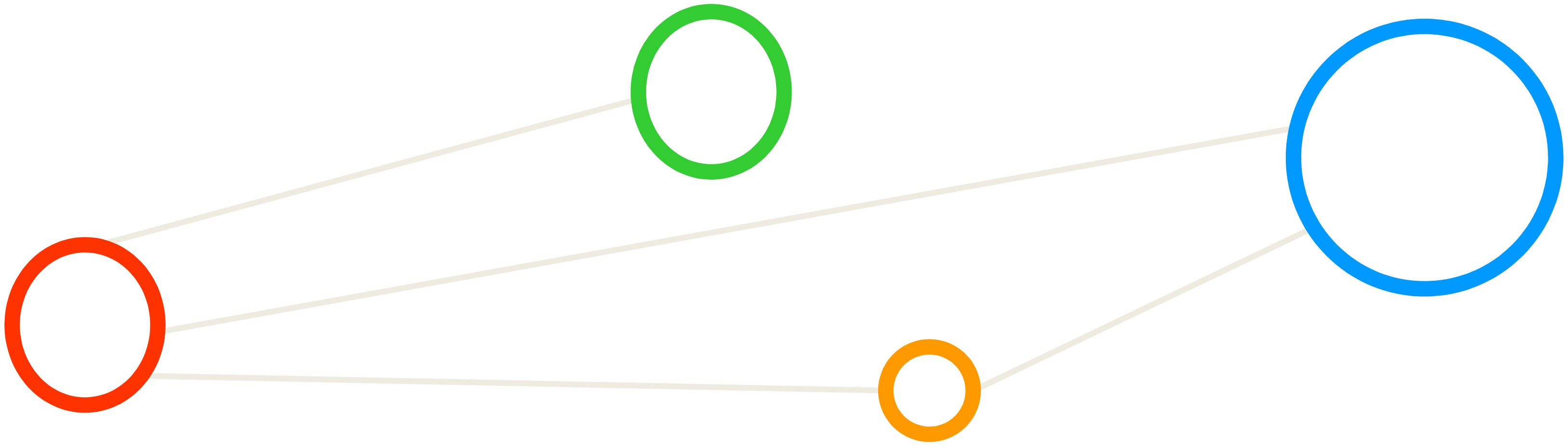
```
10000/10000 [==============================] - 0s 21us/step
Test score: 0.18584023508876563
Test accuracy: 0.9462
```

[Video] Overfitting in Deep Neural Networks



[4] **Overfitting and Regularization For Deep Learning,** ***YouTube***

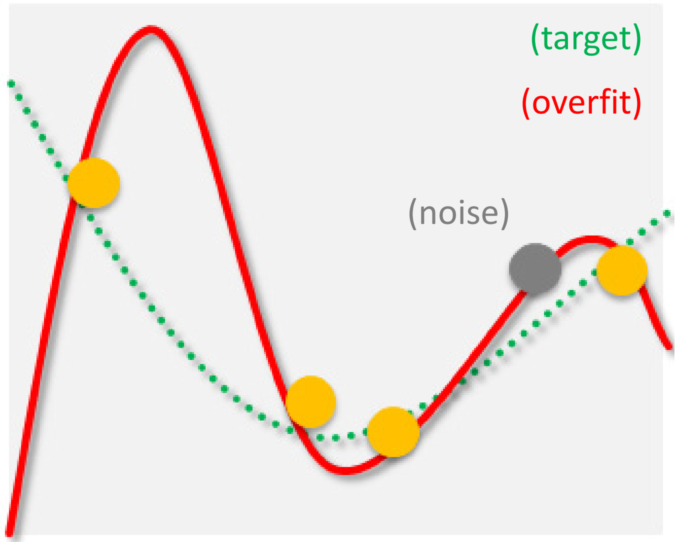
Regularization



Machine Learning Challenges – Problem of Overfitting

- **<u>Over</u>fitting refers to fit the data too well – more than is warranted – thus may misguide the learning**
- **Overfitting is not just 'bad generalization' - e.g. the VC dimension covers noiseless & noise targets**
- **Theory of Regularization are approaches against overfitting and prevent it using different methods**

- Key problem: noise in the target function leads to overfitting
  - Effect: 'noisy target function' and its noise misguides the fit in learning
  - There is always 'some noise' in the data
  - Consequence: poor target function ('distribution') approximation
- Example: Target functions is second order polynomial (i.e. parabola)
  - Using a higher-order polynomial fit
  - Perfect fit: low $E_{in}(g)$, but large $E_{out}(g)$



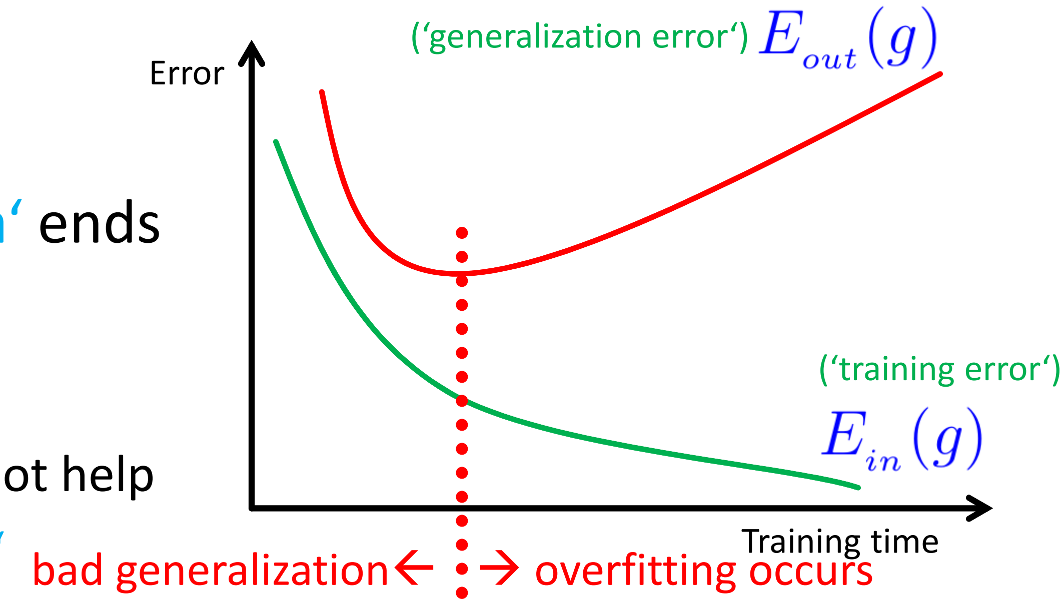
(but simple polynomial works good enough)
('over': here meant as 4th order, a 3rd order would be better, 2nd best)

Problem of Overfitting – Clarifying Terms

- A good model must have low training error (E_{in}) and low generalization error (E_{out})
- Model overfitting is if a model fits the data too well (E_{in}) with a poorer generalization error (E_{out}) than another model with a higher training error (E_{in})

[1] Introduction to Data Mining

- Overfitting & Errors
  - $E_{in}(g)$ goes down
  - $E_{out}(g)$ goes up
- 'Bad generalization area' ends
  - Good to reduce $E_{in}(g)$
- 'Overfitting area' starts
  - Reducing $E_{in}(g)$ does not help
  - Reason 'fitting the noise'

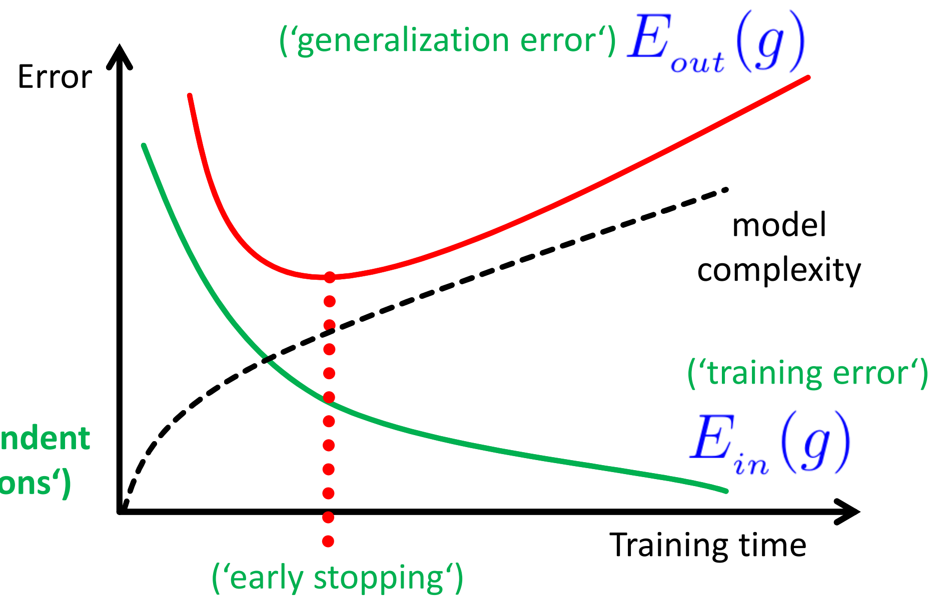


- The two general approaches to prevent overfitting are (1) regularization and (2) validation

Problem of Overfitting – Model Relationships

- Review 'overfitting situations'
  - When comparing 'various models' and related to 'model complexity'
  - Different models are used, e.g. 2nd and 4th order polynomial
  - Same model is used with e.g. two different instances (e.g. two neural networks but with different parameters)
- Intuitive solution
  - Detect when it happens
  - 'Early stopping regularization term' to stop the training
  - Early stopping method (later)

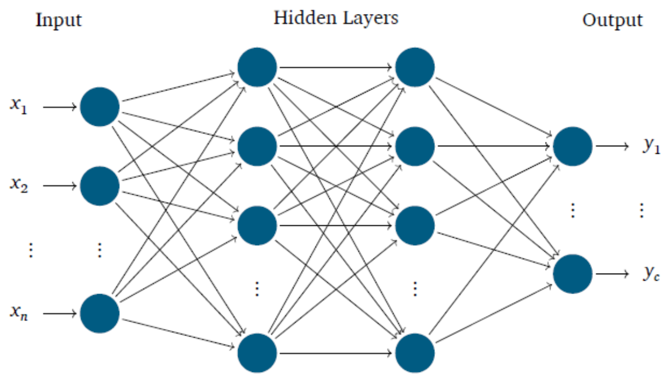
('model complexity measure: the VC analysis was independent of a specific target function – bound for all target functions')



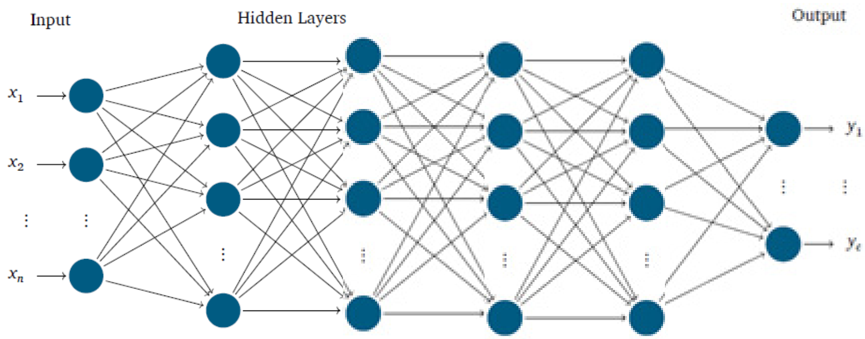
- **'Early stopping' approach is part of the theory of regularization, but based on validation methods**

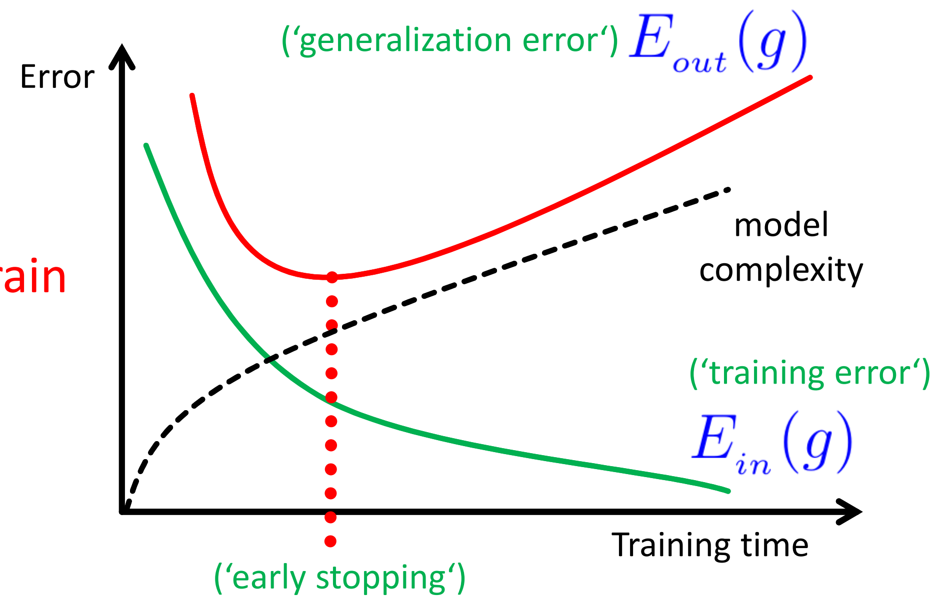
Problem of Overfitting – ANN Model Example

- Two Hidden Layers
  - Good accuracy and works well
  - Model complexity seem to match the application & data

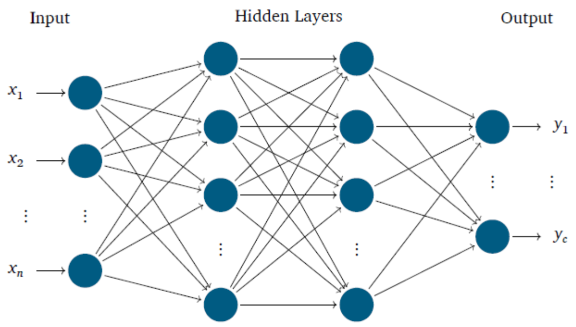


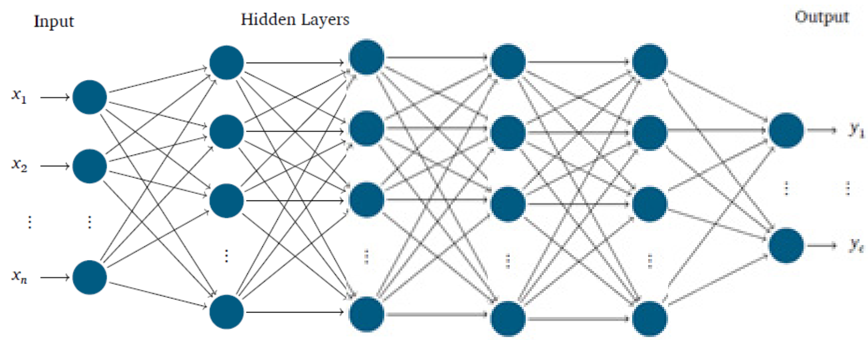
- Four Hidden Layers
  - Accuracy goes down
  - $E_{in}(g)$ goes down
  - $E_{out}(g)$ goes up
  - Significantly more weights to train
  - Higher model complexity





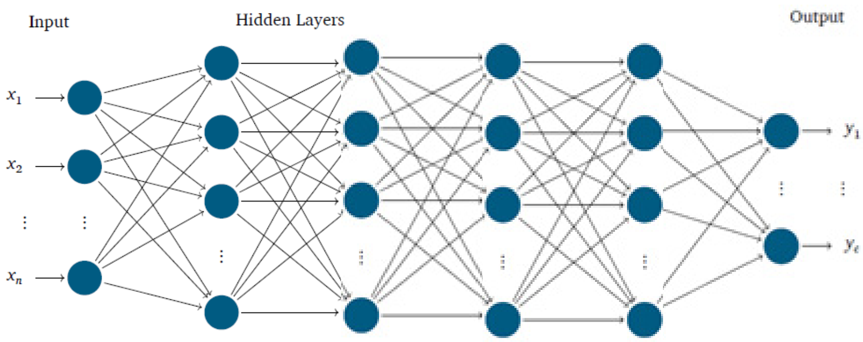
Exercises - Add more Hidden Layers – Accuracy?





MNIST Dataset & Model Summary & Parameters

- Four Hidden Layers
  - Each hidden layers has 128 neurons




```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 128)               100480
_________________________________________________________________
activation_1 (Activation)    (None, 128)               0
_________________________________________________________________
dense_2 (Dense)              (None, 128)               16512
_________________________________________________________________
activation_2 (Activation)    (None, 128)               0
_________________________________________________________________
dense_3 (Dense)              (None, 128)               16512
_________________________________________________________________
activation_3 (Activation)    (None, 128)               0
_________________________________________________________________
dense_4 (Dense)              (None, 128)               16512
_________________________________________________________________
activation_4 (Activation)    (None, 128)               0
_________________________________________________________________
dense_5 (Dense)              (None, 10)                1290
_________________________________________________________________
activation_5 (Activation)    (None, 10)                0
=================================================================
Total params: 151,306
Trainable params: 151,306
Non-trainable params: 0
_________________________________________________________________
```

```
# printout a summary of the model to understand model complexity
model.summary()
```

Exercises - Add more Hidden Layers – 4 Hidden Layers

```
Epoch 7/20
48000/48000 [==============================] - 1s 24us/step - loss: 0.2614 - acc: 0.9237 - val_loss: 0.2364 - val_acc: 0.9323
Epoch 8/20
48000/48000 [==============================] - 1s 24us/step - loss: 0.2431 - acc: 0.9290 - val_loss: 0.2243 - val_acc: 0.9347
Epoch 9/20
48000/48000 [==============================] - 1s 24us/step - loss: 0.2270 - acc: 0.9339 - val_loss: 0.2158 - val_acc: 0.9377
Epoch 10/20
48000/48000 [==============================] - 1s 24us/step - loss: 0.2130 - acc: 0.9385 - val_loss: 0.1995 - val_acc: 0.9427
Epoch 11/20
48000/48000 [==============================] - 1s 23us/step - loss: 0.2001 - acc: 0.9425 - val_loss: 0.1908 - val_acc: 0.9451
Epoch 12/20
48000/48000 [==============================] - 1s 24us/step - loss: 0.1888 - acc: 0.9445 - val_loss: 0.1866 - val_acc: 0.9464
Epoch 13/20
48000/48000 [==============================] - 1s 24us/step - loss: 0.1783 - acc: 0.9479 - val_loss: 0.1750 - val_acc: 0.9497
Epoch 14/20
48000/48000 [==============================] - 1s 24us/step - loss: 0.1701 - acc: 0.9507 - val_loss: 0.1675 - val_acc: 0.9529
Epoch 15/20
48000/48000 [==============================] - 1s 24us/step - loss: 0.1615 - acc: 0.9533 - val_loss: 0.1631 - val_acc: 0.9537
Epoch 16/20
48000/48000 [==============================] - 1s 24us/step - loss: 0.1539 - acc: 0.9555 - val_loss: 0.1553 - val_acc: 0.9555
Epoch 17/20
48000/48000 [==============================] - 1s 24us/step - loss: 0.1469 - acc: 0.9575 - val_loss: 0.1536 - val_acc: 0.9558
Epoch 18/20
48000/48000 [==============================] - 1s 24us/step - loss: 0.1405 - acc: 0.9590 - val_loss: 0.1505 - val_acc: 0.9560
Epoch 19/20
48000/48000 [==============================] - 1s 24us/step - loss: 0.1351 - acc: 0.9609 - val_loss: 0.1456 - val_acc: 0.9574
Epoch 20/20
48000/48000 [==============================] - 1s 24us/step - loss: 0.1295 - acc: 0.9625 - val_loss: 0.1398 - val_acc: 0.9600
```

```python
# model evaluation
score = model.evaluate(X_test, Y_test, verbose=VERBOSE)
print("Test score:", score[0])
print('Test accuracy:', score[1])
```

```
10000/10000 [==============================] - 0s 33us/step
Test score: 0.13893915132246912
Test accuracy: 0.9571
```

- **Training accuracy should still be above the test accuracy – otherwise overfitting starts!**

Exercises - Add more Hidden Layers – 6 Hidden Layers

```
Epoch 7/20
48000/48000 [==============================] - 1s 28us/step - loss: 0.2567 - acc: 0.9231 - val_loss: 0.2370 - val_acc: 0.9311
Epoch 8/20
48000/48000 [==============================] - 1s 28us/step - loss: 0.2333 - acc: 0.9312 - val_loss: 0.2229 - val_acc: 0.9342
Epoch 9/20
48000/48000 [==============================] - 1s 28us/step - loss: 0.2141 - acc: 0.9372 - val_loss: 0.1979 - val_acc: 0.9429
Epoch 10/20
48000/48000 [==============================] - 1s 28us/step - loss: 0.1963 - acc: 0.9415 - val_loss: 0.1860 - val_acc: 0.9461
Epoch 11/20
48000/48000 [==============================] - 1s 28us/step - loss: 0.1812 - acc: 0.9470 - val_loss: 0.1779 - val_acc: 0.9487
Epoch 12/20
48000/48000 [==============================] - 1s 28us/step - loss: 0.1693 - acc: 0.9496 - val_loss: 0.1717 - val_acc: 0.9504
Epoch 13/20
48000/48000 [==============================] - 1s 28us/step - loss: 0.1580 - acc: 0.9540 - val_loss: 0.1651 - val_acc: 0.9543
Epoch 14/20
48000/48000 [==============================] - 1s 28us/step - loss: 0.1477 - acc: 0.9573 - val_loss: 0.1535 - val_acc: 0.9552
Epoch 15/20
48000/48000 [==============================] - 1s 28us/step - loss: 0.1381 - acc: 0.9594 - val_loss: 0.1461 - val_acc: 0.9577
Epoch 16/20
48000/48000 [==============================] - 1s 28us/step - loss: 0.1309 - acc: 0.9616 - val_loss: 0.1427 - val_acc: 0.9582
Epoch 17/20
48000/48000 [==============================] - 1s 28us/step - loss: 0.1240 - acc: 0.9630 - val_loss: 0.1495 - val_acc: 0.9573
Epoch 18/20
48000/48000 [==============================] - 1s 27us/step - loss: 0.1170 - acc: 0.9663 - val_loss: 0.1447 - val_acc: 0.9563
Epoch 19/20
48000/48000 [==============================] - 1s 27us/step - loss: 0.1114 - acc: 0.9674 - val_loss: 0.1391 - val_acc: 0.9587
Epoch 20/20
48000/48000 [==============================] - 1s 27us/step - loss: 0.1053 - acc: 0.9696 - val_loss: 0.1355 - val_acc: 0.9601
```

```
# model evaluation
score = model.evaluate(X_test, Y_test, verbose=VERBOSE)
print("Test score:", score[0])
print('Test accuracy:', score[1])
```

```
10000/10000 [==============================] - 0s 34us/step
Test score: 0.13102742895036937
Test accuracy: 0.9614
```

- **Training accuracy should still be above the test accuracy – otherwise overfitting starts!**

Problem of Overfitting – Noise Term Revisited

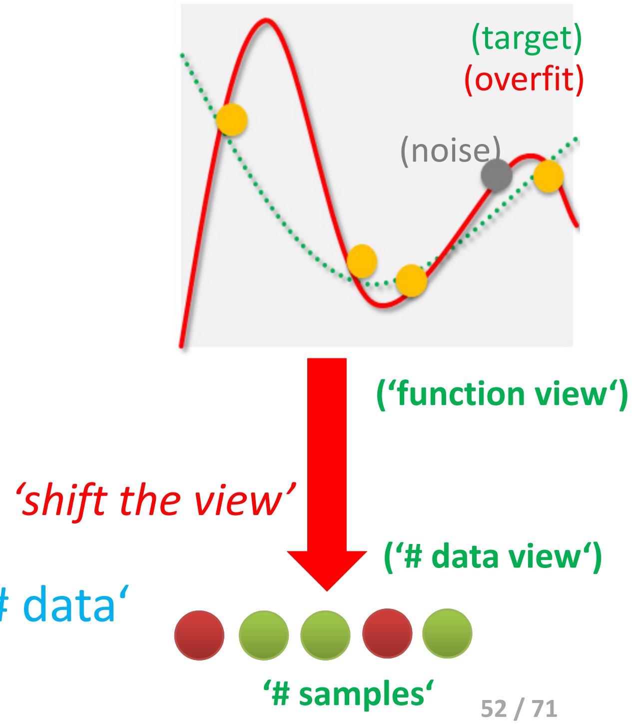
- ‘(Noisy) Target function‘ is not a (deterministic) function
  - Getting with ‘same x in‘ the ‘same y out‘ is not always given in practice
  - Idea: Use a ‘target distribution‘ instead of ‘target function‘

Unknown Target Distribution $P(y|\mathbf{x})$

target function $f : X \rightarrow Y$ plus noise

(ideal function)

> - **Fitting some noise in the data is the basic reason for overfitting and harms the learning process**
> - **Big datasets tend to have more noise in the data so the overfitting problem might occur even more intense**



- ‘Different types of some noise‘ in data
  - Key to understand overfitting & preventing it
  - ‘Shift of view‘: refinement of noise term
  - Learning from data: ‘matching properties of # data‘

Problem of Overfitting – Stochastic Noise

- Stoachastic noise is a part 'on top of' each learnable function
  - Noise in the data that can not be captured and thus not modelled by f
  - Random noise : aka 'non-deterministic noise'
  - Conventional understanding established early in this course
  - Finding a 'non-existing pattern in noise not feasible in learning'

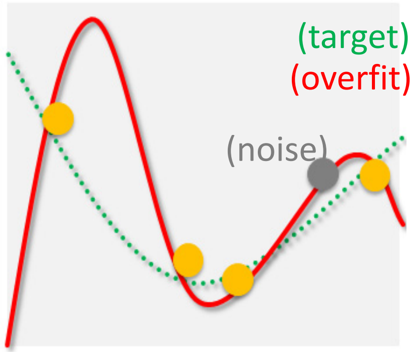
Unknown Target Distribution $P(y|\mathbf{x})$

target function $f: X \rightarrow Y$ plus noise

(ideal function)

- Practice Example
  - Random fluctuations and/or measurement errors in data
  - Fitting a pattern that not exists 'out-of-sample'
  - Puts learning progress 'off-track' and 'away from f'



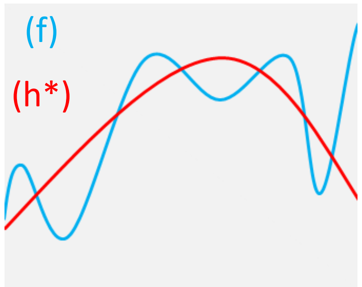


- **Stochastic noise here means noise that can't be captured, because it's just pure 'noise as is' (nothing to look for) – aka no pattern in the data to understand or to learn from**

Problem of Overfitting – Deterministic Noise

- Part of target function f that H can not capture: $f(\mathbf{x}) - h^*(\mathbf{x})$
  - Hypothesis set H is limited so best h* can not fully approximate f
  - h* approximates f, but fails to pick certain parts of the target f
  - 'Behaves like noise', existing even if data is 'stochastic noiseless'
- Different 'type of noise' than stochastic noise
  - Deterministic noise depends on $\mathcal{H}$ **(determines how much more can be captured by h*)**
  - E.g. same f, and more sophisticated $\mathcal{H}$: noise is smaller (stochastic noise remains the same, nothing can capture it)
  - Fixed for a given $\mathbf{x}$, clearly measurable (stochastic noise may vary for values of $\mathbf{x}$)

(learning deterministic noise is outside the ability to learn for a given h*)



- **Deterministic noise here means noise that can't be captured, because it is a limited model (out of the league of this particular model), e.g. 'learning with a toddler statistical learning theory'**

Problem of Overfitting – Impacts on Learning

- **The higher the degree of the polynomial (cf. model complexity), the more degrees of freedom are existing and thus the more capacity exists to overfit the training data**

- Understanding deterministic noise & target complexity
  - Increasing target complexity increases deterministic noise (at some level)
  - Increasing the number of data N decreases the deterministic noise
- Finite N case: $\mathcal{H}$ tries to fit the noise
  - Fitting the noise straightforward (e.g. Perceptron Learning Algorithm)
  - Stochastic (in data) and deterministic (simple model) noise will be part of it
- Two 'solution methods' for avoiding overfitting
  - Regularization: 'Putting the brakes in learning', e.g. early stopping (more theoretical, hence 'theory of regularization')
  - Validation: 'Checking the bottom line', e.g. other hints for out-of-sample (more practical, methods on data that provides 'hints')

High-level Tools – Keras – Regularization Techniques

- **Keras is a high-level deep learning library implemented in Python that works on top of existing other rather low-level deep learning frameworks like Tensorflow, CNTK, or Theano**
- **The key idea behind the Keras tool is to enable faster experimentation with deep networks**
- **Created deep learning models run seamlessly on CPU and GPU via low-level frameworks**

```
keras.layers.Dropout(rate,
                     noise_shape=None,
                     seed=None)
```

- **Dropout is randomly setting a fraction of input units to 0 at each update during training time, which helps prevent overfitting (using parameter rate)**

```
from keras import regularizers
model.add(Dense(64, input_dim=64,
kernel_regularizer=regularizers.l2(0.01),
activity_regularizer=regularizers.l1(0.01)))
```

- **L2 regularizers allow to apply penalties on layer parameter or layer activity during optimization itself – therefore the penalties are incorporated in the loss function during optimization**

Keras

[2] Keras Python Deep Learning Library

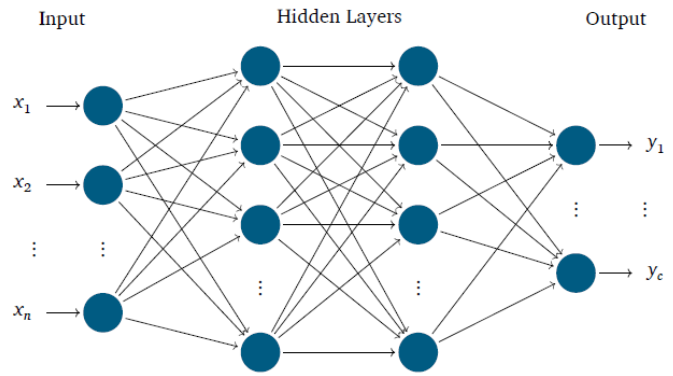
Exercises – Underfitting & Add Dropout Regularizer

- Run with 20 Epochs first (not trained enough); then 200 Epochs
  - Training accuracy should be above the test accuracy – otherwise ‘underfitting‘

VAL_SPLIT	Dropout	Accuracy Groups
0.0	0.10	98,0%
0.1	0.20	97,8%
0.2	0.25	97,78%
0.3	0.30	97,47 %
0.4	0.40	97,23%

ANN – MNIST Dataset – Add Weight Dropout Regularizer

```
# parameter setup
NB_EPOCH = 20
BATCH_SIZE = 128
NB_CLASSES = 10 # number of outputs = number of digits
OPTIMIZER = SGD() # optimization technique
VERBOSE = 1
N_HIDDEN = 128 # number of neurons in one hidden layer
VAL_SPLIT = 0.2 # 1/5 for validation rule of thumb
DROPOUT = 0.3 # regularization
```



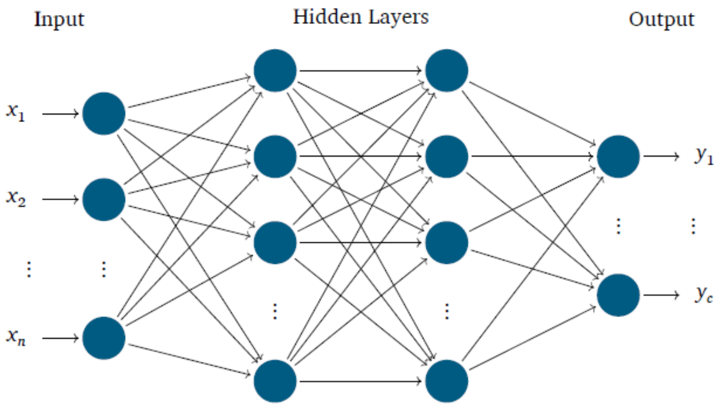

```
# modeling step
# 2 hidden layers each N_HIDDEN neurons
model.add(Dense(N_HIDDEN, input_shape=(RESHAPED,)))
model.add(Activation('relu'))
model.add(Dropout(DROPOUT))
model.add(Dense(N_HIDDEN))
model.add(Activation('relu'))
model.add(Dropout(DROPOUT))
model.add(Dense(NB_CLASSES))
```

- **A Dropout() regularizer randomly drops with ist dropout probability some of the values propagated inside the Dense network hidden layers improving accuracy again**
- **Our standard model is already modified in the python script but needs to set the DROPOUT rate**
- **A Dropout() regularizer randomly drops with ist dropout probability some of the values propagated inside the Dense network hidden layers improving accuracy again**

```
model.add(Activation('relu'))
model.add(Dropout(DROPOUT))
```

MNIST Dataset & Model Summary & Parameters

- Only two Hidden Layers but with Dropout
  - Each hidden layers has 128 neurons




```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 128)               100480
_________________________________________________________________
activation_1 (Activation)    (None, 128)               0
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_2 (Dense)              (None, 128)               16512
_________________________________________________________________
activation_2 (Activation)    (None, 128)               0
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_3 (Dense)              (None, 10)                1290
_________________________________________________________________
activation_3 (Activation)    (None, 10)                0
=================================================================
Total params: 118,282
Trainable params: 118,282
Non-trainable params: 0
```

```
# printout a summary of the model to understand model complexity
model.summary()
```

ANN – MNIST – DROPOUT (20 Epochs)

```
Epoch 7/20
48000/48000 [==============================] - 1s 22us/step - loss: 0.4616 - acc: 0.8628 - val_loss: 0.3048 - val_acc: 0.9127
Epoch 8/20
48000/48000 [==============================] - 1s 22us/step - loss: 0.4386 - acc: 0.8688 - val_loss: 0.2896 - val_acc: 0.9172
Epoch 9/20
48000/48000 [==============================] - 1s 22us/step - loss: 0.4181 - acc: 0.8762 - val_loss: 0.2776 - val_acc: 0.9198
Epoch 10/20
48000/48000 [==============================] - 1s 22us/step - loss: 0.3990 - acc: 0.8838 - val_loss: 0.2657 - val_acc: 0.9234
Epoch 11/20
48000/48000 [==============================] - 1s 22us/step - loss: 0.3819 - acc: 0.8876 - val_loss: 0.2551 - val_acc: 0.9258
Epoch 12/20
48000/48000 [==============================] - 1s 22us/step - loss: 0.3688 - acc: 0.8920 - val_loss: 0.2465 - val_acc: 0.9283
Epoch 13/20
48000/48000 [==============================] - 1s 22us/step - loss: 0.3571 - acc: 0.8943 - val_loss: 0.2388 - val_acc: 0.9299
Epoch 14/20
48000/48000 [==============================] - 1s 22us/step - loss: 0.3466 - acc: 0.8991 - val_loss: 0.2319 - val_acc: 0.9323
Epoch 15/20
48000/48000 [==============================] - 1s 22us/step - loss: 0.3359 - acc: 0.9015 - val_loss: 0.2261 - val_acc: 0.9339
Epoch 16/20
48000/48000 [==============================] - 1s 22us/step - loss: 0.3244 - acc: 0.9055 - val_loss: 0.2180 - val_acc: 0.9352
Epoch 17/20
48000/48000 [==============================] - 1s 22us/step - loss: 0.3142 - acc: 0.9085 - val_loss: 0.2122 - val_acc: 0.9375
Epoch 18/20
48000/48000 [==============================] - 1s 21us/step - loss: 0.3103 - acc: 0.9095 - val_loss: 0.2076 - val_acc: 0.9390
Epoch 19/20
48000/48000 [==============================] - 1s 21us/step - loss: 0.3019 - acc: 0.9118 - val_loss: 0.2018 - val_acc: 0.9409
Epoch 20/20
48000/48000 [==============================] - 1s 21us/step - loss: 0.2931 - acc: 0.9132 - val_loss: 0.1974 - val_acc: 0.9419
```

```python
# model evaluation
score = model.evaluate(X_test, Y_test, verbose=VERBOSE)
print("Test score:", score[0])
print('Test accuracy:', score[1])
```

```
10000/10000 [==============================] - 0s 29us/step
Test score: 0.19944561417847873
Test accuracy: 0.9404
```

- **Regularization effect not yet because too little training time (i.e. other regularlization ‚early stopping' here)**

ANN – MNIST – DROPOUT (200 Epochs)

```
Epoch 187/200
48000/48000 [==============================] - 1s 21us/step - loss: 0.0780 - acc: 0.9755 - val_loss: 0.0810 - val_acc: 0.9764
Epoch 188/200
48000/48000 [==============================] - 1s 21us/step - loss: 0.0795 - acc: 0.9753 - val_loss: 0.0799 - val_acc: 0.9765
Epoch 189/200
48000/48000 [==============================] - 1s 21us/step - loss: 0.0774 - acc: 0.9763 - val_loss: 0.0802 - val_acc: 0.9763
Epoch 190/200
48000/48000 [==============================] - 1s 21us/step - loss: 0.0773 - acc: 0.9770 - val_loss: 0.0799 - val_acc: 0.9758
Epoch 191/200
48000/48000 [==============================] - 1s 21us/step - loss: 0.0746 - acc: 0.9771 - val_loss: 0.0804 - val_acc: 0.9762
Epoch 192/200
48000/48000 [==============================] - 1s 21us/step - loss: 0.0761 - acc: 0.9771 - val_loss: 0.0805 - val_acc: 0.9762
Epoch 193/200
48000/48000 [==============================] - 1s 21us/step - loss: 0.0750 - acc: 0.9772 - val_loss: 0.0800 - val_acc: 0.9763
Epoch 194/200
48000/48000 [==============================] - 1s 21us/step - loss: 0.0753 - acc: 0.9766 - val_loss: 0.0804 - val_acc: 0.9767
Epoch 195/200
48000/48000 [==============================] - 1s 21us/step - loss: 0.0748 - acc: 0.9768 - val_loss: 0.0799 - val_acc: 0.9767
Epoch 196/200
48000/48000 [==============================] - 1s 21us/step - loss: 0.0755 - acc: 0.9767 - val_loss: 0.0795 - val_acc: 0.9765
Epoch 197/200
48000/48000 [==============================] - 1s 21us/step - loss: 0.0740 - acc: 0.9771 - val_loss: 0.0799 - val_acc: 0.9767
Epoch 198/200
48000/48000 [==============================] - 1s 21us/step - loss: 0.0744 - acc: 0.9769 - val_loss: 0.0792 - val_acc: 0.9772
Epoch 199/200
48000/48000 [==============================] - 1s 21us/step - loss: 0.0759 - acc: 0.9769 - val_loss: 0.0794 - val_acc: 0.9767
Epoch 200/200
48000/48000 [==============================] - 1s 21us/step - loss: 0.0730 - acc: 0.9778 - val_loss: 0.0794 - val_acc: 0.9771
```

```python
# model evaluation
score = model.evaluate(X_test, Y_test, verbose=VERBOSE)
print("Test score:", score[0])
print('Test accuracy:', score[1])
```

```
10000/10000 [==============================] - 0s 27us/step
Test score: 0.07506137332450598
Test accuracy: 0.9775
```

- **Regularization effect visible by long training time using dropouts and achieving highest accuracy**
- **Note: Convolutional Neural Networks: 99,1 %**

ANN – MNIST – w/o DROPOUT (200 Epochs)

```
Epoch 187/200
48000/48000 [==============================] - 1s 20us/step - loss: 0.0173 - acc: 0.9973 - val_loss: 0.0888 - val_acc: 0.9753
Epoch 188/200
48000/48000 [==============================] - 1s 20us/step - loss: 0.0170 - acc: 0.9975 - val_loss: 0.0896 - val_acc: 0.9742
Epoch 189/200
48000/48000 [==============================] - 1s 20us/step - loss: 0.0169 - acc: 0.9975 - val_loss: 0.0888 - val_acc: 0.9750
Epoch 190/200
48000/48000 [==============================] - 1s 19us/step - loss: 0.0168 - acc: 0.9973 - val_loss: 0.0880 - val_acc: 0.9752
Epoch 191/200
48000/48000 [==============================] - 1s 20us/step - loss: 0.0165 - acc: 0.9977 - val_loss: 0.0884 - val_acc: 0.9747
Epoch 192/200
48000/48000 [==============================] - 1s 20us/step - loss: 0.0164 - acc: 0.9976 - val_loss: 0.0887 - val_acc: 0.9751
Epoch 193/200
48000/48000 [==============================] - 1s 20us/step - loss: 0.0162 - acc: 0.9976 - val_loss: 0.0888 - val_acc: 0.9747
Epoch 194/200
48000/48000 [==============================] - 1s 20us/step - loss: 0.0160 - acc: 0.9977 - val_loss: 0.0891 - val_acc: 0.9752
Epoch 195/200
48000/48000 [==============================] - 1s 20us/step - loss: 0.0159 - acc: 0.9977 - val_loss: 0.0889 - val_acc: 0.9752
Epoch 196/200
48000/48000 [==============================] - 1s 20us/step - loss: 0.0157 - acc: 0.9979 - val_loss: 0.0886 - val_acc: 0.9752
Epoch 197/200
48000/48000 [==============================] - 1s 20us/step - loss: 0.0155 - acc: 0.9980 - val_loss: 0.0890 - val_acc: 0.9748
Epoch 198/200
48000/48000 [==============================] - 1s 19us/step - loss: 0.0153 - acc: 0.9980 - val_loss: 0.0893 - val_acc: 0.9747
Epoch 199/200
48000/48000 [==============================] - 1s 19us/step - loss: 0.0152 - acc: 0.9980 - val_loss: 0.0892 - val_acc: 0.9746
Epoch 200/200
48000/48000 [==============================] - 1s 20us/step - loss: 0.0151 - acc: 0.9980 - val_loss: 0.0894 - val_acc: 0.9749
```

```python
# model evaluation
score = model.evaluate(X_test, Y_test, verbose=VERBOSE)
print("Test score:", score[0])
print('Test accuracy:', score[1])
```

```
10000/10000 [==============================] - 0s 27us/step
Test score: 0.07599342362476745
Test accuracy: 0.9764
```

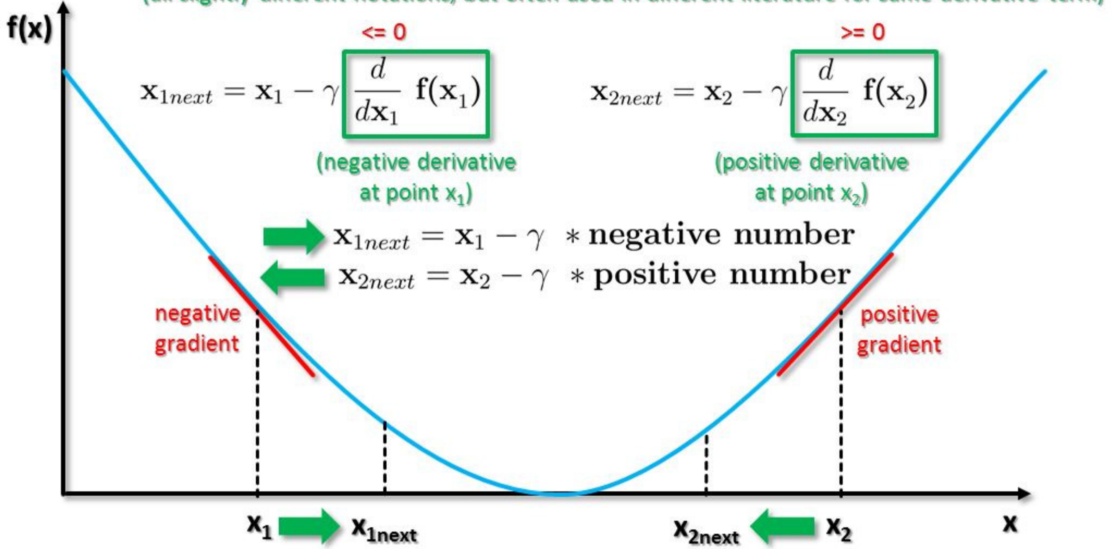
- **No regularization method by long training time for comparison – slight drop in accuracy since simple dataset**

MNIST Dataset & SGD Method – Revisited

- **Gradient Descent (GD) uses all the training samples available for a step within a iteration**
- **Stochastic Gradient Descent (SGD) converges faster: only one training samples used per iteration**

$$\mathrm{b} = \mathrm{a} - \gamma\ \nabla\ \mathrm{f(a)} \qquad \mathrm{b} = \mathrm{a} - \gamma\ \frac{\partial}{\partial \mathrm{a}}\ \mathrm{f(a)} \qquad \mathrm{b} = \mathrm{a} - \gamma\ \frac{d}{d\mathrm{a}}\ \mathrm{f(a)}$$

(all slightly different notations, but often used in different literature for same derivative term)




```python
from keras.optimizers import SGD
```

```python
OPTIMIZER = SGD() # optimization technique
```

[4] Big Data Tips, Gradient Descent

MNIST Dataset & RMSprop & Adam Optimization Methods

- **RMSProp is an advanced optimization technique that in many cases enable earlier convergence**
- **Adam includes a concept of momentum (i.e. veloctity) in addition to the acceleration of SGD**

```
Epoch 7/20
48000/48000 [==============================] - 1s 25us/step - loss: 0.1127 - acc: 0.9668 - val_loss: 0.1014 - val_acc: 0.9723
Epoch 8/20
48000/48000 [==============================] - 1s 25us/step - loss: 0.1051 - acc: 0.9690 - val_loss: 0.0984 - val_acc: 0.9735
Epoch 9/20
48000/48000 [==============================] - 1s 25us/step - loss: 0.0970 - acc: 0.9706 - val_loss: 0.0996 - val_acc: 0.9747
Epoch 10/20
48000/48000 [==============================] - 1s 25us/step - loss: 0.0949 - acc: 0.9716 - val_loss: 0.0958 - val_acc: 0.9754
Epoch 11/20
48000/48000 [==============================] - 1s 25us/step - loss: 0.0880 - acc: 0.9734 - val_loss: 0.0945 - val_acc: 0.9763
Epoch 12/20
48000/48000 [==============================] - 1s 25us/step - loss: 0.0873 - acc: 0.9745 - val_loss: 0.0957 - val_acc: 0.9761
Epoch 13/20
48000/48000 [==============================] - 1s 25us/step - loss: 0.0842 - acc: 0.9745 - val_loss: 0.0952 - val_acc: 0.9757
Epoch 14/20
48000/48000 [==============================] - 1s 25us/step - loss: 0.0804 - acc: 0.9763 - val_loss: 0.1002 - val_acc: 0.9767
Epoch 15/20
48000/48000 [==============================] - 1s 25us/step - loss: 0.0788 - acc: 0.9771 - val_loss: 0.0991 - val_acc: 0.9772
Epoch 16/20
48000/48000 [==============================] - 1s 25us/step - loss: 0.0756 - acc: 0.9772 - val_loss: 0.0988 - val_acc: 0.9761
Epoch 17/20
48000/48000 [==============================] - 1s 25us/step - loss: 0.0758 - acc: 0.9776 - val_loss: 0.1033 - val_acc: 0.9753
Epoch 18/20
48000/48000 [==============================] - 1s 26us/step - loss: 0.0755 - acc: 0.9781 - val_loss: 0.0996 - val_acc: 0.9773
Epoch 19/20
48000/48000 [==============================] - 1s 26us/step - loss: 0.0725 - acc: 0.9784 - val_loss: 0.1055 - val_acc: 0.9764
Epoch 20/20
48000/48000 [==============================] - 1s 26us/step - loss: 0.0712 - acc: 0.9791 - val_loss: 0.1014 - val_acc: 0.9778
```

```python
# model evaluation
score = model.evaluate(X_test, Y_test, verbose=VERBOSE)
print("Test score:", score[0])
print('Test accuracy:', score[1])
```

```
10000/10000 [==============================] - 0s 33us/step
Test score: 0.09596708530617616
Test accuracy: 0.9779
```

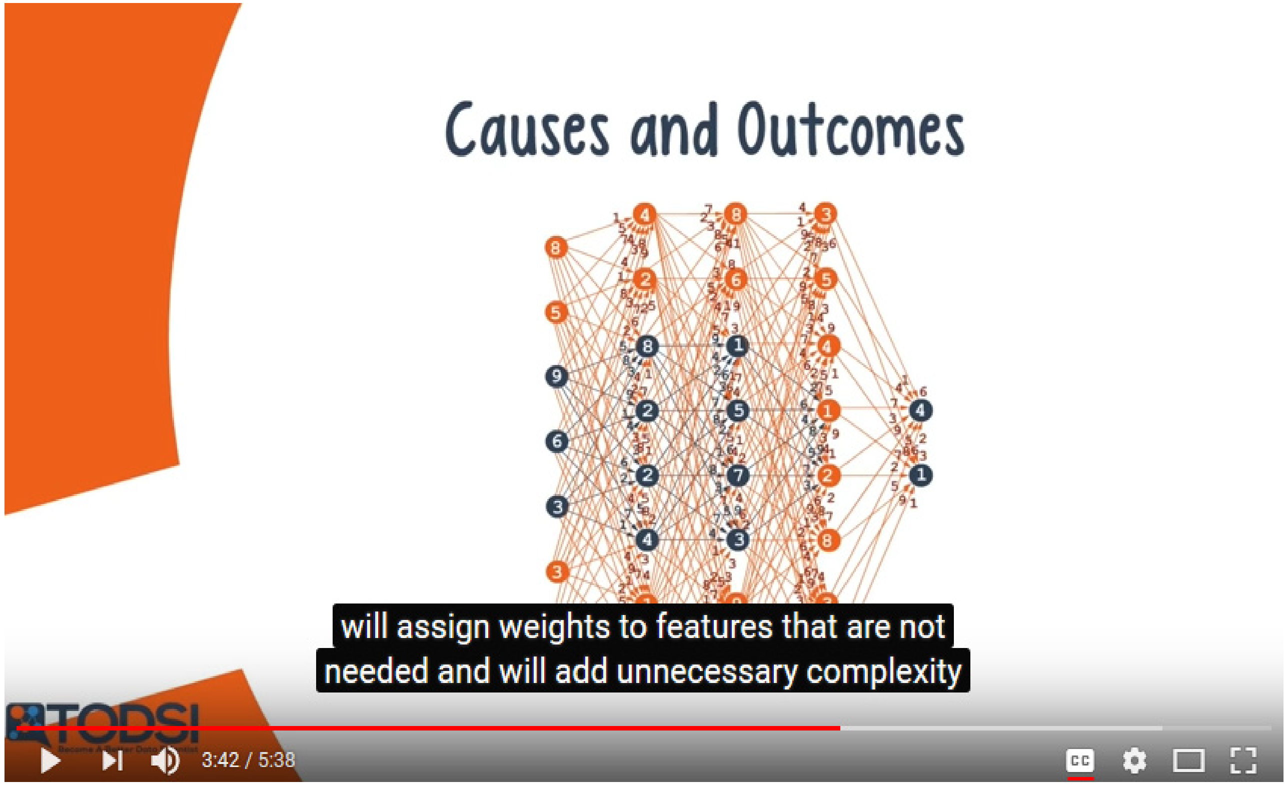
```python
from keras.optimizers import RMSprop
```

```python
OPTIMIZER = RMSprop() # optimization technique
```

Exercises – Underfitting & Change to Adam

- Run with 20 Epochs With Adam Optimizer

[Video] Overfitting in Deep Neural Networks



[3] How good is your fit?, YouTube

Appendix A – SSH Commands JURECA

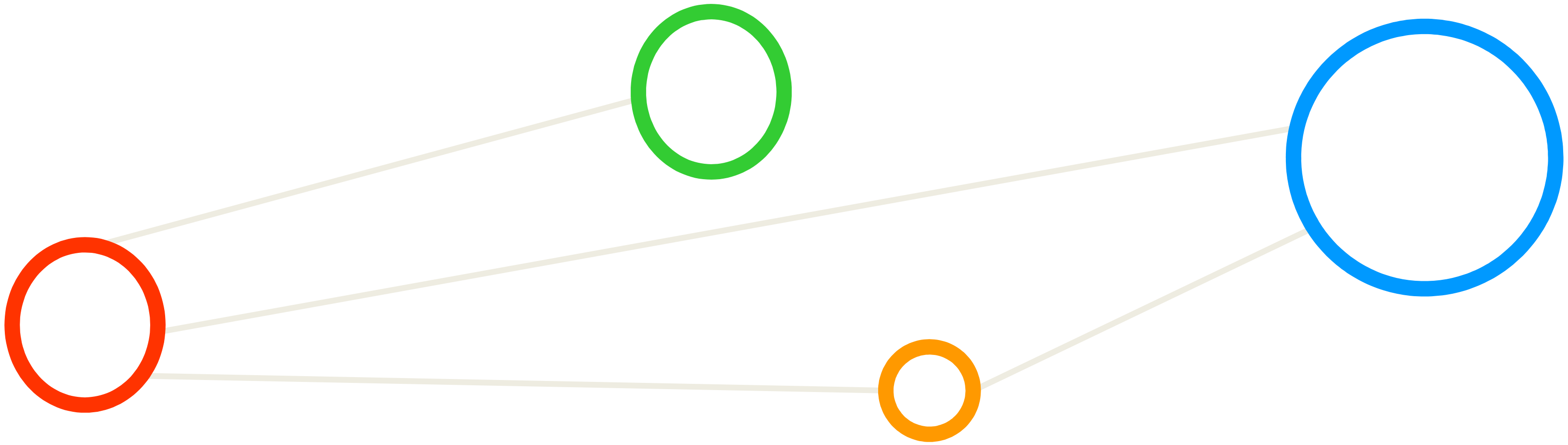
Appendix A – SSH Commands JURECA

- salloc --gres=gpu:4 --partition=gpus --nodes=1 --account=training1904 --time=00:30:00 --reservation=prace_ml_gpus_tue

- module --force purge;
  module use /usr/local/software/jureca/OtherStages
  module load Stages/Devel-2018b GCCcore/.7.3.0
  module load TensorFlow/1.12.0-GPU-Python-3.6.6
  module load Keras/2.2.4-GPU-Python-3.6.6

- srun python PYTHONSCRIPTNAME

Lecture Bibliography



Lecture Bibliography

- [1] An Introduction to Statistical Learning with Applications in R,
  Online: http://www-bcf.usc.edu/~gareth/ISL/index.html
- [2] Keras Python Deep Learning Library,
  Online: https://keras.io/
- [3] YouTube Video, 'How good is your fit? - Ep. 21 (Deep Learning SIMPLIFIED)',
  Online: https://www.youtube.com/watch?v=cJA5IHIIL30
- [4] YouTube Video, 'Overfitting and Regularization For Deep Learning | Two Minute Papers #56',
  Online: https://www.youtube.com/watch?v=6aF9sJrzxaM
- [5] www.big-data.tips, 'Relu Neural Network'
  Online: http://www.big-data.tips/relu-neural-network
- [6] www.big-data.tips, 'tanh',
  Online: http://www.big-data.tips/tanh

Slides Available at http://www.morrisriedel.de/teaching