Statistisches Data Mining (StDM)
Woche 13

Zurich University of Applied Sciences

Oliver Dürr

Institut für Datenanalyse und Prozessdesign

Zürcher Hochschule für Angewandte Wissenschaften

oliver.duerr@zhaw.ch

Winterthur, 13 Dezember 2016

No laptops, no phones, no problems

Multitasking senkt Lerneffizienz:

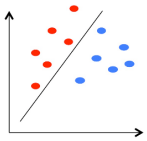
- **Keine Laptops im Theorie-Unterricht Deckel zu oder fast zu (Sleep modus)**

Praktikumsaufgabe

- Bitte bis heute Abend Teams bekannt geben.

Overview of classification (until the end to the semester)

Classifiers

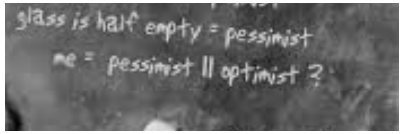


K-Nearest-Neighbors (KNN)
Logistic Regression
Linear discriminant analysis
Support Vector Machine (SVM)
Classification Trees
Neural networks NN
Deep Neural Networks (e.g. CNN, RNN)
...

Evaluation

Cross validation
Performance measures
ROC Analysis / Lift Charts

Theoretical Guidance / General Ideas

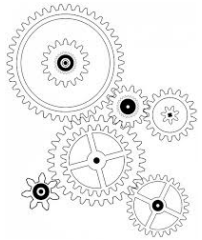


Bayes Classifier
Bias Variance Trade off (Overfitting)

Combining classifiers

Bagging
Random Forest
Boosting

Feature Engineering



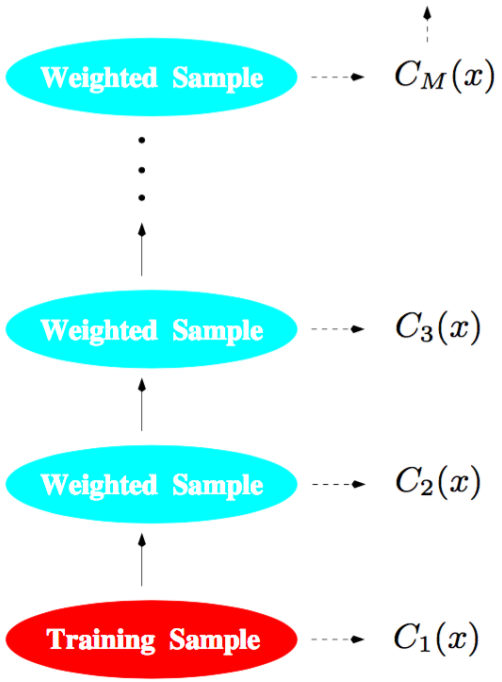
Feature Extraction
Feature Selection

Overview of Ensemble Methods

- Many instances of the same classifier
  - Bagging (bootstrapping & aggregating)
    - Create “new” data using bootstrap
    - Train classifiers on new data
    - Average over all predictions (e.g. take majority vote)
  - Bagged Trees
    - Use bagging with decision trees
  - Random Forest
    - Bagged trees with special trick
  - Boosting
    - An iterative procedure to adaptively change distribution of training data by focusing more on previously misclassified records.
- Combining classifiers
  - Weighted averaging over predictions
  - Stacking classifiers
    - Use output of classifiers as input for a new classifier

Boosting

- Consider binary problem with classes coded as +1,-1
- A sequence of classifiers C_m is learnt
- The training data is reweighted (depending on misclassification of example)



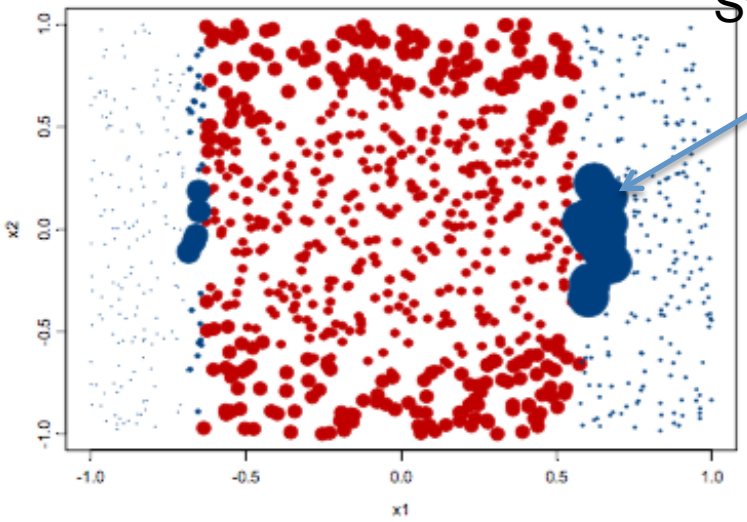
- Average many trees, each grown to re-weighted versions of the training data.
- Final Classifier is weighted average of classifiers:

$$C(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m C_m(x)\right]$$

Slides: Trevor Hastie

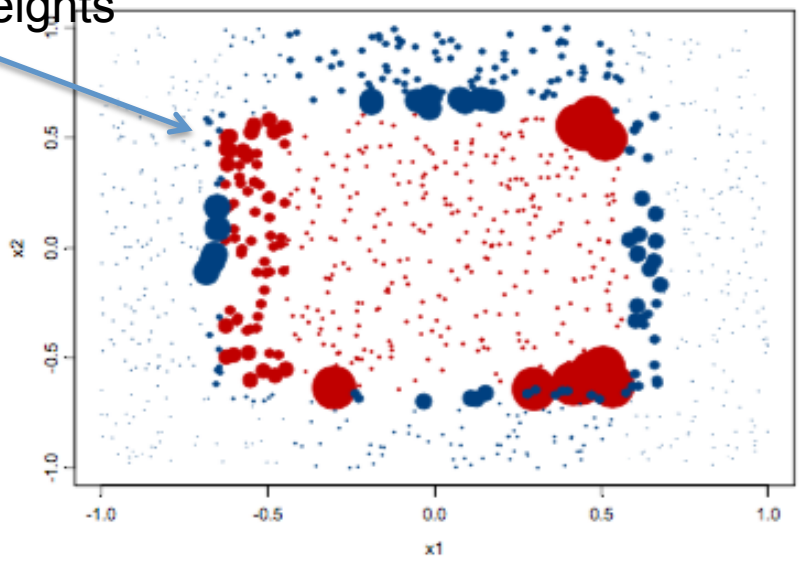
An experiment

Decision Boundary after 1 iteration

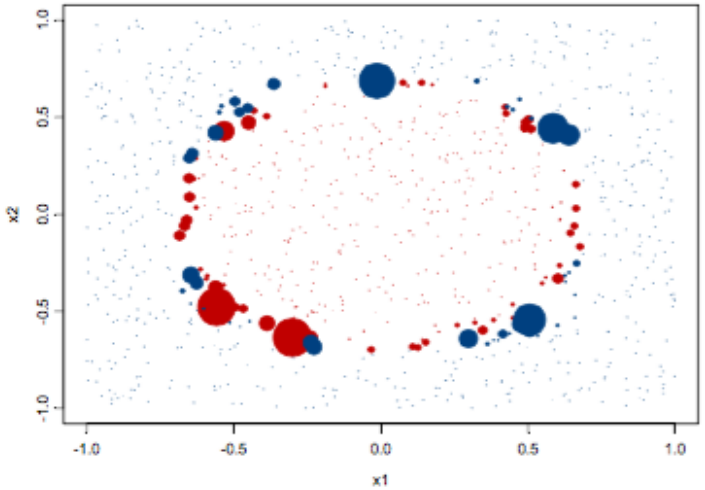


Strong Weights

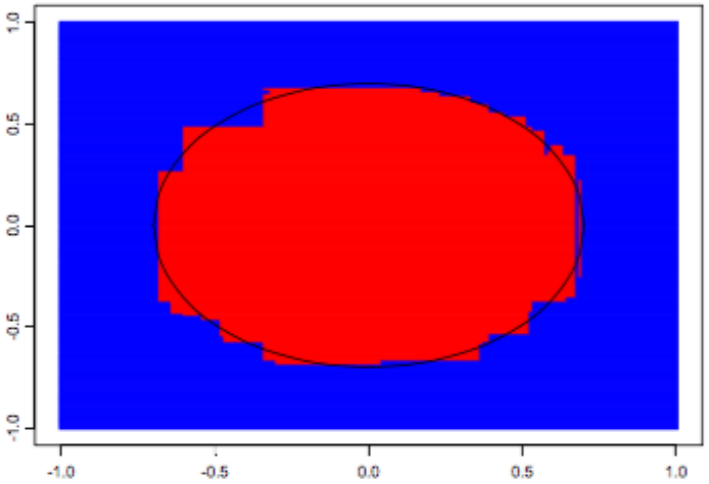
Decision Boundary after 3 iterations



Decision Boundary after 20 iterations



Decision Boundary after 100 iterations



Details of Ada Boost Algorithm

Algorithm:

1) Set $y_i \in \{-1,+1\}$ and start with identical weights $w_i = 1/n$

2) Repeat for $m = 1,2,...,M$:

   a) Fit the classifier $f_m(x) \in \{-1,+1\}$ using weights w_i

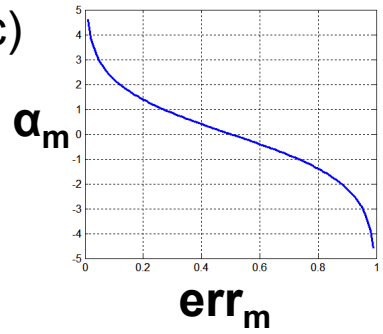
   b) Compute the weighted error $err_m = \sum_i w_i \cdot I[y_i \neq f_m(x_i)]$

   c) Compute the aggregation weight $\alpha_m = \log((1-err_m)/err_m)$

   d) Set $w_i \leftarrow w_i \cdot \exp(\alpha_m \cdot I[y_i \neq f_m(x_i)])$; normalize to $\sum_i w_i = 1$

3) Output $F_M(x) = sign\sum_{m=1}^{M} \alpha_m f_m(x)$

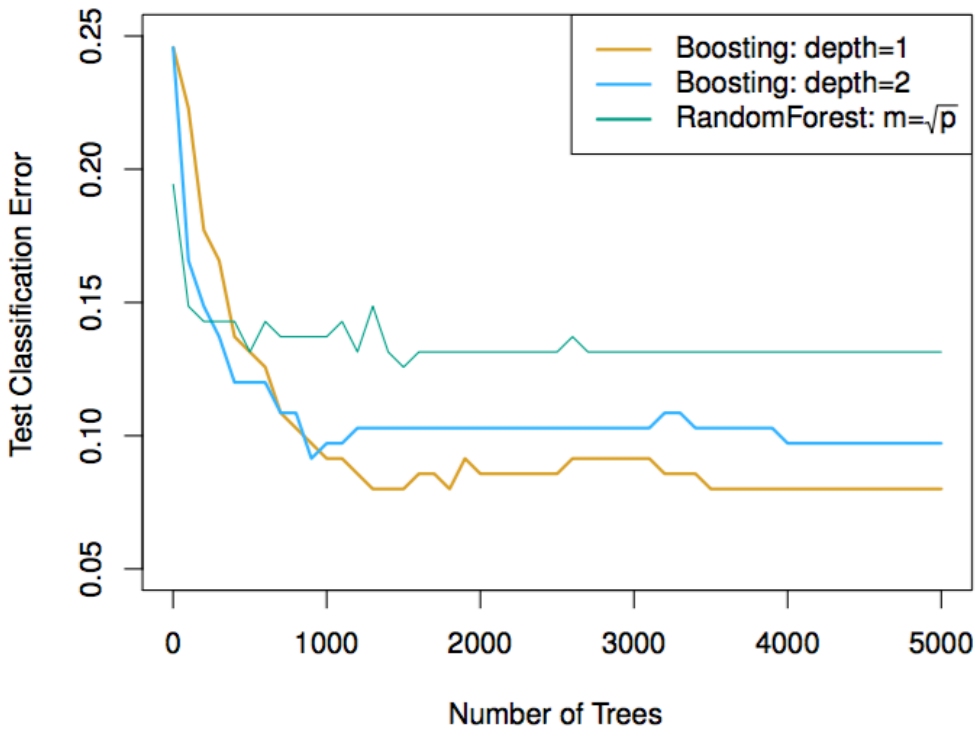
Zu c)



Error small ➔large weight

Error > 0.5 opposite is taken (quite uncommon only for small m)

Performance of Boosting



Boosting most frequently used with trees (not necessary)
Trees are typically only grown to a certain depth (1,2).
If trees of depth 2 would be better than trees of depth 1 (stubs) interactions between features would be important

Historical View

- The AdaBoosting Algorithm:
  - Freund & Schapire, 1990-1997
  - An *algorithmic* method based on **iterative data reweighting**

- Gradient Boosting:
  - Breiman (1999) and Friedman/Hastie/Tibshirani (2000)
  - A minimization of a loss function

- Extreme Gradient Boosting
  - Tianqi Chen (2014 code, 2016 published arXiv://1603.02754)
  - Theory similar to GB (also trees), a bit more regularisation
  - Much better implantation (distributed, 10x faster on single machine)
  - Often used in Kaggle Competitions as part of the winning solution

Gradient Boosting (just as sketch)

- Iterative procedure in which trees (often only stumps are added)
- Start from constant prediction:

$$
\begin{aligned}
\hat{y}_i^{(0)} &= 0 \\
\hat{y}_i^{(1)} &= f_1(x_i) = \hat{y}_i^{(0)} + \lambda f_1(x_i) \\
\hat{y}_i^{(2)} &= f_1(x_i) + f_2(x_i) = \hat{y}_i^{(1)} + \lambda f_2(x_i) \\
&\dots \\
\hat{y}_i^{(t)} &= \sum_{k=1}^{t} f_k(x_i) = \hat{y}_i^{(t-1)} + \lambda f_t(x_i)
\end{aligned}
$$

New function

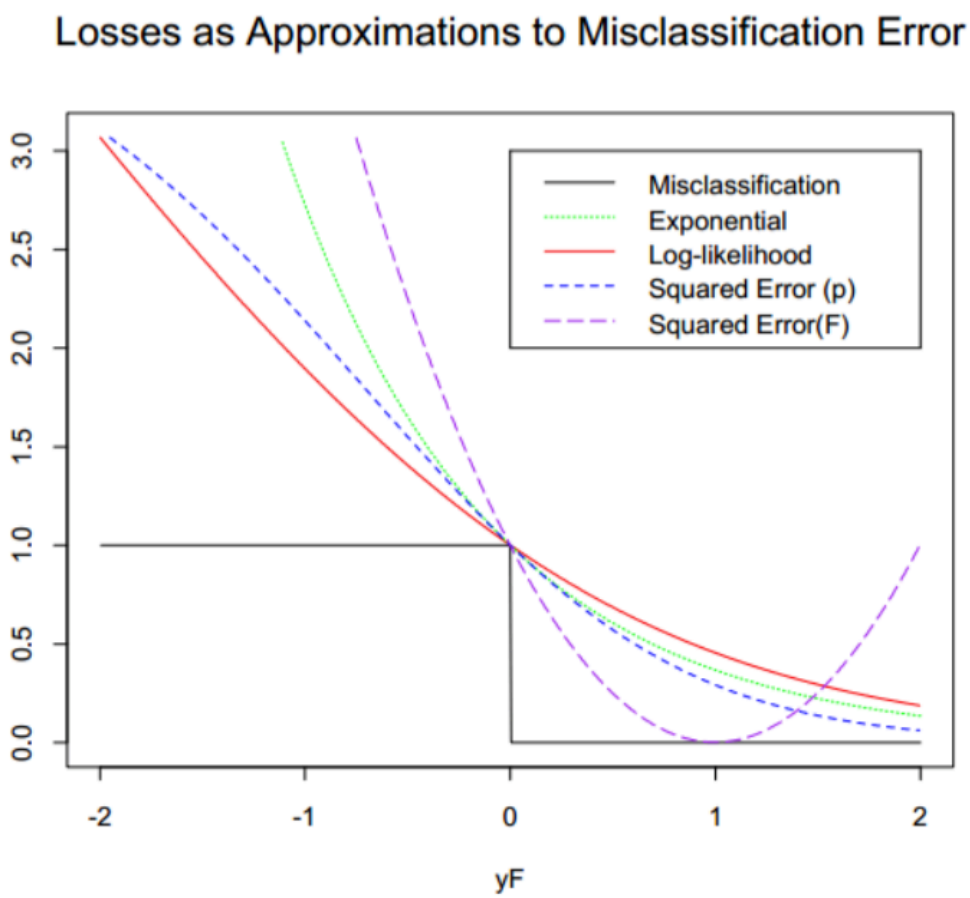
Model at training round t

Keep functions added in previous round

- Add that tree which minimizes the loss function with parameter λ
- Parameter λ “shrinkage”
- Possibly also with other functions
- Possible also for more then two classes

Boosting as minimization problem

Loss Functions



Misclassification

$$L(y,F) = I[y \neq sign(F)]$$

Exponential / AdaBoost

$$L(y,F) = \exp(-yF)]$$

LogLik / LogitBoost

$$L(y,F) = \log(1+\exp(-2yF))$$

Quadratic / L2-Boost

$$L(y,F) = (y-F)^2$$

Tuning Parameters for Boosted Trees

1. The *number of trees* B. Unlike bagging and random forests, boosting can overfit if B is too large, although this overfitting tends to occur slowly if at all. We use cross-validation to select B.
2. The *shrinkage parameter* λ, a small positive number. This controls the rate at which boosting learns. Typical values are 0.01 or 0.001, and the right choice can depend on the problem. Very small λ can require using a very large value of B in order to achieve good performance.
3. The *number of splits* d in each tree, which controls the complexity of the boosted ensemble. Often $d = 1$ works well, in which case each tree is a *stump*, consisting of a single split and resulting in an additive model. More generally d is the *interaction depth*, and controls the interaction order of the boosted model, since d splits can involve at most d variables.

Boosting in R (gbm)

```
### Just a Demo (Error estimated in-place!)
library(rpart) #For the data set
library(gbm)
str(kyphosis)
kyphosis$Kyphosis = as.numeric(kyphosis$Kyphosis) - 1# Nur die Werte 0,1

fit = gbm(Kyphosis ~ ., data=kyphosis,   n.trees=5000, shrinkage=0.001,
interaction.depth=2, distribution = "adaboost")

res = predict(fit, n.trees = 5000, type='response') #Is Probability
sum((res > 0.5) == (kyphosis$Kyphosis == 1))/length(res)
```

Boosting: XGBoost (Just for completeness)

- Similar to gbm (R and Python interfaces)
- Popular in Kaggle competitions
- Also a gradient descent procedure
- 10x faster then gbm
- Takes regulation differently into account (avoiding overfitting)
- Hyperparameters:

Hyperparameters	Grid	Random Domain	Description
Max Iterations	$[10, 20, 30]$	Uniform Integer $[5, 35]$	Number of trees to grow
Step Size	$[0.5, 0.7, 0.9]$	Uniform $[0.4, 1]$	Also called shrinkage
Max Depth	$[5, 7, 9]$	Uniform Integer $[4, 10]$	Max depth of a single tree
Row Subsample	$[0.5, 0.7, 0.9]$	Uniform $[0.4, 1]$	Portion of rows sampled to build one tree
Column Subsample	$[0.5, 0.7, 0.9]$	Uniform $[0.4, 1]$	Portion of columns sampled to build one tree

Combining classifiers
Tricks of the trade

Weighting Averages of predicted probabilities

- Idea:
  - Do a weighted average of the model predictions to get the final probability
- Solution described in https://www.kaggle.com/hsperr/otto-group-product-classification-challenge/finding-ensamble-weights
- Leave about 5% of the data away
- Learn a set of classifiers on the remaining 95%
- Make probability predictions p_{ik} on the 5% for the k=1,…,#Classifiers
- Prediction is:

$$p_i = \sum_{k=1}^{\#\text{Classifiers}} \alpha_k \, p_{ik}$$

- Optimize on the 5%with constraint sum $\alpha_k = 1$
- Use the optimized weights α_k for the final prediction

Alternative to weighted average: logistic regression

- Taken from write up Hoang Duong 6th place Otto competition

- At the beginning, I tried simple averaging. This works reasonably well. **I then learned the trick of transforming my probability estimate using inverse sigmoid, then fitting a logistic regression on top of it.** This work better. What I did at the end that work even better, is fitting a one hidden layer neural network on top of the inverse sigmoid transform of the probability estimate.

Stacking

- Basic Idea:
  - learn a function that combines the predictions of the individual classifiers
- Algorithm:
  - train n different classifiers $C_1 \ldots C_n$ (the *base classifiers*)
  - obtain predictions of the classifiers for the training examples
    - better do this with a cross-validation!
  - form a new data set (the *meta data*)
    - **classes**
      - the same as the original dataset
    - **attributes**
      - one attribute for each base classifier
      - value is the prediction of this classifier on the example
  - train a separate classifier M (the *meta classifier*)

Stacking

- Example:

Attributes			Class
x_{11}	...	x_{1n_a}	t
x_{21}	...	x_{2n_a}	f
...	...	...	...
x_{n_c1}	...	$x_{n_cn_a}$	t

(a) training set

C_1	C_2	...	C_{n_c}
t	t	...	f
f	t	...	t
...	...	...	...
f	f	...	t

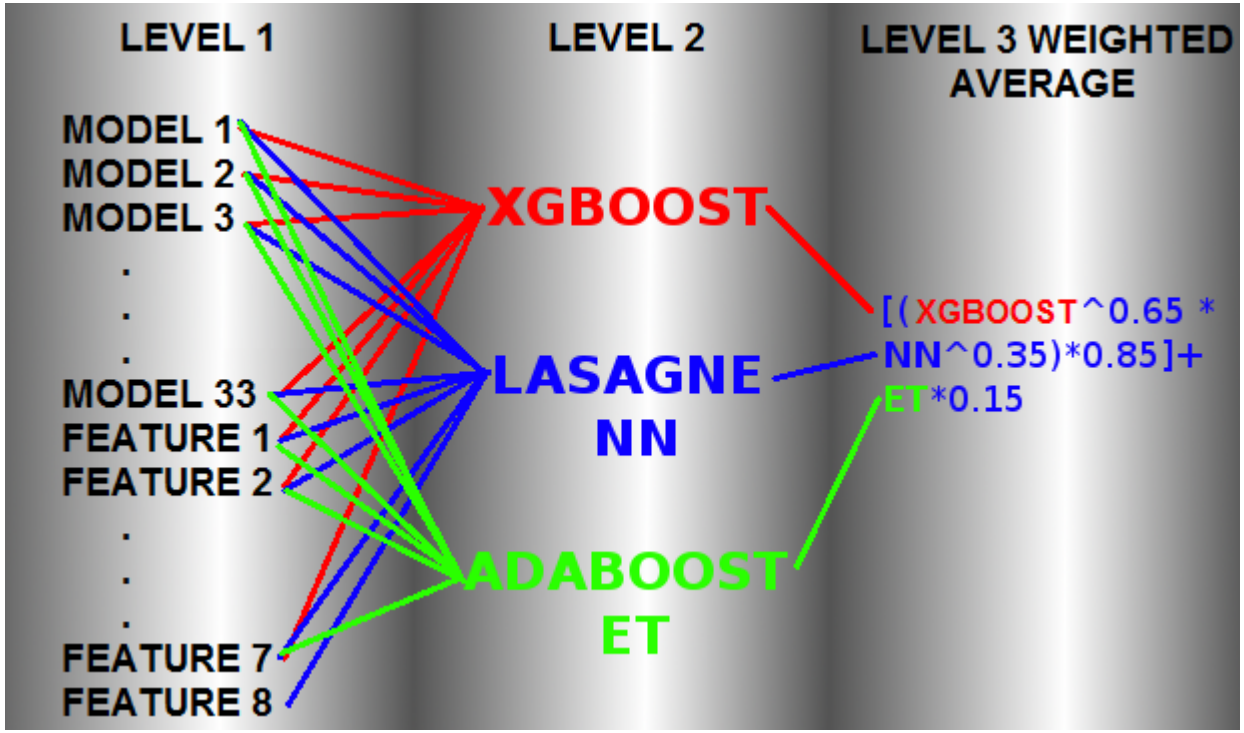
(b) predictions of the classfiers

C_1	C_2	...	C_{n_c}	Class
t	t	...	f	t
f	t	...	t	f
...	...	...	...	...
f	f	...	t	t

(d) training set for stacking

- Using a stacked classifier:
  - try each of the classifiers $C_1 ... C_n$
  - form a feature vector consisting of their predictions
  - submit this feature vectors to the meta classifier M

Winning solution for the Otto Challenge



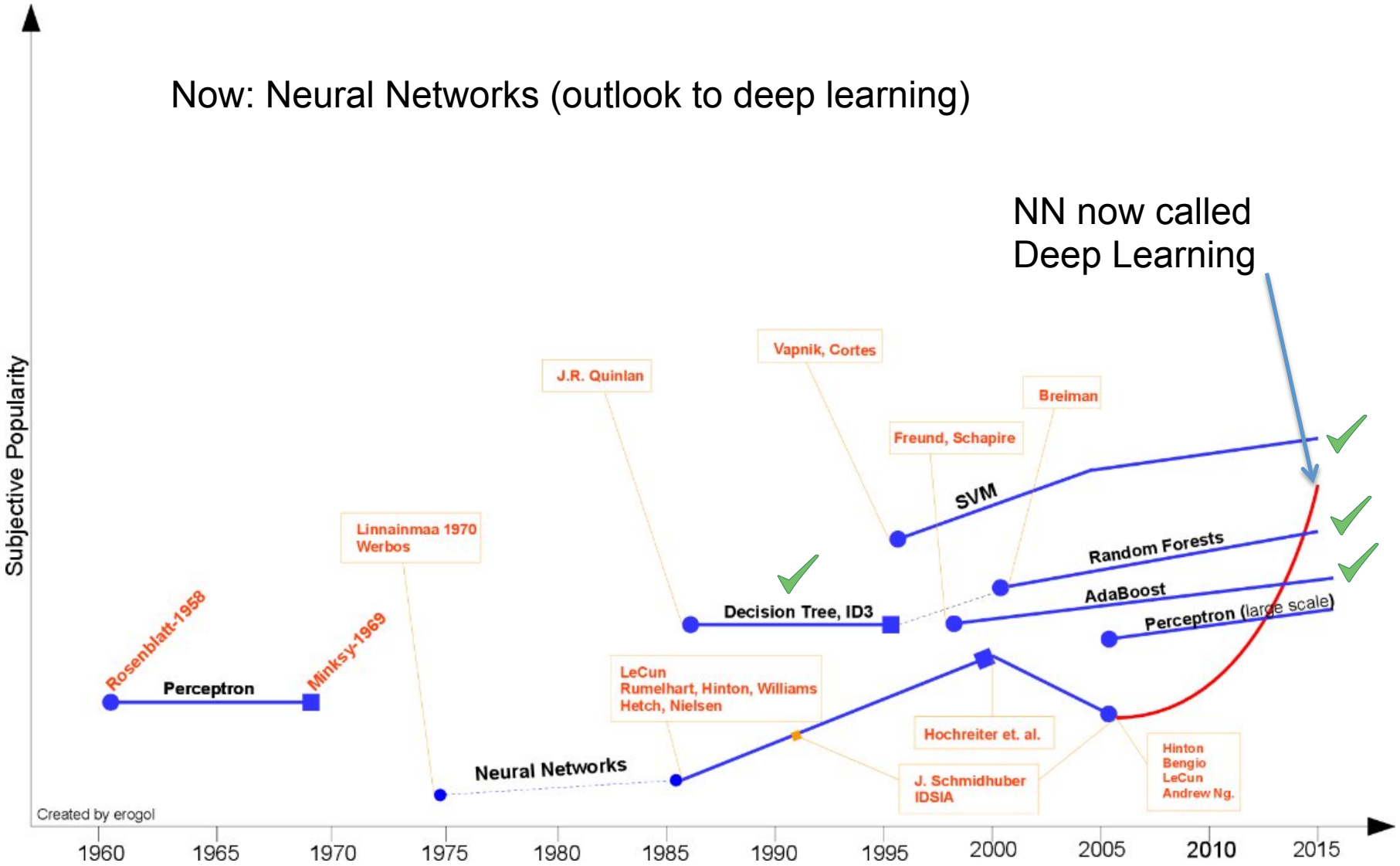
Use outcome from models (e.g. random forest) as meta features

See: https://www.kaggle.com/c/otto-group-product-classification-challenge/forums/t/14335/1st-place-winner-solution-gilberto-titericz-stanislav-semenov/79598#post79598

Neural Networks

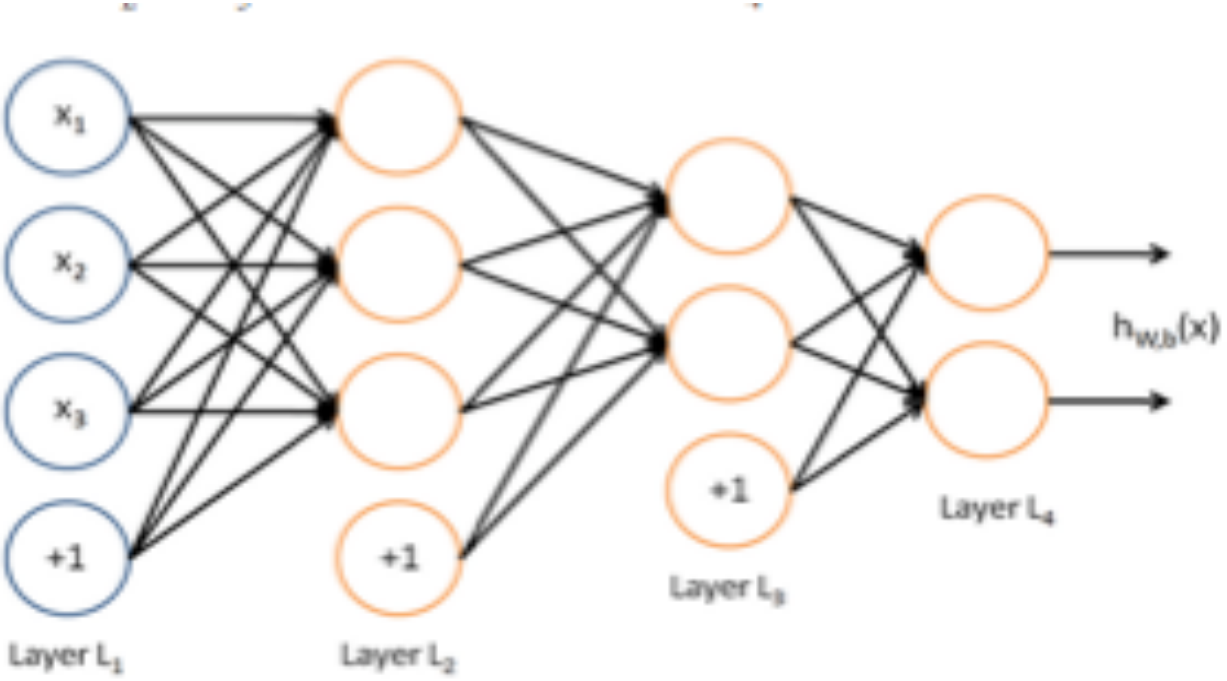
Brief History of Machine Learning (supervised learning)

Now: Neural Networks (outlook to deep learning)

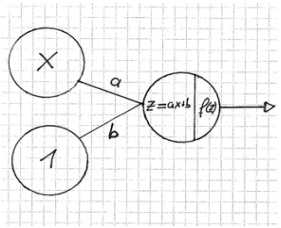


Taken from http://www.erogol.com/brief-history-machine-learning/

Architecture of a NN



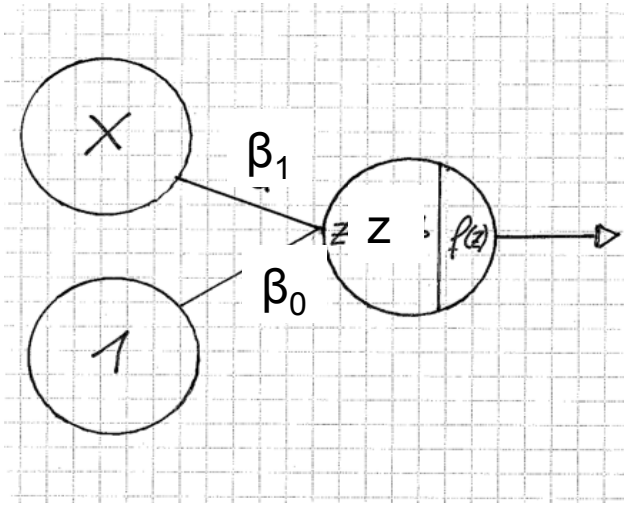
We start with....



1-D Logistic Regression

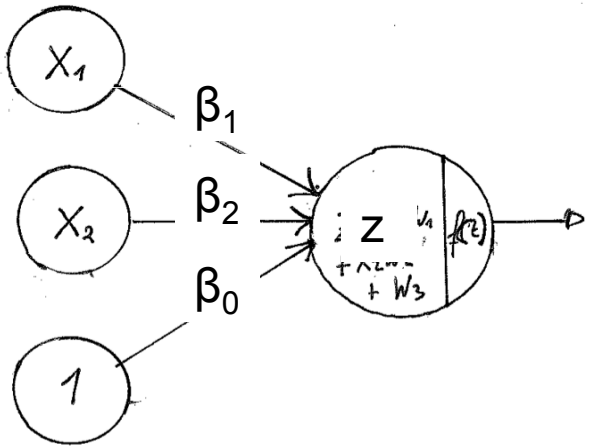
Logistic Regression the mother of all neural networks

1-D log Regression



$$z = \beta_0 + \beta_1 x$$

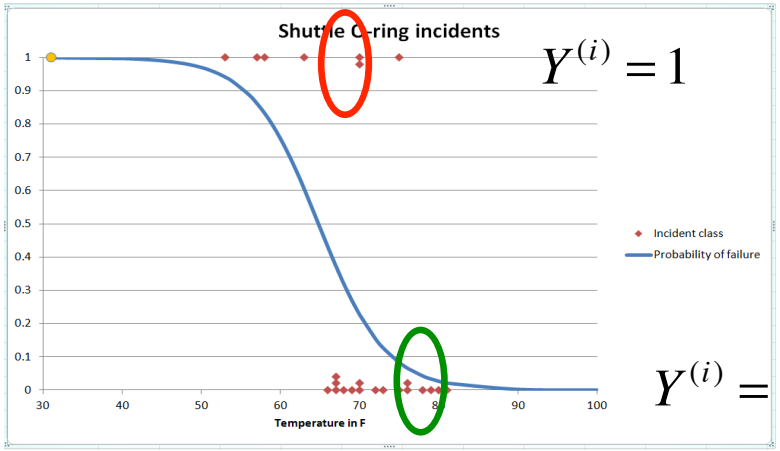
Multivariate Log.-Regression



$$z = \beta_0 + x_1\beta_1 + x_2\beta_2 = \beta^T x$$

$$p_1(x) = P(Y = 1 \mid X = x) = [1 + \exp(-\beta^T x)]^{-1} = \frac{\exp(\beta^T x)}{1 + \exp(\beta^T x)} = f(\beta^T x)$$

Likelihood: Probability of the training set (Training)



$Y^{(i)} = 1$

$Y^{(i)} = 0$

Training Data i = 1...N

$X^{(i)}, Y^{(i)}$

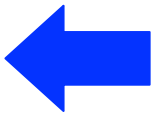
$$p_1(X) = P(Y = 1 \mid X) = (1 + e^{-(a \cdot x + b)})^{-1} = (1 + e^{-z})^{-1} = f(x)$$

Probability to find Y=1 for a given values X (single data point) and a, b

$p_0(X) = 1 - p_1(X)$ Probability to find Y=0 for a given value X (single data point)

Likelihood (probability+ of the training set given the parameters)

$$L(\beta_0, \beta_1) = \prod_{i \in All\ ones} p_1(x^{(i)}) * \prod_{i \notin All\ Zeros} p_0(x^{(j)})$$



Let‘s maximize this probability

Maximizing the Likelihood (Training)

Likelihood (prob of a given training set) want to maximized wrt. parameters

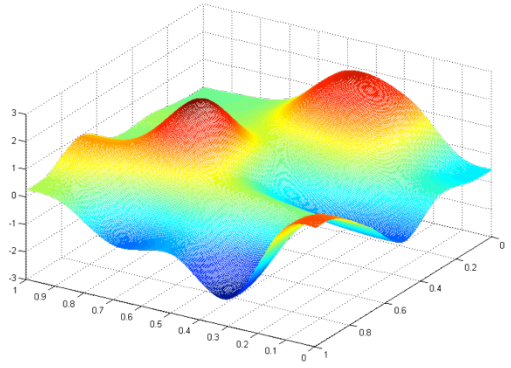
$$L(\beta_0,\beta_1) = \prod_{i \in All\ ones} p_1(x^{(i)}) \quad * \quad \prod_{i \notin All\ Zeros} p_0(x^{(j)})$$

Taking log (maximum of log is at same position)

$$-nJ(\beta) = L(\beta) = L(\beta_0,\beta_1) = \sum_{i \in All\ ones} \log(p_1(x^{(i)})) \quad + \quad \sum_{i \in All\ zeros} \log(p_0(x^{(i)})) = \sum_{i \in All\ Training} y_i \log(p_1(x^{(i)})) + (1-y_i)\log(p_0(x^{(i)}))$$

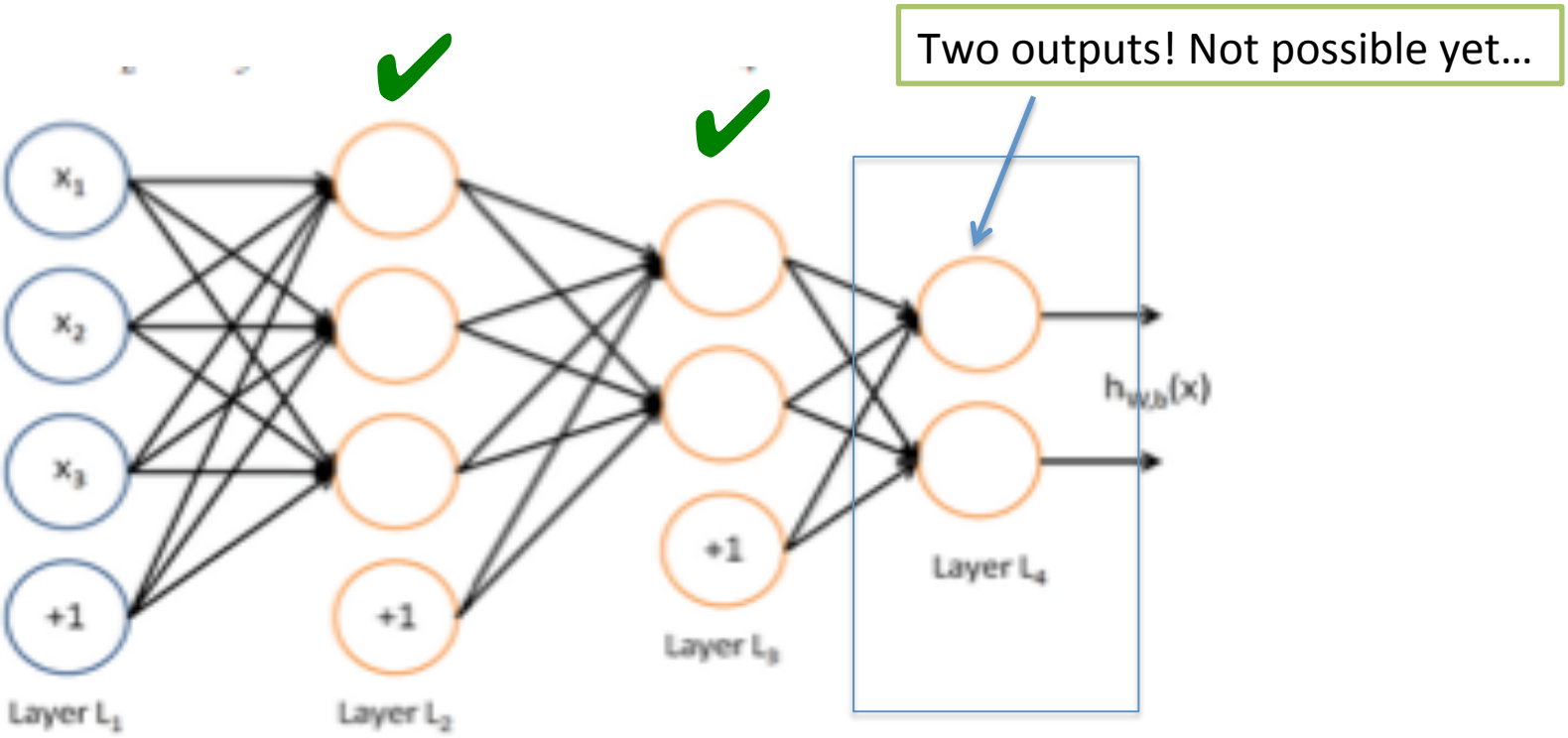
Gradient Descent for Minimum of J

$$\beta_i'' \leftarrow \beta_i' - \alpha \left. \frac{\partial J(\beta)}{\partial \beta_i} \right|_{\beta_i = \beta_i'}$$

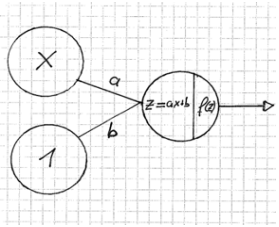


Ende der Wiederholung

Architecture of a NN



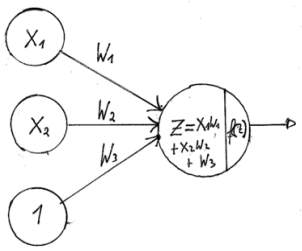
We started with....



1-D Logistic Regression

Cost function for multinomial regression

Training Examples Y=1
or Y=0

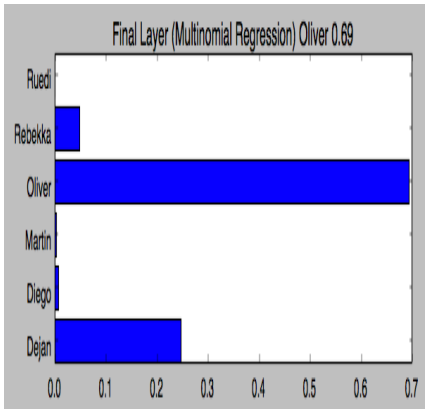


$$-nJ(\theta) = L(\theta) = L(a,b) = \sum_{i \in All\ ones} \log(p_1(x^{(i)})) \quad + \sum_{i \in All\ zeros} \log(p_0(x^{(i)}))$$

N Training Examples classes (1,2,3,…,K)

Output of last layer

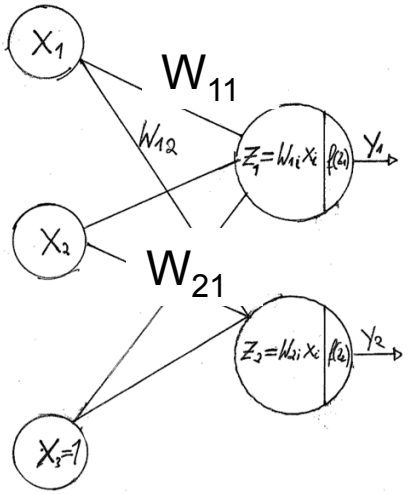
p_i



$$-nJ(\theta) = L(\theta) = L(a,b) = \sum_{i \in y_j=1} \log(p_1(x^{(i)})) \quad + \sum_{i \in y_j=2} \log(p_2(x^{(i)})) + \ldots + \sum_{i \in y_j=K} \log(p_K(x^{(i)}))$$

Example: Look at class of single training example. Say it's Dejan, if classified correctly p_dejan = 1 ➔ Loss = 0. Real bad classifier put's p_dejan=0 ➔ Loss = Inf.

Loss Function for multinomial regression



Function to maximize prob. to a certain class in last layer

$$-nJ(\theta) = L(\theta) = L(a,b) = \sum_{i \in y_j = 1} \log(p_1(x^{(i)})) \quad + \sum_{i \in y_j = 2} \log(p_2(x^{(i)})) + \ldots + \sum_{i \in y_j = K} \log(p_K(x^{(i)}))$$

Sum is over all training data with belong to class 1.

$$\theta_i'' \leftarrow \theta_i' - \alpha \left. \frac{\partial J(\theta)}{\partial \theta_i} \right|_{\theta_i = \theta_i'}$$

- Same procedure as before

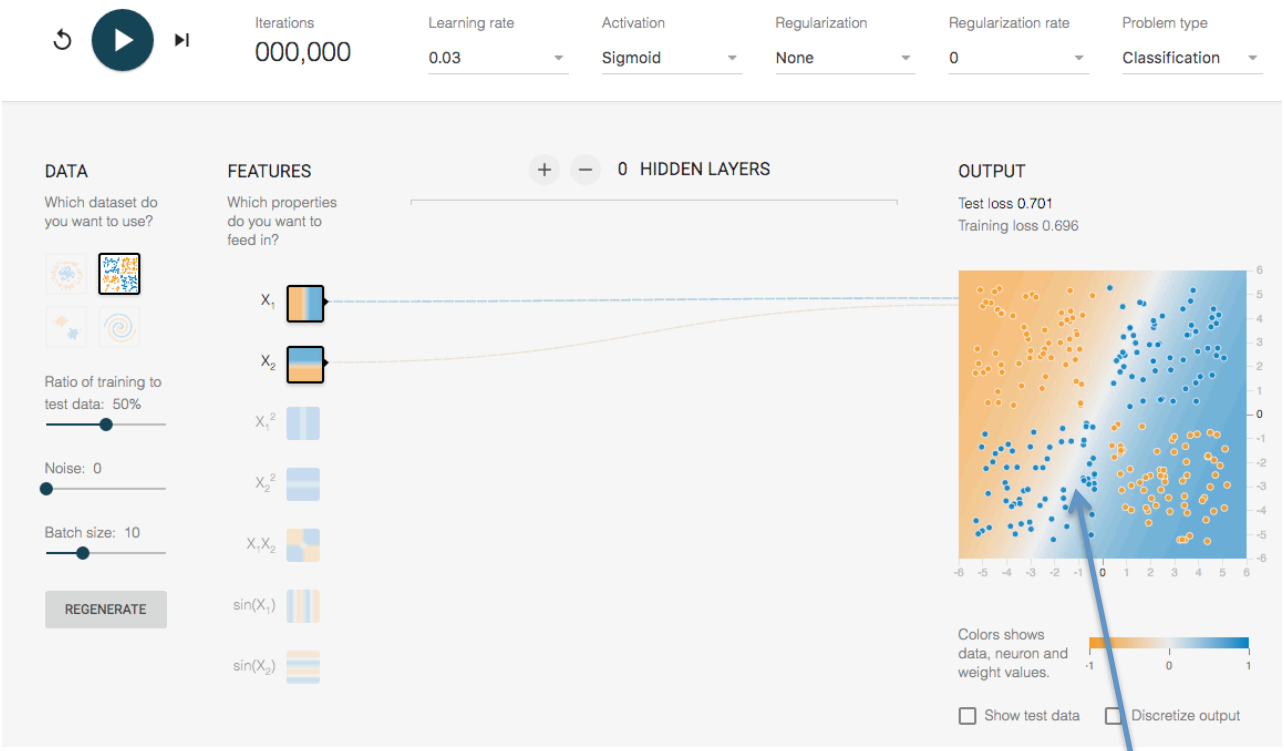
Multinomial Logistic Regression in R

```
library(nnet)
#An iterative procedure minimizing the loss function
fit  = multinom(Species ~ ., data = iris)
predict(fit, iris, type='class')[1:5]
preds = predict(fit, iris, type='prob')
true_column = as.integer(iris$Species)

# Check if we really optimized the loss function
p_class = c(preds[1:50,1], preds[51:100,2],
preds[101:150,3]) #These are sorted
-sum(log(p_class)) # The cost function
```

Limitations of (multinomial) logistics regression

Linear regression in NN speak: “on hidden layer”



Network taken from

Linear Boundary!

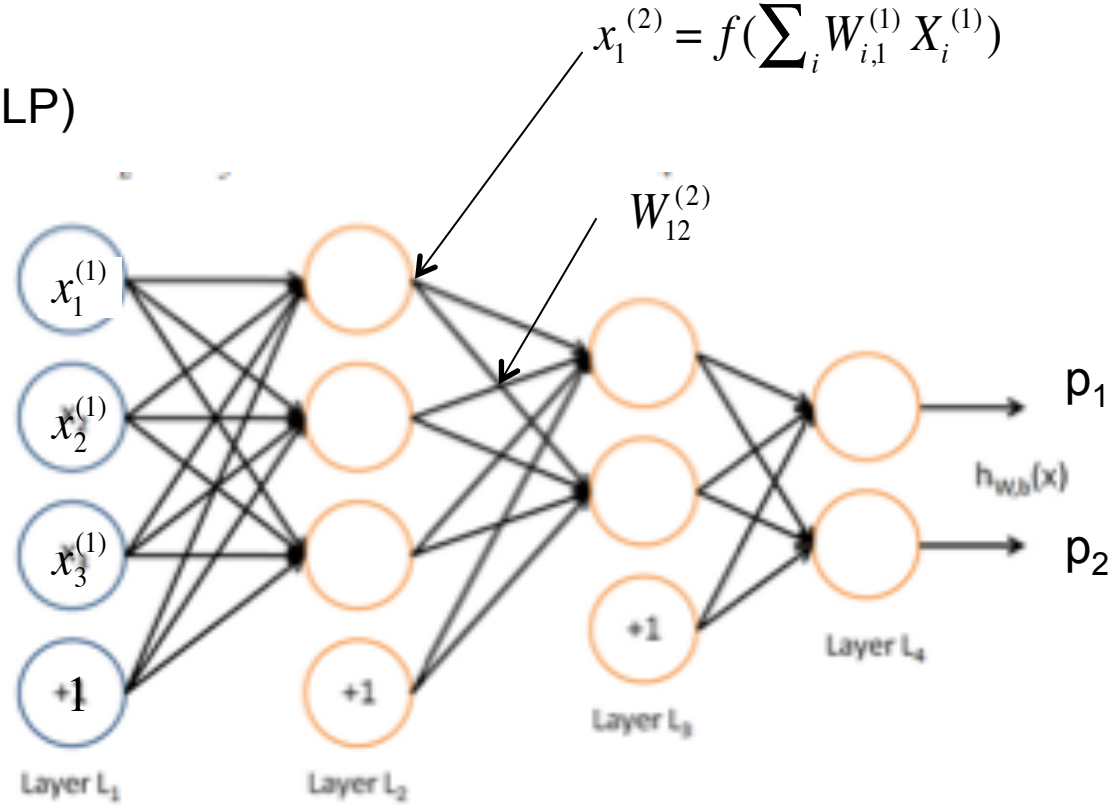
More than one layer

We have all the building blocks

- Use outputs as new inputs
- At the end use multin. logistic regression
- Names:
  - Fully connected network
  - Multi Layer Perceptron (MLP)

- Training via gradient descent

$$\theta_i'' \leftarrow \theta_i' - \alpha \left.\frac{\partial J(\theta)}{\partial \theta_i}\right|_{\theta_i=\theta_i'}$$



Simple chaining for afficinados (Einstein sum convention):

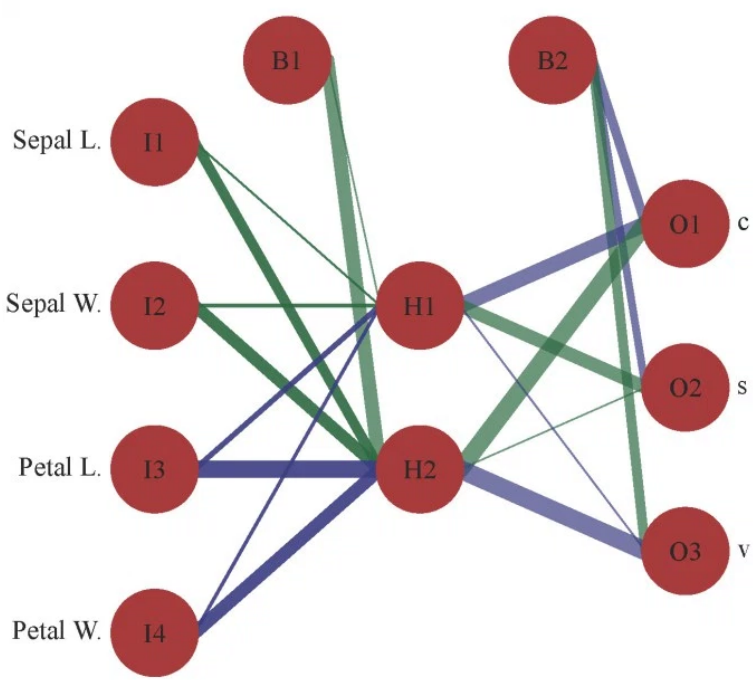
$$x_j^{(l)} = f(W_{j'j}^{(l-1)} f(W_{j''j'}^{(l-2)} f(W_{j'''j''}^{(l-3)}) \cdots f(W_{j''''j'''}^{(1)}))))$$

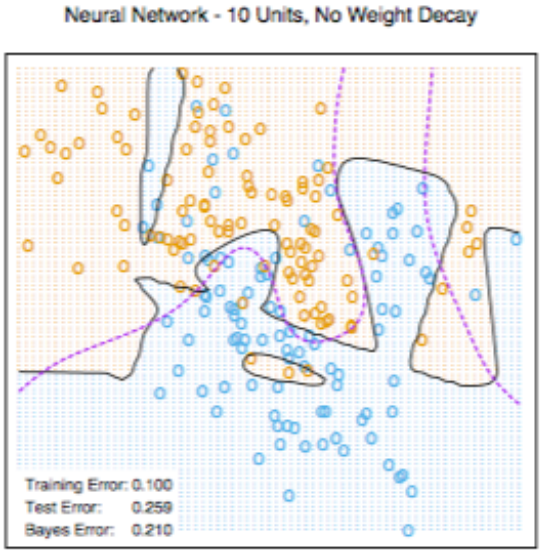
Neural Network with hidden units

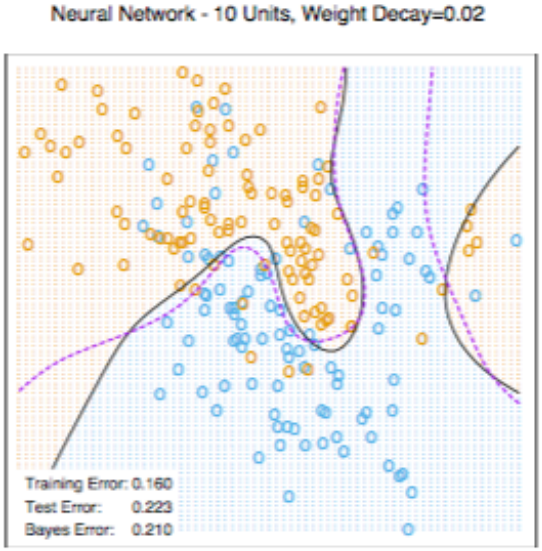
- Go to http://playground.tensorflow.org and train a neural network for the data:

- Use sigmoid activation and start with 0 hidden layers. Increase the number of hidden layers.

One hidden Layer







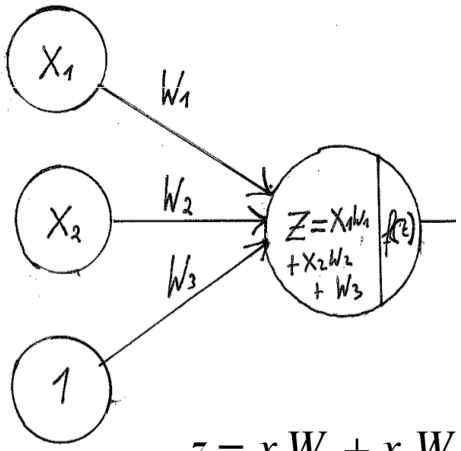
A network with one hidden layer is a universal function approximator

NN with one hidden layer in R

```
#### Fit a neural net with one hidden layer (size 12)
spam.fit = nnet(spam ~ ., data = spam[-testIdx,],
size=12, maxit=500)
spam.pred <- predict(spam.fit, spam[testIdx,],
type='class')
# Result
table(spam$spam[testIdx], spam.pred)
```

Different Activations

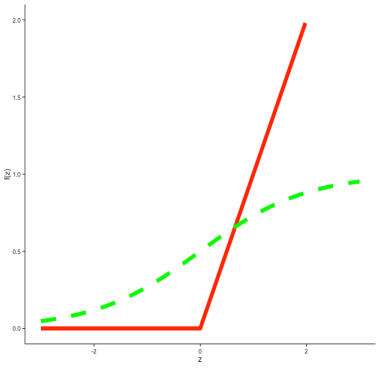
N-D log regression



$$f(z) = \begin{cases} \dfrac{\exp(z)}{1+\exp(z)} \\ \max(0,z) \end{cases}$$

$$z = x_1 W_1 + x_2 W_2 + W_3 = \theta^T x$$

Activation function a.k.a. Nonlinearity f(z)



Motivation:

Green:
logistic regression.

Red:
ReLU faster convergence

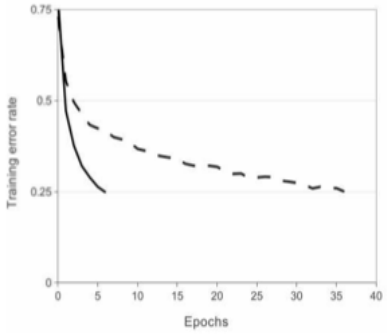
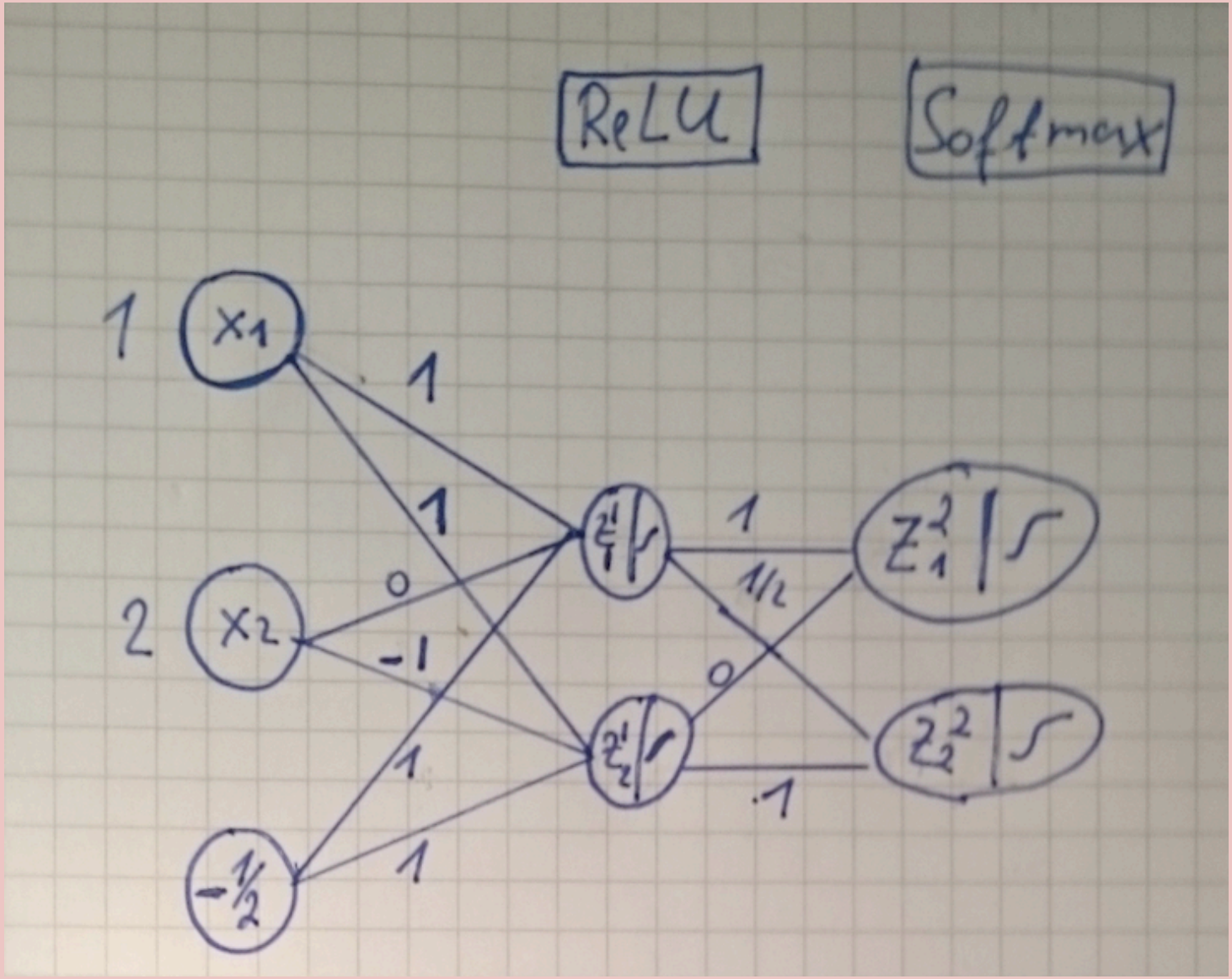


Figure 1: A four-layer convolutional neural network with ReLUs **(solid line)** reaches a 25% training error rate on CIFAR-10 six times faster than an equivalent network with tanh neurons

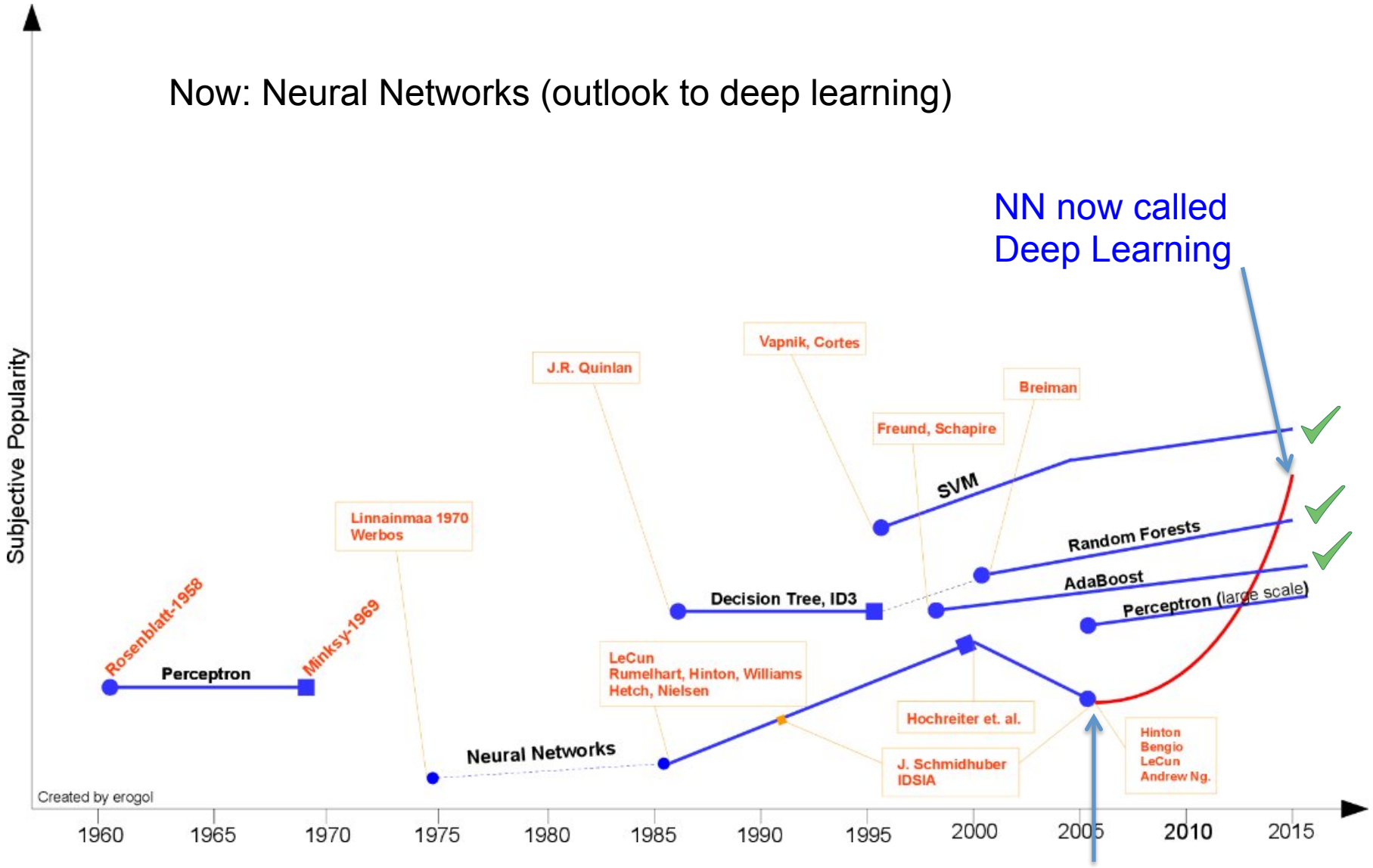
Source:
Alexnet
Krizhevsky et al 2012

Aufgabe: Rechnen von Hand



Brief History of Machine Learning (supervised learning)

Now: Neural Networks (outlook to deep learning)



Taken from http://www.erogol.com/brief-history-machine-learning/

With R 1 hidden layer ☹


WE NEED TO GO
DEEPER
memegene