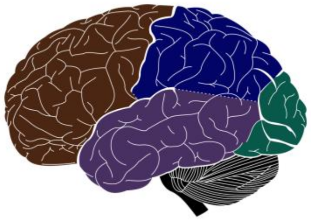
Neural networks (NN)

Ruxandra Stoean

Additional bibliography

- Simon O. Haykin , Neural Networks and Learning Machines (3rd Edition), Prentice Hall, 2008
- Charu C. Aggarwal, Neural Networks and Deep Learning: A Textbook, Springer, 2018
- Christopher M. Bishop, Neural Networks for Pattern Recognition, Oxford University Press, 1996

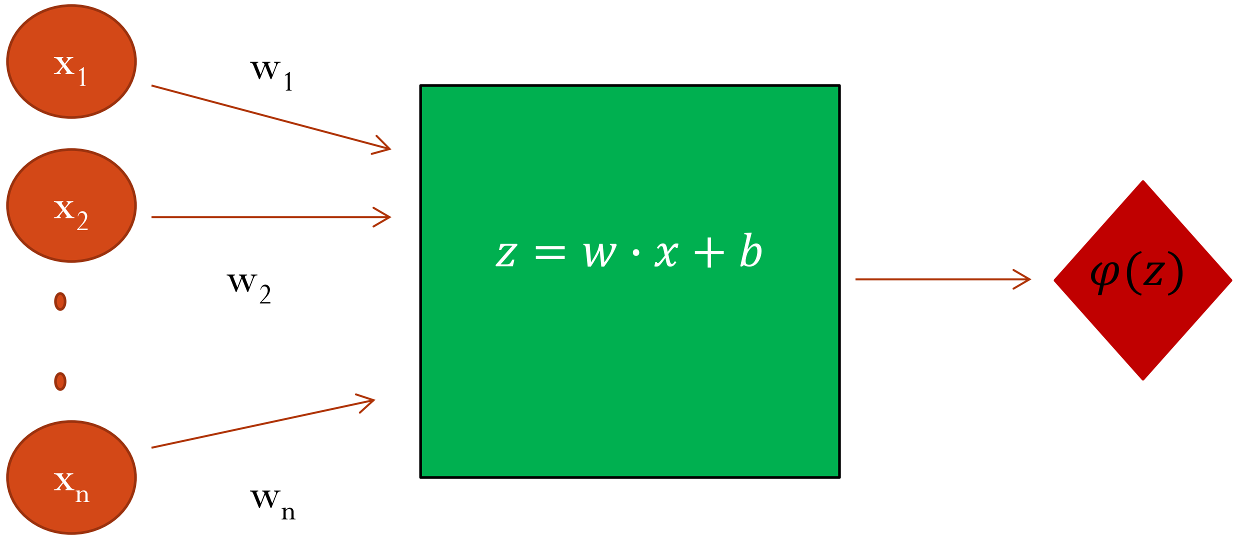
Modelling the human brain



- Artificial neural networks simulate the mode of neural interaction within the human brain, directed towards learning:
  - Base units – neurons
  - Connections between them – synapses

- The intensities (weights) of the synapses determine the performance of learning.

Artificial neuron McCulloch-Pitts



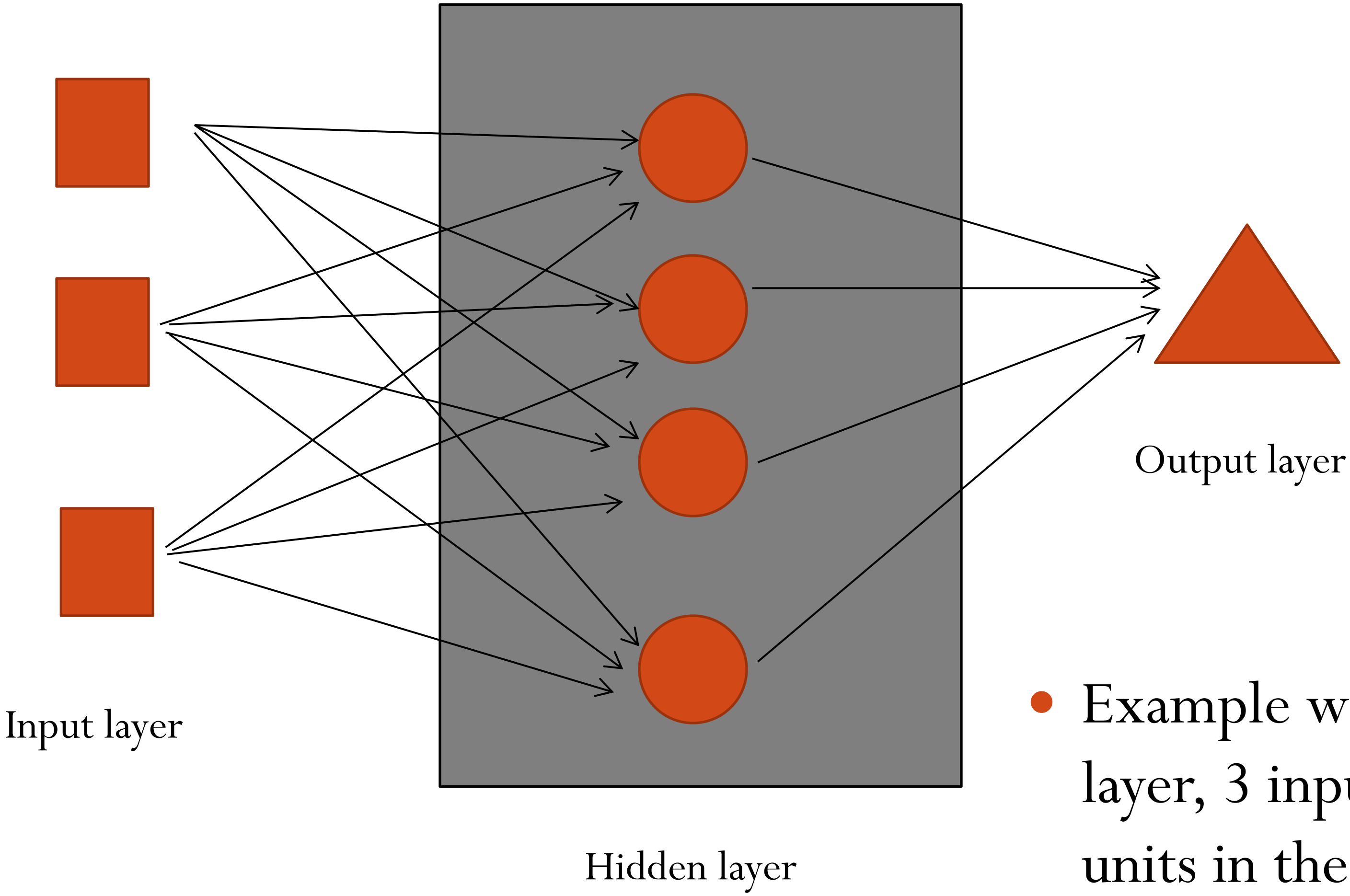
- x – input vector
- w – synaptic weights
- b – bias

- z – linear combination unit
- φ – activation function of the neuron
- Output $\varphi(z)$

Structure of a neural network

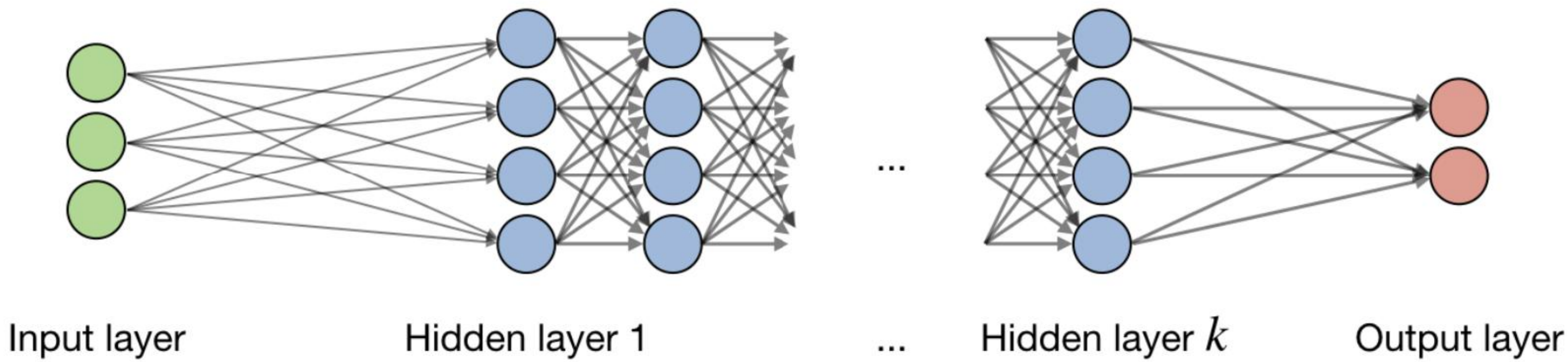
- Input neurons (units) – input layer (input data attributes)
- Hidden neurons in the black-box of learning (inside data components to be learnt)
  - In one or more hidden layers
  - The output of a layer becomes the input for the next layer
- Output neurons – output layer (network output - classes/response)
- The (supervised) learning starts from the problem data and optimizes the weights of the neurons on the base of the difference between the predicted and the real response.

Simple NN architecture



- Example with 1 hidden layer, 3 input neurons, 4 units in the hidden layer and 1 output

General NN architecture



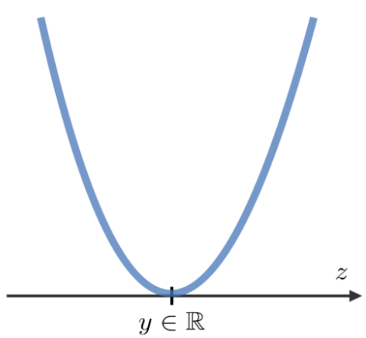
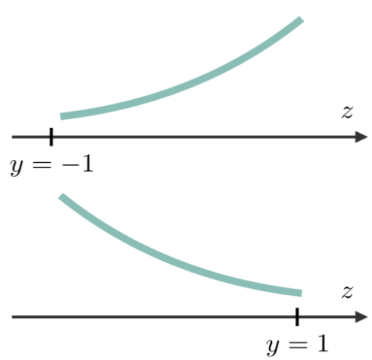
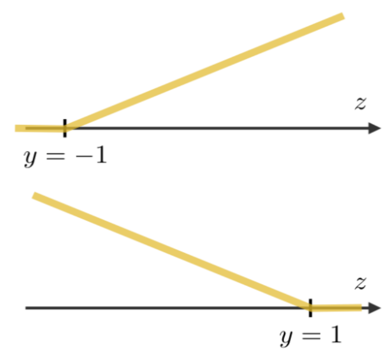
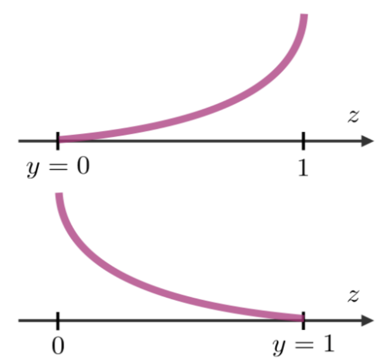
https://stanford.edu/~shervine/teaching/cs-229

NN Flow

- Batches of input data are given in the input layer
  - Batch size – amount of data passed in one forward pass iteration
- Neurons are multiplied by their weights
  - The product is further transformed by the activation function
  - The new collection of neurons are fed to the next layer
- The output of the last layer – the prediction – is compared to the output -> the loss -> to be minimized
  - Weights are initially randomly initialized
  - <u>But</u>, with the loss, the gradient of the cost function is calculated for the weights
  - Weights are updated after each iteration
- All data batches pass – one epoch ends
- The process is repeated for a number of epochs

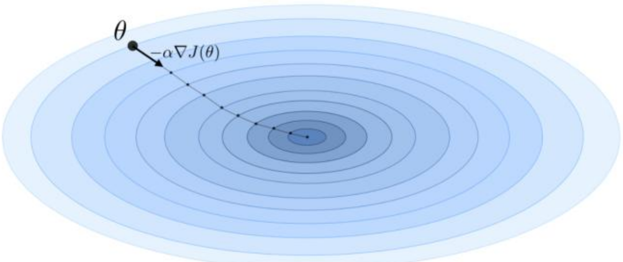
Loss function

- The difference between the predicted value of the model z and the corresponding data real output y
- Cost function – sum of loss over all training data (or batch)
- NN use (binary, categorical) cross-entropy loss since it allows probability estimates; MSE for regression

Least squared error	**Logistic loss**	**Hinge loss**	**Cross-entropy**
$\frac{1}{2}(y-z)^2$	$\log(1+\exp(-yz))$	$\max(0,1-yz)$	$-\left[y\log(z)+(1-y)\log(1-z)\right]$
			
Linear regression	Logistic regression	SVM	Neural Network

https://stanford.edu/~shervine/teaching/cs-229

Backpropagation



https://stanford.edu/~shervine/teaching/cs-229

- Backpropagation
  - The process of propagating the error backward in order to update the weights
  - It computes the gradient of the cost function with respect to the model weights through the chain rule from calculus

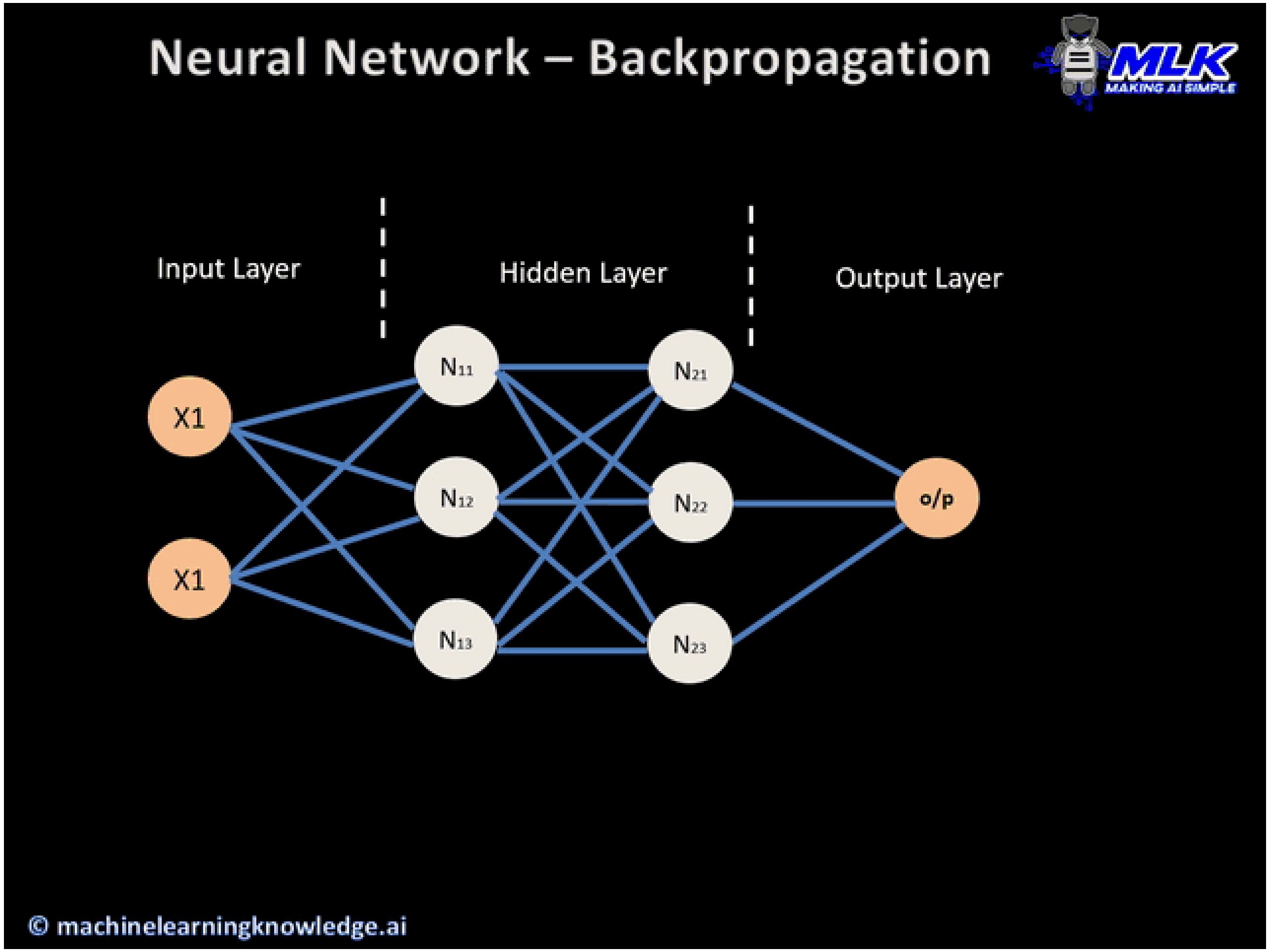
- Gradient descent
  - Optimization algorithm to find the best weight values based on the gradient
  - Descends down the cost function, by the learning rate, to the minimum
  - Examples: Stochastic gradient descent, Adam, AdaGrad, RMSProp

Learning rate

- The rate of updating the weights
  - Constant
  - Small – converges slowly and gets stuck in local minima
  - Large – unstable

- Better let is decrease gradually by time step
- Or even better – adaptive (e.g. Adam), depending on the decrease of the training loss

Illustration

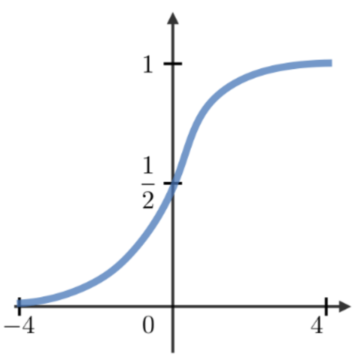
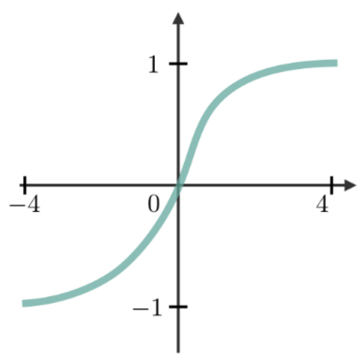
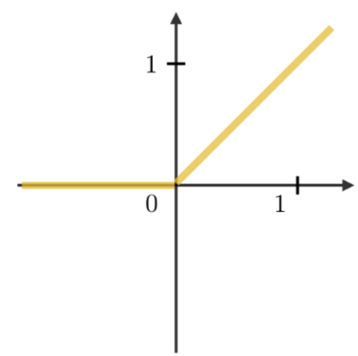
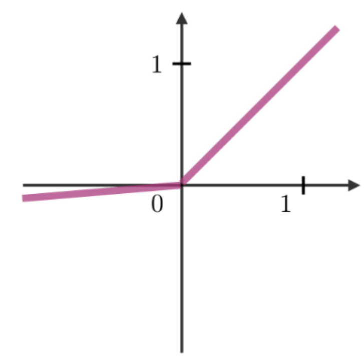


Training with SGD – batches in bold

https://www.kaggle.com/learn

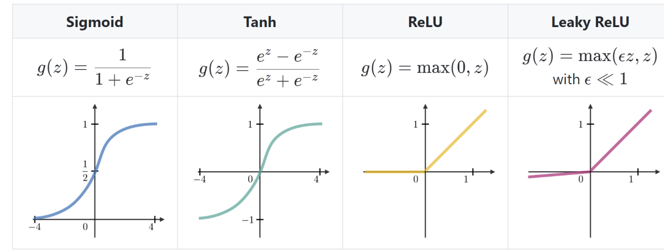
Activation function

- It induces non-linearity to the computed unit z.

Sigmoid	Tanh	ReLU	Leaky ReLU
$g(z) = \dfrac{1}{1+e^{-z}}$	$g(z) = \dfrac{e^z - e^{-z}}{e^z + e^{-z}}$	$g(z) = \max(0, z)$	$g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$
			

https://stanford.edu/~shervine/teaching/cs-229

Choice of activation

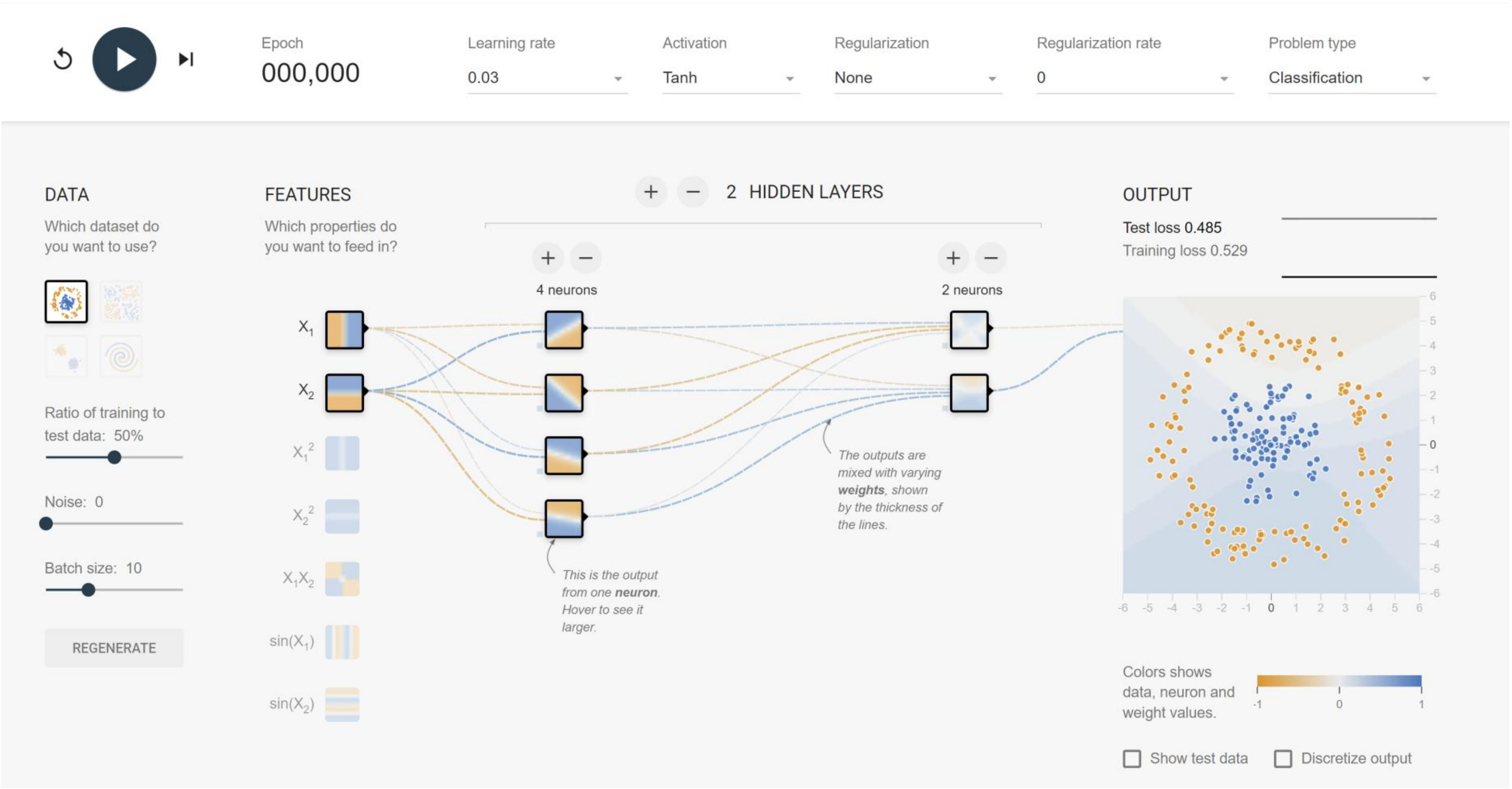


- **Sigmoid (logistic)**: smooth gradient; normalized output (between 0 and 1); clear prediction
  - **Softmax** – the extension to multi-class; used usually for last layer
  - But *vanishing gradient* for very high or low input (i.e., slow or no further learning); non-zero centered output
  - **TanH**: variant for zero centered
- **ReLU**: computationally faster
  - But *dying ReLU* for negative or zero input (i.e., gradient is 0 -> no learning)
  - Leaky ReLU: variant for negative or zero input to still have backpropagation

Regularization

- Assumption: a model with smaller weights is less complex than one with larger weights
- Penalization for the weights added to the cost function
- λ - regularization rate
- L1 norm is the sum of absolute values for the weights $\lambda \sum |w_i|$
- L2 norm is the sum of squared values for the weights $\lambda \sum w_i^2$

Example of NN flow



http://playground.tensorflow.org/

Package neuralnet in R

- Parameter *stepmax* denotes the maximum steps for training until stop.
- If the error *threshold* is not reached, then convergence is not achieved and no weights are given.
- If this happens, the default values of either *stepmax* = 1e+05 or *threshold* = 0.01 have to be increased.
- Function detects classification or regression based on the type of the variable to be predicted.

NN in R

```r
library(neuralnet)
library(datasets)
library(e1071)
library(caret)

data(iris)
dat <- iris

repeats <- 30
classColumn <- 5
accuracies <- vector(mode="numeric", length = repeats)
index <- 1:nrow(dat)
```

- Resilient backpropagation (Rprop) algorithm as the gradient descent algorithm for backpropagation
- Default activation function is logistic (if linear.output = FALSE)

Scaling

- NN sensitive to scaling
- Scaling is not done by default in package neuralnet
- Repeat same principle, fit on the training set and then transform both training and test sets
  - Use function *preprocess* from library caret
  - Use method *range* for Min-Max scaler

- The function *predict* returns class probabilities for classification
  - Hence the maximum probability has to be taken to output the predicted class

NN in R

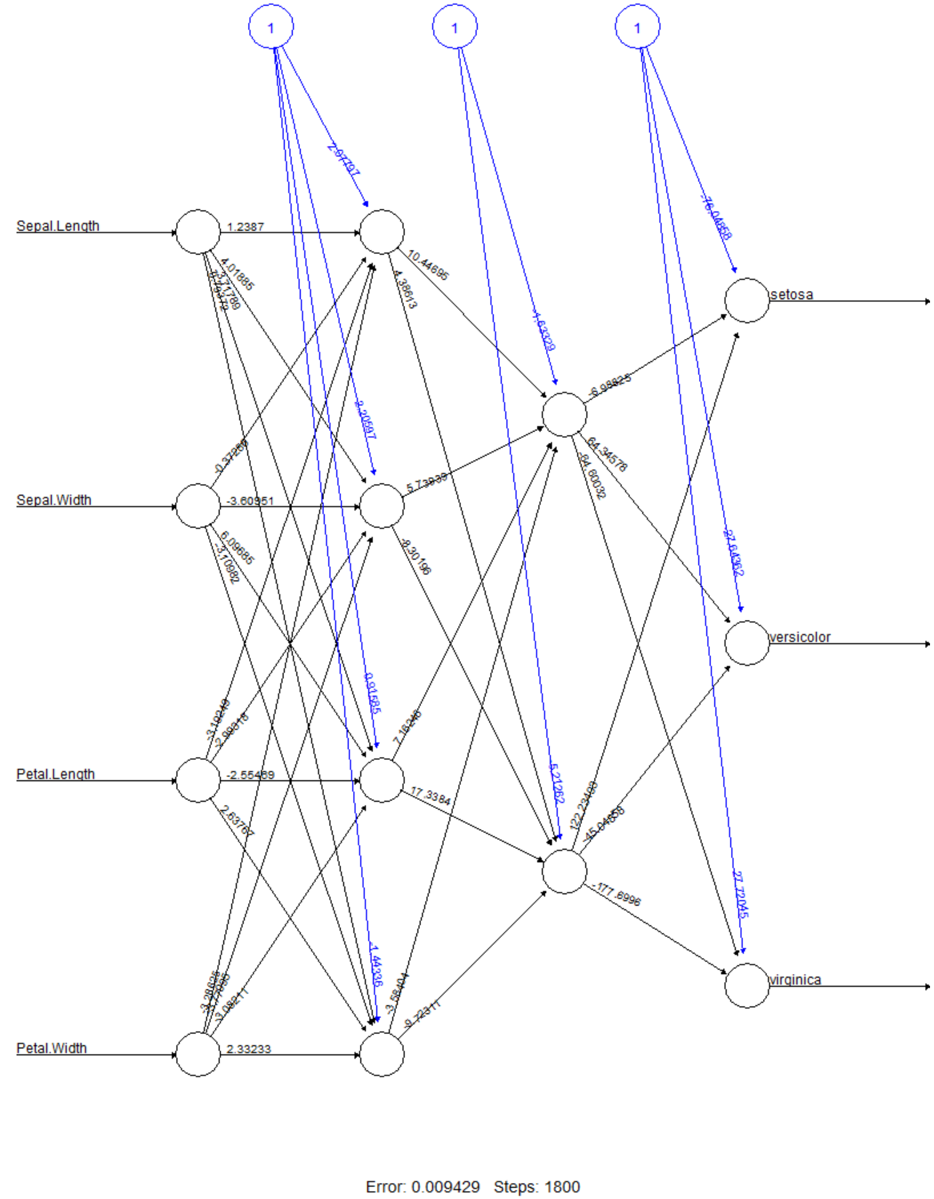
```
for (i in 1:repeats){
  testindex <- sample(index, trunc(length(index) / 3))
  testset <- dat[testindex, ]
  trainset <- dat[-testindex, ]

  process <- preProcess(as.data.frame(trainset), method = c("range"))
  train_scaled <- predict(process, as.data.frame(trainset))
  test_scaled <- predict(process, as.data.frame(testset))

  # Hidden specifies the number of neurons in each hidden layer
  # the length of the vector gives the number of hidden layers

  # If activation function should not be applied to the output neurons,
  # linear_output = TRUE, otherwise FALSE.
  nn_model <- neuralnet(Species ~ ., data = train_scaled, hidden = c(4, 2), linear.output = FALSE, stepmax = 1e+06)
  nn_pred <- predict(nn_model, test_scaled[, -classColumn])
  labels <- c("setosa", "versicolor", "virginica")
  pred_label <- labels[max.col(nn_pred)]
  contab <- table(pred = pred_label, true = test_scaled[, classColumn])
  accuracies[i] <- classAgreement(contab)$diag
}

# plot best representation of last model
plot(nn_model)

print(accuracies)
print(mean(accuracies))
print(sqrt(var(accuracies)))
print(summary(accuracies))

print("Confusion matrix of last run")
print(contab)
```

Classification results

```
 [1] 0.96 0.96 0.98 0.94 0.98 0.98 0.98 0.96 0.94 0.92 0.98 0.96 0.96 1.00 0.94
[16] 0.98 0.98 0.94 0.96 0.92 0.96 0.96 0.96 0.98 0.88 0.96 0.90 0.94 0.94 0.96
[1] 0.9553333
[1] 0.02609444
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.8800  0.9400  0.9600  0.9553  0.9800  1.0000
[1] "Confusion matrix of last run"
            true
pred         setosa versicolor virginica
  setosa         18          0         0
  versicolor      0         13         0
  virginica       0          2        17
```

Plot NN

- Black arrows – variable weight values (without activation)
- Blue lines – bias values



NN in Python

- In *sklearn* and the dedicated sublibrary *neural_network*, there is the function *MLPClassifier*.

- Gradient algorithms:
  - Stochastic gradient descent (SGD)
  - Adam

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn import datasets
from sklearn.preprocessing import MinMaxScaler

repeats = 30
accuracies = []
scaler = MinMaxScaler()

ix, iy = datasets.load_iris(return_X_y=True)

for i in range(0, repeats):
    #random generation of training and test, 66%-33%
    ix_train, ix_test, iy_train, iy_test = train_test_split(ix, iy, test_size = 0.33)

    #scale data
    fit_scalar = scaler.fit(ix_train)
    ix_train_scaled = fit_scalar.transform(ix_train)

    nnm = MLPClassifier(hidden_layer_sizes=(4, 2), solver='adam', activation = 'tanh', max_iter = 10000)
    nnm.fit(ix_train_scaled,iy_train)

    ix_test_scaled = fit_scalar.transform(ix_test)
    iy_pred = nnm.predict(ix_test_scaled)
    accuracies.append(accuracy_score(iy_test, iy_pred))

print(accuracies)
```

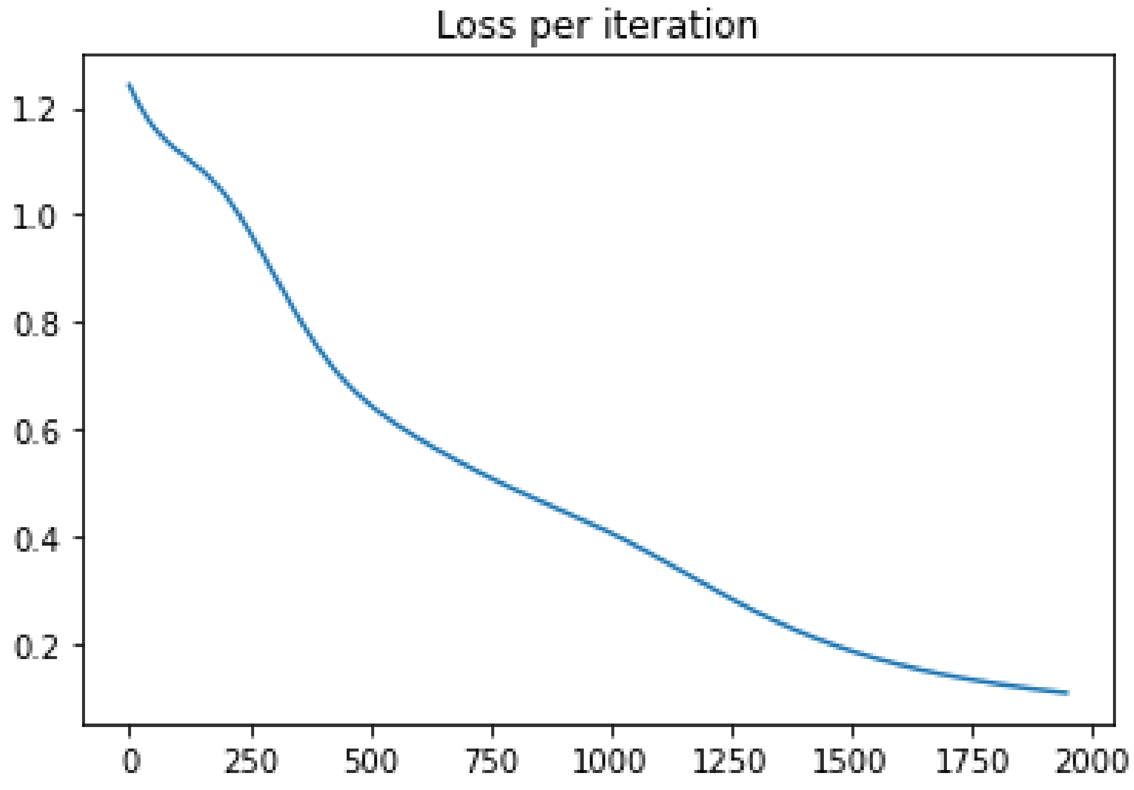
Results

```
print("Mean accuracy", np.mean(accuracies))
print("Standard deviation", np.std(accuracies))
confusion_matrix(iy_test,iy_pred)
```

```
Mean accuracy 0.9606666666666668
Standard deviation 0.023371397523944144

array([[13,  0,  0],
       [ 0, 13,  3],
       [ 0,  0, 21]], dtype=int64)
```

Loss decrease

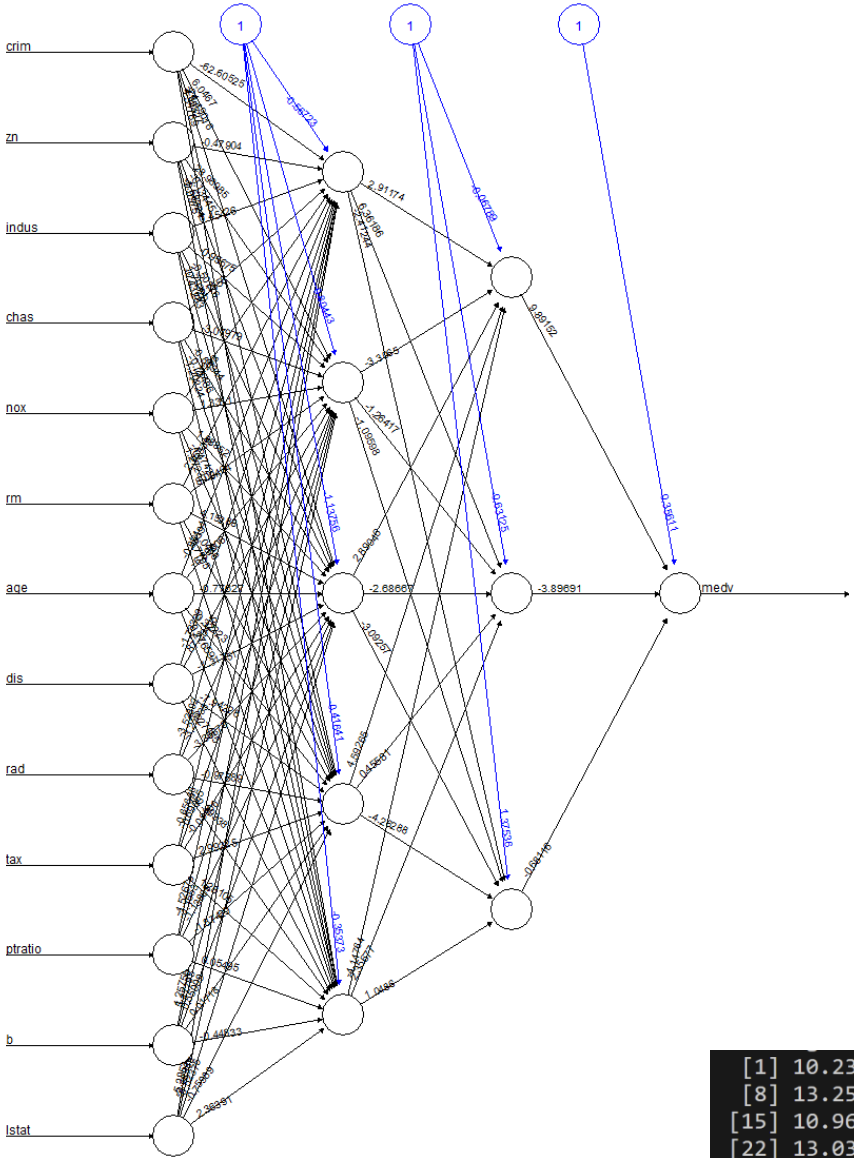
```
plt.plot(nnm.loss_curve_)
plt.title("Loss per iteration")
```



NN regression in R

```r
library(neuralnet)
library(mlbench)
library(caret)

data(BostonHousing)

# transform factor columns to integer for neuralnet
dat <- data.frame(lapply(BostonHousing, as.numeric))

repeats <- 30
classColumn <- 14
mses <- vector(mode = "numeric", length = 30)
index <- 1:nrow(dat)

for (i in 1:repeats){
  testindex <- sample(index, trunc(length(index) / 4))
  testset <- dat[testindex, ]
  trainset <- dat[-testindex, ]

  dif <- max(trainset[classColumn]) - min(trainset[classColumn])

  process <- preProcess(as.data.frame(trainset), method = c("range"))
  train_scaled <- predict(process, as.data.frame(trainset))
  test_scaled <- predict(process, as.data.frame(testset))

  nn_model <- neuralnet(medv ~ ., data = train_scaled, hidden = c(5, 3), linear.output = FALSE, stepmax = 1e+06)
  nn_pred <- predict(nn_model, test_scaled[, -classColumn])
  mses[i] <- mean((dif * (nn_pred - test_scaled[, classColumn]))^2)
}

plot(nn_model)

print(mses)
print(mean(mses))
print(sqrt(var(mses)))
```

Results Boston housing




```
 [1] 10.233588 10.128329 20.632011 31.248014  9.511252  9.487106 16.691137
 [8] 13.259778 21.159252 10.389862 22.701482 24.300544 12.448325 17.646360
[15] 10.965124 10.858534 15.872975 21.218057  7.524402 22.229145 11.683342
[22] 13.037122 22.236972 15.936926  7.215073 14.217332  8.488198 21.259411
[29] 11.298770  9.324497
[1] 15.10676
[1] 6.058824
```

NN regression in Python

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPRegressor
from sklearn import datasets
from sklearn.preprocessing import MinMaxScaler

repeats = 30
mses = []
r2s = []

bx, by = datasets.load_boston(return_X_y=True)

scaler = MinMaxScaler()

for i in range(0, repeats):
    #random generation of training and test, 75-25%
    bx_train, bx_test, by_train, by_test = train_test_split(bx, by, test_size = 0.25)

    fit_scalar = scaler.fit(bx_train)
    bx_train_scaled = fit_scalar.transform(bx_train)

    nnm = MLPRegressor(hidden_layer_sizes=(10, 10, 10), solver='adam', activation = 'relu', max_iter = 100000)
    nnm.fit(bx_train_scaled,by_train)

    bx_test_scaled = fit_scalar.transform(bx_test)
    by_pred = nnm.predict(bx_test_scaled)
    mses.append(mean_squared_error(by_test, by_pred))
    r2s.append(r2_score(by_test, by_pred))

print(mses)
print(r2s)
```

Boston results

[17.492755007264936, 30.234505625912554, 10.02046071680119, 19.037853523236794, 18.448766845564116, 20.87579946196899, 8.826236048369347, 24.001715622499553, 18.878682321430265, 12.0241620352213 1, 22.035370209214424, 9.329690404116555, 12.149204856579493, 23.005473435020107, 8.454154531507836, 14.288911607219333, 13.012196991202496, 10.710487549511683, 17.1441801066268 3, 14.955289395359628, 23.199785268773645, 11.473790827049333, 11.602084873366444, 11.009107901891278, 14.504382535062557, 21.252839627988767, 10.808259854302339, 18.340272996999875, 13.727357417027171, 14.585422802948548]

```
print("Mean MSE:", np.mean(mses))
print("Standard deviation:", np.std(mses))
print("R^2:", np.mean(r2s))
```

```
Mean MSE: 15.84764001333458
Standard deviation: 5.318173954845035
R^2: 0.809998718540234
```

Catastrophic forgetting

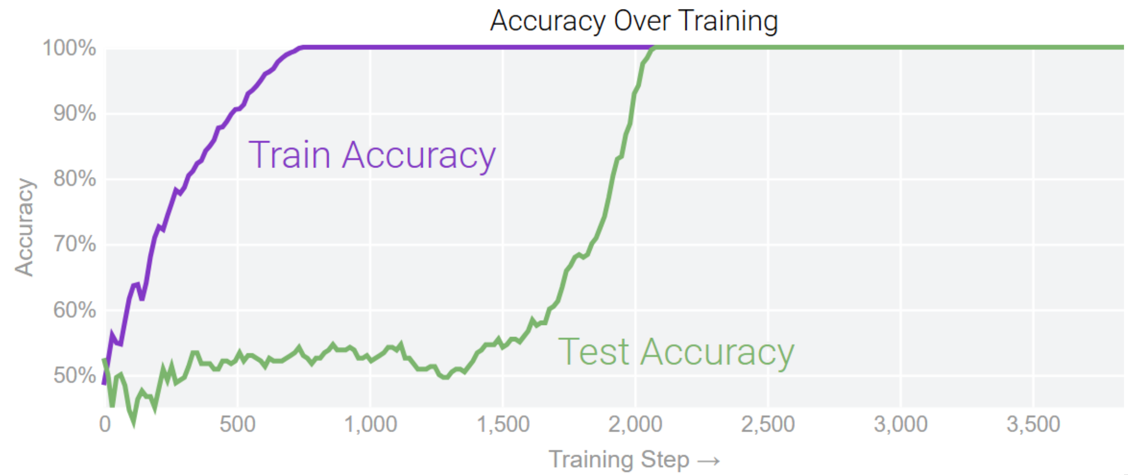
- When NN forget previously learned example after new data is learnt or when they are fine-tuned on more specific tasks.
  - Because of substantial weight re-adaptation to new examples
- Affecting large models more than smaller models
- Solutions:
  - Regularization: adding penalties for major weight adjustment
  - Progressive NN architecture (adding network parts for new tasks)
  - Ensembles
  - Rehearsal techniques (exposing the model to random old data)
  - Memory-augmented NN (NN + external memory storage)

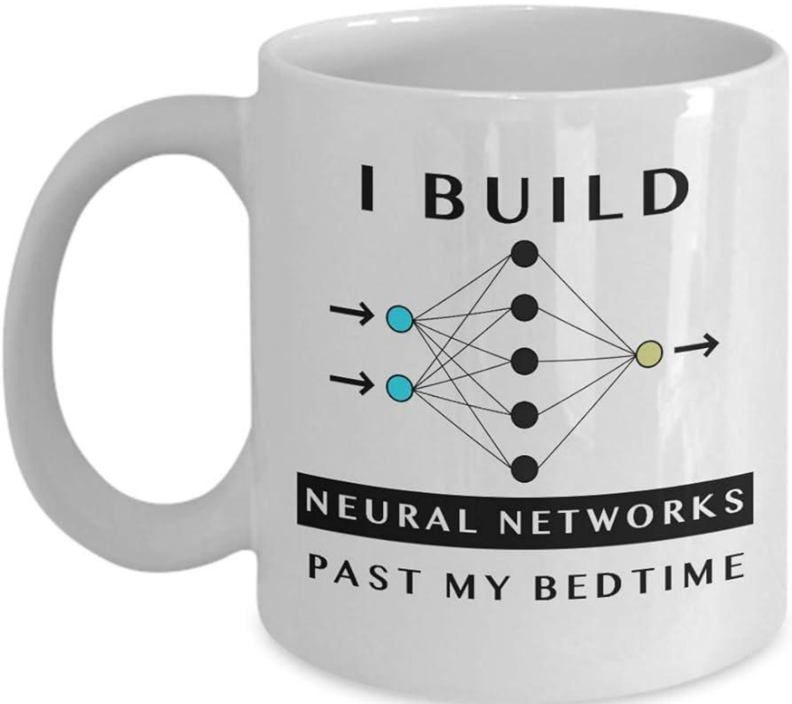
Curriculum learning

- At the data level:
  - Introduces training data from simple to complex examples
  - Question of deciding the order of the data
    - Given by expert (text, images)
    - According to model performance in time
    - Ensuring diversity
- At the model level
  - Augment the complexity of the architecture
    - Progressive model
    - Teacher-student
      - The student learns the task
      - The teacher determines the optimal learning parameters

Grokking

- Delayed generalization
- Do models memorize or generalize?
- Models initially overfitting for a long time, but then achieving high performance on validation data
- Regularization (e.g. weight decay – L2 type) and learning rate seem to play a big role
- https://pair.withgoogle.com/explorables/grokking/




I BUILD
NEURAL NETWORKS
PAST MY BEDTIME