

# MY474: Applied Machine Learning for Social Science

## Lecture 10: Introduction to Neural Networks

Friedrich Geiecke

28 March 2022

## Weeks 9 - 11

1. Bagging, random forests, and boosting
2. Unsupervised learning
3. **Neural networks**

# Outline

1. Introduction
2. Fundamental architectures
3. Training
4. In practise
5. Variants
6. Guided coding

## 1. Introduction

# This lecture

- ▶ Tries to give a high level overview of many topics around neural networks and deep learning
- ▶ Due to the short amount of time, concepts can only be mentioned briefly
- ▶ In addition to giving an overview, the approach hopefully allows students to identify topics that can be of interest for further study

## Some history

- ▶ McCulloch and Pitts (1943) created a computational model for neural networks
- ▶ Research on the topic continued throughout the 20th century, however, at times as a niche field
- ▶ In the early 21st century the models began to outperform others at a large scale
- ▶ For example AlexNet, a convolutional neural network, won the ImageNet (an image classification) competition in 2012 by a very high margin (for many researchers at the time that was unexpected)
- ▶ The paper has been cited almost 80,000 times since then

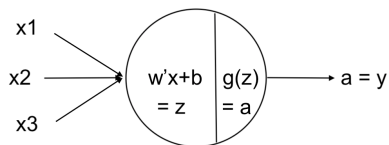
# Why have neural networks become so important in recent years

- ▶ Data: Much more (labeled) data available
- ▶ Hardware: Faster computation; extensive use of GPUs
- ▶ Software: Improved architectures, libraries, optimisation algorithms, etc.

## 2. Fundamental architectures



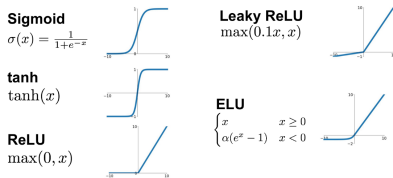
# Single layer perceptron



- ▶  $z = x'w + b$  with  $w = (w_1, w_2, w_3)$
- ▶  $a = g(z)$
- ▶  $g(z)$  is called an “activation function”
- ▶ Neurons (nodes) in these commonly shown figures actually depict both  $z$  and  $a$

# Activation functions

- ▶ Activation functions introduce non-linearities
- ▶ Stacking only linear layers would just create a linear model
- ▶ Today, the benchmark activation/nonlinearity is often ReLU (rectified linear unit) which has advantages for learning due to its constant positive gradient for positive inputs
- ▶ Sigmoid had been used earlier, but often suffers from close to zero gradients and can imply very slow learning



Some common activation functions. Image from Jadon (2018)

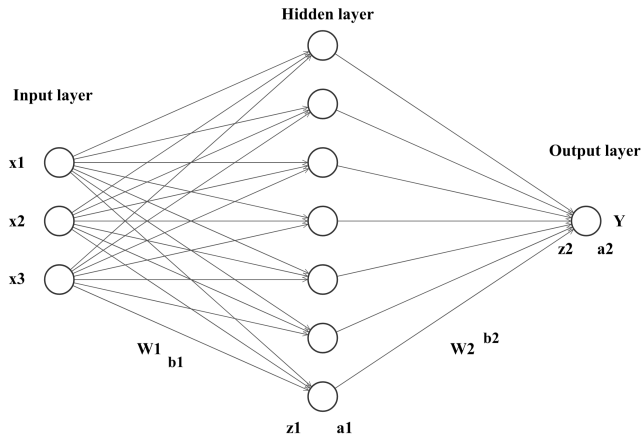
# Limitations

- ▶ A single layer perceptron can already perform well in some classification tasks
- ▶ Yet, it can only correctly classify observations that are linearly separable
- ▶ Good website for building intuition  
<https://playground.tensorflow.org/>

# Multi layer perceptron

- ▶ We generally count hidden and output layers, so a network with one hidden and one output layer is already a multi layer perceptron (MLP)
- ▶ Cells are called neurons, layers with many neurons are called *wide* and with few neurons are called *narrow*
- ▶ Network with few hidden layers are called *shallow*
- ▶ Networks with many hidden layers are called *deep*

# MLP with one hidden layer

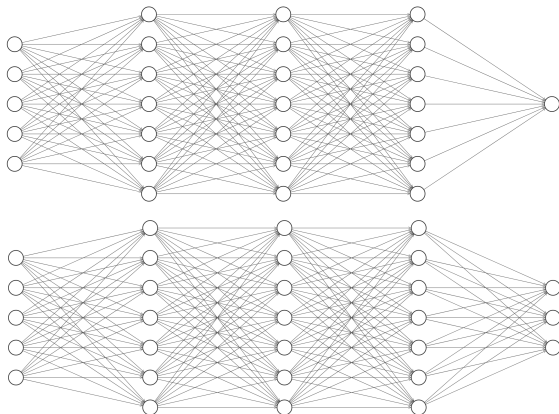


Drawn with <http://alexlenail.me/NN-SVG/index.html> and annotated

# MLP with one hidden layer

- ▶ Input layer:  $\underbrace{x}_{3 \times 1}$
- ▶ Hidden layer z:  $\underbrace{W^{(1)}}_{7 \times 3} \underbrace{x}_{3 \times 1} + \underbrace{b^{(1)}}_{7 \times 1} = \underbrace{z^{(1)}}_{7 \times 1}$
- ▶ Hidden layer activation:  $\underbrace{a^{(1)}}_{7 \times 1} = \underbrace{g(z^{(1)})}_{7 \times 1}$  (applied element-wise)
- ▶ Output layer z:  $\underbrace{W^{(2)}}_{1 \times 7} \underbrace{a^{(1)}}_{7 \times 1} + \underbrace{b^{(2)}}_{1 \times 1} = \underbrace{z^{(2)}}_{1 \times 1}$
- ▶ Output layer activation:  $\underbrace{g(z^{(2)})}_{1 \times 1} = \underbrace{a^{(2)}}_{1 \times 1} = \underbrace{\hat{y}}_{1 \times 1}$
- ▶ Expressed in one piece:  
$$x \xrightarrow{\underbrace{W^{(1)}, b^{(1)}}} z^{(1)} \xrightarrow{\underbrace{g(\cdot)}} a^{(1)} \xrightarrow{\underbrace{W^{(2)}, b^{(2)}}} z^{(2)} \xrightarrow{\underbrace{g(\cdot)}} a^{(2)} = \hat{y}$$

## MLP with three hidden layers



Drawn with <http://alexlenail.me/NN-SVG/index.html>

# Regression and classification

- ▶ Neural networks are used for both regression and classification
- ▶ Most commonly for regression, the activation function of the last layer is linear, i.e.  $g(z) = z$
- ▶ For two class classification, it is usually sigmoid, i.e.  $g(z) = \frac{1}{1+e^{-z}}$
- ▶ For multi class classification its generalisation is used, the softmax:  $g(z_i) = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}$



# Universal approximation theorem

- ▶ In fact, an important theorem states that, under some regularity conditions, already a single hidden layer neural network can approximate continuous functions, that map from  $[0, 1]^K$  to the real number line, arbitrarily closely (see Cybenko, 1989, and Hornik, 1991)
- ▶ Note that this said nothing about the how we can practically find the optimal weights of such a network, just that it exists

# Deep vs shallow networks

- ▶ Why do researchers then generally prefer deep networks over very wide one hidden layer networks?
- ▶ Because they have been found to outperform shallow networks empirically
- ▶ Montúfar et al., 2014 also provide theoretical evidence why the ability to separate observations in features space can react more quickly to adding depth than width

### 3. Training

# Loss

- ▶ To train the neural network, we first need to define a loss function
- ▶ Typical loss for observation  $i$  in regression: Squared error  
$$L(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2$$
- ▶ Typical loss for observation  $i$  classification: Cross entropy  
$$L(y_i, \hat{y}_i) = - \sum_{k=1}^K y_{i,k} \log(\hat{y}_{i,k})$$

# Cost

- ▶ One convention in the field is to name the loss aggregated over training observations the *cost function*
- ▶ Summarise all weights,  $W^{(1)}, W^{(2)}, \dots$  and biases  $b^{(1)}, b^{(2)}, \dots$  in one vector  $\theta$  for convenience
- ▶  $J(\theta) = \frac{1}{n} \sum_{i=1}^n L(y_i, f(x_i, \theta))$

# Backpropagation

- ▶ Next we need to derive the gradient vector of the cost function
- ▶ This is done with the backpropagation algorithm
- ▶ Recall as an example the structure of a shallow neural network with one hidden layer:

$$\text{▶ } x \xrightarrow{\underbrace{\hspace{1.5cm}}_{W^{(1)}, b^{(1)}}} z^{(1)} \xrightarrow{\underbrace{\hspace{1.5cm}}_{g(\cdot)}} a^{(1)} \xrightarrow{\underbrace{\hspace{1.5cm}}_{W^{(2)}, b^{(2)}}} z^{(2)} \xrightarrow{\underbrace{\hspace{1.5cm}}_{g(\cdot)}} a^{(2)} = \hat{y}$$

- ▶ Furthermore, recall the chain rule: If  $f(g(x))$ , then  $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} \frac{\partial g}{\partial x}$

# Backpropagation

- Carried over from the last slide. Forward pass:

$$x \xrightarrow{W^{(1)}, b^{(1)}} z^{(1)} \xrightarrow{g(\cdot)} a^{(1)} \xrightarrow{W^{(2)}, b^{(2)}} z^{(2)} \xrightarrow{g(\cdot)} a^{(2)} = \hat{y} \quad \text{Backward:}$$

$$x \xleftarrow{W^{(1)}, b^{(1)}} z^{(1)} \xleftarrow{g(\cdot)} a^{(1)} \xleftarrow{W^{(2)}, b^{(2)}} z^{(2)} \xleftarrow{g(\cdot)} a^{(2)} = \hat{y}$$

- Thus, we get:

$$\frac{\partial J(\theta)}{\partial W^{(2)}} = \frac{\partial J(\theta)}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial W^{(2)}}$$

$$\frac{\partial J(\theta)}{\partial b^{(2)}} = \frac{\partial J(\theta)}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial b^{(2)}}$$

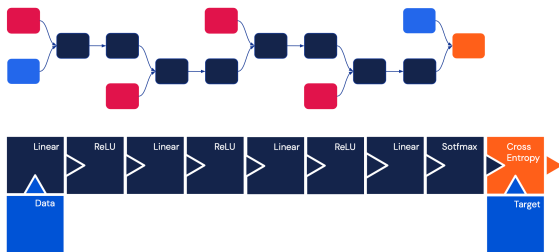
$$\frac{\partial J(\theta)}{\partial W^{(1)}} = \frac{\partial J(\theta)}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial W^{(1)}}$$

$$\frac{\partial J(\theta)}{\partial b^{(1)}} = \frac{\partial J(\theta)}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial b^{(1)}}$$

- Combine to get  $\frac{\partial J(\theta)}{\partial \theta}$  or  $\nabla_{\theta} J(\theta)$

# Computation graphs

- ▶ Modern libraries such as Tensorflow depict networks as computation graphs
- ▶ The gradients they compute flow through the individual elements of that graph
- ▶ This modular structure allows to easily process gradients in much more complex architecture than the one shown below



From Czarnecki (2020)



# Gradient descent

- ▶ Computing the gradient for the entire dataset (full batch gradient descent) is usually computationally infeasible
- ▶ Instead, we approximate the gradient of the full training data with either the gradient of a single observations (stochastic gradient descent) or with the gradient over a small batch of data (mini batch gradient descent)
- ▶ Importantly, note that as loss functions for neural networks are mostly non convex, these algorithms converge to local optima, but not global ones

# Stochastic gradient descent (SGD)

- ▶ Randomly initialise weights and choose learning rate  $\alpha$
- ▶ Repeat the following for E epochs or until approximate convergence:
  - ▶ Shuffle all observations in the training dataset
  - ▶ For observation  $i = 1, 2, \dots, n$ :
    - ▶ Update  $\theta \leftarrow \theta - \alpha \nabla_{\theta} L(y_i, f(x_i, \theta))$

# Mini batch gradient descent

- ▶ Randomly initialise weights and choose learning rate  $\alpha$
- ▶ Repeat the following for E epochs or until approximate convergence:
  - ▶ Shuffle all observations in the training dataset
  - ▶ For each batch with  $B \ll n$  observations:
    - ▶ Update  $\theta \leftarrow \theta - \alpha \nabla_{\theta} \frac{1}{B} \sum_{i=1}^B L(y_i, f(x_i, \theta))$

# Most popular optimisers

- ▶ SGD is too noisy, mini batch optimisation is the most common choice
- ▶ There are some particularly popular optimisers which have proven robust and applicable to a wide range of tasks and models, e.g. Adam or RMSprop which include moving average based momentum of gradients
- ▶ A good starting point is usually Adam

## 4. In practise

# Regularisation

- ▶ With their very high numbers of parameters, neural networks can memorise large amounts of data and relatively easy over-fit
- ▶ Possible approaches to counter over-fitting are:
- ▶ Add norms of weights to objective function
- ▶ Dropout: Randomly set a fraction of  $p$  neurons in a layer to zero during training and thereby force a wider set of neurons to learn patterns
- ▶ Early stopping: Stop the gradient descent once the loss on some validation set increases
- ▶ Add noise to activities during training

## Further approaches that can be helpful

- ▶ Batch normalisation: Batch specific normalisation of inputs or activations; can speed up training and make model more robust to changes in learning rates
- ▶ Often helpful to first fit the model to a very small sample of the data and see whether it can perfectly fit it. If not, then there are probably some issues
- ▶ Weights are randomly initialised, but initialisation matters
- ▶ Heuristics such as e.g. the “He” or “Xavier” initialisations

# Hyper-parameter search

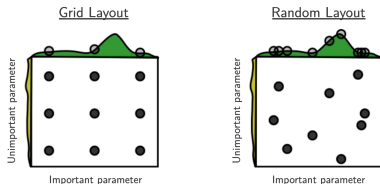
- ▶ In addition to their weights/parameters, neural networks have many so called hyper-parameters
- ▶ The amount of hidden layers, neurons per hidden layer, the learning rate, dropout percentages, degree of regularisation with norms, etc.
- ▶ Imagine you have 10 hyper-parameters and for each of them 10 possible values in mind, then you would have to search through a grid with  $10^{10}$  possible combinations and train a model for each of them (curse of dimensionality)



# Random hyper-parameter search

► Two takeaways:

- 1) Because of the curse of dimensionality, a good first approach is to search randomly and then narrow down on parts of the space that seem promising
- 2) Evenly spaced out grids with pre-specified values are not usually a good idea (see figure below from Bergstra and Bengio, 2012)



Bergstra and Bengio (2012)

## 5. Variants

# Convolutional neural networks

- ▶ Convolutional neural networks are mainly used for computer vision
- ▶ They learn filters and combine operations such as convolutions and pooling

## Convolutions with filters

1	1	1	0	0
0 <sub>x1</sub>	1 <sub>x0</sub>	1 <sub>x1</sub>	1	0
0 <sub>x0</sub>	0 <sub>x1</sub>	1 <sub>x0</sub>	1	1
0 <sub>x1</sub>	0 <sub>x0</sub>	1 <sub>x1</sub>	1	0
0	1	1	0	0

Image

4	3	4
2		

Convolved  
Feature

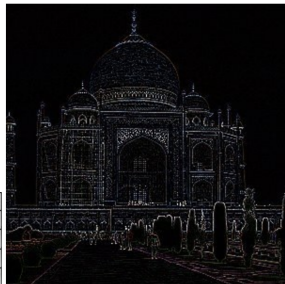
From: Stanford UFLDL wiki

# Filter examples

0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0



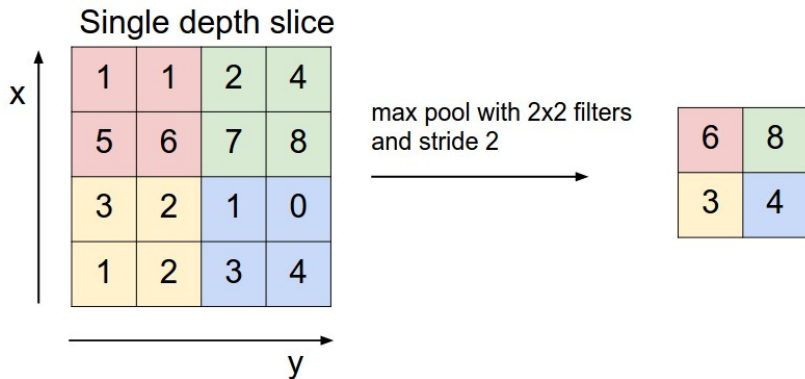
	0	1	0	
	1	-4	1	
	0	1	0	



From: Gimp documentation

- ▶ The first filter makes the image blurry, the second one detects edges
- ▶ CNNs learn these filters through training, minimising e.g. the classification loss

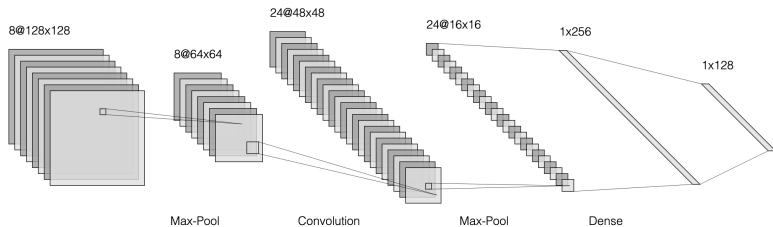
# Max pooling



From: <https://cs231n.github.io/convolutional-networks/>

- ▶ Another option is e.g. average pooling

# Exemplary CNN architecture



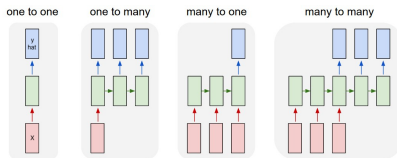
Drawn with <http://alexlenail.me/NN-SVG/index.html>

# Recurrent neural networks

- ▶ Recurrent neural networks (RNNs) can process sequences of inputs and predict sequences of outputs
- ▶ RNNs are e.g. used in machine translation, sentence completion, sentiment analysis, image captioning, etc.
- ▶ Common types of RNNs such as Long Short-Term Memory (LSTM) or Gated Recurrent Unit (GRU) are based on cells which improve the model's ability to remember long term dependencies



# Recurrent neural networks



From Andrej Karpathy's blog; slightly edited

- ▶ Arrows are functions/transformations, rectangles are vectors, green rectangles hold states
- ▶ One to one: Standard feed forward neural network/MLP
- ▶ One to many: RNN that e.g. takes an image as input and then outputs a sentence describing it
- ▶ Many to one: RNN that e.g. inputs a sequence of words and outputs a sentiment label
- ▶ Many to many: RNN that e.g. inputs a sentence in one language and outputs it in another language

# Generative adversarial networks (GANs)

- ▶ Consist of a pair of neural networks, a generator and a discriminator
- ▶ The generator generates samples (e.g. images, music, artwork etc.) and the discriminator has to distinguish these fake samples from a set of true samples
- ▶ Through training, the samples produced by the generator can become very realistic
- ▶ <https://thispersondoesnotexist.com/> by Wang (2019)

## 6. Guided coding

# Neural network libraries in R

- ▶ More libraries in Python for neural networks, but increasing support also for R:
- ▶ 1. **TensorFlow** and **Keras**:  
<https://tensorflow.rstudio.com/tutorials/> and  
<https://keras.rstudio.com/>
- ▶ 2. **PyTorch**: <https://torch.mlverse.org/start/>
- ▶ Excellent repo with many baseline models in Keras for R (also used in coding examples): <https://github.com/rstudio/keras/tree/master/vignettes/examples>

## R files

- ▶ 01-mlp.Rmd
- ▶ 02-cnn.Rmd

# References

- ▶ Bergstra, James and Yoshua Bengio, Random Search for Hyper-Parameter Optimization, Journal of Machine Learning Research, 2012
- ▶ Cybenko., G., Approximations by superpositions of sigmoidal functions, Mathematics of Control, Signals, and Systems, 1989
- ▶ Czarnecki, Wojciech , Neural Network Foundations, Lecture, <https://deepmind.com/learning-resources/deep-learning-lecture-series-2020>, 2020
- ▶ Hornik, Kurt, Approximation Capabilities of Multilayer Feedforward Networks, Neural Networks, 1991
- ▶ McCulloch, Warren S. and Walter Pitts, A logical calculus of the ideas immanent in nervous activity, Bulletin of Mathematical Biophysics, 1943
- ▶ Montúfar, Guido and Razvan Pascanu, Kyunghyun Cho, Yoshua Bengio. On the Number of Linear Regions of Deep Neural Networks, Arxiv, 2014
- ▶ Osindero, Simon, Neural Network Foundations, Lecture, <https://deepmind.com/learning-resources/deep-learning-lectures-series-2018>, 2018