MY474: Applied Machine Learning for Social Science

Lecture 9: Introduction to Neural Networks

Friedrich Geiecke

22 March 2023

Part II: Weeks 9 - 11

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

Outline

1. Introduction

This lecture

- Tries to give a high level overview of neural networks and deep learning
- Can only discuss concepts relatively briefly due to the short amount of time available
- Tries to introduce terminology on the way that is used in the field
- Hopefully allows interested students to identify many topics and resources for further study

Some history

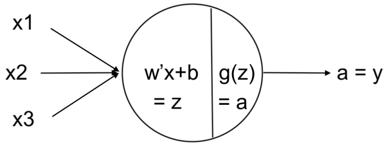
- McCulloch and Pitts (1943) created a computational model for neural networks
- Research on the topic continued throughout the 20th century, however, at times as a niche field
- In the early 21st century the models began to outperform others at a large scale
- For example AlexNet, a convolutional neural 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 130,000 times as of March 2023

Why have neural networks become so important in recent years

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

2. Fundamental architectures

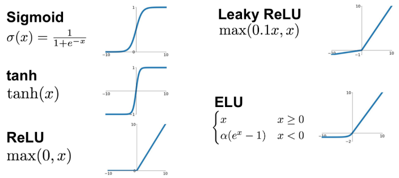
Single layer perceptron with general activation



- **Single layer perceptron (SLP)** here: A (**feedforward**) neural network with no hidden layer and an output layer with a single neuron and activation
- $z = x'w + b$ with 3-dimensional vector $w = (w_1, w_2, w_3)$ of **weights** and **bias** b
- $a = g(z)$ where $g(z)$ is called **activation function**
- Circles in figures commonly depict both z and a values
- Original perceptron: Linear activation $g(z) = z$ and output 1 if $z \geq 0$ (0 otherwise)
- Note: With **sigmoid activation** $g(z) = \frac{1}{1+e^{-z}}$, this is just logistic regression!

Activation functions

- Activation functions introduce non-linearities
- Stacking just linear layers could only create a linear model
- Sigmoid had been used earlier, but often suffers from close to zero gradients and can imply very slow learning
- Today, one common activation/nonlinearity is **ReLU (rectified linear unit)** which can have some advantages for learning e.g. due to its constant positive gradient for positive inputs. More options:



Some common activation functions. Image from Jadon (2018)

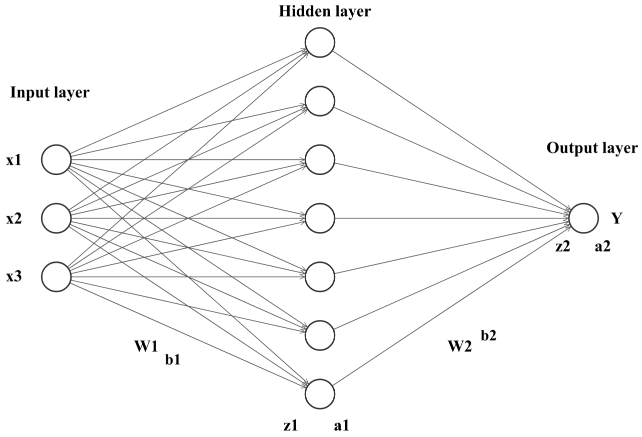
Limitations

- A network with a single output layer and single output neuron can already perform well in some binary classification tasks
- Yet, it can only correctly classify observations that are linearly separable. Why?
- Good website for building intuition https://playground.tensorflow.org/

Multi layer perceptron and some terminology

- We generally count hidden and output layers, so network with one hidden and one output later is already a multi layer perceptron (MLP). **MLPs** are cases of so called **feedforward neural networks** (where information flows into one direction), but you will often see these two terms used interchangeably
- Cells are called **neurons/unit/node**, 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

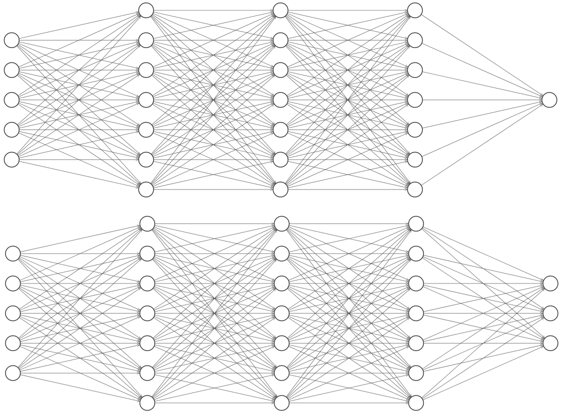


Drawn with http://alexlenail.me/NN-SVG/index.html and annotated

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

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 says 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 often 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**, but it is also very often simply called **loss function**
- Summarise all weights, $W^{(1)}, W^{(2)}, \dots$ and biases $b^{(1)}, b^{(2)}, \dots$ in one vector θ for convenience
- $J(\theta) = \frac{1}{n} \sum_{i=1}^{n} L(y_i, f(x_i, \theta))$

Forward pass

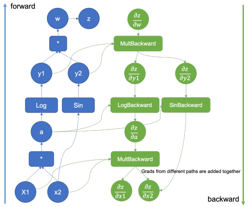
- Recall our example of a neural network with one hidden layer:
- $x \underbrace{\rightarrow}_{W^{(1)},b^{(1)}} z^{(1)} \underbrace{\rightarrow}_{g(\cdot)} a^{(1)} \underbrace{\rightarrow}_{W^{(2)},b^{(2)}} z^{(2)} \underbrace{\rightarrow}_{g(\cdot)} a^{(2)} = \hat{y}$
- This computation of output values from some input samples is called **forward pass**
- Furthermore, recall the chain rule: If $f(g(x))$, then $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} \frac{\partial g}{\partial x}$
- Next we will derive the gradient vector of the cost function to use it in gradient descent

Backpropagation

- Forward pass / network structure carried over from the last slide: $x \underbrace{\rightarrow}_{W^{(1)},b^{(1)}} z^{(1)} \underbrace{\rightarrow}_{g(\cdot)} a^{(1)} \underbrace{\rightarrow}_{W^{(2)},b^{(2)}} z^{(2)} \underbrace{\rightarrow}_{g(\cdot)} a^{(2)} = \hat{y}$
- Thus, applying the chain rule we get (assume scalars for simplicity here):
- $\frac{\partial J(\theta)}{\partial W^{(2)}} = \frac{\partial J(\theta)}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial W^{(2)}}$
- $\frac{\partial J(\theta)}{\partial b^{(2)}} = \frac{\partial J(\theta)}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial b^{(2)}}$
- $\frac{\partial J(\theta)}{\partial W^{(1)}} = \frac{\partial J(\theta)}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial W^{(1)}}$
- $\frac{\partial J(\theta)}{\partial b^{(1)}} = \frac{\partial J(\theta)}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial b^{(1)}}$
- Combine/stack to get $\frac{\partial J(\theta)}{\partial \theta}$ or also often denoted $\nabla_\theta J(\theta)$

Computation graphs

- Modern libraries such as PyTorch and 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 also in much more complex architecture than the one shown below


forward
w
z
*
y1
y2
Log
Sin
a
*
X1
x2
$\frac{\partial z}{\partial w}$
MultBackward
$\frac{\partial z}{\partial y1}$
$\frac{\partial z}{\partial y2}$
LogBackward
SinBackward
$\frac{\partial z}{\partial a}$
MultBackward
Grads from different paths are added together
$\frac{\partial z}{\partial x1}$
$\frac{\partial z}{\partial x2}$
backward

Example from PyTorch blog

Gradient descent

- Computing the gradient for the entire dataset (full batch gradient descent) is usually computationally too costly
- 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 cost/loss functions for neural networks are mostly non convex, these algorithms may converge to local optima, but typically not global ones
- There are also likely many cost/loss saddle points in high dimensional parameter spaces. Why?

Stochastic gradient descent (SGD)

- Randomly initialise weights and choose learning rate α
- Repeate the following for E epochs or until approximate convergence:
  - Shuffle all observations in the training dataset
  - For observation i = 1,2, . . . ,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 α
- Repeat the following for E epochs or until approximate convergence:
  - Shuffle all observations in the training dataset
  - For each batch with $B << n$ 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
- Include moving average based momentum of gradients that is not part of plain vanilla gradient descent
- A good starting point is usually Adam

4. In practice

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 L1 or L2 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 warying sets of neurons to learn patterns. In essence trains an ensemble
  - Early stopping: Stop the gradient descent once the loss on some validation set increases
  - Add noise to activities during training

Many further approaches to look into when training networks

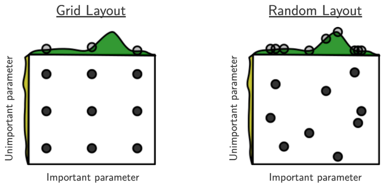
- Batch normalisation: Normalise over observations in a batch
- Layer normalisation: Normalise over activations in a layer for each observation
- Weights are initialised randomly, but the approach matters. For example, have a look at heuristics such as “He” or “Xavier” initialisations
- Residual connections/skip connections/shortcut connections: Skip some layers and directly connect output of one layer to another layer allowing some information to flow more easily
- In general, 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 might be some issues

Hyper-parameter search

- In addition to their weights/parameters, neural networks have many so called hyper-parameters
- The amount of hidden layers, activations, 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 trough a grid with 10ˆ10 possible combinations and train a model for each of them (this quickly becomes infeasible and is refered to as the **curse of dimensionalty**)

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)

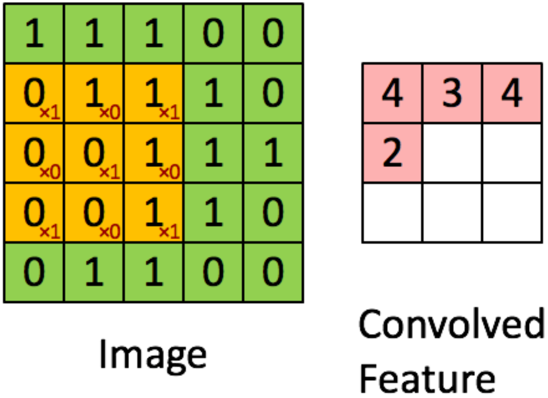
5. Variants

Convolutional neural networks

Convolutional neural networks

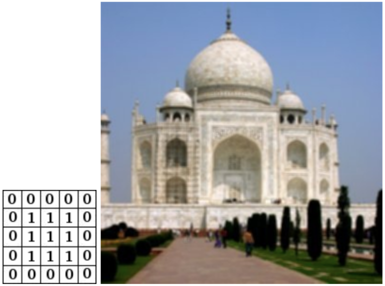
- Convolutional neural networks are often used for computer vision, but can also be used for other tasks
- They learn filters and combine operations such as convolutions and pooling

Convolutions with filters

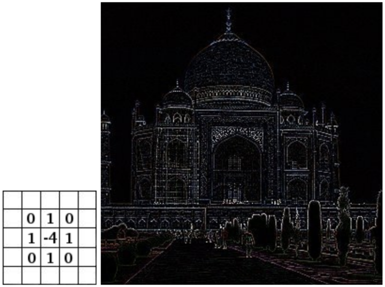


From: Stanford UFLDL wiki

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

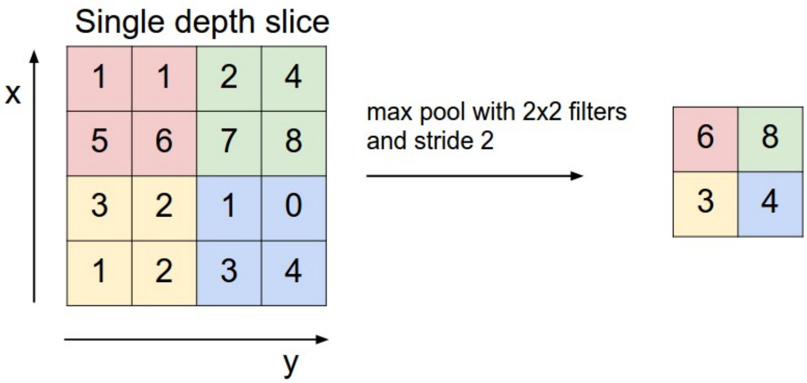


	0	1	0	
	1	-4	1	
	0	1	0	

From: Gimp documentation

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

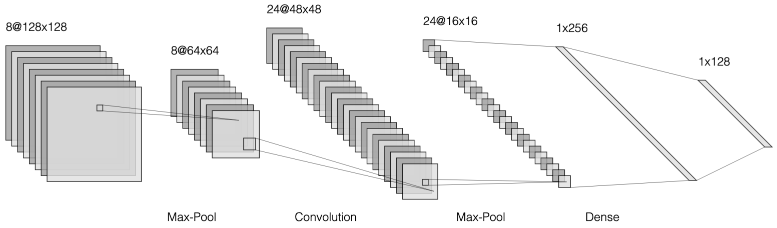
Max pooling



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

- Another option is e.g. average pooling

Exemplary CNN architecture

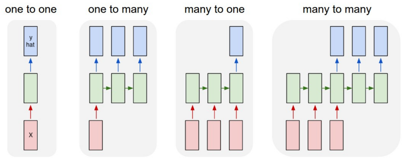


# Recurrent neural networks

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 those with Gated Recurrent Units (GRUs) are based on cells which improve the model's ability to remember long term dependencies
- Potential challenges: Vanishing/exploding gradients and limited parallelisability of sequential computation
- More recently, transformer neural networks have taken over a lot of these tasks (discussed in a bit)

Recurrent neural networks



From Andrej Karpathy's 2015 **blog post**; slightly edited

- Arrows are functions/transformations, rectangles are vectors, green rectangles hold states
- One to one: Standard feedforward 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)

General setup

- 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

Transformers

Transformers

- A key innovation in machine learning in the last years has been the transformer architecture
- It is also the basis of much of the recent public and scientific discussions surrounding AI
- This section provides a brief non-technical overview and also resources for further study

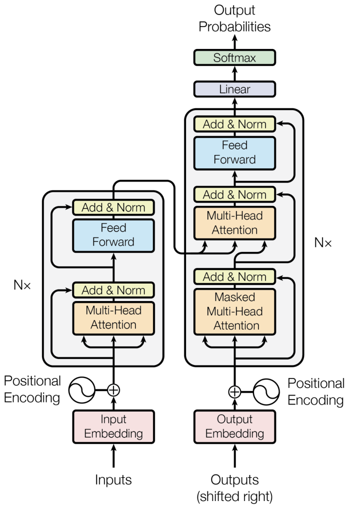
Some key concepts (1/2)

- **Context**: Text up to a maximum length in which dependencies between words can be considered. Let's use "Machine learning is interesting" as an example
- **Tokenisation**: "Machine learning is interesting" might become [4, 42, 17, 100] with each integer referring to a word in an overall vocabulary (assume for simplicity here that tokenisation is done at the word level). Hence, these models actually just input sequences of integers and also output integers and these have to be translated back into text
- **Embeddings**: Each token is represented by a numerical vector of some chosen dimension - a so called embedding (in transformers there are actually also additional vectors representing each position in the context)

Some key concepts (2/2)

- **Attention** (much simplified): Weights on vectors in a context that encapsulate how relevant each is for each other's meaning. For the text "not a good day", an updated vector for "not" should depend more on/communicate with/pay attention to/have a higher weight on the vector of "good" than on that of "day". A set of attention weights sums up to one
- **Feedforward neural network**: Just the MLPs discussed before
- Attention weights and other network parameters are learned through training on very large amounts of text and with typical variants of gradient descent
- For further details (normalisation etc.), see resources at the end of this section

General architecture



"Attention Is All You Need" by Vaswani et al (2017)

Notes

- Very influential paper; cited almost 70,000 times as of March 2023
- Repeated applications of attention weighting of vectors and transformations through feedforward neural networks
- Original application was in machine translation with both encoder and decoder blocks
- So an input “Machine learning is interesting” might output tokens that represent “Maschinelles Lernen ist interessant”
- Today encoders and decoders are often used separately

Encoders (1/2)

- Common examples of encoder transformers that you might see are models like BERT (2018)
- Each token of “Machine learning is interesting” would typically first be parsed into the encoder as an uncontextual embedding
- The output of the encoder then still returns four embeddings in this example, however, these are now contextual embeddings, i.e. their values can depend on other words in the text
- Crucially, attention in encoders can be backward and forward looking, i.e. attention weights relative to tokens before and after a given token can all be non-zero
- So the output value of the embedding for “learning” could depend on “machine” as well as “interesting”

Encoders (2/2)

- The contextual embeddings outputted by encoders can then be used as inputs in many downstream tasks, e.g. classification models
- Often, a decoder pretrained with very large amounts of data and compute is made available
- Afterwards, a much smaller neural network could be trained with some additional labelled data, e.g. using BERT embeddings as text inputs and predicting a label such as sentiment

Decoders (1/2)

- Many current large language models (LLMs) are based on decoders transformers e.g. GPT (generative pre-trained transformer)
- These decoders just predict next tokens in a sequence
- Yet, they are incredibly flexible functions making these predictions: GPT-3 contained 175 billion parameters and was trained on a good chunk of the text of "the internet"
- In detail, the models predict a probability for each token from the vocabulary as it could follow as the next one after an input text:
- Pr(next_token | sequence/context of input tokens) for each possible "next_token"

Decoders (2/2)

- Sampling the next token from this distribution and then iterating forward generates the text
- This text can be different every time because of the randomness in sampling
- Crucially, attention in decoders can only be backward looking, i.e. attention weights on tokens after the end of the sequence have to be zero as otherwise the model would already know information about the tokens to predict

ChatGPT

- The underlying basis of models like ChatGPT is a decoder transformer predicting the next token
- The input prompt given by a user is the context which is completed by the transformer
- Issue: This might well mean that the transformer follows up a question with another question, writes very wordy output etc.
- Additional work therefore went into aligning the model's generated output better with the input queries
- This used human examples of input and outputs, ranking of potential outputs by humans, and variants of reinforcement learning

Resources for further study

- An excellent free lecture series on neural network was recently published by Andrej Karpathy which you can find **here**
- It builds many key parts from first principles, such as automatic differentiation, computation graphs, or backpropagation, and then builds the transformer architecture behind GPT
- Requires knowledge of Python up to classes and some knowledge in linear algebra and calculus
- Many pre-trained transformers which can be used or adjusted for downstream tasks e.g. available via websites such as https://huggingface.co/

6. Guided coding

Neural network libraries in R

- For neural networks, most libraries are available in Python, but key functionalities can also be used through R:
- 1. **TensorFlow** and **Keras**: https://tensorflow.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
- 03-nn-from-scratch.Rmd

References (1/2)

- Bergstra, James and Yoshua Bengio, Random Search for Hyper-Parameter Optimization, Journal of Machine Learning Research, 2012
- Brown, Tom, et al., Language models are few-shot learners, Advances in neural information processing systems 33, 2020
- 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
- Devlin, Jacob, et al., Bert: Pre-training of deep bidirectional transformers for language understanding, Arxiv, 2018
- Hornik, Kurt, Approximation Capabilities of Multilayer Feedforward Networks, Neural Networks, 1991

References (2/2)

- 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
- Vaswani, Ashish, et al., Attention is all you need, Advances in neural information processing systems 30, 2017