Neural Networks 1 - Multilayer neural networks

18NES1 - Lecture 8, Summer semester 2024/25

Zuzana Petříčková

April 8th, 2025

What We Covered Last Week

Multilayer Neural Network (MLP) and the Backpropagation Algorithm

1. Overview of Python libraries for deep learning, including code examples and instructions of local installation
2. Interactive visualizations with TensorFlow Playground
3. Brief analysis of the multi-layer neural network model with the backpropagation algorithm
4. Step-by-step example using Keras on a sample task (binary classification). Setting hyperparameters and understanding their impact on the training process. Using TensorBoard.

This Week

1. Recall and finish the Keras example
2. More notes on hyperparameter setting
    - Learning algorithms for multilayer neural networks
3. Examples of various types of tasks:
    - Binary classification (already covered), multiclass classification, regression, time series prediction
    - Specifics of each task and data preprocessing
4. Generalization in MLPs and techniques for preventing overfitting (with demonstrations and examples)

Example from the last week

keras_simple_example.ipynb

- A detailed example in Keras – step-by-step learning procedure (on a binary classification task)
- Data preprocessing and analysis. Model creation and hyperparameter tuning. Training progress. Visualization. Evaluation.
- Hyperparameter tuning.
- Visualization using TensorBoard.

We will switch between the slides and the example notebook during the session.

Key Hyperparameters of an MLP Model

Architecture

- **Model size:** Number of hidden layers and number of neurons per layer
- **Activation functions in each layer:** relu, sigmoid, tanh, softmax, ...

Other key hyperparameters

- **Loss function:** MSE, binary crossentropy, ...
- **Evaluation metrics:** accuracy, MSE, precision, ...
- **Optimization algorithm:** SGD, Adam, RMSProp, ...
- **Learning rate**, and possibly other optimizer-specific parameters
- **Batch size**
- **Number of epochs**
- **Weight initialization:** Typically small random values
- **Regularization:** L2, Dropout, Early stopping, ...

Optimizers for Deep Neural Networks

- Based on gradient descent; often use adaptive and local learning rates
    - **SGD (Stochastic Gradient Descent)** – basic optimizer, uses mini-batches; stable
    - **Adam** – currently the most popular; adaptive learning rate; faster convergence
    - **RMSprop** – suitable for sequential and online data
    - AdaGrad, Adadelta, AdaMax, NAdam, FTRL, ...
- Each optimizer has additional hyperparameters (e.g., SGD: **learning_rate, momentum, nesterov**)
  In most cases, the default settings work well.

https://keras.io/api/optimizers/

TensorBoard

- A tool for visualizing training and evaluation of neural networks.
- Displays loss curves, metrics, model graph, weight distributions, and more.
- Enables real-time monitoring during training.
- Supports comparison of multiple model runs.

Usage in Keras:

```
from keras.callbacks import TensorBoard
log_dir = "logs/fit/" + ...
tensorboard_callback = TensorBoard(log_dir=log_dir,...)
model.fit(..., callbacks=[tensorboard_callback,...])
```

Run from terminal:

```
tensorboard --logdir=logs/fit
```

Documentation: tensorflow.org/tensorboard

Example – Optional Homework from the last week

keras_simple_example.ipynb

- Use this notebook to explore how different hyperparameter settings (architecture, learning rate, etc.) affect the training process and the final results.
- Suggestions for experiments are included directly in the notebook.
- Try working with TensorBoard (you can also run it locally on your own machine).
- Optionally, modify the code and try training an MLP on different datasets from the scikit-learn repository (e.g., iris, diabetes, wine).

MLP Model Analysis
Learning Speed and Approximation Capabilities

- Backpropagation is relatively slow.
- Poor hyperparameter choice can make it even slower.
- Nevertheless, it often outperforms many "fast algorithms", especially when:
  - The task has realistic complexity
  - The training set size exceeds a critical threshold

MLP Model Analysis
Learning Speed and Approximation Capabilities

How to speed up learning while maintaining good approximation

- Proper initialization of weights and biases
- Preprocessing and normalization of input data
- Proper learning rate selection
- Use of fast learning algorithms
- Simultaneous adaptation of weights, biases, and network architecture

Choosing a Suitable Learning Rate

- The learning rate α is a key parameter in training
- It controls how quickly the model learns

How to choose the learning rate?

- Too small – slow learning (tiny weight updates), risk of getting stuck in suboptimal local minima
- Too large – large jumps, risk of oscillations and skipping over minima of the loss function

What helps?

- Tuning the learning rate for a specific task
- Using momentum (momentum, nesterov)
- Adaptive learning rate methods (Adam, RMSprop,...)

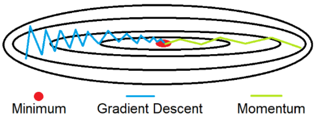
Backpropagation with Momentum

Problem: In narrow valleys of the error surface, following the gradient may cause large and frequent oscillations:

- The gradient fluctuates → slows convergence

Solution: Add a **momentum** term

- In addition to the current gradient, incorporate the previous weight updates
- Effect: Increased inertia helps maintain direction, reduces oscillations



https://www.andreaperlato.com/aipost/gradient-descent-with-momentum/

Backpropagation with Momentum

Updated weight update rule:

- Weight change from neuron i to neuron j at time $t+1$:

$$\begin{aligned}\Delta w_{ij}(t+1) &= -\alpha \frac{\partial E_t}{\partial w_{ij}} + \alpha_m \Delta w_{ij}(t) \\ &= -\alpha \frac{\partial E}{\partial w_{ij}} + \alpha_m (w_{ij}(t) - w_{ij}(t-1))\end{aligned}$$

- α ... learning rate
- α_m ... momentum coefficient

Backpropagation with Momentum

Momentum Term:

$$\Delta w_{ij}(t+1) = -\alpha \frac{\partial E_t}{\partial w_{ij}} + \alpha_m \Delta w_{ij}(t)$$

- Helps maintain direction in narrow valleys of the loss surface
- Reduces the risk of getting stuck in unstable states (local minima, saddle points)
- Speeds up convergence (longer stretches with consistent gradient direction)
- Too large α_m – excessive inertia, may overshoot the minimum

Nesterov Momentum

Improvement over classical momentum:

- "Looks ahead"– computes gradient at the anticipated next position
- Provides better stability and faster convergence in practice

Weight update steps:

1. Compute the **lookahead position**:

$$\tilde{w}_{ij}(t) = w_{ij}(t) + \alpha_m \cdot \Delta w_{ij}(t-1)$$

2. Compute gradient at this anticipated position:

$$\Delta w_{ij}(t) = -\alpha \cdot \left.\frac{\partial E}{\partial w_{ij}}\right|_{w=\tilde{w}(t)} + \alpha_m \cdot \Delta w_{ij}(t-1)$$

3. Update the weight:

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$$

Nesterov Momentum

Weight Update Summary:

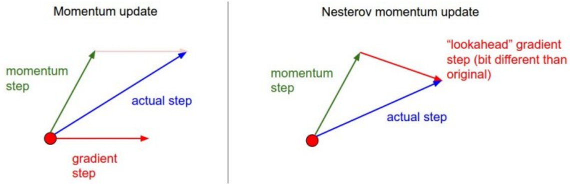
1. Compute the **lookahead position**:

$$\tilde{w}_{ij}(t) = w_{ij}(t) + \alpha_m \cdot \Delta w_{ij}(t-1)$$

2. Evaluate the gradient at that position:

$$\Delta w_{ij}(t) = -\alpha \cdot \left. \frac{\partial E}{\partial w_{ij}} \right|_{w=\tilde{w}(t)} + \alpha_m \cdot \Delta w_{ij}(t-1)$$

3. Update the weight: $w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$



https://github.com/cs231n/cs231n.github.io/blob/master/assets/nn3/nesterov.jpeg

Momentum vs. Nesterov Momentum – Summary

Classical Momentum:

- Reacts after the update
- Simpler to implement

Nesterov Momentum:

- Looks ahead – computes gradient before the update
- Better convergence in practice

In Keras: both variants available in SGD

optimizer = SGD(learning_rate=0.01, momentum=0.9, nesterov=True)

https://keras.io/api/optimizers/sgd/

Setting Learning Parameters

- **Learning rate** α
  - Small $\alpha \rightarrow$ slow learning, risk of getting stuck in local minima
  - Large $\alpha \rightarrow$ fast learning, but risk of oscillations and instability
- **Momentum** α_m
  - Helps overcome flat areas, stabilizes learning in steep regions
  - Too large $\rightarrow$ may overshoot the minimum
  - Typical values: $0.8 \leq \alpha_m \leq 0.95$
- **Use Nesterov Momentum?**
  - **Yes, when:** the model is deep or the error surface is complex
  - Useful when standard momentum leads to oscillations or slowdowns
  - Often a good default choice in modern frameworks

Challenge: Proper parameter tuning is difficult and task-dependent

Solution: Adaptive learning rate methods

Adaptive Learning Rate Control

- Various strategies exist — from simple heuristics to sophisticated methods

Primitive Heuristic (already discussed):

- The learning rate should decrease as the number of epochs increases
- **Initial learning rate:** $0 \ll \alpha < 1$
  - Helps escape shallow local minima
  - Enables rapid early learning
- **Final learning rate:** $\alpha \sim 0$
  - Prevents oscillations
  - Should not decrease too quickly; it is sufficient to ensure: $\sum_{t=0}^{\infty} \alpha_t = \infty, \quad \sum_{t=0}^{\infty} \alpha_t^2 < \infty$

Adaptive Learning Rate

Local learning rate for each weight

- Weight update from neuron i to neuron j at time $t+1$:

$$\Delta w_{ij}(t+1) = -\alpha_{ij,t+1} \frac{\partial E_t}{\partial w_{ij}} + \alpha_m \Delta w_{ij}(t)$$

Resilient Propagation (Rprop) – Silva & Almeida

Learning rule based on the sign change of the partial derivative:

- Initialize $\alpha_{ij,0}$ with small random values
- Accelerate learning if the sign of $\frac{\partial E}{\partial w_{ij}}$ remains the same for two consecutive iterations
- Slow down learning if the sign changes

Variants:

- Basic Rprop (Silva & Almeida)
- Super SAB
- Rprop+
- ...

Resilient Propagation (Rprop) – Silva & Almeida

Learning rate adaptation at time $(t+1)$

- $\alpha_{ij,t+1} = u \cdot \alpha_{ij,t}$, if $\quad \frac{\partial E_t}{\partial w_{ij}} \cdot \frac{\partial E_{t-1}}{\partial w_{ij}} > 0$
- $\alpha_{ij,t+1} = d \cdot \alpha_{ij,t}$, if $\quad \frac{\partial E_t}{\partial w_{ij}} \cdot \frac{\partial E_{t-1}}{\partial w_{ij}} < 0$
- Constants: $u > 1$, $d < 1$

Problems:

- Learning rate grows or shrinks exponentially due to u and d
  $\rightarrow$ Issues may arise if many consecutive accelerations occur

Resilient Propagation (Rprop) – Super SAB

Algorithm:

- Initialize all α_{ij}^0 to a starting value α_{start}
- Perform step t of the backpropagation algorithm with momentum
- If $\frac{\partial E_t}{\partial w_{ij}} \cdot \frac{\partial E_{t-1}}{\partial w_{ij}} > 0$: $\alpha_{ij,t+1} = u \cdot \alpha_{ij,t}$
- If $\frac{\partial E_t}{\partial w_{ij}} \cdot \frac{\partial E_{t-1}}{\partial w_{ij}} < 0$:
  - Cancel previous weight update: $w_{ij}(t+1) = w_{ij}(t) - \Delta w_{ij}(t)$
  - Decrease learning rate: $\alpha_{ij,t+1} = d \cdot \alpha_{ij,t}$

Properties:

- Orders of magnitude faster than standard backpropagation
- Relatively stable
- Robust to choice of initial parameters

Resilient Propagation (Rprop+)

- Learning rate is adjusted based on changes in the error value, rather than the sign of the gradient
- Faster than Super SAB

Algorithm:

- Initialize all α_{ij}^0 to a starting value α_{start}
- Perform step t of the backpropagation algorithm with momentum
- If $E_t < E_{t-1}$:
  - Increase learning rate: $\alpha_{ij,t+1} = u \cdot \alpha_{ij,t}$
- If $E_t > c \cdot E_{t-1}$:
  - Cancel previous weight update: $w_{ij}(t+1) = w_{ij}(t) - \Delta w_{ij}(t)$
  - Decrease learning rate: $\alpha_{ij,t+1} = d \cdot \alpha_{ij,t}$
- Constants $c > 1$, $u > 1$, $d < 1$

Modern Optimization Algorithms

Foundation:

- **SGD (Stochastic Gradient Descent)** – the basic algorithm, improved using momentum (**Momentum / Nesterov Momentum**)

Modern optimizers in libraries like Keras and PyTorch:

- **RMSprop** – adaptive learning rate, suitable especially for recurrent models (RNNs)
- **Adam (Adaptive Moment Estimation)** – currently the most widely used
- **NAdam** – Adam combined with Nesterov momentum
- and many others (**AdaGrad, Adadelta, AdaMax, FTRL, ...**)

Typical advantages of modern optimizers

- faster convergence thanks to adaptive learning rates
- lower sensitivity to hyperparameter settings

https://keras.io/api/optimizers/

RMSprop – Root Mean Square Propagation

Key Idea:

- The error surface may have various steep and flat regions in different directions (flat vs steep vs bumpy surface)
- We don't want to use the same learning step in all directions
- RMSprop adjusts the step size for each weight individually, depending on how much the gradient varies in that direction
- It acts as an **adaptive shock absorber** for rough terrains

How does it work?

- It tracks an exponential moving average of squared gradients for each weight:

$$v_{ij}(t) = \beta \cdot v_{ij}(t-1) + (1-\beta) \cdot \left(\frac{\partial E_t}{\partial w_{ij}} \right)^2$$

RMSprop – Root Mean Square Propagation

How does it work?

- It tracks an exponential moving average of squared gradients:

$$v_{ij}(t) = \beta \cdot v_{ij}(t-1) + (1-\beta) \cdot \left(\frac{\partial E_t}{\partial w_{ij}}\right)^2$$

- If gradients are frequently large $\rightarrow$ step size is reduced (slower learning)
- If gradients are small $\rightarrow$ step size remains larger (faster learning)

Weight update:

$$\Delta w_{ij}(t) = -\frac{\alpha}{\sqrt{v_{ij}(t)} + \varepsilon} \cdot \frac{\partial E_t}{\partial w_{ij}}$$

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$$

RMSprop – Root Mean Square Propagation

Result:

- More stable learning, especially in networks with gradients of varying scale (e.g., RNNs)
- Less dependent on manual tuning of the learning rate α

Use Cases:

- Commonly used for recurrent neural networks (RNNs)
- Handles non-stationary gradients well

Adam – Adaptive Moment Estimation

Principle:

- Combines the benefits of momentum (1st moment) and RMSprop (2nd moment)
- Estimates the mean and variance of gradients using exponential moving averages
- Tracks:
  - $m_{ij}(t)$ – moving average of gradients (1st moment estimate)
  - $v_{ij}(t)$ – moving average of squared gradients (2nd moment estimate)
- Includes bias correction to account for initialization effects

Adam – Adaptive Moment Estimation

Weight update:

$$\Delta w_{ij}(t) = -\alpha \cdot \frac{\hat{m}_{ij}(t)}{\sqrt{\hat{v}_{ij}(t)} + \varepsilon} \qquad w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$$

Where:

$$g_{ij}(t) = \frac{\partial E_t}{\partial w_{ij}}$$

$$m_{ij}(t) = \beta_1 \cdot m_{ij}(t-1) + (1 - \beta_1) \cdot g_{ij}(t)$$

$$v_{ij}(t) = \beta_2 \cdot v_{ij}(t-1) + (1 - \beta_2) \cdot g_{ij}^2(t)$$

$$\hat{m}_{ij}(t) = \frac{m_{ij}(t)}{1 - \beta_1^t}, \quad \hat{v}_{ij}(t) = \frac{v_{ij}(t)}{1 - \beta_2^t}$$

Typical values: $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-7}$

Adam – Adaptive Moment Estimation

- **General-purpose, reliable optimizer** – works well in most scenarios without much parameter tuning
- **Combines the benefits of:**
  - Momentum – smoother, more stable weight updates
  - Adaptive step size – like RMSprop
- Default choice in frameworks like Keras and PyTorch
- For some tasks (e.g., convex loss surfaces), it may lead to suboptimal solutions
  $\rightarrow$ in such cases, SGD or RMSprop may perform better

Keras:
optimizer = Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-7)

Nadam – Nesterov-accelerated Adam

Combines the benefits of:

- Adam (adaptive learning + momentum)
- Nesterov momentum (lookahead gradient evaluation)

Effect:

- Faster and more stable convergence compared to standard Adam
- Suitable for deep networks and complex learning problems

Summary of Modern Optimizers

Optimizer	**Advantages**	**When to Use**
SGD	simple, transparent	small models, manual tuning
SGD+Moment.	faster convergence	deeper networks
RMSprop	adaptive step size, stable	RNNs, sequential data
Adam Nadam	robust, minimal tuning even more stable	universal default deep and complex models

Other Tricks to Improve Learning

Run training multiple times

- Use different random weight initializations and select the best-performing model

If the network fails to converge or converges very slowly

- Try adding more neurons or layers

Present important training examples more frequently

- Helps reduce error for critical patterns

Other Tricks to Improve Learning

“Network annealing” = injecting random noise

- Use when weights and biases settle but the error remains high
- Adapt weight from neuron i to j:

$$w_{ij}(t+1) = w_{ij}(t) + N(0, \epsilon)$$

- Choose ϵ carefully:
  - Too small → ineffective
  - Too large → network may require full retraining

Practical Examples of Different Task Types

- Binary classification: Breast Cancer (already covered)
- Multiclass classification: MNIST
- Regression task: Wine Quality
- Time series prediction: Daily minimum temperatures in Melbourne

Binary Classification Example: Breast Cancer

- Already covered last time

Model setup:

- **Sigmoid** activation in the output layer
- **ReLU** (or **tanh**) activation in hidden layers
- Loss function: **BinaryCrossentropy**; Metrics: **Accuracy**, **Precision**, **Recall**, ...

Multiclass Classification Example: MNIST

- 60,000 grayscale images of handwritten digits (28x28 pixels)
- Centered images, uniform digit size
- 10,000 test images written by different people
- Output label: digit 0–9 (10 classes)
- Data characteristics:
    - All images have the same size $\rightarrow$ no resizing needed
    - Input data is 3D $\rightarrow$ needs to be flattened into vectors (vectorized)
    - Pixel values range from 0 to 255 $\rightarrow$ need to normalize to $[0, 1]$ or $[-1, 1]$

Multiclass Classification Example: MNIST

Model setup:

- **Softmax** activation in the output layer
- **ReLU** (or **tanh**) activation in hidden layers
- Loss function: **SparseCategoricalCrossentropy** with **SparseCategoricalAccuracy** (if labels are integers)
- Or: **CategoricalCrossentropy** with **CategoricalAccuracy** (if labels are one-hot vectors)

Observations:

- Test accuracy is typically around 85 %
- Similar training and validation errors $\rightarrow$ model generalizes well
- Accuracy can be improved by increasing the learning rate, number of epochs, or changing the optimizer

Regression Example: Wine Quality Data

- 11 numerical input features (e.g., acidity, alcohol content, sulfur dioxide, ...)
- 1 output feature – wine quality (rating from 0–10, most values 3–8)

Data characteristics:

- Features have different value ranges $\rightarrow$ normalization required (e.g., using StandardScaler)
- Sufficient number of samples (4900)
    1. Allows training of larger models without immediate risk of overfitting
    2. Dataset can be split into training, validation, and test sets

Regression Example: Wine Quality Data

Model setup for regression:

- ReLU (or tanh) activation in hidden layers
- Linear (identity) activation in the output layer
- Loss function: **MeanSquaredError (MSE)**
- Metrics: **MeanAbsoluteError (MAE)**, **RootMeanSquaredError (RMSE)**

Regression Example: Wine Quality Data

Observations:

- The model learns better for values in the center of the range than for the edges (due to data imbalance) $\rightarrow$ possible solution: oversampling
- Without early stopping, overfitting may occur (especially with larger architectures)
- Performance is sensitive to normalization of input features

Summary:

1. For regression tasks, use a linear output and MSE as the loss function.
2. Normalize input features with varying ranges to improve model training.
3. In addition to MSE, consider using MAE or RMSE for better interpretability.

Alternative Approach: Wine Quality as Classification

Same dataset, different perspective: Predicting wine quality as a classification task instead of regression.

- Convert quality (0–10) into classes, e.g.:
  - **0–4**: low quality
  - **5–6**: medium quality
  - **7–10**: high quality

Question: If we switch from a regression model to a classification model, what changes?

- Output layer, loss function, and evaluation metrics

Alternative Approach: Wine Quality as Classification

Model setup for classification:

- Output layer: 3 neurons with **softmax** activation
- Loss function: **categorical_crossentropy** or **sparse_categorical_crossentropy** (if using class indices)
- Metrics: **accuracy**, possibly **precision**, **recall**, **F1-score**

Note:

- Class distribution is imbalanced → track multiple metrics, not just accuracy

Classification vs. Regression – which makes more sense for this task?

- Consider the application and interpretability when choosing the approach

Time Series Example: Daily Minimum Temperatures

- Dataset: Daily minimum temperatures in Melbourne, years 1981–1990
- Goal: predict the temperature of the next day based on previous values

Data characteristics:

- Single feature (temperature time series)
- Over 3600 records – sufficient for training and testing
- Data points are not independent – temporal dependencies must be captured

Using the Sliding Window Method:

- For each training sample, use e.g. the last 10 days to predict the next day
- This creates a tabular representation suitable even for MLP

Daily Min. Temperatures – Model and Configuration

Input preparation:

- Create training, validation, and test sets using sliding windows (with separate time period for each set!)
- Normalize input values (e.g., MinMaxScaler)

Model configuration:

- Input layer: number of neurons = window size (e.g., 10)
- Hidden layers with ReLU or tanh activation
- Output layer with linear activation (1 value = temperature prediction)
- Loss function: MSE; Metrics: MSE / MAE / RMSE

Observations:

- The MLP needs tuning to outperform a baseline model
- The choice of window size significantly affects prediction quality
- The model can overfit if the architecture is too complex

Summary: Time Series and Sliding Window Method

1. Time series do not consist of independent samples – training, validation, and test sets must preserve temporal order.
2. The sliding window method allows transformation of a time series into a training set for MLP.
3. Window size (number of input values) is an important hyperparameter.
4. Use a linear output function, just like in regression.
5. Although specialized architectures (RNNs, LSTMs) perform better on time series, an MLP with sliding window is a simple and intuitive starting point.

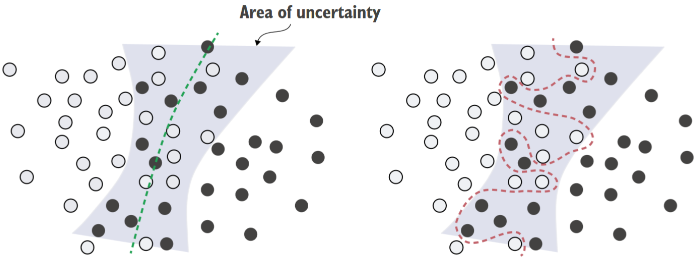
Summary: Time Series and Sliding Window Method

Possible extensions:

- Model input can include not only past values of the target variable, but also other features (e.g., pressure, humidity, etc.)
- Instead of predicting just one future value, you can predict a longer time horizon or multiple future values

Generalization of Neural Networks

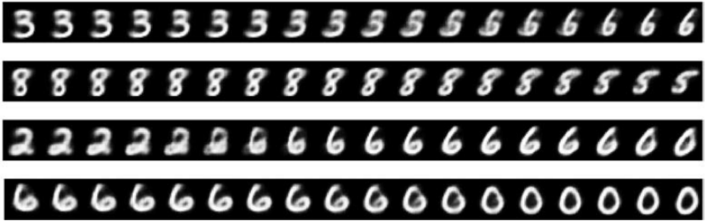
- The ability to produce correct outputs for inputs not seen during training
- Illustration: well-trained model vs. overfitted model



F. Chollet: Deep Learning with Python, Fig. 5.5

Generalization of Neural Networks

- Class boundaries are often hard to define:

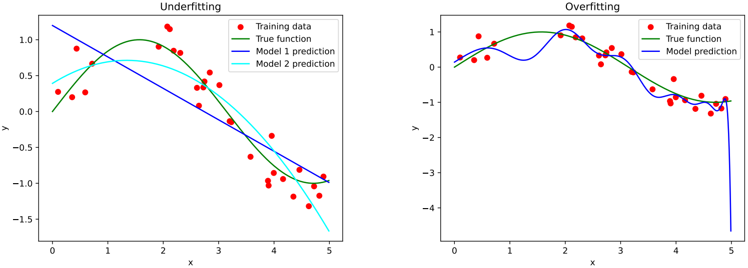


F.

Chollet: Deep Learning with Python, Fig. 5.7

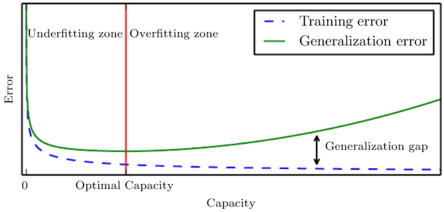
Underfitting vs. Overfitting – Regression Example

- Typical illustration of underfitting and overfitting in regression tasks:



Generalization and Model Capacity

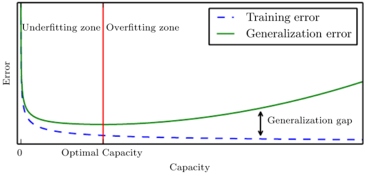
- Generalization depends on the network's architecture and model capacity (i.e., number of parameters)
- **Small model:**
  - Potentially stable but inaccurate predictions
  - Risk of **underfitting**
- **Large model:**
  - Greater variability in performance
  - Risk of **overfitting** – poor generalization



https://www.deeplearningbook.org/, Figure 5.3

Model Capacity and Dataset Size

- The required training set size depends on model capacity
- **Small model:**
  - Stable but potentially underfit
  - Needs fewer training samples to generalize well
- **Large model:**
  - Risk of overfitting
  - Requires more training data to generalize properly



https://www.deeplearningbook.org/, Figure 5.3

Theoretical Insight: Generalization and Training Set Size

Theorem: Relationship between model capacity and required number of training examples

- For a network with one hidden layer, w parameters, h hidden units, and generalization error ϵ, the minimum number of training samples N should satisfy:

$$N \geq \frac{w}{\epsilon} log_2(\frac{h}{\epsilon})$$

$\rightarrow$ If $N < \frac{w}{\epsilon}$, the model cannot generalize properly

- For target accuracy $\geq 90\%$, choose at least $10 \cdot w$ training samples

Generalization in Deep Networks

Estimated training set size for deeper architectures:

$$N \geq O\left(\frac{w \cdot \log w}{\epsilon}\right)$$

- More layers $\rightarrow$ more parameters $\rightarrow$ more data needed
- Empirical rule: **Often we need significantly more training samples than parameters**
- To achieve good generalization:
  - Use a sufficiently large training set, or
  - Apply suitable regularization techniques