Neural Networks 3 - Neural Networks

18NES2 - Lecture 3, Winter semester 2025/26

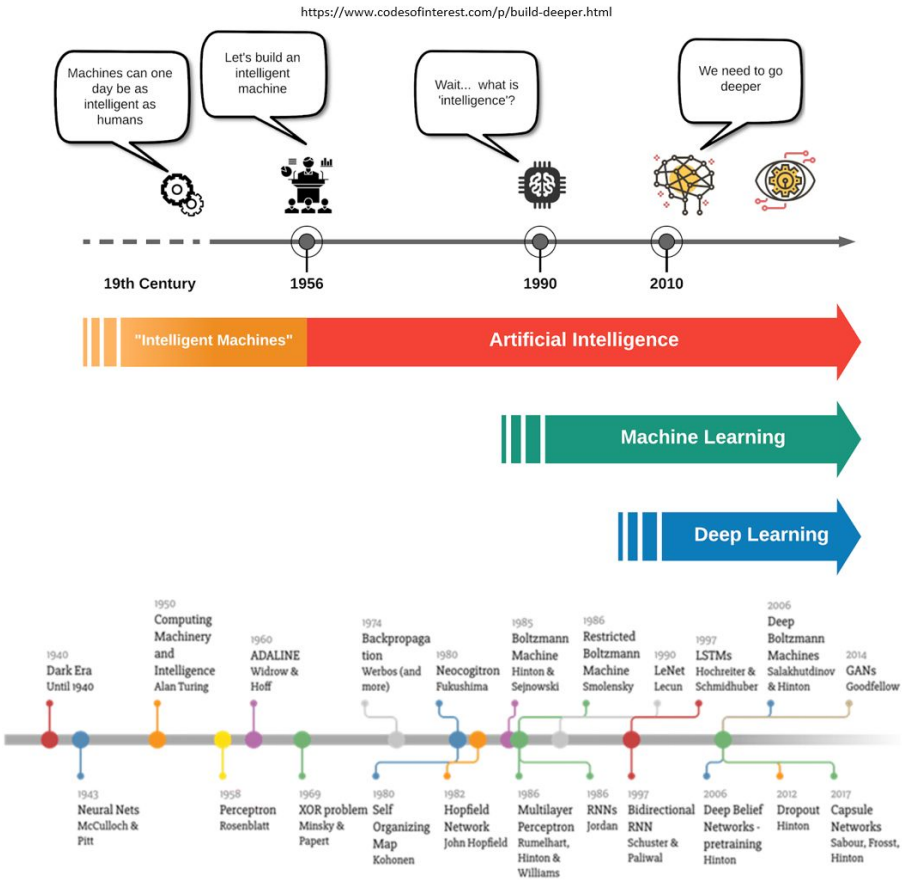
Zuzana Petříčková

October 7, 2025

Neural Networks 2 - Neural Networks

What We Covered Last Time

- A Brief History of Neural Networks
- Introduction to Artificial Neural Networks
  - Artificial Neurons
  - Neural Networks and Their Architecture
  - Multi-layer Neural Network Model (MLP)
  - Introduction To Training

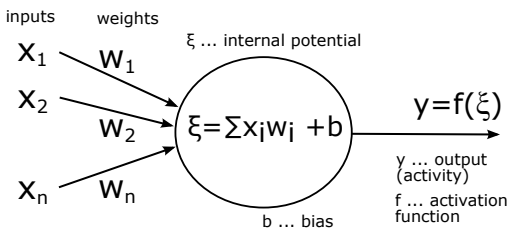


Mourtzis, Dimitris & Angelopoulos, John. (2020). An intelligent framework for modelling and simulation of artificial neural networks (ANNs) based on augmented reality. International Journal of Advanced Manufacturing Technology. 111. 10.1007/s00170-020-06192-y.

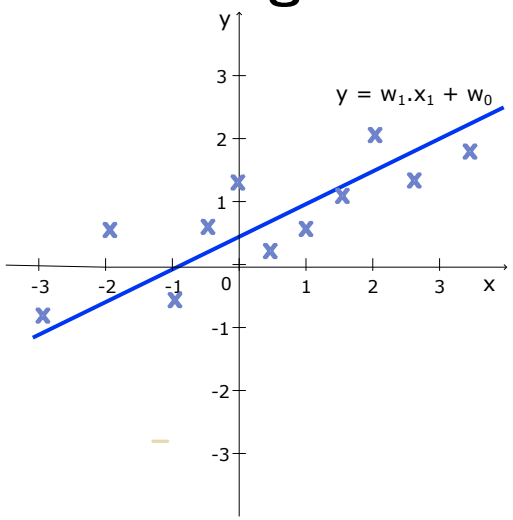
Review — Artificial Neurons

Artificial Neurons, their interpretation and activation functions

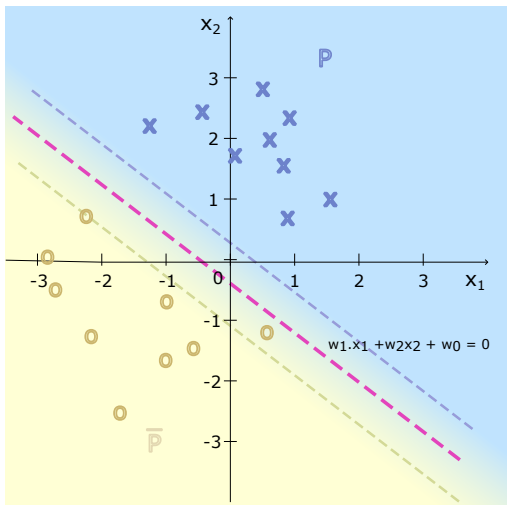
Artificial neuron



Linear regression



Linear classification



- Internal potential: $\xi = \sum_{i=1}^{n} w_i x_i + b = \mathbf{w}^\top \mathbf{x} + b$
- Output: $y = f(\xi)$ (where f is the activation function)

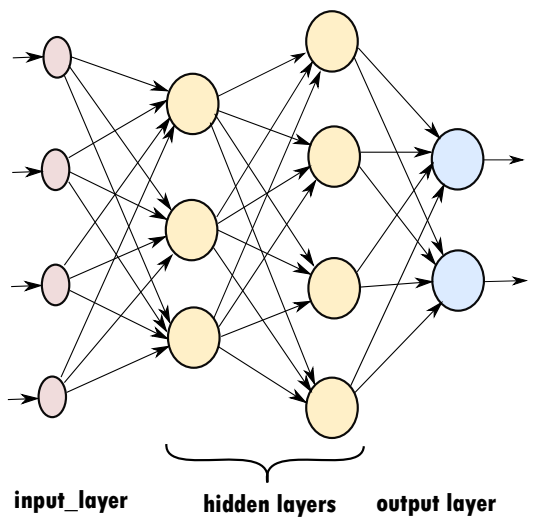
Review — Multi-Layer Neural Network (Multi-Layer Perceptron, MLP, 1980)

- Hierarchical **sequential** architecture: neurons are arranged in layers
- **Dense (fully connected) layers**: every neuron in one layer is connected to every neuron in the next layer

- Special **input layer:** corresponds to the inputs of the neural network
- The last layer is called **output layer**, the remaining layers are **hidden layers**.

Output (response) of the model:

- corresponds to the activities of the output neurons



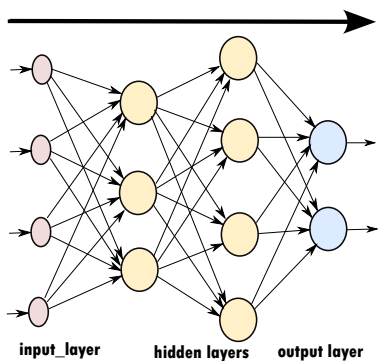
Multi-Layer Neural Network (Multi-Layer Perceptron)

How to represent the sequential NN model?

- A sequence (list) of layers $L_0, ..., L_{max}$ (see the Keras example)
- Each layer is represented by a tensor (2D matrix) of weights (and biases)

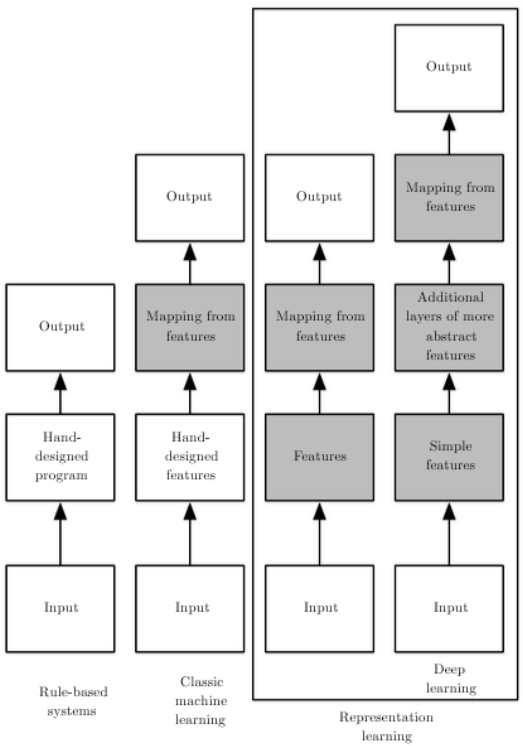
How to compute the model output (response):

- by performing a **forward pass**
- we process one layer at a time, starting from the input layer and going toward the output:
  - present the input tensor to the current layer
  - compute the layer's output
  - use this output as the input to the next layer



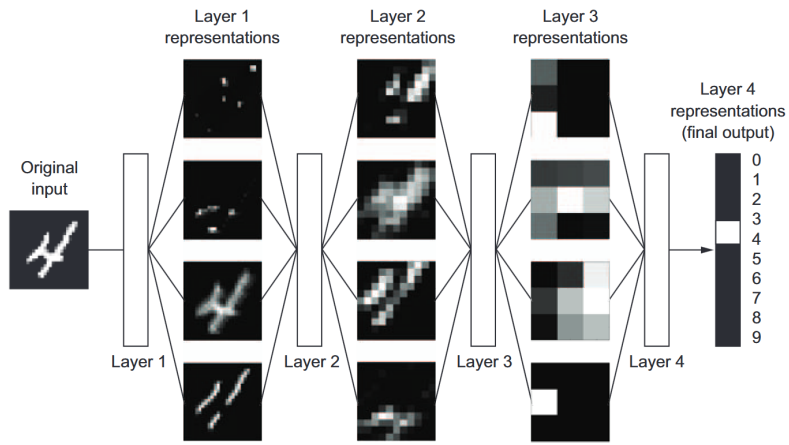
Neural Network Architecture (Topology)

- **Shallow model** – one hidden layer
- **Deep model** – more (or many) hidden layers
  - Automatically extracts features from data, reducing the need for extensive preprocessing.



I. Goodfellow, Y. Bengio, and A. Courville: Deep Learning, 2016, Figure 1.5

Convolutional neural network



F. Chollet: Deep Learning with Python, Fig. 1.6

Training a Neural Network (Supervised Learning)

Data used for training the model

- **Training set** T
  - a set of N training samples
    $T = (X, D) = \{(x_1, d_1), ..., (x_N, d_N)\}$
  - X ... input data (tensor), D ... desired output (tensor)
- **Training sample (pattern)** (x_i, d_i)
  - x_i ... input pattern (tensor)
  - d_i ... target / expected / desired output
- Examples of input data tensors:
  - vector data — 2D tensor (samples, features)
  - time series and sequential data — 3D tensor (samples, timesteps, features)
  - images — 4D tensor (samples, height, width, channels)
  - video — 5D tensor (samples, frames, height, width, channels)

Training a Neural Network (Supervised Learning)

The number and shape of output features depend on the task:

- **Regression:**
  output tensor shape (samples, 1) or (samples, output features)
  e.g. prediction of values (stock price, temperature,...)
- **Classification:**
  output tensor shape (samples, number of classes)
  e.g. binary classification $\rightarrow$ shape (samples, 1)
  multiclass classification $\rightarrow$ shape (samples, number of classes)
- **Sequential data:**
  output tensor shape (samples, timesteps, output features)
  e.g. sentence translation $\rightarrow$ shape (samples, output sequence length, vocabulary size)
- images, video, ...

Training a Layered Neural Network (MLP Model)

- Let us now focus on the classical Multi-Layer Neural Network model with n inputs and m output neurons:

Training data

- Training set $T = (X, D)$
  - X ... input patterns: 2D tensor of shape (N, n), where N is the number of training samples, n is the number of input features
  - D ... desired outputs: 2D tensor of shape (N, m), where m is the number of output features
- Training sample (pattern) $(\vec{x_i}, \vec{d_i})$
  - $\vec{x_i}$ ... vector of length n (input pattern)
  - $\vec{d_i}$ ... vector of length m (target / expected output)

Learning objective:

- Adjust the weights (and biases) of all neurons in the network so that the actual output Y matches the desired output D.

Training a Layered Neural Network (MLP Model)

Basic principle (simplified)

1. Randomly initialize the model parameters (weights and biases of all neurons).
2. Repeat the training cycle:
   - prepare a batch of training inputs X and corresponding targets D
   - compute the actual output (prediction) of the model Y
   - calculate the model error (difference between Y and D)
   - update weights and biases to make the error slightly smaller

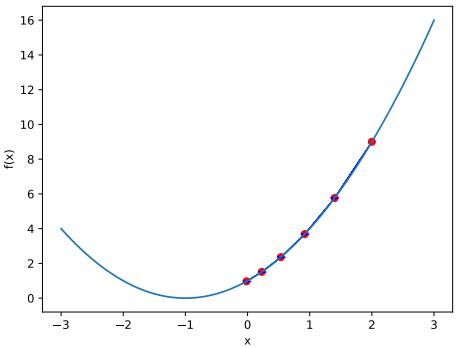
$\rightarrow$ **gradient descent method** (or its variants)

- the need of continuous and differenciable activation functions

Gradient Descent Method (Steepest Descent)

Problem Definition:

- We have a function $f(\vec{x}) : \mathbb{R}^n \rightarrow \mathbb{R}$
- We seek $\vec{x}$ such that $f(\vec{x})$ is minimized



→ **Solution (gradient descent method):**

1. Start at an (random) initial point $\vec{x}(0)$
2. Compute the gradient: $\nabla f(\vec{x}) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right)$ The gradient represents the direction and magnitude of the greatest increase in $f(\vec{x})$
3. Iteratively move in small steps opposite to the gradient direction: $\vec{x}(t+1) = \vec{x}(t) - \alpha \nabla f(\vec{x})$ α is a small positive number (step size, learning rate)
4. For a single input feature: $x_i(t+1) = x_i(t) - \alpha \frac{\partial f}{\partial x_i}$

Challenges in Gradient Descent

Common Issues:

- May converge to a local minimum instead of the global minimum
- The method has an important hyperparameter α ... **learning rate** (step size)
  - small $\alpha \rightarrow$ slow convergence
  - large $\alpha \rightarrow$ oscillations / overshooting
  - the optimal value depends on the problem **and on the training phase**
- The learning rate is a critical hyperparameter
  $\rightarrow$ advanced optimization algorithms adjust it adaptively (e.g., Adam, RMSProp)

Loss Function in Neural Networks

Goal of training:

- Optimize the parameters (weights and biases) of the network
- So that the predictions Y are as close as possible to the desired outputs D
- This is done by minimizing a **loss function** (cost function)

Examples of loss functions:

- **Regression:** Mean Squared Error (MSE), Mean Absolute Error (MAE)

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - d_i)^2$$

- **Binary classification:** Binary Cross-Entropy
- **Multiclass classification:** Categorical Cross-Entropy

Backpropagation Algorithm

Core principle: it is a standard gradient descent method

1. Randomly initialize the model parameters (weights and biases)
2. Repeat for training epochs:
   - Prepare a batch of input samples X and their corresponding target outputs D
   - Compute the model's actual outputs Y
   - Compute the model error (based on the difference between Y and D)
   - Update parameters (weights and biases) to slightly reduce the error (i.e., move in the opposite direction of the loss gradient):

$$w_i(t+1) = w_i(t) - \alpha_t \frac{\partial E_t}{\partial w_i}$$

   - Adjust the learning rate $\alpha_t \rightarrow \alpha_{t+1}$

Nice visualizations of loss surfaces:

jithinjk.github.io/blog/nn_loss_visualized.md.html

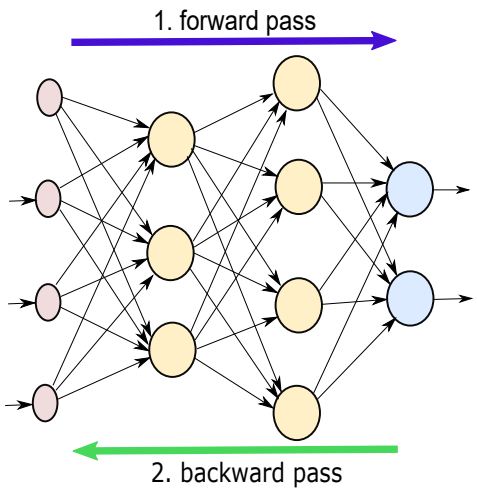
izmailovpavel.github.io/curves_blogpost

Backpropagation Algorithm

Core principle: backpropagation is gradient descent done efficiently.

Key idea:

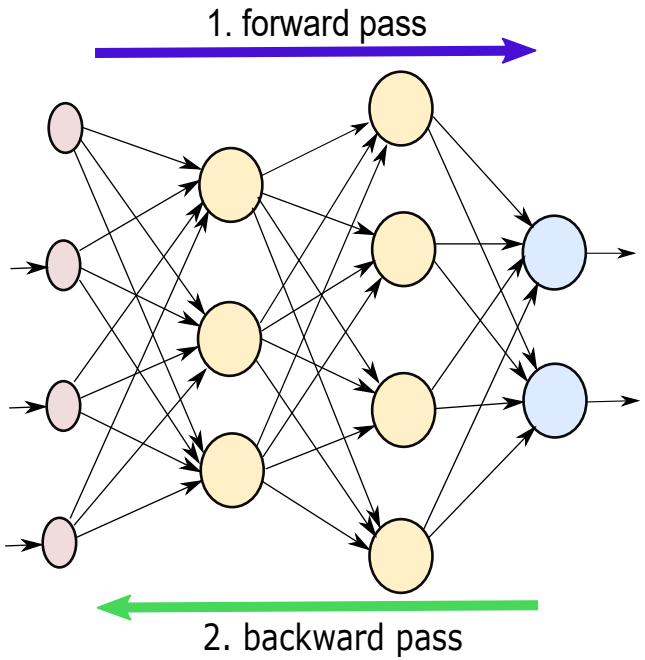
- We do not calculate the error derivative separately for every single weight — that would be extremely inefficient.
- Instead, each layer reuses error information from the layer above.
- By passing these error terms backwards through the network (layer by layer), we can update all weights efficiently in a single backward pass.
- This makes it possible to train deep neural networks in practice.



Backpropagation Algorithm

Basic principle of backpropagation:

1. Compute the actual network output for the given batch of training samples
   - by a single pass from the input to the output layer (forward pass)
2. Compare the actual and desired outputs
3. Update the weights and biases:
   - in the direction opposite to the gradient of the loss
   - using a single pass from the output to the input layer (backward pass)



Backpropagation – How to Present Training Samples

Classical Presentation strategies:

1. **Sample-wise per epoch (Online GD):** Each sample is presented once per epoch, samples are shuffled every epoch.
   - Maximum number of epochs = how many times the full dataset is presented.
   - Fast training, but relatively unstable.
2. **Batch-wise per epoch (Batch GD):**
   - The entire training set is used at once to compute and apply a single weight update.
   - More stable, but computationally and memory intensive for large datasets.

Deep learning: Mini-batch training (Stochastic GD, SGD):

- Training set is randomly split into small batches that are processed iteratively.
- Combines advantages of both previous methods.

Backpropagation Algorithm

Stopping conditions:

- Maximum number of epochs
- Training error drops below threshold: $E < E_{min}$
- Validation error stops decreasing (early stopping)
- Weight updates become very small

Multi-Layer Neural Network (MLP) – Model Analysis

Advantages:

- Simple yet powerful universal model with good approximation and generalization ability
  - Suitable for both classification and regression tasks
  - Able to capture complex nonlinear relationships
- Universal approximator – can approximate any continuous function (with nonlinear activation functions, a single hidden layer is sufficient). However, the learning problem is NP-complete.
- Uses backpropagation for efficient gradient-based learning
- Generalizes reasonably well on unseen data

Multi-Layer Neural Network (MLP) – Model Analysis

Disadvantages:

- The model is highly sensitive to weight initialization, training data, and hyperparameters, which need to be carefully tuned.
- Input and output data must be in vectorized numerical form.
- Slow convergence – although faster variants exist (e.g., Adam optimizer).
- Local learning method – can end up in suboptimal solutions
- Prone to overfitting – mitigated by regularization, early stopping, etc.
- Lacks built-in mechanisms to exploit spatial or sequential structure in data ($\rightarrow$ need for CNNs, RNNs, Transformers)
- "Black box" – the internal knowledge representation (weights and biases) is difficult for humans to interpret.

Neural Networks 2 - Neural Networks

Main Deep Learning Frameworks in Python

- **TensorFlow** – Open-source library by Google.
  - Powerful framework for AI applications (mobile, server).
  - Supports both static and dynamic computation graphs.
- **PyTorch** – Open-source library by Meta (Facebook).
  - Flexible and intuitive, ideal for research and academia.
  - Dynamic computation graphs, easy debugging.
- **Keras** – High-level universal API.
  - Beginner-friendly and easy to understand.
  - Great for fast prototyping. Runs on top of TensorFlow, JAX, or PyTorch.
- **PyTorch Lightning** – High-level wrapper for PyTorch.
  - Reduces boilerplate code in training routines.
  - Supports multi-GPU training, scaling, and reproducibility.
- **JAX** (Google, Nvidia,...) – Optimized for speed and experimental research
- Previously popular **Theano** – now deprecated.

TensorFlow vs PyTorch – Comparison

TensorFlow – robust and production-ready, but more rigid:

- Part of a broader ecosystem (TensorBoard, TF Lite, etc.).
- Very efficient (C++/Python hybrid), supports distributed training, native TPU support.
- Optimized for deployment, mobile support (TF Lite), model compilation.
- Less developer-friendly: more code, harder to define custom models.
- Difficult debugging of complex models (C++ backend).

PyTorch – newer, rapidly evolving, research-focused:

- Pythonic, concise, and easier to use; gaining feature parity.
- Slightly less performant (pure Python), but highly flexible.
- Custom models and layers are very easy to implement and debug.

Other Useful Libraries

Data manipulation and numerical computing:

- **Scikit-learn (sklearn)** – classic ML algorithms; tools for data processing and model evaluation.
- **NumPy** – efficient numerical computing with arrays and tensors.
- **Pandas** – powerful data manipulation library for structured data (categorical, missing values).

Visualization:

- **Matplotlib** – general-purpose plotting (static, animated, interactive).
- **Plotly** – interactive visualizations.
- **Seaborn** – statistical data visualization (correlations, distributions, etc.).
- **TensorBoard** – visualization of the training progress, especially for TensorFlow.

Practical Examples

useful_python_libraries.ipynb

- Commented examples of the usage of useful Python libraries (NumPy, Pandas, Matplotlib...)

NN_libraries.ipynb

- Commented examples comparing major deep learning frameworks (Keras, TensorFlow, PyTorch, Lightning) on a simple binary classification task.
- Demonstration of automatic symbolic tensor differentiation in TensorFlow and PyTorch.
- Frameworks and GPU support in practice.

NN_libraries_installation.ipynb

- Brief installation guide for running the examples locally on your own machine.

Training Workflow in Keras: A Complete Example

What is essential for the success of the training?

- Proper preprocessing of the training data
- Careful tuning of model hyperparameters for the specific task

keras_extended_example.ipynb

- A comprehensive, step-by-step Keras example demonstrating the complete learning workflow for a MLP model (on a binary classification task)
- Data preprocessing and analysis, model creation and hyperparameter tuning, training process monitoring, visualization, and evaluation
- Visualization using TensorBoard.

During the session, we will switch between the slides and the example notebook.

Preprocessing Training Data

Key preprocessing steps:

- **Serialization (specific to MLP models):**
  - Convert input and output data into 2D tensors of shape (samples, numerical features)
- **Handling categorical variables:**
  - **Ordinal encoding:** If categories can be ordered, convert each category into a numeric value and normalize it
  - **One-hot encoding:** Convert categorical variables into binary vectors (e.g., categories "A", "B", "C" become [1, 0, 0], [0, 1, 0], [0, 0, 1])
- **Ensuring data consistency:**
  - Check that all input vectors have the same length and no missing values.
  - Replace missing values using the mean, median, or more advanced imputation techniques.

Preprocessing Training Data

Key preprocessing steps (continued):

- **Normalization/Standardization of inputs:**
  - **Normalization**: Scale features to a fixed range, such as [0, 1] or [-1, 1], depending on the activation function (e.g., ReLU vs. tanh).
  - **Standardization**: Typically adjust features to have zero mean and unit variance.
  - Normalization is crucial for stable and efficient training.
- **Training set should be sufficiently large and balanced.**
  - In some cases, **data augmentation** is necessary to increase the number of training samples.
- **Split data into training, validation, and test sets:**
  - A common split is 70% training, 15% validation, and 15% test set.

Key Hyperparameters of a 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, ...

Architecture of a Multi-layer Neural Network

Creating a model in Keras

- **Sequential** – the simplest way to build a model, stacking layers sequentially.
- **Input** – input layer (can be omitted in simple cases)
- **Dense** – fully connected layer
  - Number of neurons
  - Activation function: **activation='relu'**, **'sigmoid'**, **'linear'** (default), ...
  - Weight initialization method: **kernel_initializer='glorot_uniform'**, **bias_initializer='zeros'** (default), ...
  - Example: **Dense(10, activation='relu', kernel_initializer='he_normal')**

Official documentation:

https://keras.io/api/layers/core_layers/dense/

Architecture of a Multi-layer Neural Network

- **Model size:** Defined by the number of layers and neurons in each layer

How to choose model size?

- **Input and output layers:** The number of neurons is determined by the data shape.
- **Hidden layers:**
  - Larger model = higher capacity $\rightarrow$ better at capturing complex patterns
  - Small model $\rightarrow$ underfitting, cannot capture complex relationships
  - Too large model with limited data $\rightarrow$ overfitting
  - The optimal number of layers and neurons depends on the task complexity and data size; usually selected experimentally.

Architecture of a Multi-layer Neural Network

- **Shallow model** – one hidden layer
  - Better suited for simpler tasks – learns faster and generalizes well
  - Performs better on small datasets (large datasets may not help much)
  - Easier to understand and interpret
  - Learns complex tasks slowly and may require many neurons
- **Deep model** – more (or many) hidden layers
  - More suitable for complex tasks with large training datasets
  - Capable of learning intricate patterns in the data
  - Requires different training strategies and poses different challenges

Model Size and Its Effect on MLP Performance

Practical recommendations:

- Start with a smaller model and gradually increase size as needed.
- Use validation data to monitor performance and avoid overfitting.
- If overfitting occurs, apply techniques like regularization, early stopping, or dropout.
- Choose a good balance between width (neurons per layer) and depth (number of layers).
- For smaller datasets, prefer smaller models.

Which Activation Functions to Use?

Which activation function for the output layer?

- Regression task: linear (**linear**)
- Binary classification: sigmoid (**sigmoid**)
- Multi-class classification: **softmax**

Which activation function for hidden layers?

- Hyperbolic tangent (**tanh**) – stable, symmetric; can suffer from saturation; popular in recurrent models
- In deep networks, **ReLU** is commonly used – fast and effective, but asymmetric and limited in expressive power (saturation risk still exists)

https://keras.io/api/layers/activations/

Proper Initialization of Weights and Biases

Rule of thumb:

- Weights and biases should be small, random, uniformly distributed, and centered around zero.

Recommendations:

- For ReLU: **He** initialization (HeUniform, HeNormal) – maintains variance of neuron outputs
- For sigmoid/tanh/linear: **Glorot (Xavier)** initialization (GlorotUniform, GlorotNormal)

Defaults in Keras:

- Dense layer → `kernel_initializer='glorot_uniform'`
- Biases → initialized to `zeros` by default

https://keras.io/api/layers/initializers/

Model Compilation in Keras (`model.compile`)

- Used to define how the model will learn.

Key arguments:

- **optimizer** – the learning algorithm (e.g., **'adam'**, **'sgd'**, or **Adam(learning_rate=0.001)**)
- **loss** – loss function to be minimized during training
- **metrics** – metrics for monitoring and evaluating model performance (e.g., **['accuracy']**, **['mae']**)

Example:

```
model.compile(optimizer='adam',
loss='binary_crossentropy', metrics=['accuracy'])
```

Which Loss Function to Use?

For regression tasks:

- **MSE (loss='mean_squared_error')**
  - Most commonly used loss function for regression
  - Sensitive to outliers
- **MAE (loss='mean_absolute_error')** – more robust to outliers
- **Huber Loss (loss='huber')** – hybrid of MSE and MAE
- ...

https://keras.io/api/losses/

Which Loss Function to Use?

For classification tasks:

- **Binary Crossentropy (loss = 'binary_crossentropy')**
  - Suitable for binary classification together with a sigmoid activation in the output layer
- **Categorical Crossentropy (loss = 'categorical_crossentropy')**
  - Suitable for multi-class classification with softmax output
  - Assumes one-hot-encoded labels
- **Sparse Categorical Crossentropy (loss = 'sparse_categorical_crossentropy')**
  - Similar to categorical crossentropy but uses integer class indices instead of one-hot encoding

Which Metric to Use for Model Evaluation?

Classification:

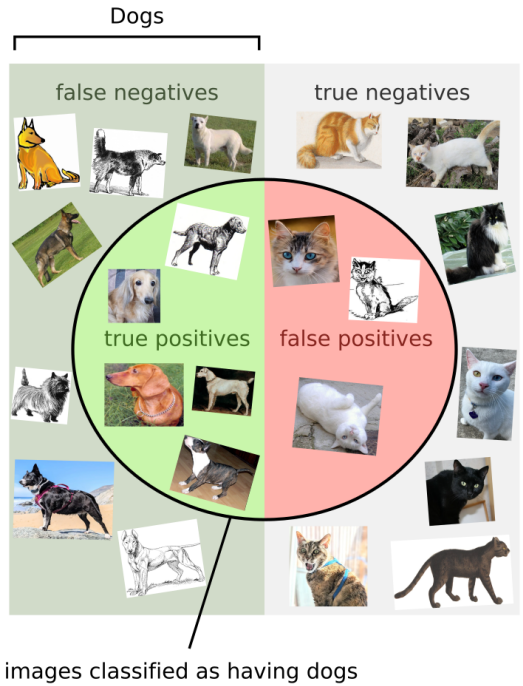
- **Accuracy** – proportion of correctly classified samples
  - **Binary classification: accuracy**, **binary_accuracy**
  - **Multi-class classification: categorical_accuracy** (for one-hot labels), **sparse_categorical_accuracy** (for integer labels)
- **Other metrics for binary classification:**
  - **AUC** – area under the ROC curve; useful for imbalanced datasets
  - **Precision**, **Recall**, **F1** – useful when minimizing false positives or false negatives

Regression:

- **mean_squared_error (MSE)** – for typical regression tasks
- **mean_absolute_error (MAE)** – better when data contains outliers

https://keras.io/api/metrics/

Which Metric to Use for Model Evaluation?



$$\text{Precision} = \frac{\text{5 true pos.}}{\text{8 total pos.}} \qquad \text{Recall} = \frac{\text{5 true pos.}}{\text{12 total dogs}}$$

$$\text{Prevalence} = \frac{\text{12 total dogs}}{\text{22 total images}}$$

$$\text{Accuracy} = \frac{\text{5 true pos. + 7 true neg.}}{\text{22 total images}}$$

Source: https://en.wikipedia.org/wiki/Precision_and_recall, license CC0

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/

Choosing the Right Learning Rate

- For SGD, setting the learning rate correctly is crucial.
- It controls how quickly the model learns.

How to choose the learning rate?

- Too small $\rightarrow$ slow learning, risk of getting stuck in a local minimum
- Too large $\rightarrow$ unstable learning, risk of overshooting the minimum and oscillations

What helps?

- Tuning the learning rate for your specific task
- Using momentum (**momentum**, **nesterov**)
- Using adaptive optimizers (Adam, RMSprop)

Training the Model in Keras (`model.fit`)

Key arguments:

- **x**, **y** – input patterns and desired outputs
- **batch_size** – number of samples processed at once (e.g., 32)
- **epochs** – number of passes through the entire dataset
- **validation_data** – validation set, e.g., **(x_val, y_val)**
- **callbacks** – functions called during training (e.g., **EarlyStopping**)
- **shuffle=True** – shuffle the data before each epoch

Example:

```
model.fit(x_train, y_train, batch_size=32, epochs=10,
validation_data=(x_val, y_val), shuffle=True)
```

Documentation:

https://keras.io/api/models/model_training_apis/#fit-method

Other Key Hyperparameters for MLP

- **Batch size** – number of samples in one mini-batch
  - Typically a power of 2 (8, 16, 32, 64, ...)
  - Small batches: slower, less stable learning; often better generalization
  - Large batches (512+): faster training, higher memory usage, increased risk of overfitting
  - **Recommendation:** Use smaller batches for small datasets. For large datasets, use the largest possible batch size that fits in memory, while monitoring generalization.
- **Number of epochs**
  - Determined experimentally; use early stopping to avoid overfitting

Callbacks in Keras

Functions called during model training
$\rightarrow$ enable monitoring, model saving, early stopping, etc.

Most commonly used callbacks:

- **EarlyStopping** – stops training when validation performance stops improving
- **ModelCheckpoint** – saves the best model based on a chosen metric
- **ReduceLROnPlateau** – reduces the learning rate when a metric has stopped improving
- **TensorBoard** – logs training metrics for interactive visualization

Usage: see example in notebook

Documentation:
https://keras.io/api/callbacks/

Keras Model – Evaluation, Prediction, Saving and Loading

evaluate() – evaluate model performance on test data

- Returns a tuple of loss and metrics values:
  `loss, accuracy = model.evaluate(x_test, y_test)`

predict() – compute model outputs

- Example for classification:
  `y_pred = model.predict(x_test)`

save() – save the model to a file

- `model.save("model.keras")`

load_model() – load a saved model

- `model = load_model("model.keras")`

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

Practice: Explore and Experiment

keras_extended_example.ipynb

- Experiment with how changing various hyperparameters (architecture, optimizer, etc.) affects the learning process and performance.
- The notebook includes suggestions for what to try.
- Explore the use of **TensorBoard** – optionally run it locally on your own computer to visualize training progress.

Further exercises:

- Train a MLP on other datasets from the scikit-learn dataset repository (e.g., **iris**, **diabetes**, **wine**).
- Try reproducing the full workflow from scratch — data loading, preprocessing, model creation, compilation, training, and evaluation.

Practice: Explore and Experiment

Interactive MLP Playground – Simple Visual Demos
https://playground.tensorflow.org/

- Interactive demo illustrating how a multilayer perceptron learns.
- Includes five classification tasks (of increasing difficulty – the spiral task is the most challenging).
- You can configure network architecture and training parameters.
- Excellent visualizations: loss over time, weight signs and magnitudes, neuron behavior.
- Great for experimenting: how many layers and neurons are needed for which task?
- **Challenge:** can you train a model that solves the spiral task?

Note: We explored this tool in detail in the previous semester (18NES1).

1st Graded Homework: MLP Training in Keras (1 point)

Task:

- Train and evaluate a MLP in Keras on a small dataset from scikit-learn.
- You can use the example from the lecture (**keras_extended_example.ipynb**) as a template and reproduce the full workflow: data loading, preprocessing, model creation, compilation, training, and evaluation.

Choose one of the following datasets:

- **Classification:**
  - `load_iris()` – 3 classes, 4 features (very small, clean)
  - `load_wine()` – 3 classes, 13 features (slightly richer)
- **Regression:**
  - `load_diabetes()` – 10 features, small sample (classic toy regression)
  - `fetch_california_housing()` – 8 features, larger sample

1st Graded Homework: MLP Training in Keras (1 point)

Requirements

- You can start from the provided example notebook and adapt it to your chosen dataset.
- Ensure correct preprocessing (e.g., `StandardScaler`); for classification report accuracy (and optionally F1/confusion matrix); for regression report MSE (and optionally MAE and R^2).
- Show training curves (loss/metric) and briefly discuss under/overfitting.
- Experiment with at least **three** hyperparameter changes (layers/units, activation, optimizer, learning rate, etc.).
- Provide a short summary of your findings in the notebook.

Submission:

- Email your **solution notebook** (.ipynb) by **Oct 14, 2025**.
- **Consultation (during lab or individually) required by Oct 17, 2025 to receive points.**