Neural Networks 1 - Multilayer neural networks

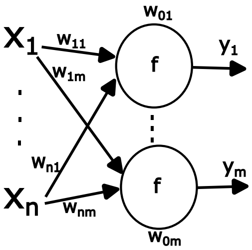
18NES1 - Lecture 6, Summer semester 2024/25

Zuzana Petříčková

April 1st, 2025

What We Covered Last Time

1. Single-layer neural network
   - Multivariate linear regression (linear neural network)
   - Multiclass linear classification / pattern recognition (single-layer perceptron)



1. Multi-layer neural network (MLP)
2. Backpropagation algorithm

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

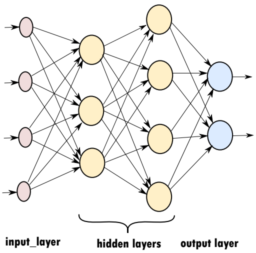
Special input layer:

- corresponds to the inputs of the neural network

Output layer:

- the output (response) of the network corresponds to the activations of the output neurons

The remaining layers are **hidden layers**.



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 (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):

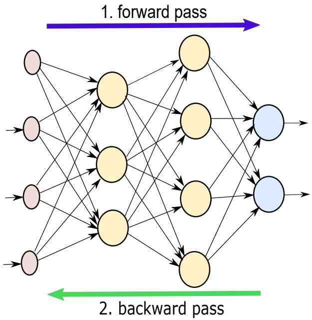
$$w_i(t+1) = w_i(t) - \alpha_t \frac{\partial E_t}{\partial w_i}$$

Nice visualizations of loss surfaces:
jithinjk.github.io/blog/nn_loss_visualized.md.html
izmailovpavel.github.io/curves_blogpost

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)



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** – Optimized for speed and experimental research.
- Previously popular **Theano** – now deprecated.

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** – learning visualization, especially for TensorFlow.

This Week

1. Finalizing examples using Python frameworks for neural networks
2. A brief overview of local library installation
3. Examples in TensorFlow Playground
4. Brief analysis of a multi-layer neural network model with the backpropagation algorithm
5. Step-by-step example using Keras on a sample task (binary classification). Setting hyperparameters and understanding their impact on the training process. Using TensorBoard.

Practical Examples

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.

Interactive MLP Playground – Simple Visual Demos

https://playground.tensorflow.org/

- Five classification tasks (of increasing difficulty – spiral is the hardest).
- You can configure network architecture and training parameters.
- Includes 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?

Multi-layer Neural Network Trained via Gradient Descent – Model Analysis

Advantages:

- A simple and **universal** model with solid approximation capabilities
  - Suitable for both classification and regression tasks, including time series prediction.
  - Capable of capturing complex nonlinear relationships.
  - Generalizes well.
- Uses backpropagation for efficient training via gradient descent.
- A universal approximator – capable of approximating any continuous function (for certain nonlinear activation functions, one or two hidden layers suffice). However, the training problem is NP-complete.

NP-completeness of the Training Problem

Theorem

- The general problem of training artificial neural networks is NP-complete. The computational complexity grows exponentially with the number of parameters.

Remarks

1. The theorem holds even for training multi-layer neural networks and for learning logical functions.
2. For some specific types of simple neural networks, the learning problem is solvable in polynomial time (e.g., via linear programming methods).

$\rightarrow$ Therefore, we must rely on local optimization methods (e.g., gradient descent).

Multi-layer Neural Network Trained via Gradient Descent – 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 – may converge to suboptimal solutions.
- Prone to overfitting – mitigated by regularization, early stopping, etc.
- No built-in mechanisms for capturing spatial data structure.
- "Black box" – the internal knowledge representation (weights and biases) is difficult for humans to interpret.

Multi-layer Neural Network Trained via Gradient Descent – Model Analysis

We want training via local (gradient-based) methods to be successful:

- Fast learning (convergence)
- The ability to learn the task (correctly capture hidden patterns in the data)
- Good generalization (accurate outputs for unseen inputs)

What is essential for the success of backpropagation?

- Proper preprocessing of training data
- Good initialization of weights and biases (e.g., $\sim N(0, 1)$)
- Careful tuning of hyperparameters for the specific task

Example

keras_simple_example.ipynb

- A more 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.

Preprocessing Training Data for MLP

Key preprocessing steps:

- **Serialization:**
  - Convert input and output data into 2D tensors of shape (samples, numerical features)
  - Handle categorical variables (e.g., ordinal encoding, one-hot encoding)
- **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 for MLP

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 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, ...

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.

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.

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

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.

Why zero mean?

- Ensures expected input to each neuron is centered at zero
- Derivative of sigmoid/tanh is maximal near zero (~ 0.25) $\rightarrow$ faster learning at start
- Reduces the risk of saturation

Why random?

- Breaks symmetry – hidden neurons should not perform identical computations

Proper Initialization of Weights and Biases

For example, using common heuristics:

- Basic idea: $w_{ij}(0) \sim N(0, 1)$
- Nguyen-Widrow method: distributes neuron weights more evenly across the input space
- Glorot et al. (2010): $w_{ij}(0) \sim N\left(0, \sqrt{\frac{6}{n_i+n_j}}\right)$ for weight matrix of shape $n_i \times n_j$; aims to preserve output variance across layers

Recommendations:

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

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

The Problem of Neuron Saturation

- If the weights and biases are too small, the propagated error is also too small and the learning is very slow.
- On the other hand, if the weights are too large, neurons may become **saturated**:
  - Neurons remain constantly highly active or inactive for all training examples, and their outputs no longer change with input.
    The derivative of the activation function is near zero.
    $\rightarrow$ **Network paralysis** and uncontrolled growth of weights.

How to reduce the risk of saturation?

- Use **ReLU** instead of **tanh** in hidden layers
- Normalize the training data; consider additional normalization techniques (batch normalization, layer normalization)
- Use proper weight initialization
- Reduce the learning rate or use an optimizer with adaptive learning rate (e.g., Adam)

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:

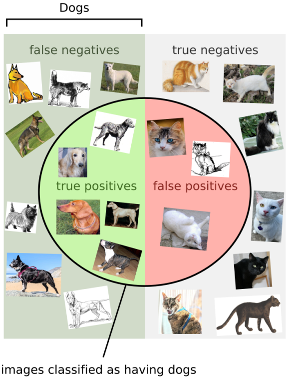
- **Accuracy** – proportion of correctly classified samples
  - **Binary classification: accuracy**, **binary_accuracy**
  - **Multi-class classification: categorical_accuracy** (for one-hot labels), **sparse_categorical_accuracy** (for integer labels)
- **Other metrics for binary classification:**
  - **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?



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

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

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

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 → slow learning, risk of getting stuck in a local minimum
- Too large → 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)

→ **More on optimizers in the next lecture**

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
→ 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

Example

keras_simple_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.
- Try out TensorBoard – optionally also run it locally on your own computer.
- Optionally modify the code and try training an MLP on other datasets from the scikit-learn dataset repository (e.g., **iris**, **diabetes**, **wine**).