

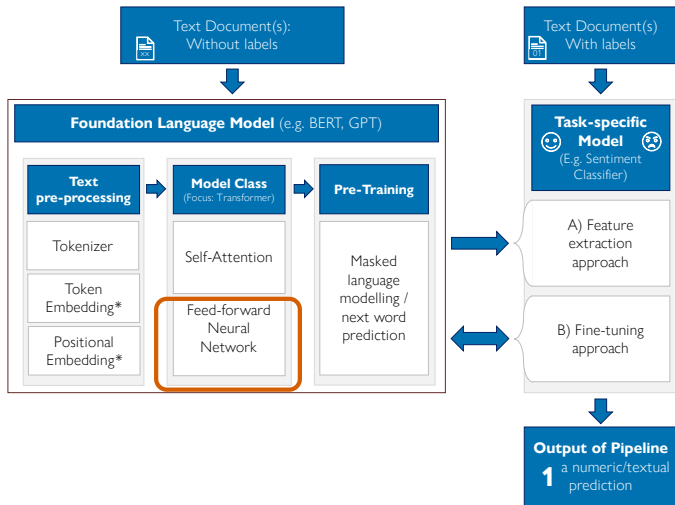
Understanding and Training Language Models: Statistical Learning and Neural Networks

Erik-Jan Senn

Faculty of Mathematics and Statistics, University of St. Gallen

CSH Autumn School at University of Hohenheim
September/October 2024

Where are we?



Language Modelling Pipeline

Roadmap

Introduction to Statistical Learning

Neural Networks

Statistical Learning and Neural Networks

A Tiny Introduction to Statistical Learning

Supervised Learning (Simplified)

Goal: Find a function $f : X \rightarrow Y$ from labeled training data (x_i, y_i) that makes good predictions of y for previously unseen data samples from the population distribution.

Supervised Learning (Simplified)

Goal: Find a function $f : X \rightarrow Y$ from labeled training data (x_i, y_i) that makes good predictions of y for previously unseen data samples from the population distribution.

- ▶ The **model** (hypothesis class) $f(X; \theta) \in \mathcal{H}$: A learnable parameterized function used to make predictions $f(\mathbf{x}; \hat{\theta}) = \hat{y}$.
- ▶ The **loss function** $\mathcal{L}(Y, \hat{Y})$ evaluates how good a prediction is.
- ▶ The **learning algorithm** estimates the optimal model parameters $\hat{\theta}$ that minimize the *sample loss*, instead of minimizing the *unobservable population loss* (empirical risk minimization).

Example: Logistic Regression for Tumor Classification

Setting

- ▶ $X \in \mathbb{R}^k$: Tumor features (e.g., radius, texture, perimeter, area, smoothness)
- ▶ $Y \in \{0, 1\}$: Tumor is benign (0) or malignant (1)
- ▶ Training sample: N tumor patients draw i.i.d. from the population of all tumor patients.

Example: Logistic Regression for Tumor Classification

Setting

- ▶ $X \in \mathbb{R}^k$: Tumor features (e.g., radius, texture, perimeter, area, smoothness)
- ▶ $Y \in \{0, 1\}$: Tumor is benign (0) or malignant (1)
- ▶ Training sample: N tumor patients draw i.i.d. from the population of all tumor patients.

Supervised learning

- ▶ Loss function for binary classification

$$\mathcal{L}(Y, \hat{Y}) = -\mathbb{E}_{(X, Y)} \left[(Y \log(p(\hat{Y} = 1))) + (1 - Y) \log(p(\hat{Y} = 0)) \right]$$

- ▶ Logistic model for conditional probabilities

$$f(\mathbf{x}, \theta = (\mathbf{w}, b)) = \sigma(\mathbf{w}^T \mathbf{x} + b) = p(y = 1 | \mathbf{x}),$$

with the sigmoid function $\sigma(z) = \frac{1}{1 + e^{-z}}$

- ▶ Learning algorithm: Any optimizer such as gradient descent to solve this (convex) optimization problem.

$$\min_{\mathbf{w}, b} \mathcal{L}(\mathbf{w}, b) = -\frac{1}{N} \sum_{i=1}^N \left[y_i \log(\sigma(\mathbf{w}^T \mathbf{x}_i + b)) + (1 - y_i) \log(1 - \sigma(\mathbf{w}^T \mathbf{x}_i + b)) \right]$$

A Learning Algorithm: Gradient Descent

Requirements

- ▶ \mathcal{L} is **differentiable** with respect to θ and gradient is sufficiently smooth.
- ▶ **Convexity** of \mathcal{L} to find global minimum

A Learning Algorithm: Gradient Descent

Requirements

- ▶ \mathcal{L} is **differentiable** with respect to θ and gradient is sufficiently smooth.
- ▶ **Convexity** of \mathcal{L} to find global minimum

The algorithm

- ▶ Pick step size (learning rate) η , tolerance ϵ and starting values θ_0 .
- ▶ **Iteratively update** the θ by taking a **step** in the direction of the **negative gradient** of the loss on the full **training sample**:

$$\theta_{k+1} := \theta_k - \eta \nabla_{\theta} \mathcal{L}(y, \hat{y}(\theta_k))$$

where $\nabla_{\theta} \mathcal{L}(y, \hat{y}(\theta_k))$ is the empirical gradient of the loss with respect to the model parameters.

- ▶ Stop when e.g. when gradients do not change much $\|\nabla_{\theta} \mathcal{L}(y, \hat{y}(\theta_k))\| < \epsilon$.

A (slightly adapted) Learning Algorithm: Stochastic Gradient Descent

Requirements

- ▶ \mathcal{L} is differentiable with respect to θ and gradient is sufficiently smooth.
- ▶ Convexity of \mathcal{L} to find global minimum.
SGD can avoid some local minima without convexity, but no guarantees.

The algorithm

- ▶ Pick step size (learning rate) η , tolerance ϵ and starting values θ_0 .
- ▶ Iteratively update the θ by taking a step in the direction of the negative gradient of the loss on a single (random) observation of the training sample:

$$\theta_{k+1} := \theta_k - \eta \nabla_{\theta} \mathcal{L}(y, \hat{y}(\theta_k))$$

where $\nabla_{\theta} \mathcal{L}(y, \hat{y}(\theta_k))$ is the empirical gradient of the loss with respect to the model parameters.

- ▶ Stop when e.g. when gradients do not change much $\|\nabla_{\theta} \mathcal{L}(y, \hat{y}(\theta_k))\| < \epsilon$.
Sometimes faster than GD, e.g. with large datasets.

Example: Logistic Regression with Gradient Descent

Can we learn the best $f_{\text{logistic}}(\theta)$ using (stochastic) gradient descent?

- ⊕ \mathcal{L} is differentiable with respect to θ .
- ⊕ Convexity of \mathcal{L} to find global minimum.

Example: Logistic Regression with Gradient Descent

Can we learn the best $f_{\text{logistic}}(\theta)$ using (stochastic) gradient descent?

⊕ \mathcal{L} is differentiable with respect to θ .

⊕ Convexity of \mathcal{L} to find global minimum.

Yes!

Update rule:

$$\mathbf{w}_{k+1} := \mathbf{w}_k - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{w}_k}, \quad b_{k+1} := b_k - \eta \frac{\partial \mathcal{L}}{\partial b_k}$$

where the gradients for one observation are:

$$\frac{\partial \mathcal{L}}{\partial w_k} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} \cdot \frac{\partial \mathcal{L}}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial (\mathbf{w}^T \mathbf{x}_i + b)} \cdot \frac{\partial (\mathbf{w}^T \mathbf{x}_i + b)}{\partial w_k} = (\hat{y}_i - y_i) \cdot x_{i,k}$$
$$\frac{\partial \mathcal{L}}{\partial b} = (\hat{y}_i - y_i) \cdot 1$$

Example: Logistic Regression with Gradient Descent

Can we learn the best $f_{\text{logistic}}(\theta)$ using (stochastic) gradient descent?

⊕ \mathcal{L} is differentiable with respect to θ .

⊕ Convexity of \mathcal{L} to find global minimum.

Yes!

Update rule:

$$\mathbf{w}_{k+1} := \mathbf{w}_k - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{w}_k}, \quad b_{k+1} := b_k - \eta \frac{\partial \mathcal{L}}{\partial b_k}$$

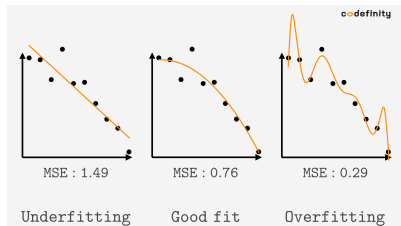
where the gradients for one observation are:

$$\frac{\partial \mathcal{L}}{\partial w_k} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} \cdot \frac{\partial \mathcal{L}}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial (\mathbf{w}^T \mathbf{x}_i + b)} \cdot \frac{\partial (\mathbf{w}^T \mathbf{x}_i + b)}{\partial w_k} = (\hat{y}_i - y_i) \cdot x_{i,k}$$
$$\frac{\partial \mathcal{L}}{\partial b} = (\hat{y}_i - y_i) \cdot 1$$

The gradients are computed using the *chain rule*: $\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$.

Overfitting

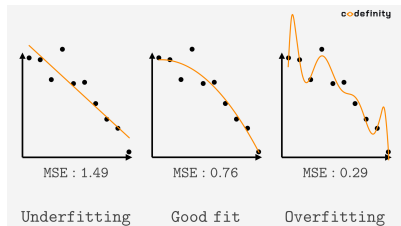
Overfitting: The model fits the training data *too well*, leading to *worse performance* on unseen (test) data.



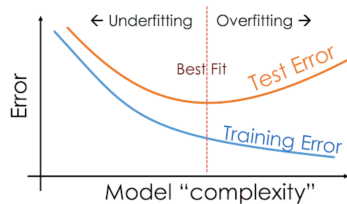
Predictions and true values [Source](#)

Overfitting

Overfitting: The model fits the training data *too well*, leading to *worse performance* on unseen (test) data.



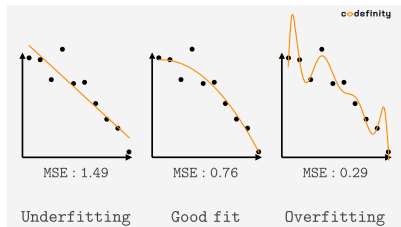
Predictions and true values [Source](#)



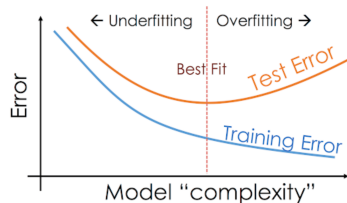
Loss on training and test data [Source](#)

Overfitting

Overfitting: The model fits the training data *too well*, leading to *worse performance* on unseen (test) data.



Predictions and true values [Source](#)



Loss on training and test data [Source](#)

We want the best model for unseen test data.

Therefore, controlling for overfitting is very important - we use **regularization**.

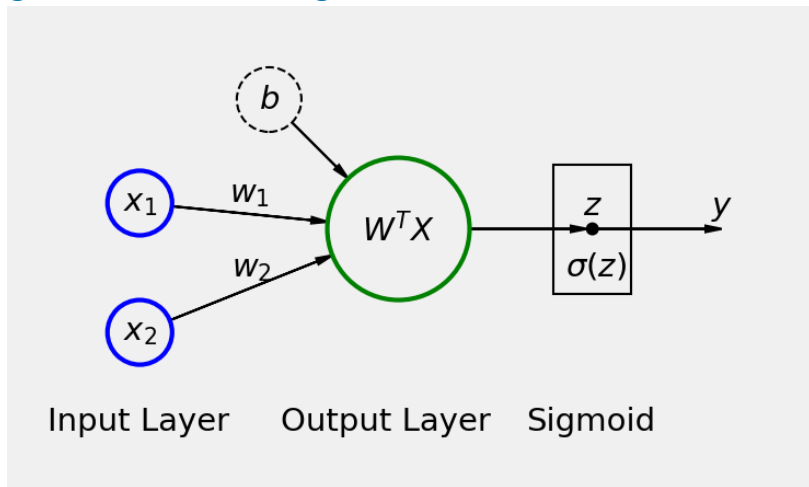
Statistical Learning and Neural Networks

Neural Networks

Neural Networks

- ▶ (Artificial) neural networks are statistical models inspired by the functionality of the human brain.
- ▶ Feed-forward neural networks are **universal function approximators**:
Any continuous function can be approximated up to arbitrary precision by a feed-forward neural network.
- ▶ The flexible functional form allows to model for **complex non-linear relationships**.

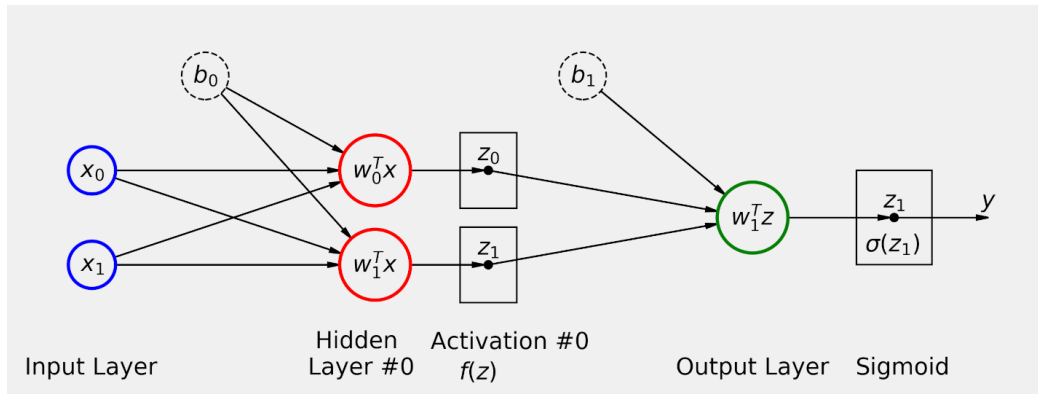
Logistic Regression as Starting Point



Logistic regression for binary classification.

[Source](#)

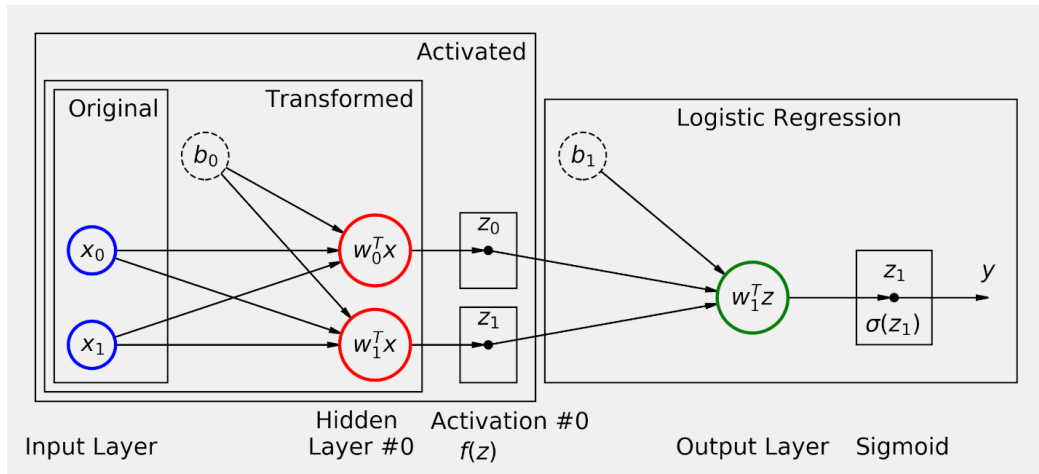
Multilayer Perceptron



Multilayer perceptron for binary classification with one hidden layer.

[Source](#)

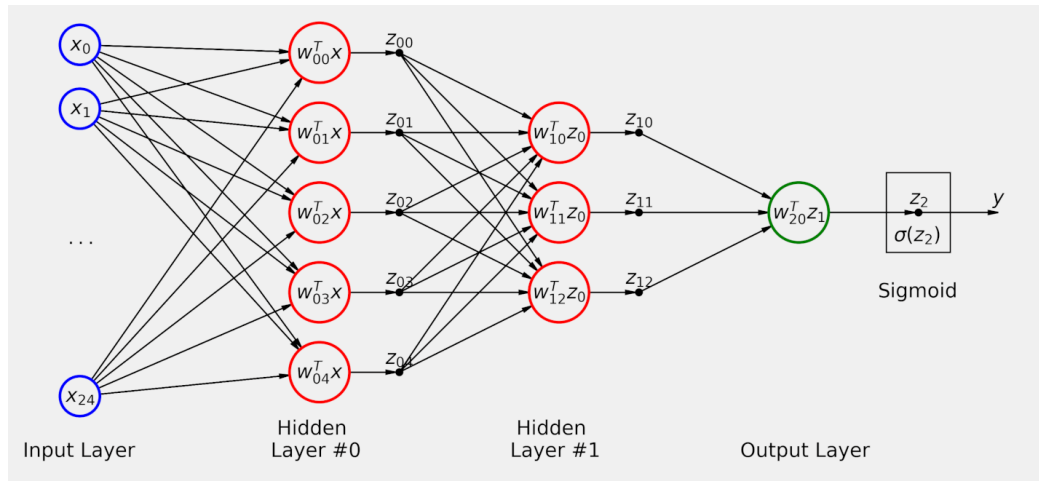
Multilayer Perceptron



Multilayer perceptron for binary classification with one hidden layer.

[Source](#)

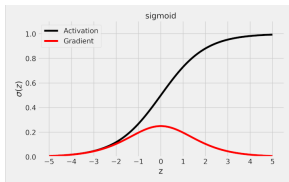
Multilayer Perceptron



Multilayer perceptron for binary classification with 24 input neurons, two hidden layers and more hidden neurons. [Source](#)

Activation Functions

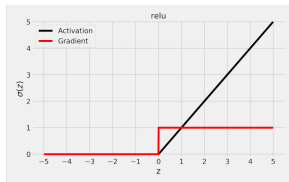
Activation functions conduct a **non-linear** transformation at each neuron in the hidden layers.



Sigmoid function [Source](#)

$$g(x) = \sigma(x) = \frac{1}{1+e^{-x}}$$

$$\text{Gradient: } \frac{d}{dx} g(x) = \sigma(x)(1 - \sigma(x))$$



Rectified linear unit (ReLU) function [Source](#)

$$g(x) = \max(0, x)$$

$$\text{Gradient: } \frac{d}{dx} g(x) = \mathbb{1}_{x>0} \quad \forall x \neq 0$$

Multilayer Perceptron with one Hidden Layer - Mathematical Formulation of the Forward Pass

A multilayer perceptron with one hidden layer $f : \mathbb{R}^{N \times K} \rightarrow \mathbb{R}^N$ can be expressed as:

$$\hat{y} = f(\mathbf{X}) = g_1(g_0(\mathbf{X}\mathbf{W}_0 + \mathbf{b}_0)\mathbf{W}_1 + \mathbf{b}_1)$$

- ▶ N , the number of observations,
- ▶ K , the number of features,
- ▶ N_Z , the number of nodes in the hidden layer,
- ▶ \mathbf{X} , a $N \times K$ matrix of data inputs,
- ▶ \mathbf{y} , a column vector of targets with N elements,
- ▶ \mathbf{W}_0 (\mathbf{W}_1), a $K \times N_Z$ ($N_Z \times 1$) **weight matrix** (slope parameters) which *passes* the input data to the hidden layer (hidden representation to the output layer) before activation,
- ▶ \mathbf{b}_0 (\mathbf{b}_1) which is column vector with N_Z (1) elements of additive **biases** (intercepts) applied before activation,
- ▶ $g_0 : \mathbb{R} \rightarrow \mathbb{R}$ ($g_1 : \mathbb{R} \rightarrow \mathbb{R}$), an activation function that is applied element-wise to the output of the first layer (second layer).

This model has $K \cdot N_Z + N_Z \cdot 1 + N_Z + 1$ trainable parameters (from \mathbf{W}_0 , \mathbf{W}_1 , \mathbf{b}_0 , \mathbf{b}_1).

Training the Neural Network

Can we learn the best f^{NN} using (stochastic) gradient descent?

- ⊕ \mathcal{L} is differentiable with respect to θ .
- ⊖ No convexity of \mathcal{L} to find the global minimum.

Training the Neural Network

Can we learn the best f^{NN} using (stochastic) gradient descent?

- ⊕ \mathcal{L} is differentiable with respect to θ .
- ⊖ No convexity of \mathcal{L} to find the global minimum.

Hopefully its good enough.

We try to avoid local minima and saddle points e.g. by using SGD, different starting values, learning rate schedulers, ...

Training the Neural Network

Can we learn the best f^{NN} using (stochastic) gradient descent?

- ⊕ \mathcal{L} is **differentiable** with respect to θ .
- ⊖ **No convexity** of \mathcal{L} to find the global minimum.

Hopefully its good enough.

We try to avoid local minima and saddle points e.g. by using SGD, different starting values, learning rate schedulers, ...

Gradient computation uses the **backpropagation** algorithm, an efficient version of the *multivariate chain rule*, to compute gradients propagating step-by-step from output (the *back*) to input.

Training the Neural Network

Can we learn the best f^{NN} using (stochastic) gradient descent?

- ⊕ \mathcal{L} is differentiable with respect to θ .
- ⊖ No convexity of \mathcal{L} to find the global minimum.

Hopefully its good enough.

We try to avoid local minima and saddle points e.g. by using SGD, different starting values, learning rate schedulers, ...

Gradient computation uses the backpropagation algorithm, an efficient version of the *multivariate chain rule*, to compute gradients propagating step-by-step from output (the *back*) to input.

Useful resources if time allows:

- ▶ Slides by Geiger (2024) on backpropagation: [Link](#)
- ▶ Interactive visualization of how neural networks learn: [Link](#)

Forward pass, backward pass and computation

Training

- ▶ **Forward pass:** $f(\mathbf{x}, \theta) \rightarrow y$
- ▶ **Backward pass:** (first-order) gradient computation *from the back* using **backpropagation**
- ▶ **Update** $\hat{\theta}$ according to optimizer step-size (and specifics).

Forward pass, backward pass and computation

Training

- ▶ **Forward pass:** $f(\mathbf{x}, \theta) \rightarrow y$
 - ▶ The output value at each neuron (*activation*) in one layer is computed (in parallel) and propagated forward, layers are computed sequentially.
 - ▶ At the same time, autograd creates the **computational graph** (DAG) and the activations ($z_{i,j}$) are saved (if required for backward pass).
 - ▶ Note for torch: Gradient can be computed only if `Tensor.requires_grad` is `True`. Required to later allow backpropagation through this tensor. Not only for trainable parameters!
- ▶ **Backward pass:** (first-order) gradient computation *from the back* using **backpropagation**
- ▶ **Update** $\hat{\theta}$ according to optimizer step-size (and specifics).

Forward pass, backward pass and computation

Training

- ▶ **Forward pass:** $f(\mathbf{x}, \theta) \rightarrow y$
- ▶ **Backward pass:** (first-order) gradient computation *from the back* using **backpropagation**
 - ▶ autograd reverts the order of computation from the forward pass by sequentially stepping *from back to front layers* and computing gradients by the (in parallel for the same layer), using the stored activations.
 - ▶ In `torch`, the gradients of components are supplied by the `backward` methods, which are implemented for most standard functions (e.g. *ReLU* (0/1), *Sigmoid* ($\sigma(1 - \sigma)$), *linear slope parameters* $w(x_i)$). For custom functions, need to define gradient manually (as symbolic or numeric derivative).
- ▶ **Update** $\hat{\theta}$ according to optimizer step-size (and specifics).

Forward pass, backward pass and computation

Training

- ▶ **Forward pass:** $f(\mathbf{x}, \theta) \rightarrow y$
- ▶ **Backward pass:** (first-order) gradient computation *from the back* using **backpropagation**
- ▶ **Update** $\hat{\theta}$ according to optimizer step-size (and specifics).

Inference (means prediction phase in ML) require only forward pass. E.g. for feature extraction from LMs.

Regularization and Hyperparameters

Neural networks can differ in many components:

- ▶ **Regularization**: e.g. number of epochs, early stopping, L1 or L2 norm penalty, dropout. Avoids overfitting, which is important for models with many trainable parameters.
- ▶ **Training**: learning rate, data normalization / transformation / augmentation, weight initialization.
- ▶ **Architectures**: e.g. standard multi-layer perceptron with different number of neurons and layers, convolutional neural networks (mainly for images), recurrent neural networks (time series), **transformer models** (most current LMs).

Regularization and Hyperparameters

Neural networks can differ in many components:

- ▶ **Regularization**: e.g. number of epochs, early stopping, L1 or L2 norm penalty, dropout. Avoids overfitting, which is important for models with many trainable parameters.
- ▶ **Training**: learning rate, data normalization / transformation / augmentation, weight initialization.
- ▶ **Architectures**: e.g. standard multi-layer perceptron with different number of neurons and layers, convolutional neural networks (mainly for images), recurrent neural networks (time series), **transformer models** (most current LMs).

These choices influence:

- ▶ model performance on a certain task,
- ▶ how long it takes to train the model.

Questions ?

References

Andreas Geiger. Deep learning lecture chapter 2 on backpropagation.

<https://drive.google.com/file/d/10lpS7rFXJ-4RS-eLswS9tAAN6tCPF1G3/view>,
2024. Accessed: 27.09.2024.