Deep Learning - WS24/25

Niklas Rodenbüsch

February 17, 2025

1 Introduction

1.1 Representation Learning vs. Deep Learning (DL)

Definition 1:

Representation Learning is a set of methods that allows a machine to be fed with **raw data** and to **automatically discover the representations** needed for detection/classification.

DL is Representation Learning with multiple levels of representations, obtained by **composing simple but nonlinear modules**.

In Deep Learning, features are learned from raw data in an **end-to-end** fashion.

1.2 Machine Learning (ML) Concepts

1.2.1 Linear Regression

Linear regression is a supervised learning algorithm that models the relationship between a dependent variable y and one or more independent variables x using a linear function.

The objective in linear regression is to minimize the **Mean Squared Error (MSE)** loss, given by:

$$\mathcal{L}_{MSE}(\hat{y}, y) = \frac{1}{2N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2,$$

where $\hat{y} = h_\theta(x)$ is the prediction of the model and N is the total number of datapoints in the dataset.

1.3 Logistic Regression

Logistic regression is a classification algorithm that estimates the probability of a binary outcome using a logistic function.

For logistic regression, the objective is to minimize the **binary cross-entropy** loss:

$$\mathcal{L}_{BCE}(\hat{y}, y) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log \hat{y}_i + (1 - y_i)\log(1 - \hat{y}_i),$$

where $\hat{y} = h_\theta(x) = g(w^T x)$ with $g(z) = \sigma(z) = \frac{1}{1+e^{-z}}$ (g is the **sigmoid** function).[1] For the Multi-Class classification case, the loss generalizes to:

$$\mathcal{L}_{CE}(\theta) = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} y_{kn} \log \hat{y}_k(x_n, \theta).$$

[1] The correct formulation of the loss functions would be $\mathcal{L}(\theta)$ or $\mathcal{L}(w)$.

1.3.1 Overfitting and Underfitting

Definition 2:

Overfitting occurs when a model memorizes the training data. This leads to poor generalization on unseen data (e.g. the validation/test set).

Underfitting on the other hand means, that the model is not able to capture the complexity of the training data (e.g. because the model is too small).

1.3.2 Cross-Validation

Cross-validation is a technique for assessing the performance of a model by dividing the dataset into multiple subsets, training the model on some subsets and testing it on others to ensure it generalizes well to unseen data.

1.4 Basics of Neural Networks

1.4.1 Types of Tasks

- **Possible Inputs:** Sound, Text, Images, Graphs
- **Possible Outputs:** Classification, Regression, Sound, Text, Images, ...

1.4.2 Single Neurons

A neuron in a neural network computes the **weighted sum** z **of its inputs** x and performs a **nonlinear transformation** $h(z)$ afterwards:

$$a = h(z) \quad , \quad z = x_1 w_1 + x_2 w_2 + ... + x_n w_n$$

2 Multi-Layer-Perceptrons (MLPs)

2.1 Entropy & Kullback-Leibler (KL)-Divergence

Definition 3:

Information entropy quantifies the uncertainty about a random variable X (assume X has K possible outcomes):

$$H(P) = -\mathbb{E}_{x \sim P} [\log P(x)]$$

$$H(P) = -\sum_{k=1}^{K} P(X = x_k) \log P(X = x_k)$$

Definition 4:

Given a true probability distribution P and a predicted distribution Q, the **Cross-Entropy** of the same random variable X is defined as:

$$H(P, Q) = -\mathbb{E}_{x \sim P} [\log Q(x)]$$

$$H(P, Q) = -\sum_{k=1}^{K} P(x_k) \log Q(x_k)$$

Definition 5:

KL-Divergence is a measure of how one probability distribution P diverges from a second, expected probability distribution Q. The KL-divergence is defined as:

$$D_{\mathrm{KL}}(P \parallel Q) = H(P, Q) - H(P) = -\mathbb{E}_{x \sim P} \left[\log \frac{Q(x)}{P(x)} \right]$$

The KL-Divergence can be interpreted as the **information loss if we use distribution Q to approximate distribution P.**
KL-Divergence is always non-negative, not symmetric and is zero if and only if $P = Q$.

2.2 Maximum Likelihood Estimation (MLE)

MLE is a method for estimating the parameters of a statistical model by maximizing the likelihood function, which represents the probability of observing the given data under various parameter values (in other words: the model "fitting" itself to match the observed data). The MLE is defined as:

$$\theta_{\mathrm{MLE}} = \arg\max_{\theta} p_{\mathrm{model}}(\mathbb{X}; \theta) = \arg\max_{\theta} \prod_{i=1}^{n} p(x_i \mid \theta) = \arg\max_{\theta} \sum_{i=1}^{n} \log p(x_i \mid \theta),$$

where $\mathbb{X} = \{x_1, ..., x_n\}$ is a set of n samples drawn from the unknown data-generating distribution p_{data}, and $p_{\mathrm{model}}(x_i \mid \theta)$ is the probability of observing the data point x_i given the parameters θ.

2.3 Multi-Layer-Perceptron

The MLP extends the simple Perceptron by adding (multiple) fully-connected hidden layers between the input and the output layer. The forward pass is computed the following way (assume the network has one hidden layer):

$$\hat{y} = g^{(2)}(W^{(2)^T} g^{(1)}(W^{(1)^T} x + b^{(1)}) + b^{(2)})$$

Note: The weight matrices in the first layer are of shape $(\mathrm{dim\_x} \times \mathrm{n\_neurons})^T$.

Theorem 1:

The Universal Function Approximation Theorem states that:

1. **Any boolean function** can be realized by an MLP with one hidden layer.
2. **Any bounded continouos function** can be approximated with arbitrary precision by a MLP with one hidden layer.

Note: The theorem does not show that any function can be learned from data!

Theorem 2:

A neural network with n_0 inputs and K layers of n units each, with Rectified Linear Unit (ReLU) activations can represent functions that have $\Omega((\frac{n}{n_0})^{(K-1)n_0} n^{n_0})$ linear regions.

2.4 Activation Functions

- **Linear:** $h_{\text{linear}}(z) = z$
- **Hyperbolic Tangent:** $h_{\tanh}(z) = \tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)} \in [-1; 1]$
- **ReLU:** $h_{\text{relu}}(z) = \max(0, z)$
- **Parametric ReLU (PReLU):** $h_{\text{relu}}(z) = \begin{cases} z, & z > 0 \\ az, & z \leq 0 \end{cases}$, where $a > 0$ is a learnable parameter.
- **Exponential Linear Unit (ELU):** $h_{\text{elu}}(z) = \begin{cases} z, & z > 0 \\ \alpha(\exp(z) - 1), & z \leq 0 \end{cases}$, with $\alpha > 0$.
- **Gaussian Error Linear Unit (GELU):** $h_{\text{gelu}}(z) = z\phi(z)$, where ϕ is the CDF.
- **Swish:** $h_{\text{swish}}(z) = z\sigma(\beta z)$, where $\beta \geq 0$ is constant or a trainable parameter.
- **Softmax:** $h_{\text{softmax}}(z) = \frac{\exp(z)}{\sum_j \exp(z_j)} = p(y_j = 1) \in [0; 1]$, for MCC output layer

3 Backpropagation

Definition 6:

The **chain rule** computes the derivatives for **compositions** of functions $g(x)$ and $f(y) = f(g(x)) = z$ as:

$$(f \circ g)'(x) = (f(g(x)))' = f'(g(x)) \cdot g'(x)$$

$$\frac{\delta z}{\delta x} = \frac{\delta z}{\delta y}\frac{\delta y}{\delta x} = \frac{\delta f(g(x))}{\delta g(x)}\frac{\delta g(x)}{\delta x}$$

Generalizing this formulation to the multidimensional case with leads us to:

$$\frac{\delta z}{\delta x_i} = \sum_j \frac{\delta z}{\delta y_j}\frac{\delta y_j}{\delta x_i}$$

where $x \in \mathbb{R}^m, y \in \mathbb{R}^n, g : \mathbb{R}^m \to \mathbb{R}^n, f : \mathbb{R}^n \to \mathbb{R}$ and $y = g(x)$ and $z = f(y)$. Transforming this formulation into vector notation gives us:

$$\nabla_x z = (\frac{\delta y}{\delta x})^T \nabla_y z,$$

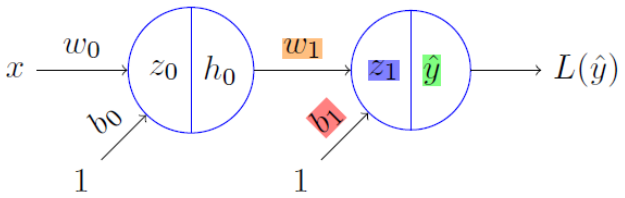
where $\frac{\delta y}{\delta x}$ is the $n \times m$ Jacobian matrix of g, and $\nabla_y z$ being the gradient of z with respect to vector y. Since $y \in \mathbb{R}^n$ this leaves us with:

$$\nabla_y z = \left[\frac{\delta z}{\delta y_1}, \frac{\delta z}{\delta y_2}, ..., \frac{\delta z}{\delta y_n}\right]^T$$

3.1 Derivations of Activations

- **Linear:** $h'(z) = 1$
- **Sigmoid:** $h'(z) = h(z)(1 - h(z))$
- **Tanh:** $h'(z) = 1 - h(z)^2$
- **ReLU:** $h'(z) = \begin{cases} 1, z > 0 \\ 0, z \leq 0 \end{cases}$

3.2 Backpropagation in Neural Networks

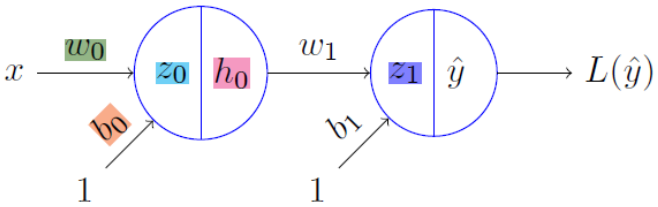


Forward pass:

$$
\begin{aligned}
L &= \frac{1}{2}(\hat{y} - y)^2 \\
\hat{y} &= g_1(z_1) = z_1 \\
z_1 &= w_1 h_0 + b_1
\end{aligned}
$$

Backward pass:

$$
\begin{aligned}
\frac{\partial L}{\partial \hat{y}} &= \hat{y} - y \\
\frac{\partial L}{\partial z_1} &= \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_1} = (\hat{y} - y) \cdot g_1'(z_1) = (\hat{y} - y) \cdot 1 \\
\frac{\partial L}{\partial w_1} &= \frac{\partial L}{\partial z_1} \frac{\partial z_1}{\partial w_1} = \frac{\partial L}{\partial z_1} h_0 \\
\frac{\partial L}{\partial b_1} &= \frac{\partial L}{\partial z_1} \frac{\partial z_1}{\partial b_1} = \frac{\partial L}{\partial z_1} \cdot 1
\end{aligned}
$$



Forward pass:

$$
\begin{aligned}
h_0 &= g_0(z_0) = \begin{cases} z_0 & \text{if } z_0 > 0 \\ 0 & \text{if } z_0 \leq 0 \end{cases} \\
z_0 &= w_0 x + b_0
\end{aligned}
$$

Backward pass:

$$
\begin{aligned}
\frac{\partial L}{\partial h_0} &= \frac{\partial L}{\partial z_1} \frac{\partial z_1}{\partial h_0} = \frac{\partial L}{\partial z_1} w_1 \\
\frac{\partial L}{\partial z_0} &= \frac{\partial L}{\partial h_0} \frac{\partial h_0}{\partial z_0} = \begin{cases} \frac{\partial L}{\partial h_0} & \text{if } z_0 > 0 \\ 0 & \text{if } z_0 \leq 0 \end{cases} \\
\frac{\partial L}{\partial w_0} &= \frac{\partial L}{\partial z_0} \frac{\partial z_0}{\partial w_0} = \frac{\partial L}{\partial z_0} x \\
\frac{\partial L}{\partial b_0} &= \frac{\partial L}{\partial z_0} \frac{\partial z_0}{\partial b_0} = \frac{\partial L}{\partial z_0} \cdot 1
\end{aligned}
$$

Note: The gradient of a skip connection is computed by summing the incoming derivatives of both the forward and skip connection.

4 Optimization

4.1 Optimization vs. Learning

The goal of optimization in DL is to maximize the performance on some measure P. Since P might be intractable (0/1 accuracy not differentiable), we introduce a surrogate loss function $\mathcal{L}$, which is **tractable, differentiable** and **might enable better generalization**.
In practice, optimizing $\mathcal{L}$ directly may not generalize well (overfitting). In order to mitigate this, we can add regularization terms to the loss function.

Definition 7:

Learning in contrast is distinguished from pure optimization by the following points:

- May have to use surrogate instead of the real objective such as softmax instead of hard-max.
- Optimization on partial data distribution instead of the real distribution.
- Often introduces regularization term to generalize.
- Requires the summation over given data points, so computationally expensive for a large dataset.

4.2 Gradient-Based Optimization

In order to minimize the loss of the model, we need to find a global minimum of $\mathcal{L}$. Even though this could be solved analytically (the gradient is 0 at a minimum), this is not feasible in practice. Instead, we use the numerical (iterative) method of **Gradient Descent**.
The algorithm is described in the following figure:

Gradient descent approach:

Require: mathematical function f, learning rate $\alpha > 0$
Ensure: returned vector is close to a local minimum of f

1: choose an initial point $\boldsymbol{x}$
2: **while** $||\nabla f(\boldsymbol{x})||$ not close to 0 **do**
3: $\quad \boldsymbol{x} \leftarrow \boldsymbol{x} - \alpha \nabla f(\boldsymbol{x})$
4: **end while**
5: **return** $\boldsymbol{x}$

4.3 Ill-Conditioning

The condition number $\kappa(Q)$ of a matrix Q with eigenvalues λ is defined as $\kappa(Q) = \frac{\lambda_{\max}}{\lambda_{\min}}$.
Matrices with large condition number are called **ill-conditioned**.
Ill-conditioned matrices drastically hurt convergence performance by causing **zig-zag behavior** (one dimension contributes more than the others)!

There are two methods to mitigate this issue in practice: **momentum** and **preconditioning**.

Definition 8:

Momentum takes into account the direction of the previous gradients by adding an extrapolation step (β) to the gradient step:

$$x_{k+1} = x_k - \alpha \nabla f(x_k) + \beta(x_k - x_{k-1}) = x_k - \alpha \sum_{i=0}^{k} \beta^{k-i} g_i$$

Definition 9:

Preconditioning makes use of a **preconditioning matrix** D_k (positiv definite and symmetric) to pre-multiply the gradients by D_k in order to scale the gradients:

$$x_{k+1} = x_k - \alpha D_k \nabla f(x_k)$$

Note: The best scaling is the inverse of the Hessian matrix $D_k = \nabla^2 f(x_k)^{-1}$ (Newton's method). The problem with Newton's method is that it is also attracted to maximizers.

4.4 Challenges of Optimization

- **Non-Convexity:** The model m is a highly non-convex function (saddle points very likely).
- **Large datasets:** In modern datasets, N is large (makes gradient calculation very expensive).
- **High dimensionality:** In modern neural networks, d is very large.
- **Structure of the network:** Can yield hard-to-optimize response surfaces.

Furthermore, the following properties in the structure of neural networks make optimizations harder:

1. **Cliff-like surfaces in the searching space:** he gradients suddenly drop or rise signifcantly at some points, especially in Recurrent Neural Networks (RNNs).
2. **Gradient vanishing and exploding:** Since we have to multiply matrices again and again, each value is going to be very small or large by the power factor.

4.5 Stochastic Gradient Descent (SGD)

While standard gradient descent handles learning by epoch, SGD computes the loss value using only a part of the dataset. Note that learning by epoch is to learn all the training data at the same time. Typically, the optimization by gradient descent is stable, but it takes a lot more time. Therefore, we use so-called **mini-batches**, i.e. a part of the dataset, for one update step. SGD stochastically picks a batch of data points and learns the batch at each step.
SGD is preferred due to the following reasons:

1. Much quicker training allows much more iterations.
2. Since the standard error of empirical loss function against true loss function decreases by the order of $O(\frac{1}{\sqrt{B}})$ where B is the batch size, a larger batch size does not make a big difference.
3. The training dataset can include many similar data points, so mini-batch learning is likely to yield similar results to that of full-set training.

4. Since mini-batch includes less data points and the standard error is larger, the neural networks are not likely to overfit the statistics of the given data distribution.

Note: We typically want to set the batch size as large as possible given memory constraints. However, small batch sizes can have a regularizing effect (small batch size requires lower learning rates). In order to mitigate biases of the data collection process, we randomly shuffle the training data before drawing batches.

4.6 Learning Rate Schedules

Setting the learning rate α correctly is very important for good model convergence. Too low a learning rate can drastically slow down convergence, while too high a learning rate might lead to divergence.

Generally, using a fixed learning rate is not a good practice, as SGD would never converge. For this reason, dynamic learning rate scheduling is used:

- **Linear decay:** Until iteration τ: $\alpha_k = (1-b)\alpha_0 + b\alpha_\tau$, with $b = k/\tau$. Then constant.
- **Exponential decay:** $\alpha_k = b\alpha_{k-1} = b^k\alpha_0$
- **Step decay:** Decay by a factor (e.g. 10) every n steps.
- **Cosine decay:** $\alpha_k = \frac{1}{2}(1 + \cos(\frac{k}{n}\pi)) \times \alpha_0$, with n being the total number of epochs.
- **SGD with warm restarts (SGDR):** Consists of multiple repeated steps
  1. Quickly cool down α to zero
  2. Heat up α again
  3. Cool down α more slowly.

4.7 SGD with Momentum

As introduced before, momentum v (velocity) is an exponentially moving average of past gradients. This makes updates smoother by keeping a history. Update step:

$$v \leftarrow \beta v - \alpha\hat{g}$$

$$\theta \leftarrow \theta + v$$

Advantages of using momentum are:

- Smoothes zig-zagging
- Accelerates learning at flat spots
- Slows down when signs of partial derivatives change

Disadvantages: additional parameter β, strong momentum might cause more zig-zagging

4.8 Adaptive Gradient Algorithms

The learning rate has to be adapted for convergence. However, to do so may require different learning rates for different dimensions. This is where adaptive gradient algorithms come into play:

1. **AdaGrad:** Update step:
   (a) $r \leftarrow r + \hat{g} \odot \hat{g}$ (accumulate squared gradient)
   (b) $\Delta\theta \leftarrow -\frac{\alpha}{\epsilon+\sqrt{r}} \odot \hat{g}$ (scale α by root of cumulative squared gradient, $\epsilon << 1$)

(c) $\theta \leftarrow \theta + \Delta\theta$ (update model weights)

- High learning rate for dimensions with small gradient.
- Sometimes decreases learning rate too early due to long influences of past gradients.

2. **RMSProp:** Update step:

(a) $r \leftarrow \rho r + (1-\rho)\hat{g} \odot \hat{g}$ (EMA of squared gradient)
(b) $\Delta\theta \leftarrow -\frac{\alpha}{\epsilon+\sqrt{r}} \odot \hat{g}$
(c) $\theta \leftarrow \theta + \Delta\theta$

- ρ is a new hyperparameter controlling how quickly past gradients are forgotten.

3. **Adam:** Update step:

(a) $t \leftarrow t+1$
(b) $s \leftarrow \rho_1 s + (1-\rho_1)\hat{g}$ (EMA of gradient)
(c) $r \leftarrow \rho_2 r + (1-\rho_2)\hat{g} \odot \hat{g}$ (EMA of squared gradient)
(d) $\hat{s} \leftarrow \frac{s}{1-\rho_1^t}$ (correct bias in moving gradient estimate)
(e) $\hat{r} \leftarrow \frac{r}{1-\rho_2^t}$ (correct bias in sq. moving bias estimate)
(f) $\Delta\theta \leftarrow -\frac{\alpha}{\epsilon+\sqrt{\hat{r}}} \odot \hat{s}$
(g) $\theta \leftarrow \theta + \Delta\theta$

- ρ_i are decay rates for first and second moment.
- Extension of RMSProp with momentum and bias correction.

5 Regularization

Usually deep neural networks habe low bias and high variance (meaning that if you change the training data a little the fitted function may change a lot). The standard bias-variance tradeoff is defined by:

$$\text{expected loss} = (\text{bias})^2 + \text{variance} + \text{noise}$$

The goal of regularization is to introduce *some* bias, to substantially reduce variance and therefore to lower the generalization loss.

Definition 10:

Early Stopping stops the training when the error on the validation set has reached its minimum (i.e. as soon as it starts to increase again). The disadvantage is that this need perpetual oberservation of the error.

5.1 Small Weights (Shrinkage Methods)

Idea: Extend the loss function with an extra **regularizer** (penalty) term of strength λ to prevent overfitting:

$$\mathcal{L}(\theta) = \mathcal{L}_D(\theta) + \lambda \mathcal{L}_{reg}(\theta)$$

$$\mathcal{L}_{reg}(\theta) = \frac{1}{q}\sum_{j=1}^{M} |\theta_j|^q \quad \rightarrow \quad \begin{cases} q = 1 : L_1 \text{ regularizer (lasso)} \\ q = 2 : L_2 \text{ regularizer (ridge)} \end{cases}$$

5.2 Decoupled Weight Decay

L_2 regularization is typically implemented by directly changing the gradient to:

$$g' = g + \lambda\theta,$$

and thus changing the SGD update to:

$$\theta \leftarrow \theta - \alpha g' = (1 - \alpha\lambda)\theta - \alpha g$$

Decoupled Weight Decay pulls the weights towards zero at each update step of SGD (note that for $\lambda = w/\alpha$ this is equivalent to L_2):

$$\theta \leftarrow (1 - w)\theta - \alpha g$$

The difference to L_2 is that in L_2 the hyperparameters λ and α are coupled!
Decoupled weight decay gave rise to the popular **AdamW** algorithm, which is essentially Adam combined with weight decay.

5.3 Parameter Sharing/Tying

Idea: Sharing parameters across the network. This is, e.g., used in Convolutional Neural Networks (CNNs) where the same filters are used across the input image, which makes feature detection invariant to translations. It is also used in RNNs where the same network is applied at each time step.
Sharing the parameters **controls the networks capacity**, **encourages the search for regular patterns** and **limits memory cost**.

Definition 11:

Parameter sharing is e.g. applied in **multitask learning**, where one network is used to tackle multiple tasks at once by an intermediate shared layer and individual models branching off of it. If the tasks are related, this can have a similar effect as additional training data.

Parameter tying allows models to be similar to each other, by adding a soft constraints to the parameters of each model. If the tasks are similar, this can be done by for example leveraging the L_2 regularization:

$$\Omega(\theta^{(A)}, \theta^{(B)}) = \|\theta^{(A)} - \theta^{(B)}\|_2^2$$

5.4 Dropout

Dropout randomly drops units from the network during training by multiplying their output by zero with a probability of p. For each datapoint a different dropout mask is sampled. This avoids co-adaption of the weights (reliance on previous units). It is important to note that we have to store the location of dropped weights in the forward pass to ignore the update in the backward pass.

Applying dropout can be seen as training an ensemble of 2^n networks with shared weights, where n is the number of dropped units.

5.5 Data Augmentation

Data augmentation is used to increase the data amount by applying some simple transformation to a training dataset and adding this transformed data to the training dataset.

Possible transformations: **translation, scaling, reflection, rotation, stretching**

Domain-Specific: replacing words with synonyms, adding background noise, change speed

Note that since CNNs have invariance with respect to the movement along each axis, trans- lation does not make sense. The drawback of data augmentation is to require more computational time proportional to the data size. Additionally, the improvement diminishes as the augmented amount increases.

There exist several techniques to find suitable augmentations automatically:

1. **AutoAugment:** Search for best augmentation combinations automatically, large comp. cost.
2. **RandAugment:** Randomly sample N augmentations with strength M, similar results as AutoAugment but much cheaper.
3. **TrivialAugment:** Randomly sample one augmentation and M for each image in a batch.

5.6 Noise Robustness

Goal: Improving robustness by training with randomly added noise. Note that data augmentations are the noise added to the inputs. Noise can be added in several more places:

- Hidden Units: Data augmentation at various levels of abstractness
- Weights: Stochastic implementation of Bayesian inference
- Outputs: Label smoothing

5.7 Adversarial Training

Adversarial data is data which can decept a trained model. The procedure for adversarial training is as follows:

1. Train network on training examples.
2. Generate adversarial examples for this network.
3. Add adversarial examples to training data and repeat.

5.8 Ensemble Methods

- **Bagging:**
  1. Randomly draw data with "zurücklegen" for each model
  2. Aggregate the output of all trained models in an ensemble classifier

  => Introduces bias but reduces variance
- **Deep Ensembles:**
  1. Train several deep networks on the same data, rely on SGD to find different minima
  2. Aggregate the output of all models (e.g. by averaging)

  => Does not introduce bias, but reduces variance

  => k-fold increase in complexity of all types (k models)
- **Hyperdeep Ensembles:**
  1. Train using SGD with weight restarts
  2. Take snapshots of weights before each restart and ensemble them

  => Reduces time complexity to a single training run

6 CNNs

CNNs are specially structured neural networks that are very well suited to process grid-like data such as images and time series. They make use of convolutional layers, which perform a **convolution operation** followed by some nonlinearity and **pooling**.

Definition 12:

The **operation of convolution** works by sliding a kernel/filter over the input and computing the dot products, thus reducing dimensionality and resulting in the so called "activation map" (one for each filter). Note that the filter always matches the depth channel of the input data. Convolutions are defined as:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$

Note that convolutions and cross-correlation are similar. Flipping the kernel by 180° and applying cross-correlation is equal to convolution.

6.1 More Properties of Convolutions

In order to **keep the spatial dimension** after applying convolution, typically we see Convolutional layers with stride 1, $F \times F$ filters and zero padding with $(F - 1/2)$ pixels.

Valid Convolution: When no padding is applied and the kernel is fully contained within the image.

Same Convolution: Applying padding to keep the image dimension constant.

With an input of shape $W_1 \times H_1 \times D_1$, the size of the convolution output $W_2 \times H_2 \times D_2$ is given by:

$$W_2 = (W_1 - F + 2P)/S + 1$$

$$H_2 = (H_1 - F + 2P)/S + 1$$

$$D_2 = K$$

Using filters of size $F \times F \times D_1$ and enabling parameter sharing, convolutions introduce $(FFD_1)K$ weights and K biases.

Convolution leverages three important ideas:

- Sparse interactions (if kernel $<$ input) $\rightarrow$ Receptive field increases with depth
- Parameter sharing
- Equivariant Representations: Change of input leads to same change of output

6.2 Miscellaneous Convolutions

1. **3D-Convolutions:** The filter can move in all three directions and the filter depth should be smaller than the input layer depth (i.e. kernel depth $<$ channel size). Useful for data such as videos or MRI images.
2. 1×1 **Convolution:** Also called feature pooling, reduces dimensionality for efficient computations to $W \times H \times 1$. Enables more complex representations using non-linearity.

3. **Transposed Convolution:** This convolution up-samples data by combining the padding and convolution.

4. **Dilated Convolution:** It in ates the kernel by inserting spaces between kernel elements while keeping the kernel size; therefore no additional cost is required. Since it convolutes elements from a larger range, we can often get much larger receptive field.

5. **Grouped Convolution:** Filters at each layer are separated into certain numbers of groups and each group is responsible for convolutions of the corresponding group. The advantage of this convolution is efficient training.

6. **Spacially Separable convolutions:** It first convolutes only along one axis and then it convolutes the convoluted data along the other axis in two separate steps. The adavantages of this convolution are less parameters and less matrix multiplications. On the other hand, training results can be suboptimal.

7. **Depthwise Separable Convolutions:** It first convolutes each channel separately, and then it applies 1×1 convolution to the data. This convolution also yields the same benefits and problems as spatially separable convolutions.

6.3 Pooling

Pooling layers (separately) reduce the size of the activation maps (downsampling) with the effect of making representations approximately invariant to small input translations.
One popular choice, namely **max-pooling** selects the maximum value in the area of the filter.
The formulas for the output size are the same as for convolutional layers, but without padding.

7 RNNs

7.1 General

RNNs allow for cycles in the connectivity graph. These cycles allow information to persist in the network for some time (state) and provide a fading memory. This makes RNNs extremely powerful at processing sequences. **RNNs are Turing-complete and can represent dynamical systems rather than function mappings.**
There are several ways to use RNNs:

- **One-to-many:** Image caption generation.
- **Many-to-one:** Giving a numerical score to a given review.
- **Many-to-many:** Video frame classification, language translation.

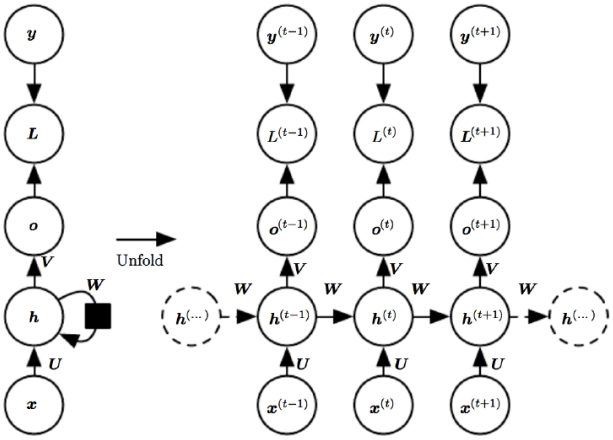
7.2 Design Patterns

RNNs produce outputs at every step t, with recurrent connections between hidden units. The unfolded network can be seen in the figure below. RNNs are trained with backpropagation through time (BPTT), which is backpropagation with weight sharing over the unrolled computational graph.
The forward pass is defined the following:

$$h^{(t)} = \tanh(Wh^{(t-1)} + Ux^{(t)} + b)$$

$$\hat{y}^{(t)} = \text{softmax}(o^{(t)}) = \text{softmax}(c + Vh^{(t)}),$$

where W, U, V are shared weight matrices of the hidden units and b, c are shared bias vectors for the hidden states and outputs respectively.



An alternative to having the recurrent connections between the hidden units, is having them go **from the output to the successive hidden unit**. Even though this is potentially less powerful, it has the advantage that it can be trained with **teacher forcing**.
Here, during training time, the hidden layers are fed with the true label of the previous step instead of the models own prediction. This enables the model to learn from the data directly, but it approximates based on the predictions during test time which makes it likely to accumulate errors over time.

7.3 Miscellaneous Architectures

1. **Many-to-One:** Recurrent connections etween hidden units, output only at last step (BPTT).
2. **One-to-Many:** Mapping static input into distribution over sequences of y. Outputs at every step.
3. **Many-to-Many:** Mapping variable length sequence of inputs into distribution over sequences of y. Can be combined with teacher forcing and BPTT.

Additional architectures: **Bidirectional RNN** with two directional hidden states can use information from both past and future. Since it requires future information, it cannot be used for online tasks. **Encoder-Decoder seq.-to-seq.** architecture are used when the length of inputs and outputs are different, such as for translation.

7.4 Problems during Training

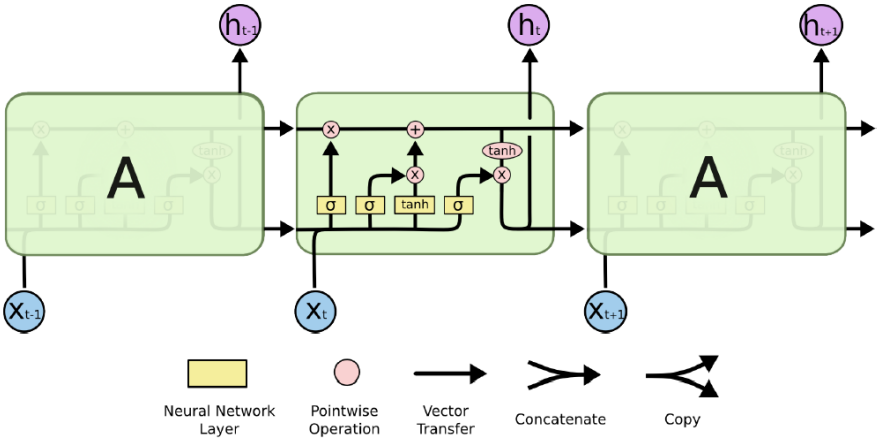
Although RNNs have powerful non-linear processing due to the repeating application of the same function, it also induces some problems. Typical problems are gradient vanishing and explosion. The solution for the explosion is gradient clipping and that for the vanishing is the LSTM.

7.5 Truncated BPTT

BPTT is expensive and possibly unstable. Furthermore it can lead to vanishing/exploding gradients. The solution is to use truncated BPTT, which truncates gradients after fixed intervals (often aligned with batch-size). Requires sequential data loading.

7.6 Long Short-Term Memory (LSTM)

The LSTM was introduced to address the vanishing gradient problem. Its architecture consists of special memory cells with **forget, input** and **outpus gates**. By now the LSTM has become the standard RNN model in many applications.



Since this model has a self-loop of inner state C without any activations, the gradient with respect to C does not vanish nearly as RNNs do. The forget gate decays less important features from the past and maintain important features. The input gate takes two vectors i_t and $\tilde{c}_t$ and controls how much new information we should store and in which direction we should change the state. The output gate controls which features to use. The advantages of LSTM are that it can selectively forget and remember features and each parameter of LSTM already have their meanings.

8 Attention & Transformers

8.1 Types of Attention

- **Implicit Attention:** A trained neural network reacts more strongly to some parts of the input than to others. This can be quantified by calculating the gradient of a class output w.r.t the input pixels to create an attention map.
- **Explicit Attention:** These mechanisms actively focus on parts of the input data (and down-weight others). Here, the network computes input-dependent attention weights (example: LSTM input gate).

8.2 RNNs with Attention

Encoder-Decoder Networks: The encoder processes the input sequence into a context vector. The decoder then takes the context vector and produces the output sequence. The context vector is usually the last hidden state of the encoder. Here the disadvantage lays in the fact that one vector has to encode the information for all inputs.

Idea: What if we were to pass all encoder hidden states to the decoder which then learns to focus on relevant parts?

Decoder-Attention: The decoder computes scores for each of the hidden states it received and thus "attends" to the relevant hidden states during each decoding step. Each hidden state is multiplied by it soft-maxed score. The scores are computed using a feedforward neural network.

8.3 Transformers

In contrast to RNNs, Transformers can handle all inputs in one single step. From a high-level overview, the architecture simply consists of a couple stacked encoder blocks followed by a couple stacked decoder blocks.

Each **encoder block** consists of a self-attention layer followed by a fully connected layer.

Each **decoder block** consists of a self-attention layer followed by an encoder-decoder attention layer and a fully connected layer.

8.4 Encoder: Flow of Vectors

In the encoder, each input word get encoded into a fixed-sized vector of size d. The self-attention blocks operate on all inputs jointly, while the FC layers operate on each word separately. Throughout the encoder block, the representation of each word gets transformed to consider the entire sentence.

8.5 Self-Attention

Attention can be seen as sort of a soft retrieval from a database. Given a list of key-value pairs stored in the database and a query q, attention compares q to the keys and returns the weighted average of the values:

$$\text{attention}(q, k, v) = \sum_{i=1}^{N} \text{softmax}(\text{Similarity}(q, k_i)) \times v_i$$

There are several popular ways to compute the similarity score:

- **Dot-Product:** $q^T k_i$
- **Scaled-Dot-Product:** $\frac{q^T k_i}{\sqrt{d}}$, with d being the dimensionality of the keys, used in Transformers
- **Generalized Dot-Product:** $g^T W k_i$
- **Additive similarity:** $w_q^T q + w_k^T k_i$

Important: In self-attention, queries, keys and values are all derived from the same input sequence! For each input x_i we compute query q_i, key k_i and value v_i vectors by linear mapping from three learnable weight matrices W^Q, W^K, and W^V.

8.6 Multi-Head Self-Attention

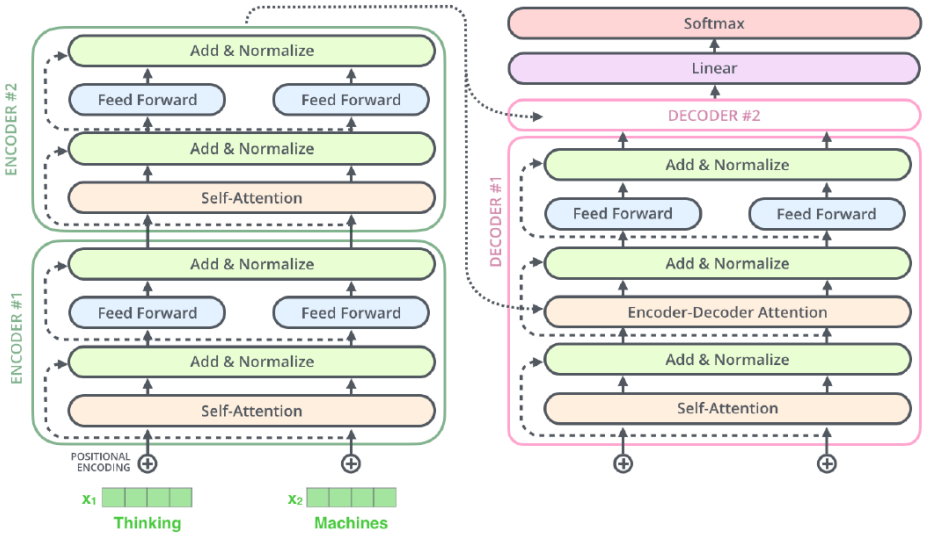
Multiple attention heads allows us to focus on more than one concept at a time. Using single attention would force us to average over concepts (e.g. in "the chicken didn't cross the road because it was too wet"). The embedding dimension d is then split equally to all N attention heads. Each head receives an input dimension of d/N.

8.7 Transformer: Encoder and Decoder

The problem with self-attention in and of itself is that it does not consider the order of the inputs, which obviously is crucial for many tasks such as translation.
Therefore in the Transformers encoder block a positional encoding is added to the input embedding before being fed into the encoder.

Additionally to the Self-Attention and FC-Layers, there are Skip Connections around both types of layers, which are followed by a LayerNorm. The weights of the FC-Layers in the same encoder block are shared.

The detailed architecture of the standard Transformer is displayed in the following figure.



The decoder of a Transformer is similar to the encoder, with the addition of encoder-decoder attention layers. The decoder takes a prefix of an output sentence as input and predicts the next word until it generated an **end token** as output. Its self-attention layer is only allowed attend to earlier predictions in the output sentence, the future positions are masked.

8.8 Limitations of Attention

- The complexity of storing the self-attention matrix of an input of length n is $O(n^2)$.
- Attention can not deal with arbitrary long inputs. The input has to be split into segments.
- Low inductive bias.
- Need to be very large to reach good performance.

9 Methodology

9.1 Batch Normalization

Problem: Training and test data and the internal representations have different variance and mean. This leads to SGD being slow and hard to optimize. Therefore, we would like our network to normalize the data, as every batch is a different distribution that must be adaptable.

Goal: Make all dimensions 0-mean and unit-variance by applying:

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}\left[x^{(k)}\right]}{\sqrt{\mathrm{Var}\left[x^{(k)}\right]}}$$

Batch Normalization should be applied after FC/Conv. layers and before the activation function. During testing batch norm becomes a linear operator.

Advantages of using batch norm:

- Makes deep networks much easier to train.
- Improves gradient flow.

- Allows for higher learning rates and thus faster convergence.
- Networks become more robust to initialization.
- Acts as regularization during training.

9.2 Layer Normalization

In contrast to batch norm layer norm normalizes across the dimensions of the input instead of the batch size. It behaves the same for train and test.

9.3 Transfer Learning

Weights from self-supervised learning can be used to initialize weights for a target task. Adding classifiers or fine-tuning layers is required.

Transfer learning methods in Vision: Self-prediction, contrastive learning, supervised image classification, ...
Transfer learning methods in NLP: Next sentence prediction, masked language modeling, ...

Definition 13:

Meta-Learning describes learning assumptions (inductive bias or prior) that can be transfered to new tasks.
It can be utilized to learn better architectures, hyperparameters, good initial weights and better loss functions.

9.4 Parameter Initialization

How we choose the initial parameters affects the convergence, minimal cost and generalization error of our model.

Current strategy: Initialize weights for each unit separately and avoid symmetry. The weights are initialized at random following a uniform or normal distribution, the biases are initialized constant. For initializing weights, the scale is important as it could lead to gradient explosion, generalization, activation flow or symmetry breaks. We e.g. use **Xavier initialization**, which is a normalized initialization defined as:

$$W_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right),$$

where m are the layers inputs and n the layers outputs. To compromise between goals of initializing all layers have the same activation variance and gradient variance.
The bias is most often set to 0, except for output units where it is set to match the desired output statistics; ReLU where it is set to 0.1 to avoid saturation; and gating units where it is set to ~ 1 to open the gate.

Other strategies:

- **Warm-starting:** Initialize with weights that have been optimized on a different task.
- **Meta-learning:** Learning the "best" initialization strategy for different tasks.

9.5 CNN Architectures

U-Net:

- Consists of a contracting and expanding path.
- Uses up-convolution to increase the resolution after the bottleneck.
- Uses 1×1 convolutions in the final layer to generate the final segmentation mask.
- Applies data augmentations.

ResNet:

- Introduces residual (identity) mappings (skip connections) around convolutional layers.
- Enables very deep networks.
- Uses batch normalization.
- Only one FC layer at the end.

10 Variational Auto-Encoders (VAEs)

10.1 Autoencoders

Autoencoders are networks that are trained to reconstruct their input. The usual architecture is something like

$$\text{encoder} \rightarrow \text{bottleneck } h \rightarrow \text{decoder}$$

where the encoder encodes the input by some function into latent representation, and the decoder creates the reconstruction based on this representation. Architecture and regularization are chosen to preven simple copying. As the network learns by the discrepancy between input and reconstruction, autoencoders learn in an unsupervised fashion.

10.2 Regularized Autoencoders

- Dimensionality $h \geq$ dimensionality of $x \rightarrow$ overcomplete representation
- Model capacity can be chosen based on the complexity of the data distribution.
- Prevent simple copying of the input e.g. by adding sparsity penalty on the hidden layer activations.

10.3 Sparse Autoencoders

- Add a sparsity penalty $\Omega(h)$ on the code h in the hidden layer to the loss function: $\mathcal{L} + \Omega(h)$.
- Often used as an unsupervised pre-processing step for downstream supervised processing.
- Sparsity as regularization to avoid overfitting and increase generalization.
- Joint distribution: $p_{\text{model}}(x, h) = p_{\text{model}}(h)p_{\text{model}}(x|h)$
- We maximize: $\log p_{\text{model}}(x, h) = \log p_{\text{model}}(h) + \log p_{\text{model}}(x|h)$
- Sparsity can be implemented using a Laplace prior: $p_{\text{model}}(h_i) = \frac{\lambda}{2} \exp(-\lambda|h_i|)$, where λ is a (learnable) hyperparameter

10.4 Denoising Autoencoders

Rather than adding an explicit regularization term, denoising autoencoders (DAE) learn to undo the effect of noise corruption when learning to reconstruct the input. To implement this, the input is corrupted by a corruption process $C(\tilde{x}|x)$ before being fed into the encoder. The optimization is performed by the gradient descent with respect to the following negative log-likelihood. As a result, we obtain the reconstructed distribution $p(\tilde{x}|x)$. Note that the latent vector is deterministic in this method, but if we make it stochastic, it is equivalent to variational auto-encoder.

10.5 VAEs

The idea behind generative models is to randomly sample from the latent space and decode to generate truly new data.
Assuming set of visible variables x and latent variables z our goal is to model the distribution of the training data:

$$p(x) = \int p(x, z)dz = \int p(x|z)p(z)dz$$

Once we have a good approximation of $p(x)$, we can sample from it and generate new data that is very similar, but different from our original training data.

But how do we get z values that are related to our input data? This is the problem of inferring the posterior distribution $p(z|x)$. Since we can't use Bayes theorem to get $p(z|x)$, we have to resort to approximate inference methods:

Definition 14:

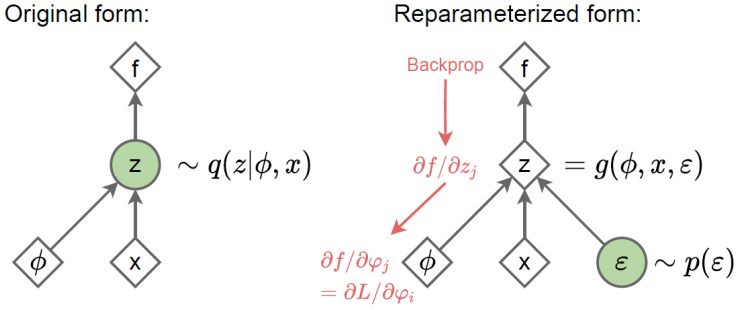
Approximate inference methods approximate the posterior with a simpler distribution $q(z|x)$ with its own set of parameters and optimize them to get as close as possible to $p(z|x)$ by maximizing the evidence lower bound (ELBO), which minimizes the KL-Divergence.

The VAE uses two neural networks: an encoder network with parameters ϕ that learns to approximate the posterior $q_\phi(z|x)$, and a decoder network with parameters θ that learns to reconstruct the input $p_\theta(x|z)$. With this parametrization, the ELBO for variational inference is:

$$\mathcal{L}(\theta, \phi, x) = -KL(q_\phi(z|x)||p_\theta(z)) + \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)]$$

where the first part is the regularization term and the second part is the reconstruction term.

To approximate the loss, we have to sample from posterior over z, but sampling is not a continuous operation. To mitigate this, we can reparametrize to express our random variable z as a deterministic variable and an external, static noise source expressed in random variable ϵ.



With this reparametrization, both the decoder and the encoder can be trained together using back-propagation to optimize the ELBO.

11 Generative Adversarial Networks (GANs)

GANs are generative models, similar to VAEs, but drop the assumption of optimizing the lower variational bound. Hence they often produce better results.
Idea: We want to sample from our training distribution, but there is now way to do this. Instead, we sample from a simple distribution, e.g. random noise, and learn the transformation to training distribution.

GANs consist of two networks: the Generator and the Discriminator. The Generator tries to fool the discriminator by generating real-looking images. The Discriminator then tries to distinguish between real and fake images. Both networks are trained jointly in an alternating fashion. We perform gradient **ascent** on the discriminator and gradient descent on the generator. The most important part during training is to have a good balance between Generator and Discriminator. It can easily happen that one outperforms/underperforms the other. After training, the generator can be used to generate new images.

11.1 Diffusion Models

Denoising diffusion models consist of two processes:

1. Forward diffusion process gradually adds noise to input data by sampling from a Gaussian T times.
2. Reverse denoising process generates output by denoising.

The forward diffusion process is a Markov process that leads to the joint probability:

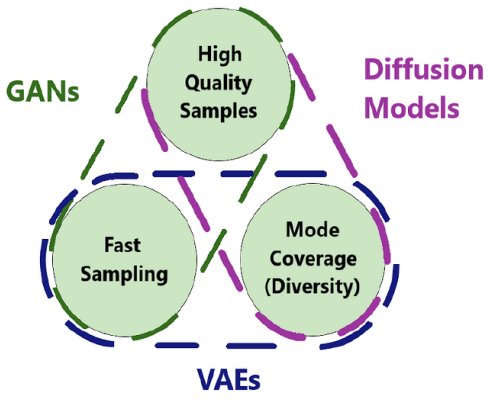
$$q(x_{1:T}|x_0) = \prod_{t=1}^{T} q(x_t|x_{t-1})$$

The schedule of the noise is designed such that $q(x_T|x_0) \approx \mathcal{N}(x_T; 0, 1)$.

The denoising process starts by sampling $x_t \sim q(x_T) \approx \mathcal{N}(x_T; 0, 1)$ and then iteratively sampling $x_{t-1} \sim q(x_{t-1}|x_t)$. Even though this is generally intractable, we can approximate this with a Gaussian if β_t is small enough. This means that we can learn the mean of a Gaussian to approximate

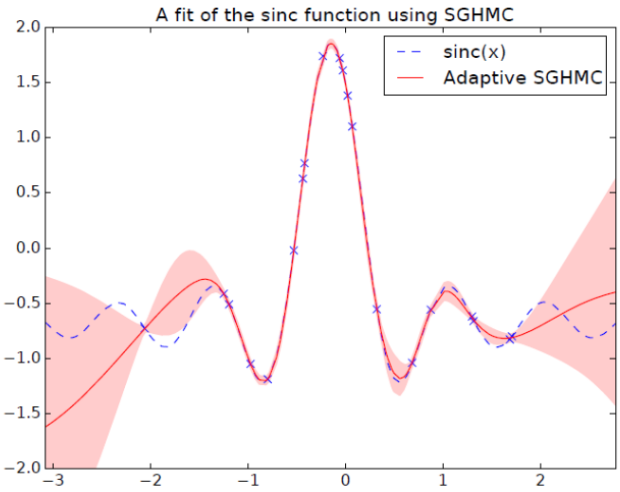
$$q(x_{t-1}|x_t) \approx \mathcal{N}(x_{t-1}; \mu_\theta(x_t, t), \sigma_t^2 I),$$

where $\mu_\theta(x_t, t)$ is the trainable network.



12 Uncertainty Estimation

Knowing model's uncertainty is crucial in safety-critical applications such as medical diagnosis, autonomous driving, etc. The output probability (softmax) of standard neural networks is typically not reliable. For example, deep neural nets can be overconfident for unrecognizable images. In these cases, the uncertainty should grow away from the typical data seen, but it doesn't.
For regression NNs, the uncertainty should look something like the following:



There are several types of uncertainty:

1. **Aleatoric:** Intrinsic observation (data) noise.
2. **Epistemic:** Uncertainty about the model, can be reduced by more data.

Definition 15:

Being Bayesian about NNs means dealing will all sources of parameter uncertainty: As more than one weight vector can explain the observed data, we need to take into account all possible explanations.

12.1 DNGO Model

The DNGO model aims at being Bayesian at least about the output layer. DNGO first trains a neural network on a given dataset and use the feature vector obtained in the last hidden layer as basis functions of the Bayesian linear regression.

12.2 Bayesian Neural Networks

Given data $\mathcal{D} = (X, y)$, parameters θ, a prior distribution $p(\theta)$ and a likelihood $p(y|X, \theta)$, the core part of Bayesian Neural Networks is to compute the posterior distribution:

$$p(\theta|\mathcal{D}) = p(\theta|X, y) = \frac{p(y|X, \theta)p(\theta)}{p(y|X)} = \frac{p(y|X, \theta)p(\theta)}{\int p(y|X, \theta)p(\theta)d\theta}$$

Problem: Since the mean of $p(y|X, \theta)$ is not a linear function of θ and Gaussians are not closed under non-linear mappings, the posterior is not a Gaussian anymore. Approximating the posterior requires variational inference or the Markov-Chain-Monte-Carlo method.

12.3 Variational Inference

Variational Inference (VI) is an optimization-based method for approximating intractable probability distributions in Bayesian inference. Instead of directly computing a complex posterior distribution $p(z \mid x)$, VI approximates it with a simpler distribution $q(z \mid \lambda)$ by minimizing the Kullback-Leibler (KL) divergence:

$$q^*(z) = \arg\min_{q \in \mathcal{Q}} D_{\mathrm{KL}}(q(z \mid \lambda) \parallel p(z \mid x)).$$

This converts Bayesian inference into an optimization problem, making it computationally feasible for high-dimensional models.
In order to minimize this KL-Divergence w.r.t to θ, we only need to maximize the ELBO. ELBO embodies a trade-off between satisfying the complexity of the data and satisfying the simplicity of the prior. ELBO is very popular since it approximates the true posterior with the variational posterior and therefore replaces the diffcult integral.

12.4 Markov-Chain-Monte-Carlo (MCMC) Method

Another posterior estimation is performed by MCMC. MCMC samples each weights according to a proposal distribution or transition distribution $p(w_{t+1}|w_t)$ and moves around the space while accepting or rejecting the proposal from the distribution. Since the next state is determined only by the current state, it is called Markov-chain. For the nal sampling, we use every t-th sample from the history to avoid the correlation between samples close to each other in terms of time steps. The goal of the sampling is to identify the stationary distribution π that is ideally the posterior distribution $p(w|\mathcal{D})$. The sufficient condition for a stationary distribution to exist is that the detailed balance $\pi(w_t)p(w_{t+1}|w_t) = \pi(w_{t+1})p(w_t|w_{t+1})$. In practice, even when the detailed balance is satisfied, it may still take time to reach the stationary distribution. This time (from the beginning) is called mixing time. Intuitively, the distribution reaches the stationary distribution when the chain forgets the beginning states. Therefore, we take samples after a burning-in phase and the length of the burn-in phase is a hyperparameter.

12.5 Prior-Fitted Networks (PFNs)

PFNs are neural networks trained to approximate Bayesian inference by learning a prior distribution over possible data-generating processes. Instead of traditional training on a fixed dataset, PFNs are trained on a diverse set of synthetic tasks sampled from a prior, enabling them to generalize to new tasks without gradient-based fine-tuning. This allows PFNs to perform rapid adaptation in a Bayesian manner while leveraging neural network expressiveness.
PFNs might be a disruptive alternative to variational inference and MCMC.

12.6 Output Ensembling

Construct an ensemble of M neural networks on the training data. For each test data point x predict with each of the M networks and combine the outputs to obtain probabilistic predictions:

$$f(x) = \frac{1}{M} \sum_{i=1}^{M} f_i(x), \quad \text{(classification case)}$$

where $f_i(x)$ is the output of the i-th network. For the regression case we can compute the empirical mean and variance across the predictions like so:

$$\mu_x = \frac{1}{M} \sum_{i=1}^{M} f_i(x)$$

$$\sigma_x^2 = \frac{1}{M}\sum_{i=1}^{M}(f_i(x) - \mu_x)^2$$

The M-Ensemble can be created in different ways: via dropout (simply by sampling M dropout masks), via MCMC, via different random seeds for SGD initialization, etc.

12.7 The Estimation of Parametric Models

We can train a network with weight θ to output the parameters w of a parametric model $p(y|x, w)$. Most commonly we have the network output μ and $\log(\sigma^2)$ of a Gaussian $\mathcal{N}(\mu, \sigma^2)$.
This can be achieved by simply maximizing the log-likelihood w.r.t. network weight θ:

$$\log p(\mathcal{D}|\theta) = \frac{1}{N}\sum_{i=1}^{N}\log p(y_i|w(x_i, \theta)).$$

The predictive distribution for an input x is then defined as:

$$p(y|x, \theta) = p(y|w(x, \theta)).$$

These predictive uncertainties can then be combined with ensembling. Each network i with weights θ_i predicts a probability distribution with mean $\mu_{x,i}$ and variance $\sigma_{x,i}^2$.
We can combine these predictions in a mixture distribution, using the law of total variance to compute the mean μ_x and variance σ_x^2 as:

$$\mu_x = \frac{1}{M}\sum_{i=1}^{M}\mu_{x,i}$$

$$\sigma_x^2 = \frac{1}{M}\sum_{i=1}^{M}\left((\mu_{x,i} - \mu_x)^2 + \sigma_{x,i}^2\right)$$

13 AutoML / Hyperparameter Optimization (HPO)

Neural networks are very sensitive to many hyperparameters, e.g.:

- Optimization: SGD variant, lr-schedule, momentum, batchsize, ...
- Regularization: dropout rates, weight decay, data augmentation, ...

Definition 16:

Let θ be the hyperparameters of an ML algorithm $\mathcal{A}$ with domain Θ, $\mathcal{D}_{opt}$ be a training set which is split into training and validation split and $\mathcal{L}(\mathcal{A}_\theta, \mathcal{D}_{train}, \mathcal{D}_{val})$ denote the loss of $\mathcal{A}_\theta$ trained on $\mathcal{D}_{train}$ and evaluated on $\mathcal{D}_{val}$.
The HPO problem is to find a hyperparameter configuration which minimizes the loss:

$$\theta^* \in \arg\min_{\theta\in\Theta}\mathcal{L}(\mathcal{A}_\theta, \mathcal{D}_{train}, \mathcal{D}_{val})$$

Hyperparameters can have different types, such as numerical (continuous, integer) or categorial (boolean, categories e.g. choice of optimizer). Additionally, some hyperparams are only active if other hyperparams take certain values. Furthermore, some hyperparams (learning rate) naturally lay on a logarithmic scale. Tuning all these parameters manually could be a tedious task.

13.1 Practical Tips

In general, HPO is useless unless the model can overfit the training dataset. For this reason, one good strategy is to make sure that the model (i.e. training loss) can overfit the subset and then move tothe HPO of validation loss on the full dataset. As another tip, the learning rate is typically important, so one heuristic is to increase learning rate until the loss diverges and then reduce it a little bit.

13.2 Black-Box Optimization

Black-box optimization is to optimize functions without derivative information. The most basic algorithm is random search and it maintains global search and can be computed in parallel. Additionally, it can deal with low intrinsic dimensionality compared to grid search. Another famous example is local search and it fixes the best configuration and changes the configuration along only a single axis.

13.3 Bayesian Optimization

General approach:

- Fit a probabilistic model to the collection functions samples
- Use the model to guide optimization, trading off exploration vs exploitation

13.4 Population-based Methods

Uses a population of configurations and the "survival of the fittest" principle. Based on evolutionary strategies.

13.5 HPO Speedup Techniques

There are several techniques to speed up black-box HPO optimization:

1. **Meta-Learning:** Follows the idea of learning about learning methods by optimizing the performance of known learning methods and generating new ones, transferring the learned knowledge in the process.
2. **Extrapolating of learning curves:** Terminate learning curves that are not promising.
3. **Multi-Fidelity optimization:** Using iterative ML algorithms, poor runs can be stopped early. Using k-fold cross-validation, poor hyperparam settings can be rejected after a few folds. Bad models can be ruled out based on their performance on data subsets.

Successive Halving:

- Try different configurations and drop the worst ones after a certain time.
- Next, drop the next generation after double the time. Repeat.
- Problem: Some configurations might be bad in the beginning but good in the end.

Hyperband:

- Same as successive halving, but takes different time budgets as lowest fidelity.
- This is done without taking already tested settings into account.

BOHB:

- Uses Bayesian optimization for choosing configurations.
- Uses Hyperband for selecting the budgets.

13.6 Combining User Beliefs with Bayesian Optimization

Prior user knowledge could be helpful for selecting well-performing hyperparameter configurations.

PriorBand:

- Combines user priors with multi-fidelity optimization
- Uses three different configuration sampling schemes: random, user prior over location of good configurations, loccaly around incumbent
- Less uniform random configurations for higher fidelities
- Over time, give higher weight to strategies that were more successful in this run
- Advantage: Even performs well (in the long run) if given a bad prior

13.7 Hyperparameter Gradient Descent

Let $\mathcal{L}_{val}(w, \theta)$ denote the validation loss of a network with weights w and hyperparameters θ; likewise, $\mathcal{L}_{train}(w, \theta)$ is the training loss.
Then, the optimization of θ can be written as the following bilevel optimization problem:

$$\min_{\theta} \mathcal{L}_{val}(w^*(\theta), \theta)$$

$$s.t. \quad w^*(\theta) \in \arg\min_{w} \mathcal{L}_{train}(w, \theta)$$

This enables us to compute gradients for θ by differentiating through the entire SGD optimization run that leads to $w^*(\theta)$. Weight and hyperparameter optimization steps can then be interleaved (does not guarantee convergence).

13.8 Neural Architecture Search (NAS)

NAS aims to automatically optimize architectural choices, making it a special case of HPO. NAS also allows for special speedup techniques: sharing & inheriting weights.
All possible architectures are subgraphs of a large supergraph: the one-shot model. Weights are shared between different architectures with common edges in the supergraph. Search costs are reduced drastically since one only has to train a single model.

DARTS: Use one-shot model with continuous architecture weight α for each operator. By optimizing the architecture weights, DARTS assigns importance to each operation. And since α is continuous, we can optimze them with gradient descent. In the end, DARTS discretizes to obtain a single architecture.

Weight Entanglement: Weight entanglement reduces memory complexity of the supernet by reusing weights, not only between architectures that share the same operation (i.e. edge in the graph) but also between operations.

Weight Inheritance: When locally expanding a network (e.g., adding a layer), we can inherit the weights of the unchanged part, which avoids costly retraining of the unchanged part. The newly introduced weights can be initialized to yield a **network morphism** (operators that change the network structure, but not the modelled function).