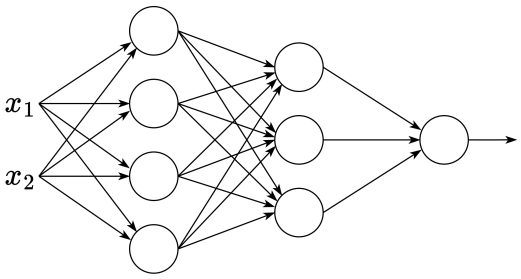
1 What is a neural network?

History of neural networks dates back at least to the 1950s when F. Rosenblatt proposed the perceptron model. The basic computational unit in such a network was the McCulloch–Pitts neuron which realized the following mapping $\mathbb{R}^n \mapsto \mathbb{R}$

$$\boxed{f(\boldsymbol{x}) = \varphi\left(\boldsymbol{x}\boldsymbol{w}^\intercal + b\right)},$$

where $\boldsymbol{x} = \begin{bmatrix} x_1 & \dots & x_n \end{bmatrix}$ is the vector of input signals, φ is some nonlinear activation function and $\boldsymbol{w}$, b are some parameters ($\boldsymbol{w}$ called the weights and b called the bias). Multilayer perceptron was built from many such neurons connected in layers so that the connections existed only between the neurons in neighboring layers and there were no connections between the neurons in the same layer.



In general neural network is any Directed Acyclic Graph (DAG) in which every vertex i has the following attributes.

1. Set of previous vertices – $\mathscr{P}_i$.
2. Set of next vertices – $\mathscr{N}_i$.
3. Parametrized tensor function $\boldsymbol{F}^{(i)}$ of the form

$$\mathbb{R}^{\left(n_1^{(1)},\dots,n_{k_1}^{(1)}\right)} \times \dots \times \mathbb{R}^{\left(n_1^{(p)},\dots,n_{k_p}^{(p)}\right)} \times \Theta \mapsto \mathbb{R}^{(m_1,\dots,m_l)}$$

The function takes p tensor arguments of dimensions $k_1, \dots, k_p$ respectively (input tensor q of dimension k_q has $n_r^{(q)}$ elements along the r axis) and parameters $\boldsymbol{\theta}^{(i)} \in \Theta$ and returns a tensor of dimension l. Obviously it must satisfy $p = |\mathscr{P}_i|$ and the tensors returned by the parent nodes must have appropriate shapes.

4. The gradient functions of the function $\boldsymbol{F}^{(i)}$ w.r.t. to the parameters and w.r.t. all the inputs, that is for all $j \in \mathscr{P}_i$ we have the gradient functions

$$\frac{\partial F_\beta^{(i)}}{\partial \theta_\alpha^{(i)}}, \quad \frac{\partial F_\beta^{(i)}}{\partial F_\alpha^{(j)}},$$

where α, β are the suitable multi-indices.

2 Loss functions

Training of a neural network consists of changing the parameters $\boldsymbol{\theta}^{(i)}$ of the nodes in such a way as to make the network perform the given task. The task is specified by a training set $\mathcal{X}$ which contains the "blueprint answers" of the network. To train the network we introduce the quantitative measure of networks performance on the dataset which implicitly (through the outputs of the network) depends on the parameters of the network (here collectively denoted by $\boldsymbol{\theta}$) $L(\mathcal{X}, \boldsymbol{\theta})$. Training can be then phrased as an optimization problem of the form

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} L(\mathcal{X}, \boldsymbol{\theta})$$

for a fixed training set $\mathcal{X}$.

There is no single established way of constructing loss functions. One of the more motivated approaches is based on the maximum likelihood criterion. The idea is that we model our data using some parametrized statistical model and express the parameters of this model as an output of a neural network. The loss function is then taken to be the *negated log-likelihood function*. In this manner one can derive the most common loss functions.

2.1 Mean Squared Error

$$\boxed{L(\mathcal{X}, \boldsymbol{\theta}) = \frac{1}{2n} \sum_\alpha \left[y_\alpha - \Phi_\alpha(\boldsymbol{X}; \boldsymbol{\theta})\right]^2}$$

where $\boldsymbol{X}$ is a tensor which can be interpreted as a stack of n 1-D feature-vectors residing in the last dimension of $\boldsymbol{X}$, $\boldsymbol{y}$ is the corresponding tensor of n scalar continuous outputs for each feature-vector (so called target) and $\boldsymbol{\Phi}$ denotes the neural network. We also show the derivation of the gradient of the loss function w.r.t. $\boldsymbol{\Phi}$

$$\begin{aligned} \frac{\partial L}{\partial \Phi_\beta} &= \frac{1}{2n} \sum_\alpha 2\left(\Phi_\alpha - y_\alpha\right) \frac{\partial \Phi_\alpha}{\partial \Phi_\beta} \\ &= \frac{1}{n} \sum_\alpha \left(\Phi_\alpha - y_\alpha\right) \delta_{\alpha\beta} \\ &= \frac{1}{n} \left(\Phi_\beta - y_\beta\right) \end{aligned}$$

so finally

$$\boxed{\frac{\partial L}{\partial \Phi_\beta} = \frac{1}{n} \left(\Phi_\beta - y_\beta\right)}$$

2.2 (Binary) Cross Entropy

$$\boxed{\begin{aligned} &L(\mathcal{X},\boldsymbol{\theta}) = -\frac{1}{n}\sum_{\alpha}\left[t_\alpha \log \pi_\alpha + (1-t_\alpha)\log(1-\pi_\alpha)\right] \\ &\boldsymbol{\pi} = \sigma\left(\boldsymbol{\Phi}(\boldsymbol{X};\boldsymbol{\theta})\right)\,, \quad \sigma(z) = \frac{1}{1+\mathrm{e}^{-z}}\,, \end{aligned}}$$

where $\boldsymbol{X}$ is a tensor which can be interpreted as a stack of n 1-D feature-vectors residing in the last dimension of $\boldsymbol{X}$, $\boldsymbol{t}$ is the corresponding tensor of n binary (i.e. 0 or 1) values denoting the class for each feature-vector, σ is the logistic function, $\boldsymbol{\Phi}$ denotes the neural network and $\boldsymbol{\pi}$ is a tensor of the same shape as $\boldsymbol{t}$ which contains the probabilities of the positive class. We also show the derivation of the gradient of the loss function w.r.t. $\boldsymbol{\Phi}$

$$\begin{aligned} \frac{\partial L}{\partial \Phi_\beta} &= \sum_\mu \frac{\partial L}{\partial \pi_\mu}\frac{\partial \pi_\mu}{\partial \Phi_\beta} \\ &= \frac{1}{n}\sum_\mu \left(\frac{1-t_\mu}{1-\pi_\mu} - \frac{t_\mu}{\pi_\mu}\right)\sigma'(\Phi_\mu)\delta_{\mu\beta} \\ &= \frac{1}{n}\left(\frac{1-t_\beta}{1-\pi_\beta} - \frac{t_\beta}{\pi_\beta}\right)\sigma'(\Phi_\beta) \quad . \end{aligned}$$

Note however that

$$\frac{\mathrm{d}\sigma}{\mathrm{d}z} = \sigma(z)\left(1-\sigma(z)\right)\,,$$

therefore $\sigma'(\Phi_\beta) = \pi_\beta(1-\pi_\beta)$ and thus

$$\boxed{\frac{\partial L}{\partial \Phi_\beta} = \frac{1}{n}(\pi_\beta - t_\beta)}$$

2.3 Cross Entropy

$$\boxed{\begin{aligned} &L(\mathcal{X},\boldsymbol{\theta}) = -\frac{1}{n}\sum_{\alpha}\sum_{\beta} t_{\alpha\beta}\log \pi_{\alpha\beta} \\ &\boldsymbol{\pi} = \boldsymbol{\sigma}\left(\boldsymbol{\Phi}(\boldsymbol{X};\boldsymbol{\theta})\right)\,, \quad \sigma_{\alpha'\alpha}(\boldsymbol{z}) = \frac{\exp z_{\alpha'\alpha}}{\sum_\beta \exp z_{\alpha'\beta}} \end{aligned}}$$

where $\boldsymbol{X}$ is a tensor which can be interpreted as a stack of n 1-D feature-vectors residing in the last dimension of $\boldsymbol{X}$, $\boldsymbol{t}$ is the corresponding thensor which can be interpreted as a stack of n 1-D one-hot-vectors residing in the last dimension of $\boldsymbol{t}$ encoding the correct class, $\boldsymbol{\sigma}$ is the soft-max function which given the stack of 1-D vectors independently normalizes each of them so that the entries are non-negative and sum to 1 and $\boldsymbol{\Phi}$ denotes the neural network. We also show the derivation of the gradient of the loss function w.r.t $\boldsymbol{\Phi}$

$$\frac{\partial L}{\partial \Phi_{\mu\nu}} = \sum_{\mu'\nu'}\frac{\partial L}{\partial \pi_{\mu'\nu'}}\frac{\partial \pi_{\mu'\nu'}}{\partial \Phi_{\mu\nu}} = -\frac{1}{n}\sum_{\mu'\nu'}\frac{t_{\mu'\nu'}}{\pi_{\mu'\nu'}}\frac{\partial \pi_{\mu'\nu'}}{\partial \Phi_{\mu\nu}}\,.$$

It can easily be shown that

$$\frac{\partial \pi_{\mu'\nu'}}{\partial \Phi_{\mu\nu}} = \delta_{\mu'\mu}\delta_{\nu'\nu}\pi_{\mu'\nu'} - \delta_{\mu'\mu}\pi_{\mu'\nu'}\pi_{\mu'\nu}$$

and thus

$$\frac{\partial L}{\partial \Phi_{\mu\nu}} = -\frac{1}{n}\left(t_{\mu\nu} - \pi_{\mu\nu}\sum_{\nu'} t_{\mu\nu'}\right)\,.$$

However from the definition of $\boldsymbol{t}$ we have $\sum_{\nu'} t_{\mu\nu'} = 1$ and thus finally

$$\boxed{\frac{\partial L}{\partial \Phi_{\mu\nu}} = \frac{1}{n}\left(\pi_{\mu\nu} - t_{\mu\nu}\right)}$$

3 Forward propagation

Let $\boldsymbol{v}^{(i)}$ be the (tensor) value of the function $\boldsymbol{F}^{(i)}$. To propagate the (tensor) inputs to the network and get the output we use the following recursive equation

$$\boxed{\boldsymbol{v}^{(i)} = \boldsymbol{F}^{(i)}\left[\left(\boldsymbol{v}^{(j)}\right)_{j\in\mathscr{P}_i};\boldsymbol{\theta}^{(i)}\right]}$$

and visit the nodes in the *topological order* as this guarantees that we visit every node exactly once. We assume here that nodes $\boldsymbol{v}^{(i)}$ such that $\mathscr{P}_i = \varnothing$ are the inputs to the network and nodes $\boldsymbol{v}^{(i)}$ such that $\mathscr{N}_i = \varnothing$ are the output of the network.

4 Backward propagation

Let L be the loss function. In order to compute the derivatives $\partial_{\boldsymbol{\theta}^{(i)}} L$ we use the following recursive equations

$$\boxed{\begin{aligned} \frac{\partial L}{\partial \theta^{(i)}_\alpha} &= \sum_\beta \frac{\partial L}{\partial F^{(i)}_\beta}\frac{\partial F^{(i)}_\beta}{\partial \theta^{(i)}_\alpha} \\ \frac{\partial L}{\partial F^{(i)}_\alpha} &= \sum_{j\in\mathscr{N}_i}\sum_\beta \frac{\partial L}{\partial F^{(j)}_\beta}\frac{\partial F^{(j)}_\beta}{\partial F^{(i)}_\alpha} \end{aligned}}$$

where α, β are the suitable multi-indices. We visit nodes in the *reversed topological order* and compute and store the values of loss function derivatives. All derivatives are computed for the current values of $\boldsymbol{v}^{(i)}$ and $\boldsymbol{\theta}^{(i)}$, therefore before backward propagation one must perform forward propagation to compute values $\boldsymbol{v}^{(i)}$.

5 Stochastic Gradient Descent

The standard optimization method used to train neural networks is the Stochastic Gradient Descent, which is an iterative, gradient-based algorithm in which in every step t we update the parameters $\boldsymbol{\theta}^{(i)}$ utilizing the gradient information. Let $\boldsymbol{\theta}^{(i,t)}$ denote the value of parameters $\boldsymbol{\theta}^{(i)}$ at step t and let $\boldsymbol{v}^{(i,t)}$ be the values of the function $\boldsymbol{F}^{(i)}$ at step t. In each step we take a batch $\mathcal{X}$ of training data, perform forward propagation to compute values $\boldsymbol{v}^{(i,t)}$ and the value of the loss function $L(\mathcal{X}, \boldsymbol{\theta}^{(t)})$, next perform backward propagation to compute the values of gradients $\boldsymbol{g}^{(i,t)} = \partial_{\boldsymbol{\theta}^{(i)}} L\left(\mathcal{X}, \boldsymbol{\theta}^{(t)}\right)$ and afterwards we update the parameters according to

$$\boxed{\theta_\alpha^{(i,t+1)} = \theta_\alpha^{(i,t)} - \eta g_\alpha^{(i,t)}}$$

where η is the learning rate.

5.1 Momentum

The problem with vanilla SGD is that it gets stuck in the local minima. To overcome this problem one can take inspiration from simple physics. We first introduce velocity tensor $\boldsymbol{V}$ with the following update rule

$$\boxed{V_\alpha^{(i,t+1)} = \gamma V_\alpha^{(i,t)} - \eta g_\alpha^{(i,t)}}$$

where $0 < \gamma < 1$ is the so called momentum term and update parameters using

$$\boxed{\theta_\alpha^{(i,t+1)} = \theta_\alpha^{(i,t)} + V_\alpha^{(i,t+1)}}$$

5.2 Adaptive Moment Estimation (Adam)

Another problem with vanilla SGD is that it uses the same learning rate for every scalar parameter. Adam algorithm solves this problem by introducing running averages with exponential forgetting of both the gradients and the second moments of the gradients.

$$\boxed{\begin{aligned}
M_\alpha^{(i,t+1)} &= \beta_1 M_\alpha^{(i,t)} + (1-\beta_1) g_\alpha^{(i,t)} \\
V_\alpha^{(i,t+1)} &= \beta_2 V_\alpha^{(i,t)} + (1-\beta_2)\left(g_\alpha^{(i,t)}\right)^2 \\
\tilde{M}_\alpha^{(i,t+1)} &= \frac{M_\alpha^{(i,t+1)}}{1-\beta_1^t} \\
\tilde{V}_\alpha^{(i,t+1)} &= \frac{V_\alpha^{(i,t+1)}}{1-\beta_2^t} \\
\theta_\alpha^{(i,t+1)} &= \theta_\alpha^{(i,t)} - \eta \frac{\tilde{M}_\alpha^{(i,t+1)}}{\sqrt{\tilde{V}_\alpha^{(i,t+1)}} + \epsilon}
\end{aligned}}$$

where $\epsilon \simeq 10^{-8}$ is used to prevent division by 0, η is the learning rate and β_1 and β_2 are the forgetting factors typically set to $\beta_1 = 0.9$ and $\beta_2 = 0.999$. Additionally popular choice for η is $\eta = 3 \cdot 10^{-4}$.

5.3 Asynchronous SGD

...

6 Regularization

Neural networks have very high capacity, meaning they are very flexible and can easily overfit. Regularization methods are methods used to fight this phenomenon.

6.1 Weight decay

Weight decay is a simple regularization method present already in classical, shallow machine learning models, in which we simply add to the loss function a suitably chosen norm of the parameters of the model

$$L^*(\mathcal{X}, \boldsymbol{\theta}) = L(\mathcal{X}, \boldsymbol{\theta}) + \lambda \|\boldsymbol{\theta}\|_p^p .$$

Typically one uses the L1 or L2 norms ($p = 1, 2$). The most important difference between the two is that L1 norm contains implicit feature selection that is it often makes the parameters exactly 0, while L2 norm only encourages the weights to be values close to 0.

6.2 Weight normalization

The problem with weight decay is that each parameter is limited independently of others. Weight normalization is a regularization technique that forces the weights to "compete" against each other so that only the most important parameters remain. The idea is following, we introduce the limit for the norm of weight vector ℓ. After each parameter update, we compute the vector p-norm

$$N_\alpha = \left(\sum_\beta |\theta_{\alpha\beta}|^p \right)^{1/p}$$

and update the weights according to

$$\theta_{\alpha\beta} \leftarrow \begin{cases} \theta_{\alpha\beta} & \text{if } N_\alpha \leq \ell \\ \frac{\ell}{N_\alpha} \theta_{\alpha\beta} & \text{if } N_\alpha > \ell \end{cases} .$$

6.3 Early stopping

The simplest, yet extremely powerful and popular form of regularization is the early stopping. The idea is that we divide the training set into a smaller training set and a validation set. We train the model on this smaller training set and at the same time measure the performance of the model on the validation set. If the loss gets smaller on both of these sets, everything is alright, but the moment the loss on the validation set starts rising, while the loss on training set gets smaller we stop the training, as this means the model is starting to overfit.

6.4 Dropout

A general method of regularization of any machine learning model is averaging the answers of an ensemble of similar models trained on different subsets of the training set which make different mistakes. The naive implementation of this method for the neural networks is not feasible as neural networks require vast computational resources in the training process. The method which effectively realizes this ensembling is dropout. The idea is to introduce layers into the computation graph which during training multiply the inputs elementwise by a binary tensor whose element can be 0 or 1 with specified probability p. During training, in each forward pass we sample such tensor and update the running sum. Having finished training we normalize the running sum tensor (i.e. divide it by the number of forward passes in training phase) and during forward pass multiply by it.

6.5 Transfer learning

When training data are limited, other datasets can be exploited to improve performance. In transfer learning, the network is pre-trained to perform a related secondary task for which data are more plentiful. The resulting model is then adapted to the original task. This is typically done by removing the last layer and adding one or more layers that produce a suitable output. The main model may be fixed, and the new layers trained for the original task, or we may finetune the entire model. The principle is that the network will build a good internal representation of the data from the secondary task, which can subsequently be exploited for the original task. Equivalently, transfer learning can be viewed as initializing most of the parameters of the final network in a sensible part of the space that is likely to produce a good solution.

6.6 Augmentation

Training set augmentation is a method of implicitly specifing certain invariances of the network by modyfing the examples from the training set before feeding them into the network. Typical examples are image transformations for convolutional neural networks trained to classify images as we want the output of the network to be invariant under rotations, zoom, reflection, etc.

7 Initialization

One important aspect of neural networks is the way we initialize the parameters before training. Since the neural network effectively realizes a highly nonlinear mapping, the loss as a function of the parametrs is also highly nonlinear and non-convex. Because of this the initial values of the parameters do matter. The simplest form of initialization is zero initialization in which we just set $\theta = 0$. This however leads to too much symmetry and thus we almost always use random initialization in which we sample weights (independently) from some probability distribution – typically uniform distribution centered at 0 with small width e.g. ± 0.005 or normal distribution with mean 0 and small variance like 0.01. Several authors proposed more specific initialization schemes (typically for fully connected layers) based on analysis of the variance of activations between the layers. Assuming n_i and n_{i+1} are the number of neurons (dimensions of outputs) in subsequent layers we have the following.

- **LeCun initialization.** We sample the weights from
$$\mathcal{U}\left(-\sqrt{3n_i^{-1}}, +\sqrt{3n_i^{-1}}\right) \quad .$$
This initialization is designed to preserve the variance of activations during the forward pass.

- **(Xavier) Glorot initialization.** We sample the weights from
$$\mathcal{U}\left(-\sqrt{6(n_i+n_{i+1})^{-1}}, +\sqrt{6(n_i+n_{i+1})^{-1}}\right) \quad .$$
This initialization is designed as a compromise between preserving the variances of activations during the forward pass and gradient variances during the backward pass.

- **(Kaiming) He initialization.** We sample the weights from $\mathcal{N}\left(0, \sqrt{2n_i^{-1}}\right)$. This initialization

is a modification of Xavier initialization for ReLU (Rectified Linear Unit) activation function

$$\boxed{\mathrm{ReLU}(z) = \max(0, z)}$$

instead of previously used sigmoid activations.

Additionally there are methods called normalization methods whose aim is to reduce the need for careful initialization, so that the neural networks have reasonable convergence for any sensible initialization e.g. $\mathcal{N}(0, 0.01)$.

8 Normalization

Activation normalization is a technique that rescales the activations of a layer of the neural network. It is used to increase the speed of convergence of the neural network, reduce overfitting and the need for careful initialization. The basic idea is to introduce a computational node (layer) which realizes the following mapping

$$\boxed{\begin{aligned} &F_{\alpha'\alpha}(\boldsymbol{X}; \boldsymbol{A}, \boldsymbol{B}) = A_\alpha \frac{X_{\alpha'\alpha} - M_\alpha}{\sqrt{S_\alpha + \epsilon}} + B_\alpha \\ &M_\alpha = \frac{1}{n}\sum_{\alpha'} X_{\alpha'\alpha}\,, \quad S_\alpha = \frac{1}{n}\sum_{\alpha'}(X_{\alpha'\alpha} - M_\alpha)^2 \end{aligned}}$$

where $\boldsymbol{A}$, $\boldsymbol{B}$ are learnable parameters, α' and α are some multi-indices which arbitrarily divide all indices of tensor $\boldsymbol{X}$, $1 = \sum_{\alpha'} n^{-1}$ and $\epsilon \simeq 10^{-8}$ is used to prevent division by 0. If the multi-index α' along which we normalize the data contains the batch dimension then such normalization is called *batch normalization*. Such normalization introduces some problems since the examples get mixed and are no longer i.i.d. Additionaly the layer behaves differently during training and inference since during inference we can no longer compute the normalization along the batch dimension. If, on the other hand, the indices α' do not contain the batch dimension then such a normalization is called *layer normalization*.

We also show the derivation of the generalized vector-jacobian product required in the backpropagation algorithm. First we compute the jacobian

$$\begin{aligned} \frac{\partial F_{\alpha'\alpha}}{\partial X_{\beta'\beta}} = A_\alpha \Big[&(S_\alpha + \epsilon)^{-\frac{1}{2}} \delta_{\alpha'\beta'}\delta_{\alpha\beta} \\ &- \frac{1}{2}(S_\alpha + \epsilon)^{-\frac{3}{2}}(X_{\alpha'\alpha} - M_\alpha)\frac{\partial S_\alpha}{\partial X_{\beta'\beta}} \\ &- (S_\alpha + \epsilon)^{-\frac{1}{2}} \frac{\partial M_\alpha}{\partial X_{\beta'\beta}} \Big] \quad . \end{aligned}$$

It is easy to show that

$$\frac{\partial M_\alpha}{\partial X_{\beta'\beta}} = \frac{1}{n}\delta_{\alpha\beta}\,, \quad \frac{\partial S_\alpha}{\partial X_{\beta'\beta}} = \frac{2}{n}(X_{\beta'\alpha} - M_\alpha)\delta_{\alpha\beta}\,,$$

thus the vector-jacobian product is given by

$$\begin{aligned} \frac{\partial L}{\partial X_{\beta'\beta}} =& \sum_{\alpha'\alpha} \frac{\partial L}{\partial F_{\alpha'\alpha}} \frac{\partial F_{\alpha'\alpha}}{\partial X_{\beta'\beta}} \\ =& A_\beta (S_\beta + \epsilon)^{-\frac{1}{2}} \frac{\partial L}{\partial F_{\beta'\beta}} \\ &- \frac{1}{n}(S_\beta + \epsilon)^{-\frac{3}{2}}(X_{\beta'\beta} - M_\beta) \sum_{\alpha'} \frac{\partial L}{\partial F_{\alpha'\beta}} \\ &- \frac{1}{n}(S_\beta + \epsilon)^{-\frac{1}{2}} \sum_{\alpha'} \frac{\partial L}{\partial F_{\alpha'\beta}} \end{aligned}$$

Denoting $\hat{X}_{\alpha'\alpha} = (X_{\alpha'\alpha} - M_\alpha)/\sqrt{S_\alpha + \epsilon}$ we can write this expression in a more elegant form

$$\boxed{\begin{aligned} &\frac{\partial L}{\partial X_{\beta'\beta}} = \\ &\frac{A_\beta}{n\sqrt{S_\beta + \epsilon}} \left[n \frac{\partial L}{\partial F_{\beta'\beta}} - \sum_{\alpha'} \frac{\partial L}{\partial F_{\alpha'\beta}} - \hat{X}_{\beta'\beta} \sum_{\alpha'} \frac{\partial L}{\partial F_{\alpha'\beta}} \hat{X}_{\alpha'\beta} \right] \end{aligned}}$$

Additionally the vector-jacobian products for the parameters are given by

$$\boxed{\begin{aligned} \frac{\partial L}{\partial A_\beta} &= \sum_{\alpha'\alpha} \frac{\partial L}{\partial F_{\alpha'\alpha}} \frac{\partial F_{\alpha'\alpha}}{\partial A_\beta} = \sum_{\alpha'} \frac{\partial L}{\partial F_{\alpha'\beta}} \hat{X}_{\alpha'\beta} \\ \frac{\partial L}{\partial B_\beta} &= \sum_{\alpha'\alpha} \frac{\partial L}{\partial F_{\alpha'\alpha}} \frac{\partial F_{\alpha'\alpha}}{\partial B_\beta} = \sum_{\alpha'} \frac{\partial L}{\partial F_{\alpha'\beta}} \end{aligned}}$$

9 Architectures

9.1 MLP

9.2 RBM

9.3 DBN

9.4 CNN

9.5 Autoencoder

9.6 VAE

9.7 GAN

9.8 DDPM

9.9 Transformer

10 Interpretability