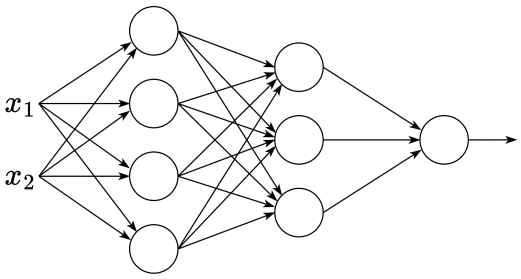
1 What is a neural network?

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 tensor 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 $\boldsymbol{\theta}$ utilizing the gradient information. Let $\boldsymbol{\theta}_t$ denote the value of parameters $\boldsymbol{\theta}$ at step t and let $\boldsymbol{v}_t$ be the values of the functions $\boldsymbol{F}$ at step t. In each step we take a batch $\mathcal{X}$ of training data, perform forward propagation to compute values $\boldsymbol{v}_t$ and the value of the loss function $L(\mathcal{X}, \boldsymbol{\theta}_t)$, next perform backward propagation to compute the values of gradients $(\partial_{\boldsymbol{\theta}} L)(\mathcal{X}, \boldsymbol{\theta}_t)$ and afterwards we update the parameters according to

$$\boxed{\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \frac{\partial L}{\partial \boldsymbol{\theta}}\bigg|_{\boldsymbol{\theta}=\boldsymbol{\theta}_t}}$$

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

$$\boxed{\boldsymbol{V}_{t+1} = \mu \boldsymbol{V}_t - \eta \frac{\partial L}{\partial \boldsymbol{\theta}}\bigg|_{\boldsymbol{\theta}=\boldsymbol{\theta}_t}}$$

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

$$\boxed{\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \boldsymbol{V}_{t+1}}$$

5.2 Nesterov's Accelerated Gradient

A simple modification of classical momentum, which (at least theoretically, for convex functions) improves the convergence rate of the SGD algorithm is the following. Whereas classical momentum updates the velocities according to

$$\boldsymbol{V}_{t+1} = \mu \boldsymbol{V}_t - \eta \frac{\partial L}{\partial \boldsymbol{\theta}}\bigg|_{\boldsymbol{\theta}=\boldsymbol{\theta}_t}$$

the Nesterov method uses

$$\boxed{\boldsymbol{V}_{t+1} = \mu \boldsymbol{V}_t - \eta \frac{\partial L}{\partial \boldsymbol{\theta}}\bigg|_{\boldsymbol{\theta}=\boldsymbol{\theta}_t + \mu \boldsymbol{V}_t}}$$

Using the equation above in the way it was written creates certain problems – the problem is that the parameters $\boldsymbol{\theta}$ we keep in the layers are not the ones we need to compute forward and backward passes. A simple solution of this complication is to use parameters $\boldsymbol{\phi}_t = \boldsymbol{\theta}_t + \mu \boldsymbol{V}_t$ instead of $\boldsymbol{\theta}_t$ since assuming the same initialization, $\boldsymbol{V}_0 = 0$ and $\boldsymbol{V}_T \approx 0$ (where T is the number of training epochs) the values of both parameters coincide and the update equations (***Bengio's equations***) for $\boldsymbol{V}_t$ and $\boldsymbol{\phi}_t$ have a very simple form

$$\boxed{\begin{aligned} \boldsymbol{V}_{t+1} &= \mu \boldsymbol{V}_t - \eta \frac{\partial L}{\partial \boldsymbol{\theta}}\bigg|_{\boldsymbol{\theta}=\boldsymbol{\phi}_t} \\ \boldsymbol{\phi}_{t+1} &= \boldsymbol{\phi}_t + \mu^2 \boldsymbol{V}_t - \eta(1+\mu) \frac{\partial L}{\partial \boldsymbol{\theta}}\bigg|_{\boldsymbol{\theta}=\boldsymbol{\phi}_t} \end{aligned}}$$

5.3 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.

$$\boxed{\begin{aligned} \boldsymbol{M}_{t+1} &= \beta_1 \boldsymbol{M}_t + (1-\beta_1) \frac{\partial L}{\partial \boldsymbol{\theta}}\bigg|_{\boldsymbol{\theta}=\boldsymbol{\theta}_t} \\ \boldsymbol{V}_{t+1} &= \beta_2 \boldsymbol{V}_t + (1-\beta_2) \left[\frac{\partial L}{\partial \boldsymbol{\theta}}\bigg|_{\boldsymbol{\theta}=\boldsymbol{\theta}_t}\right]^2 \\ \widetilde{\boldsymbol{M}}_{t+1} &= \frac{\boldsymbol{M}_{t+1}}{1-\beta_1^t} \\ \widetilde{\boldsymbol{V}}_{t+1} &= \frac{\boldsymbol{V}_{t+1}}{1-\beta_2^t} \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta \frac{\widetilde{\boldsymbol{M}}_{t+1}}{\sqrt{\widetilde{\boldsymbol{V}}_{t+1}} + \epsilon} \end{aligned}}$$

where all operations are performed element-wise, $\epsilon \simeq 10^{-8}$ 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 the learning rate η is $\eta = 3\text{e}{-4}$.

5.4 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 ensemble 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 specifying certain invariances of the network by modifying 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 parameters 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. Additionally 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

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

Denoting $\hat{X}_{\alpha'\alpha} = (X_{\alpha'\alpha} - M_\alpha)/\sqrt{S_\alpha + \epsilon}$ we can write the vector-jacobian product as

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

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 Architecture

9.1 MLP

Multilayer perceptron (MLP) is perhaps the simplest feed-forward neural network architecture. Its computational graph is a simple directed path with computational nodes consisting (in the vanilla case) of two types – ***linear*** (or ***affine***) nodes $\boldsymbol{A}$ which realize the following affine mapping

$$A_{\alpha'\alpha}(\boldsymbol{X};\boldsymbol{W},\boldsymbol{b}) = b_\alpha + \sum_\beta X_{\alpha'\beta}W_{\beta\alpha}\,,$$

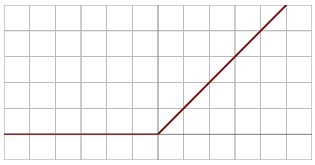
parametrized by weights' matrix $\boldsymbol{W}$ and bias vector $\boldsymbol{b}$, and ***elementwise non-linear activations*** $\boldsymbol{F}$ which for given scalar activation function $f : \mathbb{R} \mapsto \mathbb{R}$ realize the following tensor mapping

$$F_{\alpha'\alpha}(\boldsymbol{X}) = f(X_{\alpha'\alpha})\,.$$

Commonly used additions to this simple setup are dropout layers described above which help to prevent overfitting. Typically used activations include:

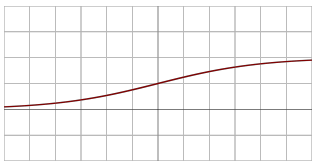
- rectified linear unit activation

$$\mathrm{ReLU} : \mathbb{R} \mapsto [0; +\infty)$$
$$\mathrm{ReLU}(z) = \max\{0, z\}$$



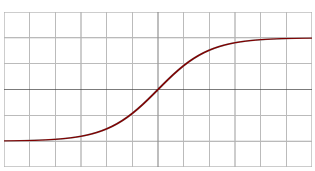
- logistic activation

$$\sigma : \mathbb{R} \mapsto (0; 1)$$
$$\sigma(z) = \frac{1}{1+\mathrm{e}^{-z}}$$



- hyperbolic tangent activation

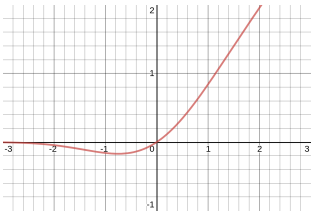
$$\tanh : \mathbb{R} \mapsto (-1; +1)$$
$$\tanh(z) = \frac{\mathrm{e}^{z}-\mathrm{e}^{-z}}{\mathrm{e}^{z}+\mathrm{e}^{-z}}$$



- Gaussian error linear unit activation

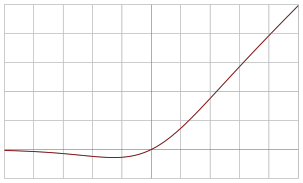
$$\mathrm{GELU}(z) = z\Phi(z)$$

where Φ is the c.d.f. of the standard normal distribution,



- sigmoid linear unit activation

$$\mathrm{SiLU}(z) = z\sigma(z)$$



The first one on the list is the most commonly used activation function, the next two were once very popular but nowadays find applications only in some specific use-cases, meanwhile the last two are relatively new modifications of the field-tested ReLU.

Here, we also derive the (very simple) expressions for the vjp-s of both the linear and activation layers required in the backpropagation algorithm. Indeed for the non-linear activation layer $\boldsymbol{F}$ we have

$$\frac{\partial F_{\alpha'\alpha}}{\partial X_{\mu\nu}} = f'(X_{\alpha'\alpha})\delta_{\alpha'\mu}\delta_{\alpha\nu}$$

and thus

$$\boxed{\frac{\partial L}{\partial X_{\mu\nu}} = \sum_{\alpha'\alpha}\frac{\partial L}{\partial F_{\alpha'\alpha}}\frac{\partial F_{\alpha'\alpha}}{\partial X_{\mu\nu}} = \frac{\partial L}{\partial F_{\mu\nu}}f'(X_{\mu\nu})\,.}$$

On the other hand for the linear layer $\boldsymbol{A}$ we have

$$\frac{\partial A_{\alpha'\alpha}}{\partial X_{\mu\nu}} = \sum_\beta W_{\beta\alpha}\delta_{\alpha'\mu}\delta_{\beta\nu} = W_{\nu\alpha}\delta_{\alpha'\mu}$$
$$\frac{\partial A_{\alpha'\alpha}}{\partial W_{\mu\nu}} = \sum_\beta X_{\alpha'\beta}\delta_{\beta\mu}\delta_{\alpha\nu} = X_{\alpha'\mu}\delta_{\alpha\nu}$$
$$\frac{\partial A_{\alpha'\alpha}}{\partial b_\mu} = \delta_{\alpha\mu}$$

and thus

$$\boxed{\begin{aligned}\frac{\partial L}{\partial X_{\mu\nu}} &= \sum_{\alpha'\alpha} \frac{\partial L}{\partial A_{\alpha'\alpha}} \frac{\partial A_{\alpha'\alpha}}{\partial X_{\mu\nu}} = \sum_{\alpha} \frac{\partial L}{\partial A_{\mu\alpha}} W_{\nu\alpha} \\ \frac{\partial L}{\partial W_{\mu\nu}} &= \sum_{\alpha'\alpha} \frac{\partial L}{\partial A_{\alpha'\alpha}} \frac{\partial A_{\alpha'\alpha}}{\partial W_{\mu\nu}} = \sum_{\alpha'} \frac{\partial L}{\partial A_{\alpha'\nu}} X_{\alpha'\mu} \\ \frac{\partial L}{\partial b_{\mu}} &= \sum_{\alpha'\alpha} \frac{\partial L}{\partial A_{\alpha'\alpha}} \frac{\partial A_{\alpha'\alpha}}{\partial b_{\mu}} = \sum_{\alpha'} \frac{\partial L}{\partial A_{\alpha'\mu}}\end{aligned}}$$

Here, we will also comment on the incorporation of the dropout layers. Note that the dropout layer $\boldsymbol{D}$ is actually just a multiplication of the input tensor by a "constant" 0-1 tensor i.e.

$$D_{\alpha'\alpha}(\boldsymbol{X}) = B_{\alpha'\alpha} X_{\alpha'\alpha} \,.$$

The tensor $\boldsymbol{B}$ is obviously different in the training phase (during which in every forward pass its every element is randomly sampled from the Bernoulli distribution with given parameter p i.e. the dropout probability) and the inference phase (during which it is just a constant matrix with value p being the dropout probability). The vjp-s for the dropout layer required in the backpropagation algorithm are thus simply given by

$$\boxed{\frac{\partial L}{\partial X_{\mu\nu}} = \sum_{\alpha'\alpha} \frac{\partial L}{\partial D_{\alpha'\alpha}} \frac{\partial D_{\alpha'\alpha}}{\partial X_{\mu\nu}} = \frac{\partial L}{\partial D_{\mu\nu}} B_{\mu\nu} \,,}$$

where $\boldsymbol{B}$ is the current 0-1 matrix sampled in the latest forward pass.

9.2 RBM

Generative model is a probabilistic model which enables us to generate new, synthetic observations similar to the one from training set. Some generative models allow us to estimate the probability density based on the finite number of samples (observations). There are also generative models in which we cannot analytically compute the value of the probability density function, however we can still sample from such model to obtain synthetic observations. Moreover a generative model can „explain" the observations using so called ***latent*** variables.

We will now focus on the so called ***energy based models***. The general idea is to introduce a parametrized ***energy function*** $E(\boldsymbol{v}, \boldsymbol{h}; \boldsymbol{\theta})$, where $\boldsymbol{v} \in V \subseteq \mathbb{R}^{d_v}$ is the observation vector, $\boldsymbol{h} \in H \subseteq \mathbb{R}^{d_h}$ is the latent variable vector and $\boldsymbol{\theta}$ are the parameters. We now assume that the joint probability is given by a Boltzmann factor

$$p(\boldsymbol{v}, \boldsymbol{h} \mid \boldsymbol{\theta}) = \frac{\mathrm{e}^{-E(\boldsymbol{v},\boldsymbol{h};\boldsymbol{\theta})}}{Z(\boldsymbol{\theta})}$$

where $Z(\boldsymbol{\theta})$ is the normalization constant

$$Z(\boldsymbol{\theta}) = \sum_{\boldsymbol{v}\in V, \boldsymbol{h}\in H} \mathrm{e}^{-E(\boldsymbol{v},\boldsymbol{h};\boldsymbol{\theta})}$$

called also (in analogy to statistical physics) – the ***partition function***. Because of the assumption of the Boltzmann-like probability distribution, these models are often called ***Boltzmann machines***. These models are trained in a similar fashion as standard neural networks. We first note that given the joint distribution we can compute the marginal distribution of observations

$$\begin{aligned} p(\boldsymbol{v} \mid \boldsymbol{\theta}) &= \sum_{\boldsymbol{h}\in H} p(\boldsymbol{v}, \boldsymbol{h} \mid \boldsymbol{\theta}) \\ &= \frac{1}{Z(\boldsymbol{\theta})} \sum_{\boldsymbol{h}\in H} \mathrm{e}^{-E(\boldsymbol{v},\boldsymbol{h};\boldsymbol{\theta})} \,. \end{aligned}$$

and thus can use the standard maximum likelihood criterion to train the model parameters i.e. given the training set $\mathcal{X} = \{\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(N)}\}$ of i.i.d. samples, we seek parameters $\boldsymbol{\theta}^*$ given by

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \left\{ -\frac{1}{N} \sum_{i=1}^{N} \log p\left(\boldsymbol{v}^{(i)} \mid \boldsymbol{\theta}\right) \right\} .$$

To find this minimum we will use one of the gradient descent algorithms and thus we will need to know the derivative of the loss function w.r.t. the parameters. We will now derive the expression for the derivative for arbitrary energy function. Indeed we have

$$\frac{\partial L(\mathcal{X}; \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = -\frac{1}{N} \sum_{i=1}^{N} \frac{1}{p\left(\boldsymbol{v}^{(i)} \mid \boldsymbol{\theta}\right)} \frac{\partial p\left(\boldsymbol{v}^{(i)} \mid \boldsymbol{\theta}\right)}{\partial \boldsymbol{\theta}} \,.$$

Expanding the expression for $p(\boldsymbol{v}^{(i)} \mid \boldsymbol{\theta})$ we get

$$\begin{aligned} \frac{\partial p(\boldsymbol{v}^{(i)} \mid \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = &- \frac{1}{Z^2(\boldsymbol{\theta})} \frac{\partial Z(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \sum_{\boldsymbol{h}\in H} \mathrm{e}^{-E(\boldsymbol{v}^{(i)},\boldsymbol{h};\boldsymbol{\theta})} \\ &- \frac{1}{Z(\boldsymbol{\theta})} \sum_{\boldsymbol{h}\in H} \mathrm{e}^{-E(\boldsymbol{v}^{(i)},\boldsymbol{h};\boldsymbol{\theta})} \frac{\partial E(\boldsymbol{v}^{(i)}, \boldsymbol{h}; \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \end{aligned}$$

and computing the derivative of partition function we obtain

$$\frac{\partial Z(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = -\sum_{\boldsymbol{v}\in V, \boldsymbol{h}\in H} \mathrm{e}^{-E(\boldsymbol{v},\boldsymbol{h};\boldsymbol{\theta})} \frac{\partial E(\boldsymbol{v}, \boldsymbol{h}; \boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$$

and thus putting it all together we get the following

expression for the derivative of loss function

$$\begin{aligned}&\frac{\partial L(\mathcal{X};\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\\&= -\frac{1}{N}\sum_{i=1}^{N}\left\{\sum_{\boldsymbol{v}\in V,\boldsymbol{h}\in H}\left[\frac{\mathrm{e}^{-E(\boldsymbol{v},\boldsymbol{h};\boldsymbol{\theta})}}{Z(\boldsymbol{\theta})}\cdot\frac{\partial E(\boldsymbol{v},\boldsymbol{h};\boldsymbol{\theta})}{\partial\boldsymbol{\theta}}\right]\right.\\&\left.-\sum_{\boldsymbol{h}\in H}\left[\frac{\mathrm{e}^{-E(\boldsymbol{v}^{(i)},\boldsymbol{h};\boldsymbol{\theta})}}{\sum_{\boldsymbol{h}'\in H}\mathrm{e}^{-E(\boldsymbol{v}^{(i)},\boldsymbol{h}';\boldsymbol{\theta})}}\cdot\frac{\partial E(\boldsymbol{v}^{(i)},\boldsymbol{h};\boldsymbol{\theta})}{\partial\boldsymbol{\theta}}\right]\right\}\end{aligned}$$

This expression looks awful, but let's notice that

$$\begin{aligned}p(\boldsymbol{h}\mid\boldsymbol{v}^{(i)},\boldsymbol{\theta})&=\frac{p(\boldsymbol{v}^{(i)},\boldsymbol{h}\mid\boldsymbol{\theta})}{\sum_{\boldsymbol{h}'\in H}p(\boldsymbol{v}^{(i)},\boldsymbol{h}'\mid\boldsymbol{\theta})}\\&=\frac{\mathrm{e}^{-E(\boldsymbol{v}^{(i)},\boldsymbol{h};\boldsymbol{\theta})}}{\sum_{\boldsymbol{h}'\in H}\mathrm{e}^{-E(\boldsymbol{v}^{(i)},\boldsymbol{h}';\boldsymbol{\theta})}}\end{aligned}$$

thus the expression can be written as

$$\begin{aligned}\frac{\partial L(\mathcal{X};\boldsymbol{\theta})}{\partial\boldsymbol{\theta}}&=-\mathbb{E}_{\boldsymbol{v},\boldsymbol{h}\sim p(\boldsymbol{v},\boldsymbol{h}\mid\boldsymbol{\theta})}\left[\frac{\partial E(\boldsymbol{v},\boldsymbol{h};\boldsymbol{\theta})}{\partial\boldsymbol{\theta}}\right]\\&\quad+\frac{1}{N}\sum_{i=1}^{N}\mathbb{E}_{\boldsymbol{h}\sim p(\boldsymbol{h}\mid\boldsymbol{v}^{(i)},\boldsymbol{\theta})}\left[\frac{\partial E(\boldsymbol{v}^{(i)},\boldsymbol{h};\boldsymbol{\theta})}{\partial\boldsymbol{\theta}}\right]\end{aligned}$$

The considerations up until now were quite general. We will now consider a specific Boltzmann machine called ***Restricted Boltzmann Machine***, introduce its' energy function and describe effective procedures to compute the gradient of the loss function. Restricted Boltzmann machine is an energy based model with binary units i.e. $\boldsymbol{v}\in V=\{0,1\}^{d_v}$ and $\boldsymbol{h}\in H=\{0,1\}^{d_h}$, described by the following energy function (for clarity of notation we don't write the parameters in the energy function arguments)

$$\begin{aligned}E(\boldsymbol{v},\boldsymbol{h})&=-\boldsymbol{v}\boldsymbol{b}^{\intercal}-\boldsymbol{h}\boldsymbol{c}^{\intercal}-\boldsymbol{v}\boldsymbol{W}\boldsymbol{h}^{\intercal}\\&=-\sum_{\alpha}b_\alpha v_\alpha-\sum_{\beta}c_\beta h_\beta-\sum_{\alpha\beta}v_\alpha W_{\alpha\beta}h_\beta\end{aligned}$$

where $\boldsymbol{W}$, $\boldsymbol{b}$, $\boldsymbol{c}$ are the parameters and the vectors are represented as row matrices. This energy function has a nice property of conditional independence. Indeed consider the conditional distribution

$$\begin{aligned}p(\boldsymbol{h}\mid\boldsymbol{v})&=\frac{\mathrm{e}^{-E(\boldsymbol{v},\boldsymbol{h})}}{\sum_{\boldsymbol{h}'\in H}\mathrm{e}^{-E(\boldsymbol{v},\boldsymbol{h}')}}\\&=\frac{\prod_\beta\mathrm{e}^{(c_\beta h_\beta+\sum_\alpha v_\alpha W_{\alpha\beta}h_\beta)}}{\sum_{\boldsymbol{h}'\in H}\prod_\beta\mathrm{e}^{(c_\beta h'_\beta+\sum_\alpha v_\alpha W_{\alpha\beta}h'_\beta)}}\end{aligned}$$

and note that

$$\begin{aligned}&\sum_{\boldsymbol{h}'\in H}\prod_\beta\mathrm{e}^{(c_\beta h'_\beta+\sum_\alpha v_\alpha W_{\alpha\beta}h'_\beta)}\\&=\prod_\beta\sum_{h'_\beta\in\{0,1\}}\mathrm{e}^{(c_\beta h'_\beta+\sum_\alpha v_\alpha W_{\alpha\beta}h'_\beta)}\end{aligned}$$

thus

$$p(\boldsymbol{h}\mid\boldsymbol{v})=\prod_\beta\frac{\mathrm{e}^{(c_\beta h_\beta+\sum_\alpha v_\alpha W_{\alpha\beta}h_\beta)}}{1+\mathrm{e}^{(c_\beta+\sum_\alpha v_\alpha W_{\alpha\beta})}}$$

and by symmetry

$$p(\boldsymbol{v}\mid\boldsymbol{h})=\prod_\alpha\frac{\mathrm{e}^{(b_\alpha v_\alpha+\sum_\beta v_\alpha W_{\alpha\beta}h_\beta)}}{1+\mathrm{e}^{(b_\alpha+\sum_\beta W_{\alpha\beta}h_\beta)}}\,.$$

We can see therefore that given $\boldsymbol{v}$ (or $\boldsymbol{h}$) each element of $\boldsymbol{h}$ (or $\boldsymbol{v}$) is given by the Bernoulli distribution with parameters

$$\begin{aligned}p(v_\alpha=1\mid\boldsymbol{h})&=\sigma\left(b_\alpha+\sum_\beta W_{\alpha\beta}h_\beta\right)\\p(h_\beta=1\mid\boldsymbol{v})&=\sigma\left(c_\beta+\sum_\alpha v_\alpha W_{\alpha\beta}\right)\end{aligned}$$

where σ is the logistic function. This observation allows us to compute the second term in the loss derivative formula

$$\begin{aligned}\mathbb{E}_{\boldsymbol{h}\sim p(\boldsymbol{h}\mid\boldsymbol{v}^{(i)})}\left[\frac{\partial E(\boldsymbol{v}^{(i)},\boldsymbol{h})}{\partial W_{\mu\nu}}\right]&=-v_\mu^{(i)}\sigma\left(c_\nu+\sum_\alpha v_\alpha^{(i)}W_{\alpha\nu}\right)\\\mathbb{E}_{\boldsymbol{h}\sim p(\boldsymbol{h}\mid\boldsymbol{v}^{(i)})}\left[\frac{\partial E(\boldsymbol{v}^{(i)},\boldsymbol{h})}{\partial b_\mu}\right]&=-v_\mu^{(i)}\\\mathbb{E}_{\boldsymbol{h}\sim p(\boldsymbol{h}\mid\boldsymbol{v}^{(i)})}\left[\frac{\partial E(\boldsymbol{v}^{(i)},\boldsymbol{h})}{\partial c_\nu}\right]&=-\sigma\left(c_\nu+\sum_\alpha v_\alpha^{(i)}W_{\alpha\nu}\right)\end{aligned}$$

The first term however is much more difficult to compute. In fact in practice it is impossible to compute as it requires summation over exponentially many values. To overcome this problem we introduce two heuristics which enable us to train RBMs in practice.

9.2.1 Contrastive Divergence

The first heuristic is based on a simple observation – we can estimate the expectation of the gradient of the energy by sampling N vectors $\underline{\boldsymbol{v}}^{(i)}$ from marginal distribution $p(\boldsymbol{v})$ and approximating the true expectation as an average

$$\begin{aligned}&\mathbb{E}_{\boldsymbol{v},\boldsymbol{h}\sim p(\boldsymbol{v},\boldsymbol{h}\mid\boldsymbol{\theta})}\left[\frac{\partial E(\boldsymbol{v},\boldsymbol{h};\boldsymbol{\theta})}{\partial\boldsymbol{\theta}}\right]\\&\approx\frac{1}{N}\sum_{i=1}^{N}\mathbb{E}_{\boldsymbol{h}\sim p(\boldsymbol{h}\mid\underline{\boldsymbol{v}}^{(i)},\boldsymbol{\theta})}\left[\frac{\partial E(\underline{\boldsymbol{v}}^{(i)},\boldsymbol{h};\boldsymbol{\theta})}{\partial\boldsymbol{\theta}}\right].\end{aligned}$$

This is obviously due to the fact that

$$\begin{aligned}\mathbb{E}_{\boldsymbol{v},\boldsymbol{h}\sim p(\boldsymbol{v},\boldsymbol{h}|\boldsymbol{\theta})}[f] &= \sum_{\boldsymbol{v}\in V}\sum_{\boldsymbol{h}\in H} fp(\boldsymbol{v},\boldsymbol{h}\mid\boldsymbol{\theta}) \\ &= \sum_{\boldsymbol{v}\in V}\left\{\left[\sum_{\boldsymbol{h}\in H} fp(\boldsymbol{h}\mid\boldsymbol{v},\boldsymbol{\theta})\right]p(\boldsymbol{v}\mid\boldsymbol{\theta})\right\} \\ &= \sum_{\boldsymbol{v}\in V}\mathbb{E}_{\boldsymbol{h}\sim p(\boldsymbol{h}|\boldsymbol{v},\boldsymbol{\theta})}[f]p(\boldsymbol{v}\mid\boldsymbol{\theta}) \\ &\approx \frac{1}{N}\sum_{i=1}^{N}\mathbb{E}_{\boldsymbol{h}\sim p(\boldsymbol{h}|\underline{\boldsymbol{v}}^{(i)},\boldsymbol{\theta})}[f]\end{aligned}.$$

Since we can easily compute the conditional expectation, using the above formula we can estimate the true gradient of the parameters. How do we sample from the marginal distribution distribution? We can use the Gibbs sampling algorithm which is an instance of general class of MCMC (Markov Chain Monte Carlo) algorithms. We start with $\underline{\boldsymbol{v}}_0^{(i)} = \boldsymbol{v}^{(i)}$, next we sample $\underline{\boldsymbol{h}}_1^{(i)}$ from the conditional distribution $p(\boldsymbol{h}\mid\underline{\boldsymbol{v}}_0^{(i)})$, after that we sample $\underline{\boldsymbol{v}}_1^{(i)}$ from $p(\boldsymbol{v}\mid\underline{\boldsymbol{h}}_1^{(i)})$ and so on. We repeat the steps k times and in the end obtain samples $\underline{\boldsymbol{v}}_k^{(i)}$. We then approximate the expectations as

$$\mathbb{E}_{\boldsymbol{v},\boldsymbol{h}\sim p(\boldsymbol{v},\boldsymbol{h})}\left[\frac{\partial E(\boldsymbol{v},\boldsymbol{h})}{\partial W_{\mu\nu}}\right] \approx -\frac{1}{N}\sum_{i=1}^{N}\underline{v}_\mu^{(i,k)}\sigma\left(c_\nu + \sum_\alpha \underline{v}_\alpha^{(i,k)}W_{\alpha\nu}\right)$$

$$\mathbb{E}_{\boldsymbol{v},\boldsymbol{h}\sim p(\boldsymbol{v},\boldsymbol{h})}\left[\frac{\partial E(\boldsymbol{v},\boldsymbol{h})}{\partial b_\mu}\right] \approx -\frac{1}{N}\sum_{i=1}^{N}\underline{v}_\mu^{(i,k)}$$

$$\mathbb{E}_{\boldsymbol{v},\boldsymbol{h}\sim p(\boldsymbol{v},\boldsymbol{h})}\left[\frac{\partial E(\boldsymbol{v},\boldsymbol{h})}{\partial c_\nu}\right] \approx -\frac{1}{N}\sum_{i=1}^{N}\sigma\left(c_\nu + \sum_\alpha \underline{v}_\alpha^{(i,k)}W_{\alpha\nu}\right)$$

and are finally able to compute the gradients and update the parameters.

9.2.2 Persistent Contrastive Divergence

The algorithm described in the previous paragraph has one serious flaw – it is extremely slow, as for it to work properly each parameters' update would require hundreds or thousands of Gibbs sampling steps. In practice it is, therefore, often implemented in such a way that we only perform few sampling steps (even as few as 1) but this comes with a serious problem – we no longer approximate the true expectation with random error – the error is now systematic.

To overcome this difficulty we introduce the idea of a ***persistent chain***. The core idea is to spread the Gibbs sampling computation over many parameters' updates, i.e. during each parameters' update we perform only a few Gibbs sampling steps but retain the set of hidden vectors $\boldsymbol{h}^{(j)}$ (which we will call the persistent chain) from which we start sampling in the next update. The outline of the algorithm is thus as follows – first we initialize the persistent chain $\{\boldsymbol{p}^{(j)} = \boldsymbol{0}\}_{j=1}^{M}$; during each parameters' update we perform k Gibbs sampling steps starting from $\underline{\boldsymbol{h}}_1^{(j)} = \boldsymbol{p}^{(j)}$ and obtain a sample $\boldsymbol{v}_k^{(j)}$, we approximate the expectations as before (noting that now we average over M samples) and in the end update the persistent chain by sampling M vectors $\boldsymbol{p}^{(j)}$ from the distribution $p(\boldsymbol{h}\mid\underline{\boldsymbol{v}}_k^{(j)})$.

9.3 DBN

Let's note that RBM can be interpreted as a network of units connected in layers as depicted in the figure below. The nodes on the left represent the visible binary units, the nodes on the right represent the hidden units and edges represent the interactions of the type $W_{\alpha\beta}v_\alpha h_\beta$ (there are also biases, not depicted in the picture).

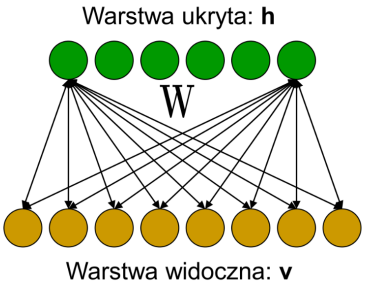


Figure 1: Schematic depiction of RBM with 8 visible units and 6 hidden units

Note that before we had not made any explicit connection between the RBM and neural networks. We simply treated it like an energy based model and the energy function was our primary object of interest. The interpretation of the RBM as an undirected, stochastic neural network is possible only due to the conditional independence property of $p(\boldsymbol{h}\mid\boldsymbol{v})$ and $p(\boldsymbol{v}\mid\boldsymbol{h})$. The network thus operates as follows – we feed it some binary example vector $\boldsymbol{v}$, then compute the ***activations*** of the hidden units $\sigma(\boldsymbol{b}+\boldsymbol{v}\boldsymbol{W})$ and then sample the hidden units given the activations from the Bernoulli distributions, we can then do the same thing in the opposite direction and sample from the visible layer. Performing these steps many times (that is performing Gibbs sampling) we obtain in the end an artificial sample from the network itself and if the network has been trained on some data $\mathcal{X}$ then these samples are similar (but not the same) to the empirical samples.

Deep Belief Network (DBN) is basically multiple RBMs stacked on top of each other as depicted in the picture below. We can implement this network as a sequence of RBMs with parameters $\boldsymbol{W}^{(1)}, \boldsymbol{b}^{(1)}, \boldsymbol{c}^{(1)}$, $\ldots$, $\boldsymbol{W}^{(l)}, \boldsymbol{b}^{(l)}, \boldsymbol{c}^{(l)}$ respectively (l being the number of layers). We can train the DBN using a simple heuristic called ***greedy layer-wise training***. The idea is simple – we sequentially train each RBM i on the dataset $\mathcal{X}$ propagated through the $i-1$ first layers (i.e. we compute only the activations and do not sample the units – this still turns out to be working but slightly differs from the theory presented above). After training we can sample DBN by propagating the visible units $\boldsymbol{v}$ up to the last RBM, performing Gibbs sampling in this last RBM and propagating the output down to the first RBM.

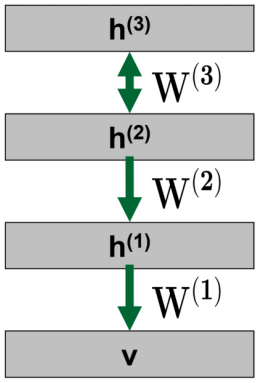


Figure 2: Schematic depiction of DBN with 4 layers

9.3.1 MLP initialization using DBN

DBNs can be used to initialize weights in the MLP layers and thus improve the overall performance of the MLP network. The improvement can be huge, especially if we have lots of unlabeled data and only a handful of labeled examples as is the case e.g. when the labeling process is costly, difficult or time-consuming. The implementation is straight-forward. We train the n-layer-deep DBN in an unsupervised manner using the greedy layer-wise training algorithm. We then construct an MLP with $n+1$ layers, initialize the first n layers with the weights and biases of the pretrained DBN and finetune the MLP on the smaller set of labeled examples. This type of initialization is also called ***greedy layer-wise pretraining***. One key aspect to remember when using this approach is that the activations of the MLP must match the "activations" of the DBN. If we use the logistic sigmoid activations in the MLP then nothing is changed in the RBM, however if we would like to use ReLU activation function then one may show that the RBM activations should given by $\mathrm{ReLU}(\boldsymbol{b}+\boldsymbol{v}\boldsymbol{W})$ and to sample the hidden units we should use the so called ***noisy rectified linear unit***

$$\text{N-ReLU}(z) = \max\{0, z + \sqrt{\sigma(z)} \cdot \epsilon\}\,,$$

where $\epsilon \sim \mathcal{N}(0,1)$.

9.4 Autoencoder

Autoencoder is a general type of neural architecture used for unsupervised learning. The idea is to construct two networks: an encoder network and a decoder network and connect them through the so called ***bottleneck layer***. The encoder, given some high-dimensional input data, compresses them into much more compact representation and the decoder can reconstruct the original data from this compact representation. The encoder and the decoder are trained jointly and learn the suitable representations together. If we care only about the learnt representations we can discard the decoder after training and use only the encoder part.

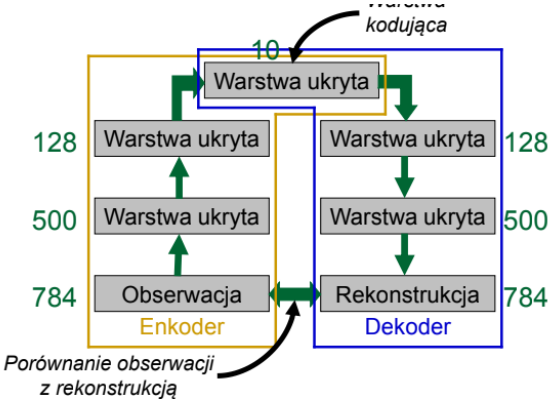


Figure 3: Schematic depiction of a general autoencoder architecture

One important aspect of the training process of the autoencoder is the choice of the loss functions. Since the output of the autoencoder should be as close as possible to the input the typical choice would be the L2 loss. If the data can be modelled using sigmoid units then perhaps the pixelwise cross-entropy loss would be more suitable.

9.5 CNN

Convolutional neural network (CNN) is a type of neural architecture adapted to processing data with certain spatial interdependencies. An obvious example of such data are images represented as 3-dimensional

arrays $X_{c,x,y}$ where the 1st dimension is the so called "channel" dimension (R, G, B) and the other two dimension contain the pixel intensities of the given channel. Processing such data using basic fully-connected networks would be incredibly inefficient as the number of weights would basically scale quadratically with the number of pixels. Convolutional and pooling layers which form the basic building blocks of a CNN were designed to remedy this problem.

$$\text{Conv2D}_{\beta,c,x,y}(\boldsymbol{X};\boldsymbol{K},\boldsymbol{b}) = b_c + \sum_{c'=0}^{C_{\text{in}}-1}\sum_{x'=0}^{H_{\text{filter}}-1}\sum_{y'=0}^{W_{\text{filter}}-1} X_{\beta,c',d_x x'+s_x x,d_y y'+s_y y}\cdot K_{c,c',x',y'}$$

The input to the layer is a stack (batch) of B 3-dimensional tensors with shapes $(C_{\text{in}}, H_{\text{in}}, W_{\text{in}})$ each. The set of the layer's parameters consists of a vector $\boldsymbol{b}$ of shape $(C_{\text{out}},)$ and a stack $\boldsymbol{K}$ of C_{out} 3-dimensional ***filters*** (or ***kernels***) of shape $(C_{\text{in}}, H_{\text{filter}}, W_{\text{filter}})$ each. Additionally the layer has specified hyperparameters: s_x, s_y (***strides***), d_x, d_y (***dilations***). The output of the layer is a batch of B 3-dimensional tensors with shapes $(C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ where

$$H_{\text{out}} = \left\lfloor \frac{H_{\text{in}} - d_x\cdot(H_{\text{filter}}-1)-1}{s_x} + 1 \right\rfloor$$
$$W_{\text{out}} = \left\lfloor \frac{W_{\text{in}} - d_y\cdot(W_{\text{filter}}-1)-1}{s_y} + 1 \right\rfloor$$

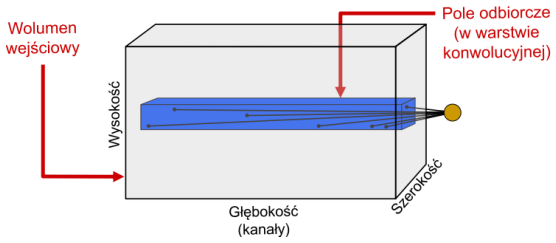


Figure 4: Schematic representation of a convolution operation

Let's note here an important property of the convolutional layer – it is ***equivariant*** to translations of the object in the picture. Indeed the translation of the object corresponds to the equivalent translation of the activations of the layer. The whole network (after applying many convolutional layers, flattening and softmax) shoud thus be ***invariant*** to such translations. Convolutional architecture has therefore a baked-in bias suitable for image data. The convolutional layer changes both the channel dimension and height / width. We can also use padding to counter the height / width shrinkage.

Pooling layers on the other hand are parameter-free layers that operate on each input channel independently and thus do not change the channel dimension but reduce the height / width of the input. Typically we use max-pooling which for a receptive field returns the maximum pixel value.

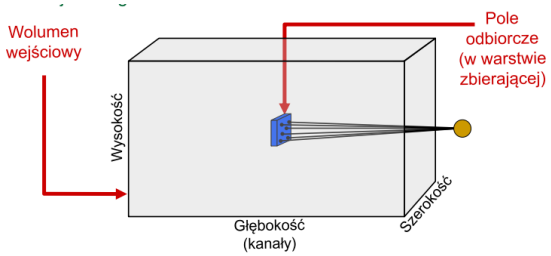


Figure 5: Schematic representation of a pooling operation

9.5.1 Residual connections

Empirical studies of the VGG (Visual Geometry Group) CNNs showed that increasing the depth of the network improved accuracy up to a certain point, beyond which the performance deteriorated. One possible heuristic used to prevent this effect was the introduction of residual connections. This trick is based on an empirical observation that given some complex, nonlinear transformation $\boldsymbol{f}(\boldsymbol{x})$ it is often easier to model and learn the residual transformation i.e. $\boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{x}$. The idea is thus to add special connections between layers of the network so that the parametrized layers learn this residual transformation which is then added to the input to model the full transformation as shown in the Figure 6.

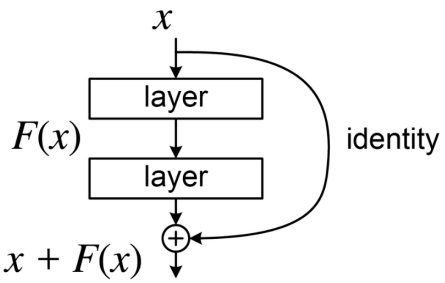


Figure 6: Schematic representation of a residual connection

Of course the output of the series of layers must have compatible shape. We can note that such mechanism certainly improves the gradients flow through

the network which might be one of the reasons for its success.

9.5.2 Some implementation details

Looking at the equation describing the convolution operation, it is not clear how we could implement it in an efficient way. Naive implementation would obviously require 7 nested for loops which access the memory in a non-contiguous way making it highly inefficient. Convolution can however be implemented efficiently using matrix multiplication. Let's focus on the case with $B = 1$. The basic idea is to rewrite the convolution as (we will denote the result of the convolution as $\boldsymbol{C}$)

$$\begin{aligned} C_{c,(x,y)} &= b_c + \sum_{c',x',y'} K_{c,c',x',y'} \cdot X_{c',d_x x'+s_x x, d_y y'+s_y y} \\ &= b_c + \sum_{(c',x',y')} K_{c,(c',x',y')} \cdot \tilde{X}_{(c',x',y'),(x,y)} \end{aligned}$$

where $\tilde{\boldsymbol{X}}$ is given by

$$\tilde{X}_{(c',x',y'),(x,y)} = X_{c',d_x x'+s_x x, d_y y'+s_y y}$$

If we now look at the multi-indices in round brackets as flattened indices, then we get the same structure as matrix multiplication. We thus only have to construct the auxiliary matrix $\tilde{\boldsymbol{X}}$ and use some highly optimized GEMM procedure to compute $\boldsymbol{C}$ (assuming we already store the kernels in the matrix form) and afterwards just reshape $\boldsymbol{C}$ to 3-D. The matrix $\tilde{\boldsymbol{X}}$ is the result of the so called ***im2col*** transformation defined by the equality above (see Figure 7). The im2col transformation can be implemented using NumPy as shown below. Whether there is a faster or simpler implementation possible is unknown to me.

One more topic which requires attention is the computation of the gradient. We can first note that given gradient $\partial L/\partial C_{c,x,y}$ we can easily compute the gradient for the reshaped convolution output. Indeed if $Y_\alpha = X_{\nu(\alpha)}$ for some multi-index α and a bijective function between multi-indices ν then

$$\frac{\partial L}{\partial X_\beta} = \sum_\alpha \frac{\partial L}{\partial Y_\alpha}\frac{\partial Y_\alpha}{\partial X_\beta} = \sum_\alpha \frac{\partial L}{\partial Y_\alpha}\delta_{\beta,\nu(\alpha)}$$

and since we assumed that ν is a bijection therefore there is only a single α s.t. $\nu(\alpha) = \beta$ and it is given by $\nu^{-1}(\beta)$ thus

$$\frac{\partial L}{\partial X_\beta} = \frac{\partial L}{\partial Y_{\nu^{-1}(\beta)}}.$$

Obviously the reshape operation is a bijection. Now, since we can compute gradient w.r.t. the reshaped convolution output, we can also easily compute the gradient w.r.t. the parameters of the layer in almost the same way as for the linear layer, namely assuming we store the kernels as a matrix $\boldsymbol{K}$ we have

$$C_{\alpha\beta} = b_\alpha + \sum_\gamma K_{\alpha\gamma}\tilde{X}_{\gamma\beta}$$

and thus

$$\begin{aligned} \frac{\partial L}{\partial \tilde{X}_{\mu\nu}} &= \sum_{\alpha\beta} \frac{\partial L}{\partial C_{\alpha\beta}}\frac{\partial C_{\alpha\beta}}{\partial \tilde{X}_{\mu\nu}} = \sum_\alpha \frac{\partial L}{\partial C_{\alpha\nu}} K_{\alpha\mu} \\ \frac{\partial L}{\partial K_{\mu\nu}} &= \sum_{\alpha\beta} \frac{\partial L}{\partial C_{\alpha\beta}}\frac{\partial C_{\alpha\beta}}{\partial K_{\mu\nu}} = \sum_\beta \frac{\partial L}{\partial C_{\mu\beta}} \tilde{X}_{\nu\beta} \\ \frac{\partial L}{\partial b_\mu} &= \sum_\beta \frac{\partial L}{\partial C_{\mu\beta}} \end{aligned}$$

The difference being only the dimensions along which we perform the summation.

```python
def im2col(
    x: NDArray,          # input tensor
    s: tuple[int, int],  # strides
    d: tuple[int, int],  # dilation
    k: tuple[int, int],  # kernel size
) -> tuple[NDArray, NDArray]:
  # Get input dimensions
  B, C_in, H_in, W_in = x.shape

  # Compute output size
  H_out = int(1 + (H_in - d[0] * (k[0] - 1) - 1) / s[0])
  W_out = int(1 + (W_in - d[1] * (k[1] - 1) - 1) / s[1])

  # The idea is to first compute an index-array of shape
  # `(B, C_in*H_ker*W_ker, H_out*W_out)` with flat index
  # (i.e. a number between 0 and prod(x.shape) - 1) of the
  # correct element to take from `x` and then use `np.take`.

  # First we compute 1-D index arrays for each dimension.
  # Basically if we would concatenate these arrays as columns
  # then each row would be a unique multi-index to the array
  # of appropriate shape i.e. `(C_in, H_ker, W_ker)` or
  # `(H_out, W_out)`.

  # `(C_in * H_ker * W_ker,)` each
  idx_c, idx_h_ker, idx_w_ker = np.indices((C_in, *k)).reshape(3, -1)

  # `(H_out * W_out,)` each
  idx_h_out, idx_w_out = np.indices((H_out, W_out)).reshape(2, -1)

  # Then we compute 4 index-arrays broadcastable to
  # `(B, C_in * H_ker * W_ker, H_out * W_out)` each, such that forall i,j,k:
  # x[idx_b[i,j,k], idx_c[i,j,k], idx_h[i,j,k], idx_w[i,j,k]] is the
  # correct element to put under index (i,j,k) in the im2col matrix.

  # broadcastable to `(B, C_in * H_ker * W_ker, H_out * W_out)`
  idx_b = np.arange(B).reshape(-1, 1, 1)
  idx_c = idx_c.reshape(-1, 1)
  idx_h = d[0] * idx_h_ker.reshape(-1, 1) + s[0] * idx_h_out
  idx_w = d[1] * idx_w_ker.reshape(-1, 1) + s[1] * idx_w_out

  # Here we could just return x[idx_b, idx_c, idx_h, idx_w]. However
  # for the backward pass it is much more convenient to have these
  # indices raveled i.e. instead of having 4 index-arrays with indices
  # along given (i.e.: 0,1,2,3) dimension we can transform them to a single
  # index-array with raveled index i.e. index to the flattened array `x`.

  # `(B, C_in * H_ker * W_ker, H_out * W_out)`
  idxs = np.ravel_multi_index((idx_b, idx_c, idx_h, idx_w), x.shape)

  # Finally we return the raveled index-array and the im2col array
  # which can be constructed using `np.take` as by default the flattened
  # input array is used to extract the values.
  return idxs, np.take(x, idxs)
```

Listing 1: Im2Col Operation

Now we face an obstacle, how to compute gradient $\partial L/\partial X_\beta$ given the gradient $\partial L/\partial \tilde{X}_\alpha$. Let's note that the index mapping of im2col operation is not a bijection thus in general we can only write

$$\frac{\partial L}{\partial X_\beta} = \sum_\alpha \frac{\partial L}{\partial \tilde{X}_\alpha}\delta_{\beta,\nu(\alpha)}$$

where ν denotes the im2col operation i.e. $\nu((c', x', y'), (x, y)) = (c', d_x x' + s_x x, d_y y' + s_y y)$. Using this expression naively would be very inefficient. Let's note however that we already have precomputed indices of im2col transformation i.e. for every α we have (flattened) index $\nu(\alpha)$. We can thus initialize $\partial L / \partial X_\beta$ with 0s, iterate over all $\alpha = ((c', x', y'), (x, y))$ and accumulate the gradient $\partial L / \partial \tilde{X}_\alpha$ under $\nu(\alpha)$. One additional optimization is to note that id we fix one of the (c', x', y'), (x, y) then for all (x, y) or (c', x', y') indices after transformation ν are distinct which can be used to accumulate the gradients in parallel using some vectorization mechanism.

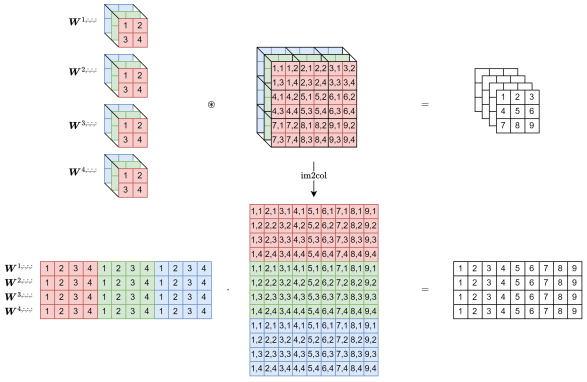


Figure 7: Schematic representation of an im2col operation. Note that for simplicity the receptive fields do not overlap in this example i.e. the strides are equal to the kernel size. In general though they may overlap and this creates duplicated entries in the im2col matrix.

9.6 GAN

We once again come back to the topic of generative models i.e. probabilistic models which enable us to sample new synthetic examples of the modeled observations. This can be achieved by constructing a model which either explicitly or implicitly models the probability distribution of the observed data. In the explicit case we have direct access to the probability density function and can compute the probability values. In the implict case we can only sample from the probability distribution but have no direct way of computing the probability values. A prominent example of the second class of models are the so called Generative Adversarial Networks (GANs). The idea is to express the generating process as a kind of game between two players: the ***generator*** who generates the synthetic observations and the ***discriminator*** whose task is to differentiate the true samples from the synthetic ones.

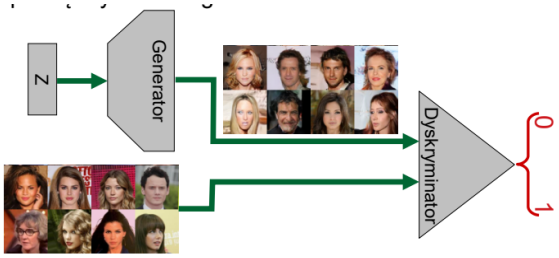


Figure 8: Schematic representation GAN architecture

The discriminator is trained using backpropgation like a simple classifier with binary cross-entropy loss and binary labels indicating whether the image is fake which can be written as

$$\begin{aligned} L_D = &- \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})}[\log D_{\boldsymbol{\phi}}(\boldsymbol{x})] \\ &- \mathbb{E}_{\boldsymbol{z} \sim p(\boldsymbol{z})}[\log(1 - D_{\boldsymbol{\phi}}(G_{\boldsymbol{\theta}}(\boldsymbol{z})))] \end{aligned}$$

In practice we obviously approximate the expectations as averages from the samples, so in reality we sample a batch of examples $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$ and a batch of latent variables $\{\boldsymbol{z}_1, \ldots, \boldsymbol{z}_N\}$ which after being passed through the generator create a batch of fake images $\{\tilde{\boldsymbol{x}}_1, \ldots, \tilde{\boldsymbol{x}}_n\}$ and then we can compute the discriminator loss as a standard binary cross entropy

$$\begin{aligned} L_D = &- \sum_{i=1}^{N} \left[1 \cdot \log D_{\boldsymbol{\phi}}(\boldsymbol{x}_i) + (1 - 1) \cdot \log\{1 - D_{\boldsymbol{\phi}}(\boldsymbol{x}_i)\} \right] \\ &- \sum_{i=1}^{N} \left[0 \cdot \log D_{\boldsymbol{\phi}}(\tilde{\boldsymbol{x}}_i) + (1 - 0) \cdot \log\{1 - D_{\boldsymbol{\phi}}(\tilde{\boldsymbol{x}}_i)\} \right] \\ = &- \sum_{i=1}^{N} \left[\log D_{\boldsymbol{\phi}}(\boldsymbol{x}_i) + \log\{1 - D_{\boldsymbol{\phi}}(\tilde{\boldsymbol{x}}_i)\} \right] \end{aligned}$$

We then update the parameters of the discriminator using backpropagation on this loss. Now the question arises, how do we train th generator (after all this is the part that we want the most). In the classical formulation the generator loss is the negated discriminator loss i.e.

$$L_G = -L_D$$

so that the training effectively realizes the zero-sum game

$$\begin{aligned} \min_{\boldsymbol{\theta}} \max_{\boldsymbol{\phi}} \Big(&\mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})}[\log D_{\boldsymbol{\phi}}(\boldsymbol{x})] \\ &+ \mathbb{E}_{\boldsymbol{z} \sim p(\boldsymbol{z})}[\log(1 - D_{\boldsymbol{\phi}}(G_{\boldsymbol{\theta}}(\boldsymbol{z})))] \Big) \end{aligned}$$

It can then be shown that in the equilibrium the probabilty distribution of the generator is the same as the

data distribution. In practice, however this formulation can be problematic. Note that if the discriminator learns much faster then the generator then after very few epochs the discriminator has no difficulty in distinguishing the true samples from the synthetic ones. Discriminator loss is thus very low and therefore the gradient to the generator will have very small value – the generator thus cannot learn. In practice therefore the generator loss is thus often defined as a discriminator cost (only the part with the generated observation) with changed changed label from 0 to 1

$$L_G = -\mathbb{E}_{\boldsymbol{z}\sim p(\boldsymbol{z})}[\log D_{\boldsymbol{\phi}}(G_{\boldsymbol{\theta}}(\boldsymbol{z}))]\,.$$

To update the generator parameters $\boldsymbol{\theta}$ we do a forward pass through the generator and discriminator, then propagate the gradient through the discriminator for the loss function with changed label (but we do not change any weights in the discriminator here) to obtain $\partial L_G/\partial\tilde{\boldsymbol{x}}$ and then backpropagate this gradient through generator and update its weights.

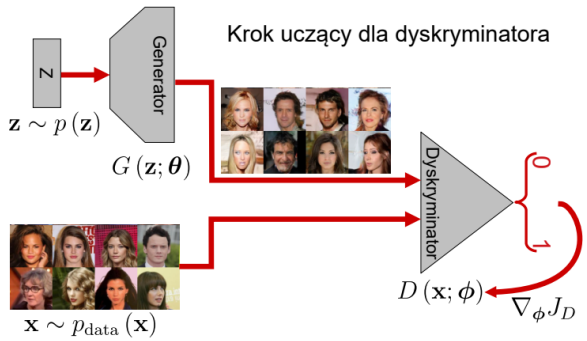


Figure 9: Discriminator training step

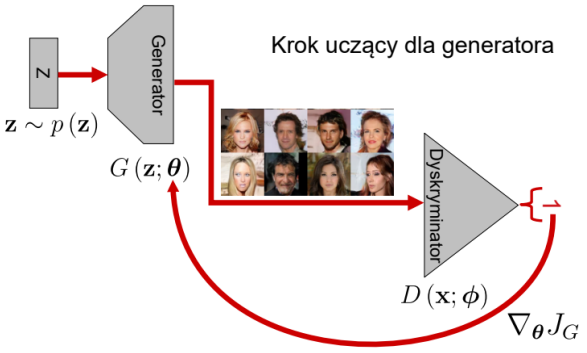


Figure 10: Generator training step

9.7 RNN

Sequence modelling is yet another topic where neural-based approaches outperform earlier, more classical solutions. We can model sequences using a Markov model of order k i.e. given a sequence $(s_0, s_1, \ldots, s_t)$ we model the probability distribution of the next element given k previous ones

$$P(s_t \mid s_{t-1}, \ldots, s_{t-k})\,.$$

This approach can be found e.g. in Transformer architecture which we describe later. Another idea is to use a ***hidden Markov model*** (HMM) in which have have a ***hidden state*** h_t and model the ***transition*** and ***emission*** probabilities

$$P(h_{t+1} \mid h_t)\,, \quad P(s_t \mid h_t)$$

respectively. A simple recurrent neural network (RNN) is a straight-forward implementation of the HMM idea.

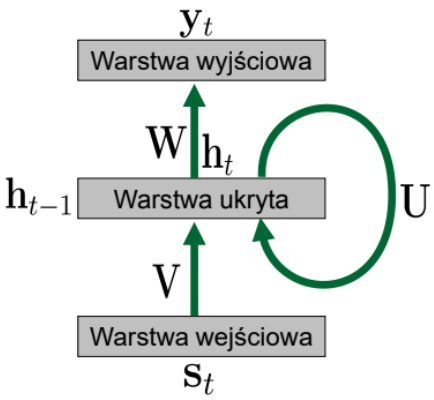


Figure 11: Simple RNN