Neural Networks

August 25, 2022

Neural Networks have the remarkable property that they are able to improve their performance with more data. In many other methods more data certainly helps, but the performance of the algorthms tends to achieve a plateau upon a certain amount of data. Furthermore nowadays more complex MM architectures can be run. So NN benefit from enhanced computer power because we are able to 1) handle more data 2)build more complex architectures, and 3) NN particularily benefit from more data. A summary of historic as well as recent developments and achievement of NN can be found in Ref. [54].

NN are typically used for supervised learning problem. They had quite some success for unstructured data (images, audio). So we can roughly say that there are the following main architectures:

- Standard NN (classification)
- Convolutional NN (image data)
- Recurrent NN (time series, sequence data)

1 Notation

We start with the notation for a classification. Todo: generalize this

- m Number of training examples
- x^i i-th feature, $x^i \in \mathbb{R}^{n_x}$
- with n_x the number of features
- y^i i-th label, $y^i \in \mathcal{Y}$
- $\tilde{y}^i$ i-th predicted label $\tilde{y}^i \in \mathcal{Y}$
- with $\mathcal{Y}$ the individual target classes.

- Feature Matrix $\mathbb{R}^{m \times n_x} \ni M = (x^1, \dots x^m)$
- Label vector $\mathbb{R}^m \ni Y = (y^1, \dots y^m)$

So the columns of the feature Matrix are the individual training features. Note that in other frameworks the feature matrix is defined as the transpose

2 Activation Functions

Non-linear activation functions

- Sigmoid function

$$f(x) = \frac{1}{1 + e^{-x}} \tag{1}$$

- tanh activation function

$$f(x) = \tanh(x) \tag{2}$$

- Rectified linear unit (Relu)

$$f(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{3}$$

- Leaky Relu

$$f(x) = \begin{cases} x & x \geq 0 \\ \alpha x & x < 0 \end{cases} \quad \alpha \geq 0 \tag{4}$$

In practice the sigmoid function is not used, except for output layers in classification tasks. The tanh function outperforms the sigmoid function, since it symmetrizes the sigmoid function and thus has mean of zero. Currently the gold standard is the Relu function, which ameliorates problems with very small derives of the tanh activation function. Since the Relu function is 0 for $z < 0$ and thus gives zero derivatives, the leaky relu function avoids this by setting a very small constant negatve slope for $z < 0$. In practice there does not seem to be much of a difference between relu and leaky relu.

Note that all these activation functions are non-linear.

Non-linear activation functions In principle there are also linear activation $a = Mz + b$ functions thinkable. These include the identity function. However note, that these destroy the advantage of hidden layers because the linear combination of linear functions is linear. The only useful usecase is to use e.g. the identity activation in the output layer in a regression problem.

Output-layer activation functions Typically in a deep neural network the Relu activation function is used. Only the output layer plays a special role as it needs to reflect the problem structure.

- Binary classification
- Multi-class classification

3 Loss and cost functions

Definition 1 (Loss function) *Let there be the true label $y \in \mathcal{Y}$ and the predicted label $\tilde{y} \in \mathcal{Y}$ for feature x. Then a loss function is a map*

$$L : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$$

Note that the loss function depends on the problem at hand and should be defined appropriately.

Definition 2 (Cost function)

$$J = \frac{1}{m} \sum_{i=1}^{m} L(y^i, \tilde{y}^i)$$

Examples

- Quadratic loss function: $L(y, \tilde{y}) = \frac{1}{2}(y - \tilde{y})^2$
- Logistic regression loss function: $L(y, \tilde{y}) = -y \ln \tilde{y} - (1 - y) \ln(1 - \tilde{y})$

Eventually the cost function needs to be minimized. Numerically this is done by using optimizers such as gradient descent. However, these are local optimizers and a **convex** loss function is essential in order to find the global optimum.

Logistic regression The Logistic Regression is given

$$\tilde{y} = \sigma(w^T x + b) \tag{5}$$

$$\sigma(z) = \frac{1}{1 + e^{-x}}, \tag{6}$$

where σ is the logist function, x the training features, and w the model parameters to be learned. $\tilde{y}$ is interpreted as the probability of beeing in class 1 (and $1 - \tilde{y}$ as the probability of beeing in class 0). Summarizing, we get

$$p(y = 1|x) = \tilde{y} \tag{7}$$
$$p(y = 0|x) = 1 - \tilde{y}, \tag{8}$$

which can be expressed more compactly as

$$p(y|x) = \tilde{y}^y (1 - \tilde{y})^{1-y} \tag{9}$$

Taking the negative logartithm of this expression just gives the logistic regression loss function. The negative because just the logarithm would yield an objective function to be maximised and the loss function is a quantity we would like to minimize. The associated cost function may be obtained by assuming iid of training data. The total probability then $p(\{y_n\}|\{x_n\}) = \prod_n p(y_n|x_n)$. Using the same argument as for the loss function gives the logistic cost function.

4 Predicting and training

Once a NN is specified (architecture, definition of units and activation functions), it needs to be trained and eventually used to compute predictions. A NN may be understood as a computational graph. Together with the the concepts of forward and backward propagation training and predicting is done on this computational graph.

Predicting. Forward propagation is used to compute the output of the NN and thus is equivalent to the prediction step.

Training. Backward propagation is used to compute the gradients. This is the basic ingredient for computing the parameters of a NN, i.e. the training step. This is achieved by minimizing the cost function. Once the gradients are computed by back propagation algorithm [52] of the NN corresponds to minimizing the cost function with respect to the parameter space of the NN. As in the specification step of the NN the gradients need to be computed individually.

Gradient descent. Gradient descent is a common method to train the NN. Note that the parameters should be **initialized randomly** in order to achieve symmetry breaking. Furthermore for these parameters should be

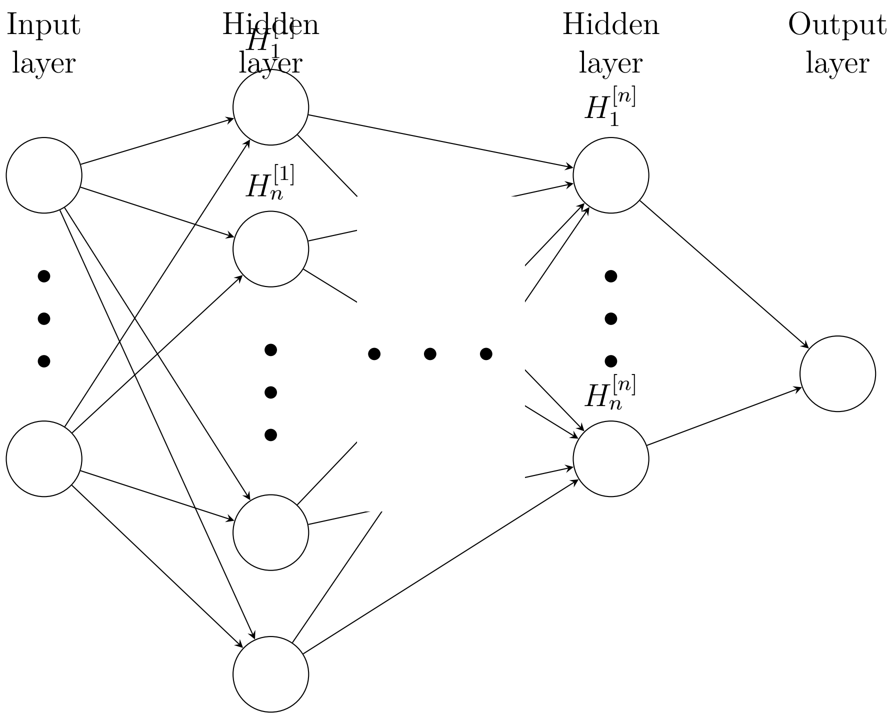

Figure 1: Illustration of a deep neural network with a single output unit.

chosen to be small in order to avoid too small gradients, which eventually results in slow learning. Strictly speaking this only holds true for activation functions such as the sigmoid or tanh, which exhibit very small gradients for large input values (that is why we want to keep the input values small upon initialization).

5 Deep neural Networks

Deep neural networks have several hidden layers (Figure 1). The intuition behind this is, that the first layer finds very local details of the input (simple functions of the input). The ensuing layers then add more complex representations of the overall output (compositional representation). There are several motivations behind such a layered architecture:

- Universal approximation theorem
- Circuit theory: one may compute functions with a small (i.e., not many hidden units per layer) deep neural network that would require exponentially many hidden units in a shallow NN [43]. The infamous exam-

ple is computing XOR on (all) the features (note that XOR itself with AND and OR functions itself already needs a deep neural network).

- An exhausitve list of arguments can be found in [21] Chap 6.4

Definition and Conventions

- $W^{[l]}, b^{[l]}$: Weights of layer l
- $g^{[l]}$: Activation function of layer l
- $n^{[l]}$: Number of units in layer l

Note that the structure depicted in the graphical representation refers to a **single** training example. In implementations we usually work with vectorized formulars.

Parameters vs Hyperparameters. Parameters are just the weights $W^{[l]}, b^{[l]}$. These are the quantity to be trained. All other quantities (Number of layers l, number of units in layer l, activation function of layer l, learning rate, ...) are called Hyperparameters. These quantities must be set empirically.

Forward and Backpropagation

Vectorized version

Initialization

6 Optimizing Neural Networks

The predictive performance of a neureal network depends on the hyperparameters and its architecture. First the necessary concepts for judging the predictive power are presented. Based on that, concepts to improve the prediction accuracy are summarized.

6.1 Train, development and test set.

It's common practice to split the Data in 3 Sets: Training set, Hold-out crossvalidation (or development) set, and Test set. The Training set is used to train the model. The development set is used to test specific models and based on that develop better models. Once the choice for the best model is made the test set is used to get an *unbiased* estimate of the model

model	M1	M2	M3	M4
Training set error [%]	1	15	15	0.5
Development set error [%]	11	16	30	1
Problem diagnosis	high variance	high bias	high bias high variance	low bias low variance

Table 1: Illustration of bias and variance for 4 models M1 ... M4. .

performance. If no unbiased estimate on the model is needed sometimes only train and dev sets are used. How to split the data depends on the amount of data:

- **Conventionally** the splitting was made in the order of $70\,\% - 30\,\%$ (omitting the test set) or $60\,\% - 20\,\% - 20\,\%$. This applies $\approx 10^5$ datapoints on order to make sure to have enough test examples.
- **Big data** usecases with $> 10^6$ datapoints typically have splittings in the order $98\,\% - 1\,\% - 1\,\%$ because there are still enough samples for the test statistics.

Since neural networks need a lot of data some applications use mismatched data distributions between the training set and development and test distributions. E.g., the training data come from webcrawling and the develpment and test sets come from the specific application at hand. However, in this case it is necessary that the development and the test set come from the same distribution.

6.2 Bias and Variance.

The **bias** of a model describes the complexity of the model. A model with high bias has a (too) simple decision boundary (e.g., linear regression). The bias may be assessed by the model performance on the training set. The **variance** describes the generalization property of model. A model with high variance tends to perform worse for new data (development set). High variance is also called **over-fitting**.

An example is given in Table 6.2. The worst case is model M3 (high bias and high variance). The ideal case is model M4 (low bias and low variance). Note that the model assessment in Table 6.2 is based on the assumption that the true labeling contained in train and development set are known without error (the Bayesian error is $0\,\%$). A counterexample to this assumption would be blurry images, where even humans make errors in labeling. Bias and variance are important quantities upon improving the model performance:

Strategies to reduce bias:

- Bigger network (number of layers / units / hyperparameter search)
- Train longer
- Use other optimization algorithm
- Different NN architecture

Strategies to reduce variance:

- More training data
- Regularization
- Different NN architecture

By iteration through bias reduction and variance reduction one can systematically improve the model performance. Note that in the case of neural networks reducing the bias and reducing the variance can be done relatively independently. This is a further advantage of neural networks. In the 'old days' one needed to find a bias variance trade off as these two quantities could not be improved independently. Probably the main reason is that nowadays there are much more data at hand. Also note that regularization also affects the bias.

The purpose of the test set is to judge the final performance of the algorithm. Ideally development set and test set error are very similar. However, if the test set error is much larger than the training set error then the development set should be increased.

6.3 Regularization

Regularization aims to reduce the variance, i.e., reduce generalization errors. There are several regularization strategies for neural networks [57, 21]:

Parameter norm penalties. This procedure is also called weight decay and follows the same ideas as standard regularization. Then this regularization scheme adds to the cost function $J(\theta)$ a penalty term

$$\tilde{J}(\theta) = J(\theta) + \lambda\Omega(\theta), \tag{10}$$

where λ is an additional hyperparameter and θ are model parameters, which are determined by minimizing the cost function. $\tilde{J}$ is then used instead of J as objective function in order to determine the modelparmeters within an optimization algorithm. In the specific case of a deep neural network a common regularization scheme is

$$\tilde{J}(\{w^{[l]}, b^{[l]}\}) = J(\{w^{[l]}, b^{[l]}\}) + \frac{\lambda}{2m} \sum_{i=1}^{L} \|w^{[i]}\|_F^2, \tag{11}$$

where $w^{[l]}, b^{[l]}$ are the weights of layer l, $\|w\|_F$ is the Frobenius norm ($\|w\|_F^2 = \sum_{i,j} w_{ij}^2$,) and m is the number of training examples. This procedure is also called weight decay. Note that in this scheme only the $w^{[l]}$ are regularized - probably because there are much more parameters than in the $b^{[l]}$ and the assumption that this leads to a sufficient regularization.

Remark: The disadvantage of this approach is the scaling properties are lost. These can be recovered by regularizing not all layers [57] ?!?

Dropout. In dropout regularization nodes and its edges are removed at random for each training example. This introduces a new hyperparamter the "survival" probability of a node. This probability can be layer dependent. In particular layers with many units that are suspicious to overfitting may be assigned lower survival probabilities than layers with only a few units. In principle dropout can be also used in the input layer. Dropout has been successful in particular in computer vision. Dropout is implemented by masking the activation with a boolean random draw of of a Bernoulli experiment with given survival probability. In practice "inverse dropout" is used, which additionally divides the activations by the the survival probability.

Remark: With dropout the cost function J is no longer well defined. So the convergence behaviour of J as monotonically decreasing function during iteration is lost.

Data augmentation. Reduces the variance by artificially creating more data via e.g., random rotations, distortions, flips, crops of images. This approach does *not* introduce new hyper parameters.

Early stopping. This approach tries to reduce over fitting by "tweaking" the optimization step. Instead of ensuring, that the cost function w.r.t to the training data is minimized, additionally the cost function or error w.r.t the development set is monitored (optimization is still performed with the training set). One then stops at the optimization step where the cost function of the development data is minimized. This approach roots in the observation, that typically the development set error decreases with increasing optimization steps but then starts to increase again. This approach does *not* introduce new hyperparameters.

Further regularization techniques[57]:

- Parameter tying and sharing
- Bagging/ensemble methods
- Tangent methods
- Adversarial training

7 Optimization algorithms

Given an objective function $J(\theta)$ an optimization algorithm tries to find a local minimizer $\theta^* = \operatorname{argmin}_\theta J(\theta)$ [44]. To this end an optimization algorithm generates a sequence of iterates $\{\theta_k\}_{k=0}^{\infty}$ that (ideally) terminate at the minimizer. Different algorithms differ the process of generating these iterates. θ_0 is often called initialization. For brevity we will use the notation $\nabla J_k \equiv \nabla_\theta J(\theta)|_{\theta_k}$.

7.1 Algorithms used for neural networks

Gradient descent. The update in gradient descent is given by

$$\theta_{k+1} = \theta_k - \alpha \nabla J_k, \tag{12}$$

where α is called learning rate (hyper parameter).

Exponentially moving average (EMA). This is not an optimization algorithm itself but a very common ingredient. Exponentially moving average is a comutationally efficient method to approximate moving averages. Gradient descent may produce oscillating gradients, which induce a reduced learning rate in order to achieve convergence. Many alogrithmic improvements of gradient descent are based on the first and second moment of the gradient.

For example, in order to soften oscillations one could use the moving average over the last m computed derivatives, $\bar{\nabla} J_k := 1/m \sum_{i=k-m}^{k} \nabla J_k$ instead of the current gradient in Equation 12. However, this would require storing $m-1$ additional gradients. Instead, the moments are approximated by the exponentially moving average.

Assume a series of data $\{q_k\}$. Then the exponentially moving average is defined iteratively as

$$\langle q\rangle_0^{\text{EMA}} = 0 \tag{13}$$

$$\langle q\rangle_k^{\text{EMA}} = \frac{1}{1-\beta^k}\left(\beta\langle q\rangle_{k-1}^{\text{EMA}} + (1-\beta)q_k\right) \tag{14}$$

There is also refined version of exponentially moving average that includes bias correction,

$$\langle q\rangle_0^{\text{cEMA}} = 0 \tag{15}$$

$$\langle q\rangle_k^{\text{cEMA}} = \frac{1}{1-\beta^k}\left(\beta\langle q\rangle_{k-1}^{\text{EMA}} + (1-\beta)q_k\right) \tag{16}$$

The term $1/\left(1-\beta^k\right)$ is called *bias correction.* It corrects detoriations for the first few data points.

Gradient descent with momentum. Gradient descent with momentum replaces the gradient by its first moment (expectation value) obtained by the exponentially weighted average 14. The update procedure of 12 gets then modified to:

$$\langle\nabla J\rangle_0^{\text{EMA}} = 0 \text{ (initialization)} \tag{17}$$

$$\langle\nabla J\rangle_k^{\text{EMA}} = \beta\langle\nabla J\rangle_{k-1}^{\text{EMA}} + (1-\beta)\nabla J_k \tag{18}$$

$$\theta_{k+1} = \theta_k - \alpha\langle\nabla J\rangle_k^{\text{EMA}}. \tag{19}$$

The term $\langle\nabla J\rangle_{k-1}^{\text{EMA}}$ is called momentum, which lends the algorithm it's name. β describes how many past gradients are used and is in principle an other hyper parameter. In practice it is often set to $\beta = 0.8$. Gradient descent with momentum converge typically faster than gradient descent.

RMSprop [58]. Gradient descent with momentum damps gradient oscillations by using the approximate averages over the previous gradients. RMSprop ("Root mean square propagation") tries to damp oscillation in a complementary fashion. It weights the gradient by the inverse of the root mean square of the previous gradients – again using as exponentially moving averages as approximations to the average.

$$\langle\nabla J \odot \nabla J\rangle_0^{\text{EMA}} = 0 \text{ (initialization)} \tag{20}$$

$$\langle\nabla J \odot \nabla J\rangle_k^{\text{EMA}} = \beta\langle\nabla J \odot \nabla J\rangle_{k-1}^{\text{EMA}} + (1-\beta)\nabla J_k \odot \nabla J_k, \tag{21}$$

$$\theta_{k+1} = \theta_k - \alpha\nabla J_k \oslash \left(\sqrt{\langle\nabla J \odot \nabla J\rangle_k^{\text{EMA}}} + \epsilon\right), \tag{22}$$

where the square rood is taken elementwise $(\sqrt{A})_{ij} = \sqrt{A_{ij}}$, and the elementwise (Hadamard) product $(A \odot B)_{ij} = A_{ij}B_{ij}$ and division $(A \oslash B)_{ij} = A_{ij}/B_{ij}$ is used. The matrix ϵ prevents division by zero and it's components are typically set to 10^{-8}. The expression $\langle \nabla J \odot \nabla J \rangle_k^{\text{EMA}}$ is the element wise second moment of the gradient within the EMA approximation. Thus, the RMSprop update 22 scales the gradients by the element wise RMS (root mean square) using EMA. As with gradient descent with momentum, β introduces a new hyper paramter but it's typically set to $\beta = 0.8$.

Adam [31]. Adaptive moment estimation (Adam) combines gradient descent with momentum with RMSprop. It takes the averaged gradients as in 19 and scales them according to RMSprop 22. However, instead of using EMA the bias corrected EMA is used. Adam then updates the optimization steps according to

- Initialize θ and first and second moments of gradients

$$\begin{aligned} \langle \nabla J \rangle_0^{\text{cEMA}} &= 0 \\ \langle \nabla J \odot \nabla J \rangle_0^{\text{cEMA}} &= 0 \end{aligned}$$

- Update Update moments

$$\begin{aligned} \langle \nabla J \rangle_k^{\text{cEMA}} &= \frac{1}{1-\beta_1^k} \left(\beta_1 \langle \nabla J \rangle_{k-1}^{\text{EMA}} + (1-\beta_1) \nabla J_k \right) \\ \langle \nabla J \odot \nabla J \rangle_k^{\text{cEMA}} &= \frac{1}{1-\beta_2^k} \left(\beta_2 \langle \nabla J \odot \nabla J \rangle_{k-1}^{\text{EMA}} + (1-\beta_2) \nabla J_k \odot \nabla J_k \right) \end{aligned}$$

- Update optimization step

$$\theta_{k+1} = \theta_k - \alpha \langle \nabla J \rangle_k^{\text{cEMA}} \oslash \left(\sqrt{\langle \nabla J \odot \nabla J \rangle_k^{\text{cEMA}}} + \epsilon \right), \tag{23}$$

Apart from the learning rate α there are in principle three additional hyper parameters: $\beta_1, \beta_2, \epsilon$. In practice these are not tuned but set to the values $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$ [31].

7.2 Problems of optimizers for neural networks

We briefly judge the optimization strategies for neural networks

- Neural networks pose a **non-convex optimization problem**. However, the algorithms above are modifications of established algorithms for convex optimization problems. So it would be desirable to have non-convex optimization algorithms.

- In principle neural networks may have **many local minima** and the optimization algorithm finds one of them (of course one would like to find the global minimum). This problem has been debated intensively in the community. However, there is an interesting counterargument: in order to have a local minimum θ^* *all* components of the second derivative at the minimum must be greater than zero $\partial_\theta^2 J|_{\theta^*} > 0$. Howver, for a very large NN where θ is high dimensional (several thousand components) this is very unlikely. So extremal points are most probably saddle points, which don't pose a problem to the the above optimizers. This is a consequence, that low dimensional intuition does not necessarily carry over to high dimensional neural networks.
- **Flat regions** of the cost functions may lead to very slow learning. Algorithms like Adam tend to reduce this problem, but it can still be severe.
- All algorithms of section 7 only use the first derivative of the cost function (and it's first and / or second moments). However in the literature the most powerful standard algorithms also use the second derivative (**Hessian**).
- Mind the paradigm: "In practice, it is more important to choose a model family that is easy to optimize than to use a powerful optimization algorithm" [21].

In summary, there seems to be quite some room for improving optimization algorithms [62].

7.3 Speed up the learning process

Training a NN is optimizing the cost function with respect ot the parameters. In practice this can be a tricky task for several reasons:

- Algorithm specific problems: learning rate
- NN architecture specific problems: vanishing or exploding gradients for (very) deep neural networks.
- Large training data (slow computations).

For these reasons optimization in neural networks is yet not well understood. We summerize some heuristics often used in the community.

Data normalization. When data are in the same numerical range a larger learning rate can be chosen.

Weight initialization. The problem of vanishing or exploding gradients can be partially reduced by carefully initializing the weights. The weights are initialized by drawing from samples from the unit normal distribution, $\mathcal{N}(1,0)$ followed by a calibration transformation. The latter depends on the activation function but it's main rational is to account for different number of nodes per layer via a variance argument. Popular initialization schemes are:

- Reilu activation: $w_{ij}^{[l]} = \mathcal{N}(1,0)\sqrt{\frac{2}{n^{[l-1]}}}$
- Tanh activation: $w_{ij}^{[l]} = \mathcal{N}(1,0)\sqrt{\frac{1}{n^{[l-1]}}}$ (Xavier initialization),

where $n^{[l-1]}$ is the number of nodes in layer $n-1$. The aim of these initialization schemes is to reduce the tendency of gradients to explode or vanish for very deep neural networks.

Mini-batch gradient descent. For moderately many training data a vectorized implementation of gradient descent may have an acceptable speed. In this version all training examples are passed into a single optimization step, which is also called batch (gradient descent). However, for many training examples (larger than $\sim 10^3$) this may become slow. Mini-batch gradient descent splits the data in mini batches of size. Each optimization step then only sees one mini batch (including the labels and cost function).

An **epoch** is an entire pass through the training set. While batch gradient descent allows only one optimization step per epoch, mini-batch gradient descent allows for batch size over mini-batch size many optimization steps in one epoch.

The extreme case of mini-batch size one is called **stochastic gradient descent**. The mini-batch size is another hyper parameter – in practice typical values are powers of two in the range 64, 128, ... 512.

Learning rate decay The algorithms in section 7 use a fixed learning rage α. Intuitively, near the optimum the learning rate should be smaller and larger at areas far away from the optimum. There are several heuristics that adapt the learning rate in terms of epochs n_e

- $\alpha = \frac{\alpha_0}{1+dn_e}$, with hyper parameters α_0 (initial learning rate) and d (decay rate).
- $\alpha = \alpha_0\, 0.95^{n_e}$, with hyper parameters α_0 (initial learning rate).
- $\alpha = \frac{\alpha_0\, d}{\sqrt{n_e}}$, with hyper parameters with hyper parameters α_0 (initial learning rate) and d.

8 Convolutional neural networks

Convolutional neural networks come from computer vision (image classification, object detection, neural style transfer, ...). If a forward deep neural network was used for such tasks it would yield a very high dimensional input space and thus parameter space. As a consequence, there would not be enough data to prevent overfitting[1].

Continuous case. Let $f, g : \mathbb{R}^n \to \mathbb{C}$, then

$$(f * g)(t) = \int f(t')g(t-t')\,dt' \quad (\textbf{convolution}) \tag{24}$$

$$(f \star g)(t) = \int \overline{f(t')}g(t+t')\,dt' \quad (\textbf{cross correlation}), \tag{25}$$

While these quantities look innocently similar the interpretation is different. Cross correlation is interpreted as the similarity of the functions f and g (shifted by t). As special case, the *auto correlation* at lag τ is just the cross correlation with itself $R_f(\tau) = (f \star f)(\tau)$. Convolution is interpreted as f being "smeared out" by a g: $(f * g)(t)$ is the weighted "mean" of $f(t')$ by a function $g(t-t')$. The convolution satisfies the following properties:

- Commutative $f * g = g * f$
- Associative
- Distributive
- Identity $f * \delta = f$. And therefore the inverse convolution (if exisits) is defined as
- Translational equivariance. $\tau_x(f * g) = (\tau_x f) * g = f * (\tau_x g)$, with the translation operator $(\tau_x(f))(y) = f(y-x)$ (convolution commutes with translation).

Introducing the mirror operater $Pf(t) = f(-t)$ these convolution and cross correlation are related by [2]

$$f \star g = \overline{Pf} * g. \tag{26}$$

[1]For example a 1000×1000 pixel input image with 1000 units in the first hidden layer would lead to a $3 \cdot 10^6 \times 1000$ dimensional weight matrix just for the first hidden layer (the factor of 3 comes from the 3 RGB channels).

[2]Because $\overline{Pf} * g = \int \overline{f(-t')}g(t-t')\,dt' = \int \overline{f(t')}g(t+t')\,dt' = f \star g$

So, if f is hermitian (symmetric for real valued functions), $\overline{f}(t) = f(-t)$, then $f \star g = f * g$. And if both, f and g are hermitian then the cross correlation is commutative, $f \star g = g \star f$. Convolution and Cross correlation is translational equivariant.

In practice, these quantities are often calculated using the convolution theorem. Consider the Fourier transform of a function $f \in L^1(\mathbb{R}^n)$

$$(\mathcal{F}f)(y) = \frac{1}{\sqrt{2\pi}^n} \int f(x) e^{-iy\cdot x}\, dx \tag{27}$$

$$(\mathcal{F}^{-1}\hat{f})(x) = \frac{1}{\sqrt{2\pi}^n} \int \hat{f}(y) e^{iy\cdot x}\, dy, \tag{28}$$

where $\mathcal{F}^{-1}$ is the inverse Fourier transform and $y \cdot x$ is the standard scalar product $(x, y \in \mathbb{R}^n)$. The *convolution theorem* states that,

$$f * g = \mathcal{F}^{-1}(\mathcal{F}f \cdot \mathcal{F}g), \tag{29}$$

$$f \star g = \mathcal{F}^{-1}(\overline{\mathcal{F}f} \cdot \mathcal{F}g). \tag{30}$$

with the pointwise product $(f_1 \cdot f_2)(x) = f_1(x)\, f_2(x)$. The expression follows from (26) and $\mathcal{F}\{\overline{Pf}\} = \overline{\mathcal{F}f}$. Thus, convolution and cross correlation may be computed from (inverse) Fourier transforms.

Discrete convolution. Let $F : \mathbb{Z}^d \to \mathbb{R}$ be a discrete function. Let $\Omega_r = [-r, r]^d \cap \mathbb{Z}^d$ and $\omega : \Omega_r \to \mathbb{R}$ be a $(2r+1)^d$ dimensional filter kernel. Then the discrete convolution operator $*$ is defined as [63, 14]

$$(F * k)(p) = \sum_{s+t=p} F(s)\, \omega(t), \tag{31}$$

where in the context of neural networks F is called input feature map and the convolution is called output feature map. The filter kernel size $(2r+1)^d$ determines the size a given filter locally looks at the input feature map. For neural networks this is also called *receptive field.*

For example, in image processing there are well known edge detection filter kernels,

$$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}, \quad \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}, \quad \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}. \tag{32}$$

Deconvolution and transposed convolution [64]

Dilated and causal convolutions [63, 9]

Convolutions in neural networks. There are several tweaks in the application of convolutions to neural networks

- Filters are learned by back propagation.
- Activation function and bias.
- Discrete correlation instead of convolution.
- Several Filters in parallel (filterbanks)
- Dimensions
- Padding and strides
- Pooling

- https://cs231n.github.io/convolutional-networks/
- https://arxiv.org/pdf/1511.07122.pdf - http://torch.ch/blog/2016/02/04/resnets.html - https://cs231n.github.io/optimization-2/

9 Sequence Models

Sequence models are largely motivated from natural language processing but also find application in time series modeling [16, 37, 50, 60, 38].

Recurrent neural nets suffer from vanishing / exploding gradient problem. Furthermore they are not easily capable of considering events in the long past. Eventually they suffer from big memory requirement and need long training. One reason is that they use the last hidden state in the encoder step. Gated recurrent units (GRU) ans long-short-term-memory (LSTM) try to incorporate more information from the past by a more sophisticated activation unit. However, NLP tasks still suffer from the last hidden state approach in the encoder step. One could heuristically incorporate more hidden encoder states and pass them into the decoder.

A more sound approach is the attention mechanism[3, 39], which weights the hidden states and passes them into the decoder. Incorporating the attention mechanism also in the encoder step is called self-attention. This leads to the concept of transformers [59, 2] and state of the art NLP models like BERT [12] or GPT2 [48, 1].

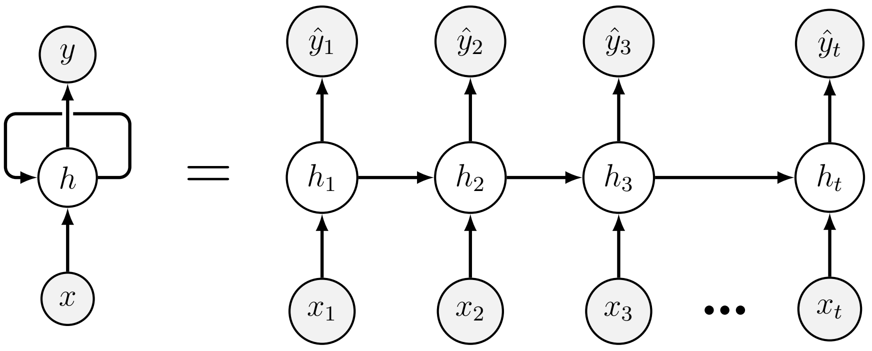

Figure 2: Illustration of a RNN. The right hand side shows the time unfolded structure, which explicitly depends on time.

9.1 Recurrent Neural Networks (RNN)

In contrast to feed forward neural networks RNNs are *in principle* capable of looking at the whole time sequence. The basic idea is that, RNN cells contain an internal memory state $h_t \in \mathbb{R}^{d_h}$ which acts as a compact summary of past information. At each time step the memory state is updated with new observations [53, 37]. Furthermore, RNNs are related to Bayesian filtering in discrete state Markov models [38, 5].

Assume that we have temporarily ordered features $\{x_t\}$ and labels $\{y_t\}$, with $x_t \in \mathbb{R}^{d_x}$ and $y_t \in \mathbb{R}^{d_y}$. There are many formulations of RNNs. One of the simplest RNNs, the Eilman RNN, takes as internal memory state h_t the expression [15, 21]

$$h_t = \gamma_h \left(W_{h_1} h_{t-1} + W_{h_2} x_t + b_h\right) \tag{33}$$

$$y_t = \gamma_y \left(W_y h_t + b_y\right), \tag{34}$$

where $W_{h_1} \in \mathbb{R}^{d_h \times d_h}, W_{h_2} \in \mathbb{R}^{d_h \times d_x}, b_h \in \mathbb{R}^{d_h}$ and $W_y \in \mathbb{R}^{d_y \times d_h}, b_y \in \mathbb{R}^{d_y}$. Training corresponds to learning these weights. Together with their usual demand for initialization the initial state is taken as $h_0 = 0$. γ_h and γ_y are activation functions (a common choics is $\gamma_h(\cdot) = \text{thanh}(\cdot)$ and $\gamma_y(\cdot) = \text{softmax}(\cdot)$ for classification). The **advantages** of RNN are

- They can process input sequences of any length. As a consequence RNNs do not require the explicit specification of a lookback window.
- The model size is invariant with respect to input size.
- The computation for step t in principle uses information from many steps back.

The **disadvantage** are

- Slow computation.
- Very limited temporal memory in practice [7, 35], which is implied by the exploding and vanishing gradient problem. [21]. One can show that the matrix norm of the partial derivatives necessary for back propagation is of the form $\|\partial h_t / \partial h_k\| \leq \beta_1 \beta_2^{t-k}$, which can easily explode or vanish for large time differences $t-k$.

There are several RNN **design patterns**,

- RNN that produce at time step an output and have recurrent connections between hidden units. (E.g., decoder part in sequene to sequence models)
- RNN that have recurrent connections between hidden units and only produce an output at the final time step. (E.g., encoder part in sequene to sequence models)
- *Bi-directional* RNN that use future information for predictions. This can be usesful in NLP tasks. The construction of bi-directional essentially doubles the number of weights and ist straight forward. It constructs forward and backward memory cells and then concatinates them. For the Eilmann RNN e.g., this yields

$$\overrightarrow{h}_t = \gamma_h \left(\overrightarrow{W}_{h_1} \overrightarrow{h}_{t-1} + \overrightarrow{W}_{h_2} x_t + \overrightarrow{b}_h \right) \tag{35}$$

$$\overleftarrow{h}_t = \gamma_h \left(\overleftarrow{W}_{h_1} \overleftarrow{h}_{t-1} + \overleftarrow{W}_{h_2} x_t + \overleftarrow{b}_h \right) \tag{36}$$

$$h_t = \overrightarrow{h}_t \oplus \overleftarrow{h}_t \tag{37}$$

$$y_t = \gamma_y \left(W_y h_t + b_y \right), \tag{38}$$

  where $\oplus$ is the direct sum (i.e., concatinate vectors).

Long-Short-Term Memory (LSTM) LSTMs [25, 20] try to overcome the limitations in learning long-range dependencies in the data by improving ”gradient flow” within the network [55]. This is achieved by a more complex memory state. LSTMs introduce an additional cell state c_t, which stores long-term information with additional temporal modulations conducted by so called gates,

$$\textit{Input gate:} \quad i_t = \sigma \left(W_{i_1} h_{t-1} + W_{i_2} x_t + b_i \right) \tag{39}$$

$$\textit{Output gate:} \quad o_t = \sigma \left(W_{o_1} h_{t-1} + W_{o_2} x_t + b_o \right) \tag{40}$$

$$\textit{Forget gate:} \quad f_t = \sigma \left(W_{f_1} h_{t-1} + W_{f_2} x_t + b_f \right), \tag{41}$$

where $W_{i_1}, W_{o_1}, W_{f_1} \in \mathbb{R}^{d_h \times d_h}$, $W_{i_2}, W_{o_2}, W_{f_2} \in \mathbb{R}^{d_h \times d_x}$, and $b_i, b_o, b_f \in \mathbb{R}^{d_h}$. h_t is called hidden state and $\sigma(\cdot)$ is the sigmoid activation function. The gates modify hidden state h_t and cell state c_t at time t according to

$$\textit{Cell state} \quad c_t = f_t \odot c_{t-1} + i_t \odot (W_{c_1} h_{t-1} + W_{c_2} x_t + b_c) \tag{42}$$
$$\textit{hidden state:} \quad h_t = o_t \odot \tanh(c_t), \tag{43}$$

where $W_{c_1} \in \mathbb{R}^{d_h \times d_h}, W_{c_2} \mathbb{R}^{d_h \times d_x}, b_c \in \mathbb{R}^{d_h}$, and $\odot$ is the component wise (Hademard) product. The initial states are set to $h_0 = c_0 = 0$. As with simple RNNs there are many kinds of LSTMs [22], in particular peephole LSTMs [19, 18] and peephole convolutional LSTMs [56].

9.2 Attention mechanism

The attention mechanism [4, 10, 40]. Assume a sequence $\{x_1, \ldots, x_T\}$ with $x_i \in \mathbb{R}^{d_x}$. Then the attention mechanism is of the form

$$x'_t = \sum_i f(x_i)\, \lambda_{it} \tag{44}$$
$$\lambda_{it} = \lambda(x_i, x_t) = \operatorname{softmax} g(x_i, x_t), \tag{45}$$

where f and g are potentially trainable functions. Different choices for them yield different attention mechanisms. The transformed $x'_t \in \mathbb{R}^{d_h}$ is sometimes called context vector or attention. It is common to parametrize the weights λ_{it} with the softmax function (45), [3] and call g similarity function.

Hard vs. soft attention The softmax parametrization enables a probabilistic interpretation of the weights $\lambda_{it} = p(i|t)$ with respect to i for a given t. The most common approach to obtain a attention function is given in (44) as a weighted linear combination over vectors $f(x_i)$. This approach is called *soft attention* and can be directly incorporated with back propagation.

However, one could also sample an ℓ of the λ-distribution $p(i|t)$ and then use $x'_t = f(x_\ell)$ as attention function instead of averaging (44). This approach is called *hard attention* and thus pays attention only to a specific part of the sequence (for example think of x_ℓ as a glimpse of a figure). Hard attention is harder to train as back propagation is not feasible and one has to resort to methods from reinforcement learning.

[3] $\operatorname{softmax} g(x_i, x_t) = \frac{e^{g(x_i, x_t)}}{\sum_i e^{g(x_i, x_t)}}$

Self Attention Self attention computes a transformed x'_t with respect to every element in the sequence $\{x_1, \dots, x_T\}$ - in particular there is also a contribution to (44) from x_t it*self.* Specifically [59] parametrizes the functions f and g with three matrices W_q, W_k, W_v, with $W_v \in \mathbb{R}^{d_h \times d_x}$ and $W_q, W_k \in \mathbb{R}^{d_s \times d_x}$. Now, define

$$q_t := W_q x_t, \quad k_i := W_k x_i, \quad v_i := W_v x_i, \tag{46}$$

and identify the functions f and g in (44) - (45) with

$$f(x_i) := v_i \tag{47}$$

$$g(x_i, x_t) := \frac{k_i^T q_t}{\sqrt{d_s}} = \frac{q_t^T k_i}{\sqrt{d_s}}, \tag{48}$$

gives as attention $x'_t = \sum_i v_i \lambda_{it}$ with $\lambda_{it} = \mathrm{softmax}(q_t^T k_i / \sqrt{d_s})$. The vectors $v_i \in \mathbb{R}^{d_h}$, $q_i \in \mathbb{R}^{d_s}$ and $k_i \in \mathbb{R}^{d_s}$ are called value, query and key in analogy with database retrieval systems.

So for a fixed point in time t the weights are essentially obtained by a scaled dot product of the query vector q_t at t with the key vectors *at all times* $k_i, i \in \{1, \dots, T\}$ (including t itself). The attention is then the weighted sum over the value vectors v_i.

The formulation can be readily extended to all points in time. Define the matrices

$$X := [x_1, \dots, x_T] \in \mathbb{R}^{d_x \times T} \tag{49}$$

$$Q := W_q X \in \mathbb{R}^{d_s \times T}, \tag{50}$$

$$K := W_k X \in \mathbb{R}^{d_s \times T}, \tag{51}$$

$$V := W_v X \in \mathbb{R}^{d_h \times T}, \tag{52}$$

which gives

$$X' =: \mathcal{A}(Q, K, V) = V\lambda, \quad \lambda = \mathrm{softmax}\left(\frac{Q^T K}{\sqrt{d_s}}\right), \tag{53}$$

where the softmax function is component wise and the normalization w.r.t to rows. This formulation computes the attention for all times simultaneously ($X' := [x'_1, \dots, x'_T] \in \mathbb{R}^{d_h \times T}$). Note that in practice rather a dual (i.e., transposed) formulation is used as it is more compatible with sequence conventions (time as leading index). Either way, **training corresponds to computing the matrices** W_q, W_k, W_v via back propagation.

Multi-head attention The idea is to perform self attention in parallel on different subspaces for increased performance. Each individual attention computation within this approach is called *head.*

Specifically for each h (*head*) in $1, \ldots H$ introduce "projection" - Matrices $P_{h,q}, P_{h,k}, P_{h,v}$, where $P_{h,q}, P_{h,k} \in \mathbb{R}^{d_{s'} \times d_s}$ and $P_{h,v} \in \mathbb{R}^{d_{h'} \times d_h}$. Now define

$$Q_h = P_{h,q} W_q X = W_{h,q} X \in \mathbb{R}^{d_{s'} \times T}, \tag{54}$$

$$K_h = P_{h,k} W_k X = W_{h,k} X \in \mathbb{R}^{d_{s'} \times T}, \tag{55}$$

$$V_h = P_{h,v} W_v X = W_{h,v} X \in \mathbb{R}^{d_{h'} \times T}, \tag{56}$$

and compute the matrix multiplications, which yield effective weight matrices $W_{h,q}, W_{h,k} \in \mathbb{R}^{d_{s'} \times d_x}$ and $W_{h,q} \in \mathbb{R}^{d_{h'} \times d_x}$. The multi-head attention is then the direct sum[4] of the individual attentions together with an additional matrix multiplication by a matrix W,

$$X' = W \oplus_{h=1}^{H} \mathcal{A}(Q_h, K_h, V_h), \tag{57}$$

where $X' \in \mathbb{R}^{Hd_{h'} \times T}$ and $W \in \mathbb{R}^{Hd_{h'} \times Hd_{h'}}$. Training corresponds to learning the $3H$ matrices $\{W_{h,q}, W_{h,k}, W_{h,v}\}$ and the matrix W via back propagation.

Dual formulation While the the above formulations seem natural if the weight matrices are thought of linear operators in the it is common to use the dual formulation (because conventionally the time index is the first index). This can be achieved by transposing the design matrix X all subsequent expressions and then rename the Matrices. So consider matrices of the form $W_v \in \mathbb{R}^{d_x \times d_h}$ and $W_q, W_k \in \mathbb{R}^{d_x \times d_s}$. Then, in the matrix formulation the attention is given by

$$X := [x_1, \ldots, x_T]^T \in \mathbb{R}^{T \times d_x} \tag{58}$$

$$Q := X W_q \in \mathbb{R}^{T \times d_s}, \tag{59}$$

$$K := X W_k \in \mathbb{R}^{T \times d_s}, \tag{60}$$

$$V := X W_v \in \mathbb{R}^{T \times d_h}, \tag{61}$$

$$X' =: \mathcal{A}(Q, K, V) = \operatorname{softmax}\left(\frac{QK^T}{\sqrt{d_s}}\right) V, \tag{62}$$

where $\mathcal{A}(Q, K, V) \in \mathbb{R}^{T \times d_h}$ and the element-wise softmax is now normalized with respect to columns. For the multi-head attention choose matrices

[4]The direct sum is with respect to columns $\mathcal{A}(Q_i, K_i, V_i) \oplus \mathcal{A}(Q_j, K_j, V_j) \in \mathbb{R}^{2d_{h'} \times T}$

$P_{h,q}, P_{h,k} \in \mathbb{R}^{d_s \times d_{s'}}$ and $P_{h,v} \in \mathbb{R}^{d_h \times d_{h'}}$ and

$$\begin{aligned} Q_h &= XW_q P_{h,q} = XW_{h,q} \in \mathbb{R}^{T \times d_{s'}}, &(63)\\ K_h &= XW_k P_{h,k} = XW_{h,k} \in \mathbb{R}^{T \times d_{s'}}, &(64)\\ V_h &= XW_v P_{h,v} = XW_{h,v} \in \mathbb{R}^{T \times d_{h'}}, &(65)\\ X' &= \oplus_{h=1}^{H} \mathcal{A}(Q_h, K_h, V_h)W, &(66) \end{aligned}$$

where the direct sum is now with respect to columns.

Causal attention So far there was no notion auf causality. For a given point in time t the attention ranges over the whole time period and thus also w.r.t to future events. In order to introduce temporal causality contributions from future events should vanish in (44): $\lambda_{it} = 0$ for $i > t$. In the parametrization of self attention this can be achieved by masking

$$\mathcal{A}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_s}} + M\right)V, \tag{67}$$

where M is $-\infty$ in the upper triangular and 0 elsewhere.

9.3 Sequence to sequence models and transformers

The problem with RNNs are i) long range dependencies ii) vanishing and exploding gradients, iii) large number of training steps and iv) the recurrence prevents from parallel training. *All* of these problems are improved by transformer networks. Transformers are entirely based on attention. They just look at contextual similarity and thus inherently neglect temporal ordering. Therefore the temporal ordering (positions) is incorporated by adding positional encoding to the input features.

- compare methods: https://jalammar.github.io/illustrated-bert/
- http://jalammar.github.io/illustrated-word2vec/
- https://www.youtube.com/watch?v=5vcj8kSwBCY

10 Normalizing flows

Baysian methods allow to model uncertainties. While there are methods of its own for Bayesian modeling via a Baysian belief network there are approaches that incorporate these ideas into neural network architectures [29].

In principle there are two sources for uncertainty. **Epistemic uncertainty** is inherent to the model due to the likelihood function, which is of the general $p(y|x, \theta)$ form for a supervised problem for a target random variable Y given covariates X and model parameters θ. **Aleotoric uncertainty** is the uncertainty on the model parameters coming from limited training data D, $p(\theta|D)$.

For likelihood functions, and hence epistemic uncertainty the traditional approach is to parametrize the likelihood function by some analytic pdf. In the context of neural networks these parameters are learned by a neural network with suitable activation functions in the output layer to fit the domain of definitions of these parameters. While this allows to incorporate features into the likelihood function in a non-linear fashion this may still limit the expressiveness of the model as it hinges on the particular choice of the likelihood function. In order to overcome this limitation recent attempts aim to learn the likelihood function in a parameter free fashion via normalizing flows [46, 34], diffusion models [24], or implicit quantile regression [11]

10.1 Transformation of random variables

Consider two random variables $X, Y \in \mathbb{R}^d$ with probability distributions p_X, p_Y. Let f be an **bijective** function such that $f(X) = Y$. Then the two probability distributions are linked by the change of variable formular.

$$p_Y(y) = p_X(f^{-1}(y))|\det Df^{-1}(y)| \tag{68}$$

where $Df^{-1}(y) = \partial f^{-1}/\partial y$ is the Jacobian evaluated at point y. The density p_Y is also sometimes called *push forward* of p_X by the function f, $f_* p_X$. So, given f and p_X one can compute the pdf p_Y. As f is bijective then the inverse transformation is

$$p_X(x) = p_Y(f(x))|\det Df(x)|. \tag{69}$$

Now we ask a more complicated question. Let there be given two probability distributions p_X, p_Y - does there exist a bijective function f that transforms these distributions into each other? However, from a practical point of view the mere existence is not enough as (10.1) dictates that the inverse and the Jacobi determinant should be cheap to compute. [8, 28]. Fortunately there is theoretical support,

Definition 3 *(Triangular map) A map $T = (T_1, \ldots, T_d) : \mathbb{R}^d \to \mathbb{R}^d$ is called triangular if T_1 is a function of x_1, T_2 a function of (x_1, x_2), and so on: T_i is a function of $(x_1, \ldots, x_i)$. A triangular map is called (strictly) increasing if every component T_i is (strictly) increasing with respect to the variable x_i (keeping the other variables $(x_1, \ldots, x_{i-1})$ fixed).*

Theorem 1 *[8, 28] For any two densities p_X, p_Y over $X = Y = \mathbb{R}^d$ there exists a unique increasing triangular map $T : X \to Y$ such that $p_Y = T_* p_X$.*

Consider some properties of triangular maps. First, the Jacobian of a triangular map is triangular because $\partial T_i / \partial x_j = 0$ for $j > i$ and therefore the Jacobian determinant can be computed from the diagonal elements $\det DT = \prod_i \partial T_i / \partial x_i$. The inverse strictly increasing triangular map is also a triangular map.

$$T^{-1} = (T_1^{-1}, \ldots, T_d^{-1}) : \mathbb{R}^d \to \mathbb{R}^d \tag{70}$$

Consider the scalar map $t_i : x_i \mapsto T_i(x_1, \ldots x_i)$, for fixed $x_1, \ldots x_{i-1}$. The inverse t_i^{-1} exists because this map is strictly increasing. This establishes the components of T^{-1}

$$\begin{aligned}
&T_1^{-1}(y_1) = t_1^{-1}(y_1) \\
&T_2^{-1}(y_1, y_2) = t_2^{-1}(T_1^{-1}(y_1), y_2) \\
&\ldots \\
&T_d^{-1}(y_1, \ldots, y_d) = t_d^{-1}(T_1^{-1}(y_1), \ldots, T_{d-1}^{-1}(y_1, \ldots, y_{d-1}), y_d)
\end{aligned} \tag{71}$$

Apparently this is a triangular map[5]. We need to show that it is indeed the inverse map $T^{-1} \circ T = T \circ T^{-1} = id$. We show this by induction in d. For $d = 1$ this is apparently true. Assume that the statment holds fof $d - 1$. Pick $x \in \mathbb{R}^d$, the components of $T(x)$ are given by $y_1 = T_1(x_1), y_2 = T_2(x_1, x_1), \ldots y_d = T_d(x_1, \ldots, x_d)$. By the induction assumption we have $T_1^{-1}(y_1) = x_1 \ldots, T_{d-1}^{-1}(y_1, \ldots y_{d-1}) = x_{d-1}$. For the last component we have $T_d^{-1}(y_1, \ldots, y_d) = t_d^{-1}(T_1^{-1}(y_1), \ldots, T_{d-1}^{-1}(y_1, \ldots, y_{d-1}), y_d) = t_d^{-1}(x_1, \ldots x_{d-1}, y_d)$ for fixed $x_1, \ldots x_{d-1}$. But this is a scalar function and its value at y_d is by construction x_d. Thus $T^{-1} \circ T = id$. And similar for $T \circ T^{-1} = id$

From a computational point of view the inverse T^{-1} needs to be computed iteratively as T_i^{-1} relies on $T_1^{-1}, \ldots, T_{i-1}^{-1}$. Therefore it cannot be computed in a parallel fashion (as opposed to T), which can be a limiting factor for models with high dimensions. However, the good news is that each component of this recursion is just the inverse of a scalar function, which can be computed fairly easy also numerically (if necessary).

The composition of triangular maps is again a triangular map. Consider two triangular maps T with $x'_i = T_i(x_1, \ldots, x_i)$ and T' then $(T' \circ T)_i = T'_i(x'_1, \ldots, x'_i) = T'_i(T_1(x_1), \ldots, T_i(x_1, \ldots, x_i))$ only depends in $x_1, \ldots, x_i$. $T' \circ T$ is (strictly) increasing if T and T' are.

In summary,

[5]Note that abuse of notation for $d = 1$: t_1^{-1} is the inverse of T_1. But T_1^{-1} is also denoted as the first component of the inverse map

- The Jacobian of a triangular map is triangular and the Jacobian Determinant can be computed from the diagonal entries only

$$\det DT = \prod_i \partial T_i / \partial x_i. \tag{72}$$

- The inverse of a strictly increasing triangular map is triangular and given by Eqs. (70) and (71).
- The composition of triangular maps $T = T^n \circ \cdots \circ T^2 \circ T^1$ is triangular and the Jacobian factorizes $DT = \prod_{i=1}^n DT^i$. If the T^i are strictly increasing then the inverse of T is given by $T = T^{1^{-1}} \circ \cdots \circ T^{n^{-1}}$.

Todo:

- look at paper again: do we need strictly increasing map T in order to assure invertibility? The paper has outlined this ...
- Also the general trafo rule requires a bijective function. The paper comes around to this and shows the formular for increasing triangular functions with a one sided derivative.
- Use examples from [28] to show that this is not so mystical at all (cdf in 1 d case¸)
- conditional version (probably a clean formulation helps with variational autoencoders)
- Batch normalization as normalizing flow
- See also Peter Abbeel on generative models:
  lectures + papers: https://sites.google.com/view/berkeley-cs294-158-sp20/home

10.2 Normalizing Flows: basic idea

Let p_X be a complicated distribution and p_Y a known, tractable pdf (uniform or normal distribution). Further assume that we only have samples of p_X $\mathcal{D} = \{x^{(i)}\}_{i=1}^N$ exists, but is unknown otherwise. We would like to learn a tringular map $f : X \to Y$ by a neural network (Fig. 3).

If we could succeed in learning f then we can use it in two fold manner. We could use it for density estimation and data generation by drawing samples from the known base distribution $y \sim p_Y$ and then obtain new samples

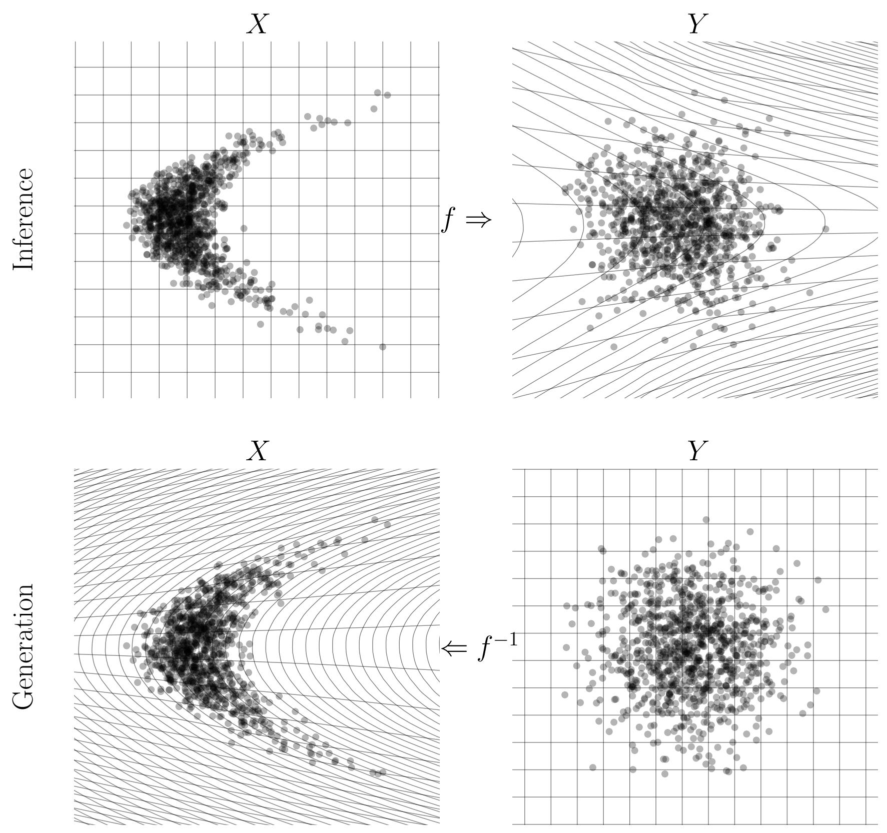

Figure 3: Illustration of normalizing flow.

in the target distribution p_X from $f^{-1}(y)$. From this samples all relevant statistical properties can be computed. Furthermore this approach can be also used to construct flexible test distributions for the approximate posterior in variational inference [51].

Let the base distribution be parametrized by parameters ϕ, such that $p_Y(y) = p_Y(y|\phi)$ and denote the neural network parameters of f by θ, $f_\theta(x) = f(x|\theta)$. The training (inference) task is to estimate the parameters of the base distribution and the function (ϕ, θ) from $\mathcal{D}$.

In order to learn the parameters θ of f one needs to define a loss function, which shall be minimized. This loss function should "compare" the pdfs from the inverse transformation induced by f from (69) $p_X(x|\phi,\theta)$ and the true, unknown distribution $p_X(x)$ (of which, we only have samples) in some sense. There are several possible loss functions and the Kullback-Leibler[6] divergence is an obvious one:

$$\begin{aligned}
\mathcal{L}(\phi,\theta,\mathcal{D}) &= \mathrm{KL}(p_X(x)\,||\,p_X(x|\phi,\theta)) \\
&= -\langle \ln p_X(x|\phi,\theta)\rangle_{p_X} + const. \\
&\approx -\frac{1}{N}\sum_i^N \ln p_X(x^{(i)}|\phi,\theta) + const. \\
&= -\frac{1}{N}\sum_i^N \left\{\ln p_Y(f(x^{(i)}|\theta)\,|\,\phi) + \ln|\det Df(x^{(i)}|\theta)|\right\} + const.,
\end{aligned} \tag{73}$$

where the constant terms are contributions independent of the paramters to optimize (ϕ,θ). In the second line the expectation value w.r.t. the unknown exact pdf is approximated by the empirical mean (Monte-Carlo approximation) given samples from $\mathcal{D}$. So (73) can be used to determine gradients on the function $\partial_\theta\mathcal{L}(\phi,\theta)$ and on the base distribution $\partial_\phi\mathcal{L}(\phi,\theta)$ needed to minimize the Kullback-Leibler divergence. Note however, that in some implementations only the parameters θ of f are learned keeping the parameters of the base distribution ϕ fixed.

Alternatively we could aim to determine the parameters (ϕ,θ) in order maximize the log-likelihood, $\ln p_X(\mathcal{D}|\phi,\theta)$, and thus minimize the negative of it,

$$\begin{aligned}
\mathcal{L}(\phi,\theta,\mathcal{D}) &= -\ln p_X(\mathcal{D}|\phi,\theta) \\
&= -\ln\prod_{i=1}^N p_X(x^{(i)}|\phi,\theta) \\
&= -\sum_i^N \left\{\ln p_Y(f(x^{(i)}|\theta)\,|\,\phi) + \ln|\det Df(x^{(i)}|\theta)|\right\},
\end{aligned} \tag{74}$$

which eventually gives up to the constant $1/N$ the same gradients as the KL divergence. Therefore, both versions share the same minimizer[7]. There are

[6]$\mathrm{KL}(q||p) = \int q(x)\ln\frac{q(x)}{p(x)}\,dx = \langle\ln q\rangle_q - \langle\ln p\rangle_q$

[7]However, when using an optimizer with given step length the optimal step length could be different as the computed gradients differ by $1/N$. On the other hand it ist most common to optimize the average of the log likelihood, which gives the same result as the KL approach.

Algorithm 1: Inference of a normalizing flow model using stochastic gradient descent (SGD)

```
Input:
    Data $\mathcal{D}$
    A base distribution $p_Y(\cdot|\phi)$
    A function $f_\theta$
Result: Learned parameters $\phi, \theta$
Initialize: $\phi, \theta$
while SGD not converged do
    $\mathcal{M} \sim \mathcal{D}$ (Random minibatch of data)
    Compute loss $\mathcal{L}(\theta, \phi, \mathcal{M})$ from Eq. (73)
    Compute gradients $\partial_\theta \mathcal{L}(\theta, \phi, \mathcal{M})$, $\partial_\phi \mathcal{L}(\theta, \phi, \mathcal{M})$
    Udate $\theta, \phi$ using SGD optimizer
end
```

Algorithm 2: Data generation with a normalizing flow

```
Input:
    A base distribution $p_Y(\cdot|\phi)$
    A function $f_\theta$
Result: Sample $x \sim p_X(\cdot|\theta)$
$y \sim p_Y(\cdot|\phi)$
$x = f_\theta^{-1}(y)$
```

several other possible loss functions like the Wasserstein metric as reviewed in [46].

This is summarized in the basic normalizing flow algorithm for inference and data generation in algorithms 1 and 2, respectively.

10.3 Constructing f

So far no statement of how to construct f has been made. Theorem 1 suggests to construct f as a triangular map. The theorem only guarantees that there is an increasing triangular map. However, as we wish to generate samples from the inverse map in practice strictly increasing triangular maps are constructed. Surprisingly some of these have been shown to be universal [28, 26][8]. While most of the normalzing flows are triangular maps, there are

[8]A flow is called universal if it can learn any target density to any required precision given sufficient capacity and data. The existence of such a map is guaranteed by Theorem 1

exceptions. In either case a common strategy to construct more complex normalizing flows is composition. It is illustrative to start with some simple transformations $f : \mathbb{R}^d \to \mathbb{R}^d$.

Example: Component wise affine linear flow. Consider the component wise map $y_i = e^{\alpha_i} x_i + \beta_i$. This is a trivial triangular map because $T_i(x_1, \ldots, x_i) = T_i(x_i)$. It's inverse is given by $x_i = e^{-\alpha_i}(y_i - \beta_i)$. Its Jacobian is given by $Df = \operatorname{diag}(e^{\alpha_1}, \ldots, e^{\alpha_d})$ and therefore $\ln |\det Df| = \ln |\prod_i e^{\alpha_i}| = \ln \prod_i e^{\alpha_i} = \sum_i \alpha_i$. This is essentially a affine linear map with the additional exponential map that ensures invertibility for any α_i.

Example: Permutation. A permuation $f_\pi : y = P_\pi x$ is given by an orthogonal permutation matrix $P_\pi^{-1} = P_{\pi^{-1}} = P_\pi^T$. It has Jacobi determinant $\det Df = \operatorname{sgn}(\pi)$ and therefore $\ln |\det Df| = \ln 1 = 0$. This is *not* a triangular map. However it is commonly used as "intertwiner" in a composition. This has been shown to be beneficial in practice. The reason is that triangular maps inherently depend on the ordering of the random variables (think of factorization of a joint pdf in terms of conditional pdfs). While this is in the limit of infinitly many training data probably not a problem, in practice changing the order of the variables may improve performance (in particular when triangular maps are composed). A generalization of the permutation groups are so called *linear normalizing flows*, which aim to determine the best ordering.

Example: batch normalization. Batch normalization [27] is used to speed up training can be understood as normalizing flow. It is of the adapted form [47]

$$y_i = \frac{x_i - \tilde{\mu}_i}{(\tilde{\sigma}_i^2 + \varepsilon)^{\frac{1}{2}}} e^{\alpha_i} + \beta_i, \; i = 1 \ldots d, \tag{75}$$

where α_i, β_i are the trainable parameters and ε is a small constant (hyperparameter) to ensure numerical stability ($\varepsilon = 10^{-5}$). $\tilde{\mu}_i$ and $\tilde{\sigma}_i$ are the empirical means and standard deviations derived from data $\mathcal{D} = \{x_i^n\}_{n=1}^N$ along dimension i for a minibatch (during training) and for test data (during prediction). Different choices for the choice of sampling space for training and prediction phase exist. As the map is defined component wise it is also triangular. It's inverse is given by

$$x_i = (y_i - \beta_i) \left(\tilde{\sigma}_i^2 + \varepsilon\right)^{\frac{1}{2}} e^{-\alpha_i} + \tilde{\mu}_i, \; i = 1 \ldots d, \tag{76}$$

and the Jacobian is diagonal $Df = \mathrm{diag}(e^{\alpha_1}/\,(\tilde{\sigma}_1^2+\varepsilon)^{\frac{1}{2}},\ldots,e^{\alpha_d}/\,(\tilde{\sigma}_d^2+\varepsilon)^{\frac{1}{2}})$, and therefore

$$\ln|\det Df| = \sum_{i=1}^{d}\left(\alpha_i - \frac{1}{2}\ln\left(\tilde{\sigma}_i^2+\varepsilon\right)\right). \tag{77}$$

Autoregressive flows. The affine linear flow extends to a more general form, which is still triangular, $y_i = e^{\alpha_i(x_{1:i-1})}x_i + \beta_i(x_{1:i-1})$, where α_i and β_i are now functions of $x_{1:i-1} = x_1,\ldots x_{i-1}$.

More generally, let $h_i(\cdot\,;\alpha_i) : \mathbb{R} \to \mathbb{R}$ be a bijections parametrized by $\theta_i \in \mathbb{R}^{|\theta_i|}$. Then a triangular map is called autoregressive flow if it is of the form

$$y_i = T_i(x_1,\ldots,x_i) = h_i(x_i;\gamma_i(x_{1:i-1})), \quad i = 1,\ldots,d, \tag{78}$$

with $\alpha_i = \gamma_i(x_{1:i-1})$. The function h_i is called coupling function or transformer and γ_i is called conditioner function. For $i = 1$ the conditioner does not depend on x. So, the conditioner "learns" the parameters for the univarite coupling function h_i.

As autoregressive flows are triangular, all properties hold – and they further simplify. In particular for the diagonal element of the Jacobian (72), $\partial y_i/\partial x_i$ only the derivative of the coupling functions needs to be comupted (keeping the conditioner fixed) in (78). And similar, the inverse needs to be computed only with respect to the coupling function: from (71)

$$\begin{aligned}T_d^{-1}(y_1,\ldots,y_d) &= t_i^{-1}(T_1^{-1}(y_1),\ldots,T_{i-1}^{-1}(y_1,\ldots,y_{i-1}),y_i)\\ &= t_i^{-1}(x_1\ldots x_{i-1},y_i)\\ &= h_i^{-1}(y_i;\gamma_i(x_1\ldots x_{i-1})),\end{aligned} \tag{79}$$

where in the last line the specific form of (78) was used. The inverse has still a recursive form as $h_i^{-1}(y_i;\gamma(x_1,\cdots,x_{i-1}))$ can only be computed once the $x_1,\ldots x_{i-1}$ (i.e. the inverses h_1^{-1}, h_{i-1}^{-1}) are known.

The property that the Jacobian and the inverse need not be evaluated for the conditioner functions is exploited in constructing autoregressive flows: the conditioner can be any "crazy" function (neural network) and only the coupling function needs fine tuning in order to be strictly increasing (bijective). From a theoretical point of view this is also beneficial as one can exploit the universal approximation theorem for the conditioner part and the universality property needs to shown only for the univariate coupling functions. [28, 61, 26]

Real NVP [13]. Fix $\ell < d$ then the real NVP normalzing flow is defined as

$$\begin{aligned} y_{1:\ell} &= x_{1:\ell} \\ y_{\ell+1:d} &= x_{\ell+1:d} \odot \exp(\alpha(x_{1:\ell})) + \beta(x_{1:\ell}), \end{aligned} \tag{80}$$

where the pair of conditioner functions $\gamma = (\alpha, \beta)$ with $\alpha, \beta : \mathbb{R}^\ell \to \mathbb{R}^{d-\ell}$ are neural networks with appropriate dimensions (whose parameters are optimized by e.g. algorithm 1). This kind of block parametrizations of normalizing flows are also called *coupling layers.* The inverse of real NVP is given by

$$\begin{aligned} x_{1:\ell} &= y_{1:\ell} \\ x_{\ell+1:d} &= (y_{\ell+1:d} - \beta(x_{1:\ell})) \odot \exp(-\alpha(y_{1:\ell})), \end{aligned} \tag{81}$$

and the diagonal elements of the Jacobian

$$\begin{pmatrix} 1_\ell & 0 \\ \cdot & \operatorname{diag}\left(e^{\alpha(x_{1:\ell})}\right), \end{pmatrix} \tag{82}$$

and therefore $\ln|\det Df| = \sum_{i=1}^{d-\ell} \alpha_i(x_{1:\ell})$, that is the sum over all components of the vector-valued function α.

To see that this is indeed an autoregressive flow write equivalently

$$y_i = T(x_{1:i}) = \begin{cases} T(x_i) = x_i & \text{for } i \leq \ell \\ T(x_{1:\ell}, x_i) = x_i \exp(\alpha_i(x_{1:\ell})) + \beta_i(x_{1:\ell}) & \text{for } i > \ell, \end{cases} \tag{83}$$

where α_i, β_i are the i-th components of $\alpha, \beta \in \mathbb{R}^{d-\ell}$, respectively. As $x_{1:\ell}$ in (84) are not transformed, in practice compositions with alternative permutations are used, $f = \ldots f_\pi \circ f_{NVP} \circ f_\pi \circ f_{NVP}$. After the permutation f_π untouched x parts enter a non trivial map. The coupling function in 84 is affine linear. The unusual exponential map ensures invertiblity for any $\alpha(x_1 : \ell)$. Therefore there are no constraints on the underlying neural network for the conditioner.

Masked autoregressive flow (MAF)[47]. The conditioners α, β in real-NVP are all the same for $i > \ell + 1$ as the take as input $x_{1:\ell}$ and therefore are not of the most general form of (78). MAF generlizes the parametrization of the conditioners in this direction:

$$y_i = x_i \exp(\alpha_i(x_{1:i-1})) + \beta_i(x_{1:i-1}), \tag{84}$$

where for each component there is now a pair of conditioner functions $\gamma_i = (\alpha_i, \beta_i),\ i = 1 \dots d$, with $\alpha_i, \beta_i : \mathbb{R}^{i-1} \to \mathbb{R}$ for $i > 1$ (and constants for $i = 1$). The logarithm of the Jacobian determinant is $\ln|\det Df| = \sum_{i=1}^{d} \alpha_i(x_{1:i-1})$. The inverse follows from (79) recursively,

$$x_i = (y_i - \beta_i(x_{1:i-1})) \exp(\alpha_i(x_{1:i-1})). \tag{85}$$

While MAF is a straightforward extension of the conditioner over real-NVP, it requires to set up $2d$ neural networks $(\alpha_i, \beta_i),\ i = 1 \dots d$ as opposed to only two in real-NVP (α, β)[9]. So for a single function evaluation of (85) one needs $2d$ neural network forward passes, which can be cumbersome for large d. However, one can combine these $2d$ neural networks into a single *masked* neural network [17].

Masked conditioner networks

Neural autoregressive flow (NAF) [26].

Sum-of-squares polynomial Flow (SOS) [28]. So far only affine coupling functions were used. In order to obtain more expressive coupling functions a more flexible strictly increasing coupling function is constructed from polynomials. The basic idea is to obtain a positive function by squaring a polynomial (these squared polynomials can be again expanded with some coefficients) and then construct a strictly increasing function by integrating it. Generally,

Proposition 1 *[28] A univariate real polynomial is increasing iff it can be written as*

$$h(x; a, b) = b + \int_0^x \sum_{j=1}^{k} \left(\sum_{i=0}^{r} a_{i,j} u^i \right)^2 du, \tag{86}$$

with $b \in \mathbb{R}, a \in \mathbb{R}^{(r+1)\times(k)}$. It is strictly increasing iff it is not constant. Furthermore let C be the space of real univariate continuous functions, equipped with the topology of compact convergence. Then, the set of increasing polynomials is dense in the cone of increasing continuous functions.

[9]One could express the pair of neural networks $\alpha, \beta : \mathbb{R}^\ell \to \mathbb{R}^{d-\ell}$ by doubling the output dimension and compute one neural network $\gamma : \mathbb{R}^\ell \to \mathbb{R}^{2(d-\ell)}$ and then use slicing in the coupling function (which can be computed vectorized). Typically one enlarges the network in this case to have a similar amount of parameters as as when working with two networks in order to ensure similar expressiveness. Similar for MAF $\gamma_i : \mathbb{R}^{i-1} \to \mathbb{R}^2$. But then there are still d of such combined networks.

The hyperparameter $r \in \mathbb{N}$ is called the degree of the polynomial, and $k \in \mathbb{N}$ is called number of polynomials A autoregressive flow can be readily constructed by

$$y_i = h_i(x_i, a_i, b_i) = h(x_i, a_i(x_{1:i-1}), b_i(x_{1:i-1})), \tag{87}$$

with the coupling function (86) and $\gamma_i(x_{1:i-1}) = (a_i(x_{1:i-1}), b_i(x_{1:i-1}))$ can be approximated by e.g., a masked conditioner network. That these polynomials are dense in the cone of increasing continuous function essentially shows the universality property of this flow.

In practice only small values for the hyper parameters k, r are used ($k = 5, r = 4$ or so). Further expressiveness can be achieved by composition (deep) or increasing r (wide), where the latter is theoretically more sound but introduces more parameters. Because there are just polynomials the integral can be performed analytically. The diagonal element of the Jacobian is given by[10]

$$\frac{\partial y_\ell}{\partial x_\ell} = \sum_{j=1}^{k} \left(\sum_{i=0}^{r} (a_\ell)_{i,j} \, x_\ell^i \right)^2 . \tag{89}$$

However the inverse cannot be computed analytically. It needs to be computed numerically for each y_i on the real line together with the general formular (79).

SOS is easier to train than NAF as there are no constraints on parameters of the coupling function. Also note that for $r = 0$ SOS reduces to the affine couling function.

Summary. While there are several non-triagonal flows, autoregressive forms are a promising avenue to construct flows.

- By their definition they are triangular and thus Theorem 1 may be exploited.
- The **conditioning function** can be approximated efficiently by masked conditioners and accurately due to the universal approximation theorem for deep neural networks.

[10]

$$\frac{\partial}{\partial x} \int_0^x f(u)\, du = \frac{\partial}{\partial x} \left(F(x) - F(0) \right) = \frac{\partial}{\partial x} F(x) = f(x) \tag{88}$$

- For the **coupling function** it is enough to construct univariate (strictly) increasing functions. This can be achieved by care carefully designing a neural network as in NAF. More generally this can also achieved by integrating non-negative functions [61] as in the Sum-of-squares polynomial flow.
- While such constructions are necessary to assure universality they have a drawback compared to affine coupling functions (or other simple, yet analytically invertible coupling functions): for sampling they typically need to be inverted numerically (for each y_i in one dimension respectively). If there are more sampling steps then inference steps, then one could also learn the f^{-1} rather than f like in the inverse autoregressive flow (IAF) [32]. If both, efficient sampling and density estimates is important then [45] train a student network for f^{-1} with f as a teacher network.

10.4 Conditional probabilities

[47]

10.5 Dequantization

The theory assumes continuous random variables. However, in practice often discrete data occur. Assume that the input data data are count data $x_i \in \mathbb{N}$. Instead of the real data one adds uniform noise, $x'_i = x_i + Unif(-1/2, 1/2))$ and feeds that data into the flow algorithm. This process is called dequantization.

11 Variational Autoencoders

[33] http://krasserm.github.io/2019/03/14/bayesian-neural-networks/

12 Time Series Neural Networks

Common practices for time series modelling

- **Split time** in context and prediction times
- **Scaling:** scale data over context time and rescale after prediction

- **Metrics for evaluation** *Continuous Ranked Probability Score* (CRPS) for probabilistic time series models
- **Dequantization:** corresponds to adding noise (e.g. from a uniform distribution) for disrecte data when using models that require contineous data (eg. normalizing flow models).
- **Categorical features:** use embeddings

Time series can be understood as subclass of sequence models. This section summarizes explicit time series models.

LSTNet. This network [36] aims to model long and short term patterns by combining neural net components with an autoregressive model. A prediction at time step t given by

$$\hat{y}_t = y_t^{AR} + y_t^{NN}, \tag{90}$$

where y_t^{NN} is the neural net component and y_t^{AR} is the auto regressive component. The auto regressive part is sensitive to the scale of the inputs (which is a hard for neural networks). This significantly improves model performance.

The neural network part models modulations with respect to this baseline model. It introduces a so called recurrent skip connection by adapting the formulae of e.g. LSTM (in the paper of GRU): LSTM uses hidden states at time $t-1$ as inputs for time t. The paper instead uses hidden states of time step $t-p$ as input for time step t. This adds a parameter p (the number of hidden states skipped through) and needs to be tuned. For periodic patterns p is obviously the periodicity. This method should increase the long term capacity of standard RNNs.

$$\text{Temporal convolution: } y_t \mapsto y'_t \tag{91}$$

$$\text{Gated recurrent network: } y'_t \mapsto y''_t \tag{92}$$

$$\text{Recurrent-skip network: } y'_t \mapsto y'''_t \tag{93}$$

$$\text{Combination } y_t^{NN} = W^1 y''_t + \sum_{i=0}^{p-1} W_i^2 y'''_{t-i} + b \tag{94}$$

The convolution shall capture short term dependencies and serves as input for following RNN. The RNN should capture long term time dependencies and even more the recurrent skip network. Finally both recurrent units are combined by a small dense NN. As input serve only the last hidden state at time t of the RNN (context vector) and the last p hidden states of the recurrent-skip network. Instead of the last two steps also the attention mechanism can be used (which also increases the long term capacity). The authors find similar accuracy within these two approaches.

13 Outlook

- Practical tips [6, 23, 42]
- Interpretation of NN [41, 30]
- (Denoising) autoencoder
- Adversarial nets (generative vs. discriminative)
- Hopfield networks [49]
- Convolution neural networks
- Hyper parameter tunining using Gaussian processes
- eg VGG http://www.robots.ox.ac.uk/ vgg/practicals/cnn/index.html

A Numerics

Log-sum-exponent trick. The logarithm of sum of exponentials may suffer from numerical overflow (due to exponent) or underflow. A numerical stable equivalent is

$$\ln \sum_i e^{x_i} = \ln \sum_i e^{x_i - c} + c, \quad c = \max\{x_i\}, \tag{95}$$

which prevents the exponent to explode (numerical overflow). This idea may be put forward compute numerical stable logarithms of matrix multiplication. More generally, consider the logarithm of a Tensor contraction

$$\begin{aligned} \ln \sum_k A_{ikl} B_{ikl} &= \ln \sum_k e^{\ln A_{ikl}} e^{\ln B_{ikl}} \qquad (96) \\ &= \ln \sum_k \exp(\ln A_{ikl} + \ln B_{ikl}) \qquad (97) \end{aligned}$$

As a special case the matrix product is given by

$$\ln AB_{ij} = \ln \sum_k A_{ik} B_{kj} = \ln \sum_k \exp(\ln A_{ik} + \ln B_{kj}), \tag{98}$$

both can be computed in a numerical stable fashion using (95). Furthermore the logarithm of a Matrix product can be written as a generalized sum of the individual Matrix logarithms.

References

[1] The illustrated gpt2. http://jalammar.github.io/illustrated-gpt2/.

[2] The illustrated transformer. http://jalammar.github.io/illustrated-transformer/.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2016.

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2016.

[5] D. Barber. *Bayesian Reasoning and Machine Learning.* Cambridge University Press, 04-2011 edition, 2011. In press.

[6] Y. Bengio. Practical recommendations for gradient-based training of deep architectures. *ArXiv e-prints*, June 2012.

[7] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[8] V I Bogachev, A V Kolesnikov, and K V Medvedev. Triangular transformations of measures. *Sbornik: Mathematics*, 196(3):309–335, apr 2005.

[9] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L. Yuille. Semantic image segmentation with deep convolutional nets and fully connected crfs, 2016.

[10] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation, 2014.

[11] Will Dabney, Georg Ostrovski, David Silver, and Remi Munos. Implicit quantile networks for distributional reinforcement learning. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1096–1105. PMLR, 10–15 Jul 2018.

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[13] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using real nvp, 2016.

[14] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning, 2018.

[15] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

[16] Valentin Flunkert, David Salinas, and Jan Gasthaus. Deepar: Probabilistic forecasting with autoregressive recurrent networks. *CoRR*, abs/1704.04110, 2017.

[17] Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle. Made: Masked autoencoder for distribution estimation. 2015.

[18] A. Felix Gers, N. Nicol Schraudolph, and Jürgen Schmidhuber. Learning precise timing with lstm recurrent networks. *Journal of Machine Learning Research*, pages 115–143, 2002.

[19] F. A. Gers and E. Schmidhuber. Lstm recurrent networks learn simple context-free and context-sensitive languages. *IEEE Transactions on Neural Networks*, 12(6):1333–1340, 2001.

[20] F.A. Gers. Learning to forget: continual prediction with lstm. *IET Conference Proceedings*, pages 850–855(5), January 1999.

[21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016.

[22] Klaus Greff, Rupesh K. Srivastava, Jan Koutnik, Bas R. Steunebrink, and Jurgen Schmidhuber. Lstm: A search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, 28(10):2222–2232, Oct 2017.

[23] Geoffrey E. Hinton. *A Practical Guide to Training Restricted Boltzmann Machines*, pages 599–619. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[24] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models, 2020.

[25] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[26] Chin-Wei Huang, David Krueger, Alexandre Lacoste, and Aaron Courville. Neural autoregressive flows, 2018.

[27] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[28] Priyank Jaini, Kira A. Selby, and Yaoliang Yu. Sum-of-squares polynomial flow, 2019.

[29] Laurent Valentin Jospin, Wray Buntine, Farid Boussaid, Hamid Laga, and Mohammed Bennamoun. Hands-on bayesian neural networks – a tutorial for deep learning users, 2020.

[30] Pieter-Jan Kindermans, Kristof T Schütt, Maximilian Alber, Klaus-Robert Müller, Dumitru Erhan, Been Kim, and Sven Dähne. Learning how to explain neural networks: Patternnet and patternattribution. In *6th International Conference on Learning Representations*, 2018.

[31] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[32] Diederik P. Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, and Max Welling. Improving variational inference with inverse autoregressive flow, 2016.

[33] Diederik P. Kingma and Max Welling. An introduction to variational autoencoders. *CoRR*, abs/1906.02691, 2019.

[34] Ivan Kobyzev, Simon J.D. Prince, and Marcus A. Brubaker. Normalizing flows: An introduction and review of current methods. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(11):3964–3979, nov 2021.

[35] J. F. Kolen and S. C. Kremer. *Gradient Flow in Recurrent Nets: The Difficulty of Learning LongTerm Dependencies*, pages 237–243. 2001.

[36] Guokun Lai, Wei-Cheng Chang, Yiming Yang, and Hanxiao Liu. Modeling long- and short-term temporal patterns with deep neural networks. *CoRR*, abs/1703.07015, 2017.

[37] Bryan Lim and Stefan Zohren. Time series forecasting with deep learning: A survey, 2020.

[38] Bryan Lim, Stefan Zohren, and Stephen Roberts. Recurrent neural filters: Learning independent bayesian filtering steps for time series prediction, 2020.

[39] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation, 2015.

[40] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation, 2015.

[41] G. Montavon, W. Samek, and K.-R. Müller. Methods for Interpreting and Understanding Deep Neural Networks. *ArXiv e-prints*, June 2017.

[42] Grégoire Montavon, Genevieve B. Orr, and Klaus-Robert Müller, editors. *Neural Networks: Tricks of the Trade - Second Edition*, volume 7700 of *Lecture Notes in Computer Science*. Springer, 2012.

[43] G. Montúfar, R. Pascanu, K. Cho, and Y. Bengio. On the Number of Linear Regions of Deep Neural Networks. *ArXiv e-prints*, February 2014.

[44] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization.* Springer, New York, NY, USA, second edition, 2006.

[45] Aaron van den Oord, Yazhe Li, Igor Babuschkin, Karen Simonyan, Oriol Vinyals, Koray Kavukcuoglu, George van den Driessche, Edward Lockhart, Luis C. Cobo, Florian Stimberg, Norman Casagrande, Dominik Grewe, Seb Noury, Sander Dieleman, Erich Elsen, Nal Kalchbrenner, Heiga Zen, Alex Graves, Helen King, Tom Walters, Dan Belov, and Demis Hassabis. Parallel wavenet: Fast high-fidelity speech synthesis, 2017.

[46] George Papamakarios, Eric Nalisnick, Danilo Jimenez Rezende, Shakir Mohamed, and Balaji Lakshminarayanan. Normalizing flows for probabilistic modeling and inference. 2019.

[47] George Papamakarios, Theo Pavlakou, and Iain Murray. Masked autoregressive flow for density estimation, 2017.

[48] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[49] Hubert Ramsauer, Bernhard Schäfl, Johannes Lehner, Philipp Seidl, Michael Widrich, Thomas Adler, Lukas Gruber, Markus Holzleitner, Milena Pavlović, Geir Kjetil Sandve, Victor Greiff, David Kreil, Michael Kopp, Günter Klambauer, Johannes Brandstetter, and Sepp Hochreiter. Hopfield networks is all you need, 2020.

[50] Syama Sundar Rangapuram, Matthias W Seeger, Jan Gasthaus, Lorenzo Stella, Yuyang Wang, and Tim Januschowski. Deep state space models for time series forecasting. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[51] Danilo Jimenez Rezende and Shakir Mohamed. Variational inference with normalizing flows, 2015.

[52] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Neurocomputing: Foundations of research. chapter Learning Representations by Back-propagating Errors, pages 696–699. MIT Press, Cambridge, MA, USA, 1988.

[53] Hojjat Salehinejad, Sharan Sankar, Joseph Barfett, Errol Colak, and Shahrokh Valaee. Recent advances in recurrent neural networks, 2018.

[54] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *CoRR*, abs/1404.7828, 2014.

[55] Alex Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, 404:132306, Mar 2020.

[56] Xingjian Shi, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai kin Wong, and Wang chun Woo. Convolutional lstm network: A machine learning approach for precipitation nowcasting, 2015.

[57] Sargur Srihari. Regularization in neural networks.

[58] T. Tieleman and G. Hinton. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

[59] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[60] Yuyang Wang, Alex Smola, Danielle C. Maddix, Jan Gasthaus, Dean Foster, and Tim Januschowski. Deep factors for forecasting, 2019.

[61] Antoine Wehenkel and Gilles Louppe. Unconstrained monotonic neural networks. 2019.

[62] P. Xu, F. Roosta-Khorasani, and M. W. Mahoney. Second-Order Optimization for Non-Convex Machine Learning: An Empirical Study. *ArXiv e-prints*, August 2017.

[63] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions, 2016.

[64] Matthew D. Zeiler, Dilip Krishnan, Graham W. Taylor, and Rob Fergus. Deconvolutional networks. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 2528–2535, 2010.