Neural Networks

May 19, 2021

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 algorithms 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. [35].

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 [33] 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

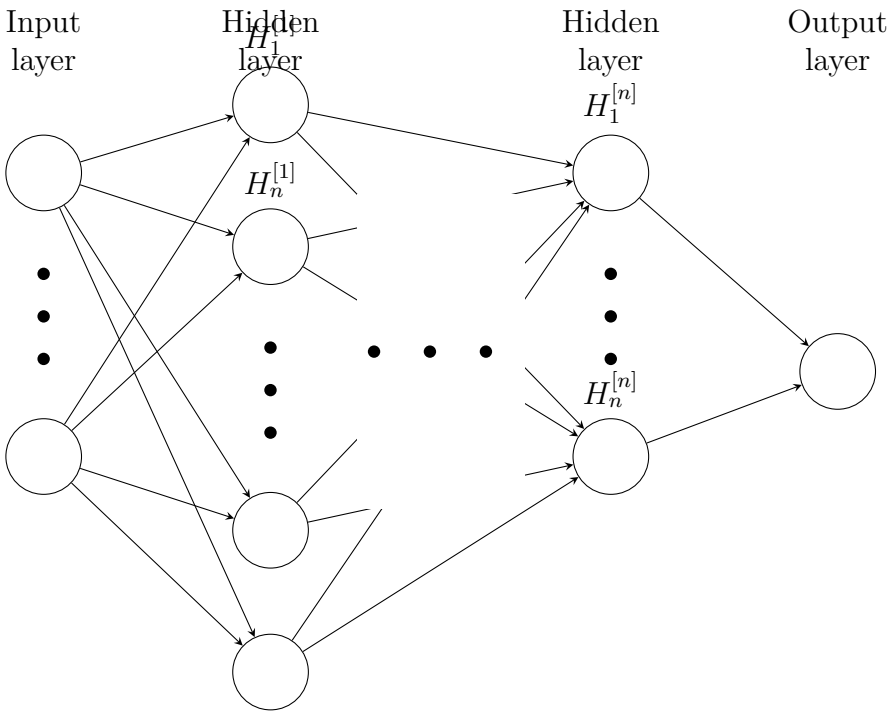

Figure 1: Illustration of a deep neural network with a single output unit.

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 [28]. 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 [15] 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 [38, 15]:

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 [38] ?!?

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[38]:

- 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 $\theta^* = \text{argmin}_\theta J(\theta)$ [29]. 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

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

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

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

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

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

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:

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

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

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

The term $\langle \nabla J \rangle_{k-1}^{\mathrm{EMA}}$ 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 [39]. 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.

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

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

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

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 [20]. 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

$$\langle \nabla J \rangle_0^{\text{cEMA}} = 0$$
$$\langle \nabla J \odot \nabla J \rangle_0^{\text{cEMA}} = 0$$

- Update Update moments

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

- Update optimization step

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

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 $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$ [20].

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" [15].

In summary, there seems to be quite some room for improving optimization algorithms [42].

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].

9 Sequence Models

Sequence models are largely motivated from natural language processing but also find application in time series modeling [11, 22, 32, 41, 23].

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, 24], 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 [40, 2] and state of the art NLP models like BERT [9] or GPT2 [30, 1].

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 [34, 22]. Furthermore, RNNs are related to Bayesian filtering in discrete state Markov models [23, 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

[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).

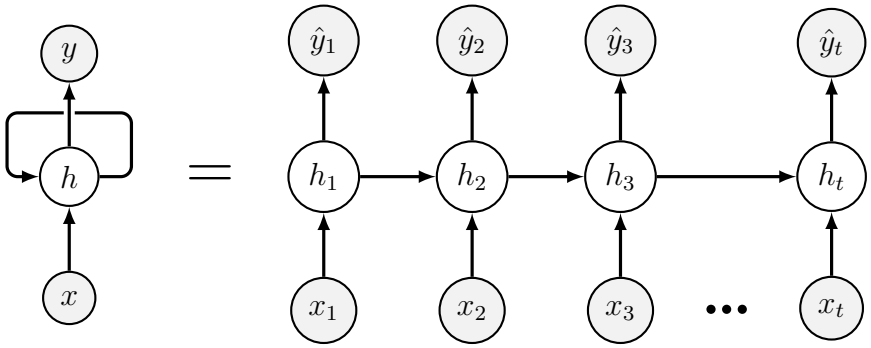

Figure 2: Illustration of a RNN. The right hand side shows the time unfolded structure, which explicitly depends on time.

the simplest RNNs, the Eilman RNN, takes as internal memory state h_t the expression [10, 15]

$$h_t = \gamma_h \left(W_{h_1} h_{t-1} + W_{h_2} x_t + b_h \right) \tag{24}$$

$$y_t = \gamma_y \left(W_y h_t + b_y \right), \tag{25}$$

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 consequnce 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, 21], which is implied by the exploding and vanishing gradient problem. [15]. 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

$$\overrightarrow{h}_t = \gamma_h \left(\overrightarrow{W}_{h_1} \overrightarrow{h}_{t-1} + \overrightarrow{W}_{h_2} x_t + \overrightarrow{b}_h \right) \tag{26}$$

$$\overleftarrow{h}_t = \gamma_h \left(\overleftarrow{W}_{h_1} \overleftarrow{h}_{t-1} + \overleftarrow{W}_{h_2} x_t + \overleftarrow{b}_h \right) \tag{27}$$

$$h_t = \overrightarrow{h}_t \oplus \overleftarrow{h}_t \tag{28}$$

$$y_t = \gamma_y \left(W_y h_t + b_y \right), \tag{29}$$

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

Long-Short-Term Memory (LSTM) LSTMs [18, 14] try to overcome the limitations in learning long-range dependencies in the data by improving "gradient flow" within the network [36]. 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,

$$\textit{Input gate:} \quad i_t = \sigma \left(W_{i_1} h_{t-1} + W_{i_2} x_t + b_i \right) \tag{30}$$

$$\textit{Output gate:} \quad o_t = \sigma \left(W_{o_1} h_{t-1} + W_{o_2} x_t + b_o \right) \tag{31}$$

$$\textit{Forget gate:} \quad f_t = \sigma \left(W_{f_1} h_{t-1} + W_{f_2} x_t + b_f \right), \tag{32}$$

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

$$\textit{Cell state} \quad c_t = f_t \odot c_{t-1} + i_t \odot \left(W_{c_1} h_{t-1} + W_{c_2} x_t + b_c \right) \tag{33}$$

$$\textit{hidden state:} \quad h_t = o_t \odot \tanh(c_t), \tag{34}$$

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 [16], in particular peephole LSTMs [13, 12] and peephole convolutional LSTMs [37].

9.2 Attention mechanism

The attention mechanism [4, 8, 25]. Assume a sequence $\{x_1, \dots, x_T\}$ with $x_i \in \mathbb{R}^{d_x}$. Then the attention mechanism is of the form

$$x'_t = \sum_i f(x_i)\, \lambda_{it} \tag{35}$$

$$\lambda_{it} = \lambda(x_i, x_t) = \operatorname{softmax} g(x_i, x_t), \tag{36}$$

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 (36), [2] 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 (35) 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 (35). 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.

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 (35) from x_t it*self*. Specifically [40] 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

$$q_t := W_q x_t, \quad k_i := W_k x_i, \quad v_i := W_v x_i, \tag{37}$$

and identify the functions f and g in (35) - (36) with

$$f(x_i) := v_i \tag{38}$$

$$g(x_i, x_t) := \frac{k_i^T q_t}{\sqrt{d_s}} = \frac{q_t^T k_i}{\sqrt{d_s}}, \tag{39}$$

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

gives as attention $x'_t = \sum_i v_i \lambda_{it}$ with $\lambda_{it} = \text{softmax}(q_t^T k_i / \sqrt{d_s})$. 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* $k_i, i \in \{1, \ldots, T\}$ (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

$$X := [x_1, \ldots, x_T] \in \mathbb{R}^{d_x \times T} \tag{40}$$

$$Q := W_q X \in \mathbb{R}^{d_s \times T}, \tag{41}$$

$$K := W_k X \in \mathbb{R}^{d_s \times T}, \tag{42}$$

$$V := W_v X \in \mathbb{R}^{d_h \times T}, \tag{43}$$

which gives

$$X' =: \mathcal{A}(Q, K, V) = V\lambda, \quad \lambda = \text{softmax}\left(\frac{Q^T K}{\sqrt{d_s}}\right), \tag{44}$$

where the softmax function is component wise and the normalization w.r.t to rows. This formulation computes the attention for all times simultaneously ($X' := [x'_1, \ldots, x'_T] \in \mathbb{R}^{d_h \times T}$). 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

$$Q_h = P_{h,q} W_q X = W_{h,q} X \in \mathbb{R}^{d_{s'} \times T}, \tag{45}$$

$$K_h = P_{h,k} W_k X = W_{h,k} X \in \mathbb{R}^{d_{s'} \times T}, \tag{46}$$

$$V_h = P_{h,v} W_v X = W_{h,v} X \in \mathbb{R}^{d_{h'} \times T}, \tag{47}$$

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[3] of the individual attentions together with an additional matrix multiplication by a matrix W,

$$X' = W \oplus_{h=1}^{H} \mathcal{A}(Q_h, K_h, V_h), \tag{48}$$

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

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

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

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

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

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

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

$$Q_h = XW_qP_{h,q} = XW_{h,q} \in \mathbb{R}^{T \times d_{s'}}, \tag{54}$$

$$K_h = XW_kP_{h,k} = XW_{h,k} \in \mathbb{R}^{T \times d_{s'}}, \tag{55}$$

$$V_h = XW_vP_{h,v} = XW_{h,v} \in \mathbb{R}^{T \times d_{h'}}, \tag{56}$$

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

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 (35): $\lambda_{it} = 0$ for $i > t$. In the

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

parametrization of self attention this can be achieved by masking

$$\mathcal{A}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_s}} + M\right) V, \tag{58}$$

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 Outlook

- Practical tips [6, 17, 27]
- Interpretation of NN [26, 19]
- (Denoising) autoencoder
- Adversarial nets (generative vs. discriminative)
- Hopfield networks [31]
- Convolution neural networks
- Hyper parameter tunining using Gaussian processes
- eg VGG http://www.robots.ox.ac.uk/ vgg/practicals/cnn/index.html

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] 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.

[9] 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.

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

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

[12] 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.

[13] 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.

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

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

[16] 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.

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

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

[19] 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.

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

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

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

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

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

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

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

[27] 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.

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

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

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

[31] 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.

[32] 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.

[33] 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.

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

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

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

[37] 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.

[38] Sargur Srihari. Regularization in neural networks.

[39] 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.

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

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

[42] 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.