Contents

List of Figures

List of Tables

1 Hadronic recoil regression with deep neural networks

In the recent years a significant progress was achieved in the field of big datasets analysis. There is a number of principles available for solving a wide variety of tasks. In this thesis a Deep Neural Network (DNN) was used for the regression of the 2-component hadronic recoil vector.

1.1 Deep neural networks

Normally a machine learning problem has a number of ingredients: a dataset $\mathbf{X}$, a set of parameters θ, a model $g(\theta)$ and a loss function $C(\mathbf{X})$ that tells us how well the model $g(\theta)$ describes the dataset. Finding the values of θ that would minimize the loss function we fit the model.

1.1.1 Gradient descent optimization

One of the most powerful and used class of methods in minimizing the loss function is called the *gradient descent*, [1] especially its sub-class, the stochastic gradient descent (SGD) [2], [3]. One of its modifications called ADAM [4] was used as an optimization algorithm in the work presented in this thesis.

Let's assume that a loss function $E(\theta)$ may be estimated as a sum over n data points:

$$E(\theta) = \sum_{i=1}^{n} e_i(x_i, \theta). \tag{1.1}$$

In the simplest case of the gradient descent (GD) algorithm we start looking for the values of parameters θ such that the sum of functions $\sum_{i=1}^{n} e_i$ is minimal. We start with a certain value θ_0 and then iteratively perform the following:

$$\begin{aligned} v_t &= \eta_t \nabla_\theta E(\theta_t), \\ \theta_{t+1} &= \theta_t - v_t, \end{aligned} \tag{1.2}$$

where $\nabla_\theta E(\theta_t)$ is the gradient of $E(\theta)$ with respect to θ; factor η_t is called the *learning rate* and defines the length of the step in the direction of θ performed with every iteration. Balancing learning rate is very important for learning process and convergence. A value too low can make the convergence "stuck" in the local minimum, it also increases the number of iterations. Picking a very high learning

rate we risk to miss the minimum so the algorithm would never converge to a minimum. Also, if the number of data points n is high, calculating the gradient is a costly task in terms of CPU time.
Some of the problems accompanying the use of GD are dealt with by using its modification - the SGD. The idea is the following: instead of using all the available data points n at each iteration of the GD, we split the data into k *minibatches*, each having M data points, such that $k = n/M$. Normally the size of the batch is few hundreds of data points, to provide a certain degree of variance and incorporating stochasticity. The transition to SGD algorithm is done in the following way:

$$\nabla_\theta E(\theta) = \sum_{i=1}^{n} \nabla_\theta e_i(x_i, \theta) \rightarrow \sum_{i \in B_l} \nabla_\theta e_i(x_i, \theta), \tag{1.3}$$

where B_l is a set of data points belonging to a minibatch $l \in 1, ..., n/M$. Now every next iteration of θ parameters update is performed over a different batch, consecutively running over all the batches:

$$\begin{aligned} &\nabla_\theta E^{EM}(\theta) = \textstyle\sum_{i \in B_l} \nabla_\theta e_i(x_i, \theta), \\ &v_t = \eta_t \nabla_\theta E^{EM}(\theta_t), \\ &\theta_{t+1} = \theta_t - v_t. \end{aligned} \tag{1.4}$$

A full iteration over all the n/M batches is called an *epoch*. Now stochasticity prevents the gradient algorithm from getting stuck in a local minimum. Also computing the gradient over fewer data point notably decreases the CPU time spent.
The algorithm may be further improved, adding a "memory", that is to say making every next step t dependent on the direction of the previous step $t-1$:

$$\begin{aligned} &v_t = \gamma v_{t-1} \eta_t \nabla_\theta E^{EM}(\theta_t), \\ &\theta_{t+1} = \theta_t - v_t. \end{aligned} \tag{1.5}$$

Thanks to analogy from physics the parameter γ is called a *momentum*, having $0 \leq \gamma \leq 1$ [5], [6]. This parameter provides a certain "inertia" in the change of the direction of the gradient descent. Introduction of the momentum helps for quicker convergence in the case of a slow but steady change of a certain parameter during the gradient descent.
The convergence of the GD may be significantly improved if the learning rate could be different in different directions, depending on the landscape of the parameter space θ: the steeper the gradient in a certain direction - the smaller the corresponding step. The optimal step could be estimated by obtaining the *Hessian matrix* in the vicinity of a point θ_0, providing a description of the local curvature in a multidimensional space. Although calculating Hessian matrix is complicated and slow-converging process [7]. However, a number of methods use the second moment of the gradient to efficiently estimate the optimal learning rate. One of such methods is called ADAM (ADAptive Momentum) [4],

its iterative relations are the following:

$$\begin{aligned}
g_t &= \nabla_\theta E(\theta_t) \\
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
s_t &= \beta_2 s_{t-1} + (1-\beta_2) g_t^2 \\
\hat{m}_t &= \frac{m_t}{1-(\beta_1)^t} \\
\hat{s}_t &= \frac{s_t}{1-(\beta_2)^t} \\
\theta_{t+1} &= \theta_t - \eta_t \frac{\hat{m}_t}{\sqrt{\hat{s}_t} + \epsilon}.
\end{aligned} \tag{1.6}$$

Here the parameters β_1 and β_2 set the memory lifetime for the first and second moment; $\eta$is the learning rate and ϵ is a small regularization constant keeping the denominators from vanishing. Like in other cases of the SGD here the iterations are performed batch-wise. Parameter s_t is linked to the variance of the gradient size. This basically means that the learning rate is proportional to the first momentum of the gradient and inverse proportional to its standard deviation.

1.1.2 DNN structure and training

A neural network is composed of single neurons, also called nodes, arranged in layers. The first layer is called the input layer, the last one is called the output layer; all the layers in between are named hidden layers (see Fig. 11).
A single node i takes a vector of k input features $\mathbf{x} = (x_1, x_2, ..., x_k)$ and produces a scalar input $a_i(\mathbf{x})$. Function a_i may have a different form, although it normally can be decomposed into two steps. The first step is a linear transformation of the inputs into a scalar value assigning each input a weight:

$$z^i = w^i_k \cdot x_k + b^i, \tag{1.7}$$

where $\mathbf{w}^i = (w^i_1, w^i_2, ..., w^i_k)$ is a set of k weights assigned to corresponding inputs. The weights $\mathbf{w}^i$ are specific to a neuron i, as well as the scalar bias b^i. The next step is where the non-linear function σ_i comes into play: we can express the output function $a_i(\mathbf{x})$ as follows:

$$a_i(\mathbf{x}) = \sigma_i(z^i). \tag{1.8}$$

There exists a number of options for the non-linear function σ; in current thesis a tanh is used. When the neurons are arranged in layers in a feed-forward neural network - the outputs from neurons of the previous layer serve as inputs for the succeeding layers neurons. The universal approximation theorem states, that a neural network with a single hidden layer can approximate any continuous multiparametric function with arbitrary accuracy [8], [9]. However, in practice it is easier to reach the possible precision having more hidden layers.
So in terms of a DNN fitting the model means tuning the weights and biases $(\mathbf{w}^i, b^i)$ in such a way that a loss function applied to the new dataset would be minimal. It is reached through iterative process called *training*, that involves the GD with an algorithm called *backpropagation* [10]. The backpropagation algorithm allows to calculate the gradients and adjust the corresponding parameters

in a very computation-efficient way.

Let us assume that there are L layers in the network $l = 1,...,L$, that w^l_{jk} and b^l_j are the weight of an

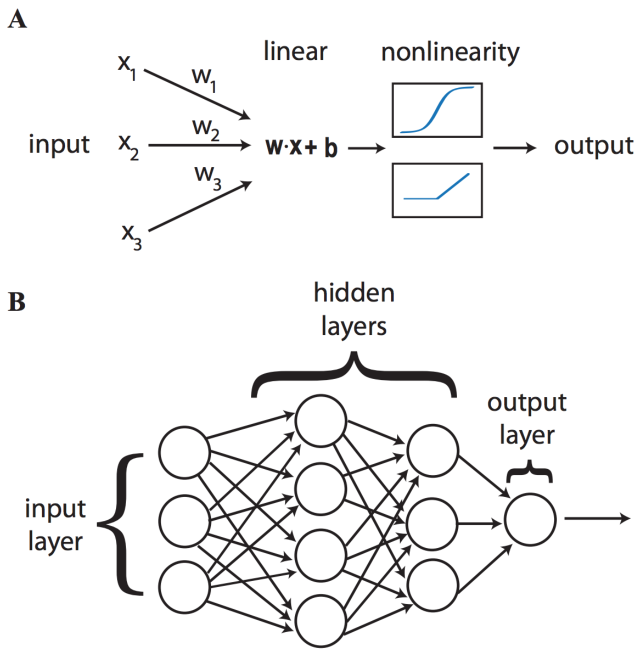

Figure 11: A: The nodes perform a linear transformation of the inputs, then apply a non-linear activation function. B: The architecture of a deep neural network: neurons are arranged into layers [11].

input parameter k and the bias for node k in layer l respectively. The layered structure of the neural network ensures that the inputs for the nodes in layer l depend only on the outputs of the nodes from layer $l-1$, hence:

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right). = \sigma(z^l_j), \tag{1.9}$$

where the linear weighted sum is denoted as:

$$\sigma(z^l_j) = \sum_k w^l_{jk} a^{l-1}_k + b^l_j. \tag{1.10}$$

The cost function E is computed from the output of the neural network, so it directly depends only on the values of a^L_j. Let us define the error Δ^L_j of the j-th node in the output (L-th) layer as a change in the cost function with respect to the weighted output of the last layer:

$$\Delta^L_j = \frac{\partial E}{\partial z^L_j}. \tag{1.11}$$

At the same time the loss depends indirectly on all the preceding layers, so keeping in mind eq. 1.9 we can define the error of an arbitrary node j in arbitrary layer l as the change in the cost function E with respect to the weighted input z_j^l:

$$\Delta_j^l = \frac{\partial E}{\partial z_j^l} = \frac{\partial E}{\partial a_j^l}\sigma'(z_j^l), \tag{1.12}$$

where $\sigma'(z_j^l)$ is the derivative of the non-linear activation function σ with respect to its input at z_j^l. But on the other hand we can also interpret the error function Δ_j^L in terms of bias partial derivatives:

$$\Delta_j^l = \frac{\partial E}{\partial z_j^l} = \frac{\partial E}{\partial b_j^l}\frac{\partial b_j^l}{\partial z_j^l} = \frac{\partial E}{\partial b_j^l} \cdot \mathbf{1}. \tag{1.13}$$

So starting from the output layer we can compute the error in any layer l, provided we know it for the subsequent layer $l+1$:

$$\begin{aligned} \Delta_j^l &= \tfrac{\partial E}{\partial z_j^l} = \textstyle\sum_k \tfrac{\partial E}{\partial z_j^{l+1}} \tfrac{\partial z_j^{l+1}}{\partial z_j^l} = \\ &= \textstyle\sum_k \Delta_j^l \tfrac{\partial z_j^{l+1}}{\partial z_j^l} \left(\textstyle\sum_k \Delta_j^l w_{kj}^{l+1}\right)\sigma'(z_j^l). \end{aligned} \tag{1.14}$$

And finally we can get the gradient of the cost function E with respect to a weight of an arbitrary neuron:

$$\frac{\partial E}{\partial w_{jk}^l} = \frac{\partial E}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} = \Delta_j^l a_k^{l-1}. \tag{1.15}$$

Using these four equations (1.11, 1.13, 1.14, 1.15) it is possible to "backpropagate" the error back from the output layer and once we can compute the gradient - we know how we should tune the weights and biases in order to minimize the loss function.

1.1.3 Batch normalization

Batch normalization is a regularization scheme that helps to improve the speed and stability of the DNN training. The main idea behind the method is to prevent an *internal covariant shift* - a change in the distribution of network activations due to the change in network parameters during training [12] by means of normalization of the parameters transferred from layer l to layer $l+1$. So let us consider a layer l that has d inputs $\mathbf{x} = (x^1, x^2, ..., x^d)$, then for every x^k we perform the following transformation:

$$\hat{x}^k = \frac{x^k - E[x^k]}{\sqrt{Var[x^k]}}, \tag{1.16}$$

where $E[x^k]$ and $Var[x^k]$ are the expectation and variance of the parameter x, calculated over the training dataset, respectively. Although we have to be sure that we preserve the non-linearity of the activation function output. In order to do this the two additional parameters are introduced:

$$y^k = \gamma\hat{x}^k + \beta^k, \tag{1.17}$$

where the parameters γ and β are trained just like the rest of the network parameters. Practically if the training is performed within the mini-batch scheme with batch size $B = x_1, ..., x_m$ the batch normalization layer is inserted between the DNN layers the transformations for the input x are the following:

$$\begin{aligned}
&\frac{1}{m}\sum_{i=1}^{m} x_i \longrightarrow \mu_B \\
&\frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_B)^2 \longrightarrow \sigma_B^2 \\
&\frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \longrightarrow \hat{x}_i \\
&\gamma \hat{x}_i + \beta \longrightarrow y_i \equiv BN_{\gamma,\beta}(x_i),
\end{aligned} \tag{1.18}$$

where ϵ is a small regularization constant.

1.2 HR regression

1.2.1 Input samples and features

1.2.2 Network parameters

1.2.3 Results

1.3 Technical details

Bibliography

[1] Yann Lecun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE*. 1998, pp. 2278–2324.

[2] Léon Bottou. “Stochastic Gradient Descent Tricks”. In: vol. 7700. Jan. 2012, pp. 421–436. DOI: 10.1007/978-3-642-35289-8_25.

[3] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning representations by back-propagating errors”. In: *Nature* 323 (1986), pp. 533–536.

[4] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015. 2014. URL: http://arxiv.org/abs/1412.6980.

[5] Yu Nesterov. “A method of solving a convex programming problem with convergence rate $O(1/k^2)$”. In: vol. 27. Jan. 1983, pp. 372–376.

[6] Boris Polyak. “Some methods of speeding up the convergence of iteration methods”. In: *Ussr Computational Mathematics and Mathematical Physics* 4 (Dec. 1964), pp. 1–17. DOI: 10.1016/0041-5553(64)90137-5.

[7] Yann LeCun et al. “Efficient BackProp”. In: *Neural Networks: Tricks of the Trade*. Ed. by Genevieve B. Orr and Klaus-Robert Müller. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 9–50. ISBN: 978-3-540-49430-0. DOI: 10.1007/3-540-49430-8_2. URL: https://doi.org/10.1007/3-540-49430-8_2.

[8] Kurt and Hornik. “Approximation capabilities of multilayer feedforward networks”. In: *Neural Networks* 4.2 (1991), pp. 251–257. DOI: 10.1016/0893-6080(91)90009-T. URL: http://www.sciencedirect.com/science/article/pii/089360809190009T.

[9] G. Cybenko. “Approximation by superpositions of a sigmoidal function”. In: *Mathematics of Control, Signals and Systems* 2.4 (Dec. 1989), pp. 303–314. ISSN: 1435-568X. DOI: 10.1007/BF02551274. URL: https://doi.org/10.1007/BF02551274.

[10] D. E. Rumelhart and D. Zipser. “Feature Discovery by Competitive Learning”. In: *Parallel Distributed Processing*. MIT Press, 1986, pp. 151–193.

[11] Pankaj Mehta et al. “A high-bias, low-variance introduction to Machine Learning for physicists”. In: *Phys. Rept.* 810 (2019), pp. 1–124. DOI: 10.1016/j.physrep.2019.03.001. arXiv: 1803.08823 [physics.comp-ph].

[12] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*. ICML15. Lille, France: JMLR.org, 2015, pp. 448–456.