Machine Learning, Deep learning and Quantum Mechanics

by

Karl Henrik Fredly

Thesis

for the degree of

Master of Science

Faculty of Mathematics and Natural Sciences
University of Oslo

June 2022

Abstract

This is an abstract text.

Acknowledgements

I acknowledge my acknowledgements.

Contents

Chapter 1

Introduction

Good luck.

Part I

Theory

Chapter 2

Quantum Mechanics

2.1 The Schrödinger Equation

2.1.1 The Born-Oppenheimer Approximation

2.1.2 The Electronic Schrödinger Equation

2.1.3 Atom Units

2.2 The Variational Principle

Chapter 3

Hartree-Fock theory

Chapter 4

Coupled-Cluster theory

Chapter 5

The Variational Monte-Carlo method

Chapter 6

Feedforward Neural Networks

A feedforward neural network (FNN) is a model central to the field of machine learning. Even though it is a relatively simple model, it still sees use among more complicated models due to its flexibility and performance.

FNNs are a type of artificial neural network which can be used for approximating any arbitrary continuous function. The universal approximation theorem for neural networks states that a neural network with only one layer can approximate any continuous function from one finite dimensional space to another [3]. The ground state wave function we are trying to approximate falls within this category of functions, so it should be possible to approximate it with a FNN. The universal approximation theorem requires an infinite number of neurons in a single layer though, which is not possible in practice. Despite this, FNNs often achieve good results with a limited number of neurons over a limited number of layers.

It is therefore worth investigating how well a feedforward neural network can solve the Schrödinger Equation, as has been done by others[4] [1].

This chapter will cover the inner workings of a FNN and how it is trained to better approximate target data from the target function. We will finish the chapter by delving into automatic differentiation, which will be used to calculate some derivatives of the network. From here on we will write "neural network", or "network" instead of "feedforward neural network", for brevity.

6.1 Network Structure

A neural network takes an input vector $\mathbf{a}^{(0)}$ and after many operations on this input returns an output vector $\mathbf{a}^{(L)}$. These operations are done sequentially, each one belonging to what we call a layer.

A layer in a neural network takes an input vector $\mathbf{a}^{(l-1)}$ and returns the output vector $\mathbf{a}^{(l)}$ which comes from the operation

$$\mathbf{a}^{(l)} = \sigma_l(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}), \tag{6.1}$$

where $\mathbf{W}^{(l)}$ is the weight matrix, $\mathbf{b}^{(l)}$ is the bias, and σ_l is the activation function for layer l. The activation function σ_l is a simple $\mathbb{R} \to \mathbb{R}$ function which acts element-wise on the elements of its input vector.

This means that each element of $\mathbf{a}^{(l)}$ is a weighted sum of the elements of $\mathbf{a}_{i-1}$, plus an element of the bias $\mathbf{b}^{(l)}$, all sent through the activation function. This is a very powerful operation. If $\mathbf{a}^{(l)}$ and $\mathbf{a}^{(l-1)}$ both contain 50 numbers, the 50 numbers of $\mathbf{a}^{(l)}$ are combined in 50 different ways, giving 50 potential useful interpretations of the data. The activation function can then dampen, amplify or transform the results in a non-linear way, introducing even more flexibility in what can be modelled.

The entire neural network of L layers can be described by the set of L operations

$$\begin{aligned}
\mathbf{a}^{(1)} &= \sigma_1(\mathbf{W}^{(1)}\mathbf{a}^{(0)} + \mathbf{b}^{(1)}), \\
\mathbf{a}^{(2)} &= \sigma_2(\mathbf{W}_2\mathbf{a}^{(1)} + \mathbf{b}^{(2)}), \\
&\dots \\
\mathbf{a}^{(L)} &= \sigma_L(\mathbf{W}^{(L)}\mathbf{a}^{(L-1)} + \mathbf{b}^{(L)}).
\end{aligned} \tag{6.2}$$

The flexibility offered by a single layer is only increased when chaining them together like this. The intermediate operations make $\mathbf{a}_i$ a more useful representation than $\mathbf{a}_{i-1}$, so that finally, the final operation can leverage the information in $\mathbf{a}_{L-1}$ to produce the desired result.

Often, the only numbers that make any intuitive sense in this process is the input and the output. Consider a network which takes a picture of an animal as input(a vector of pixel values), and then returns a "cat"-score. Naturally, we want the network to give pictures of cats a high cat score and pictures of other animals a low cat score. The input of the network has a simple interpretation, it is the pixel values of a picture of an animal. The output of the network also has a simple interpretation: A high, medium or low score means the network has high, middling or low confidence the picture is of a cat. But what about the other numbers in the operation? What do the weights $\mathbf{W}^{(l)}$ and biases $\mathbf{b}^{(l)}$ say about what the network knows about cats? What can an intermediate value $\mathbf{a}^{(l)}$ tell us about anything?

These questions belong to the field of model explainability, and sometimes we can get answers [2], maybe we can for instance single out a part of the network which is sensitive to the appearance of whiskers. But sometimes, the inner workings of a neural network are a mystery. If it gives the right answer we don't know how, and if it gives the wrong answer we don't know why. The use of heuristics and trial and error are therefore common when choosing the number of layers and the shape of their weight matrices.

6.2 Activation Functions

Activation function in a FNN serve multiple purposes. They can introduce non-linearity to the model as is the case of the ReLU activation function:

$$\mathrm{ReLU}(x) = \max(0, x). \tag{6.3}$$

They can also make model results less volatile by dampening large values when they appear as is the case of the Sigmoid and $\tanh$ activation functions (which are also non-linear)

$$\begin{aligned} \mathrm{Sigmoid}(x) &= \frac{1}{1 + e^{-x}}, \\ \tanh(x) &= \frac{e^{2x} - 1}{e^{2x} + 1}. \end{aligned} \tag{6.4}$$

These are commonly used for these features, as well as for their simple derivatives. ReLU especially is very fast to compute, and is therefore often used in very large networks to gain non-linearity at a low cost. The plots of these three functions is shown in figure 6.1.

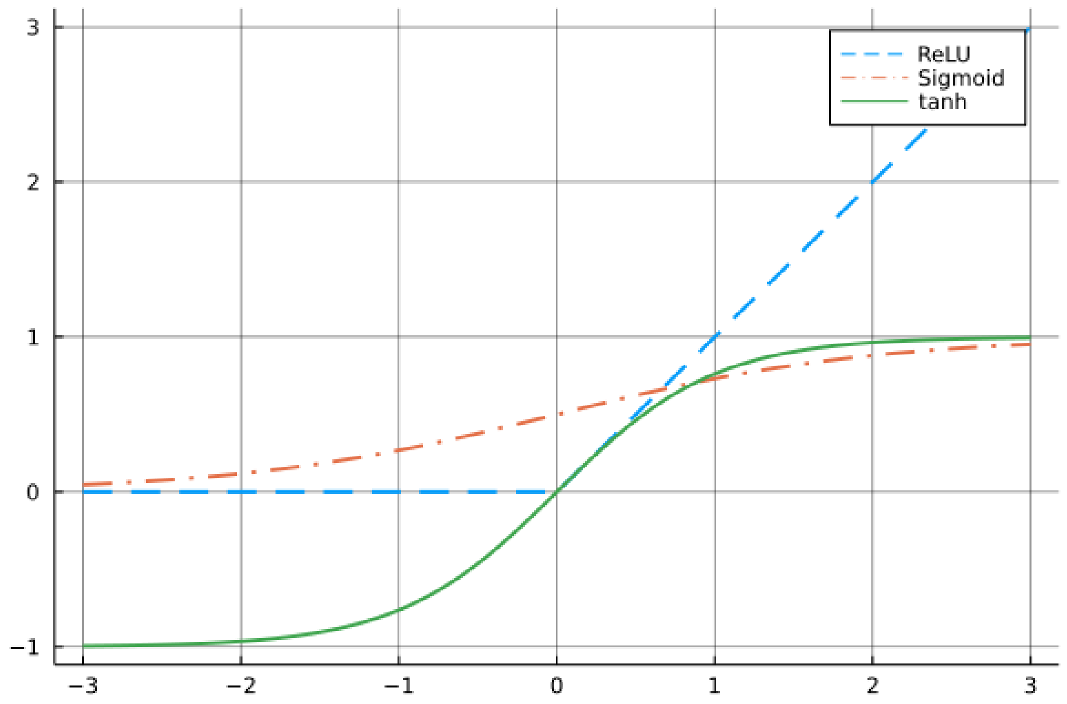

Figure 6.1: The ReLU, Sigmoid and tanh activation functions. Notice the ReLUs sharp angle at 0, where negative values are all set to zero. The Sigmoid function has an upper bound at 1 and lower bound at 0, which it approaches asymptotically. tanh is very similar, but has a lower bound at -1, and is much steeper around 0.

For our final layer, which produces the wavefunction output, we employ the exponential function as activation, inspired by [5]. This function will hopefully amplify the network output and make the output more sensitive to subtle differences in input. We will also try other activation functions, to see what works best.

6.3 Optimization

Now that we have shown how the input to a network is turned into the output, let's look at how we change the network so that it gives the output we want.

6.3.1 Stochastic Gradient Descent

Stochastic gradient descent (SGD) is a method of minimizing a function by calculating its gradient using a small random subset of data (even though the actual gradient requires the full data set), and updating the function parameters by the negative gradient. The following section uses a simple example model to show how and why SGD is used.

Consider the linear function

$$y(x, \alpha, \beta) = \alpha x + \beta. \tag{6.5}$$

We can use it to model something we think has a linear nature, for instance the height (model input x) and weight (model output y) of an adult.

To use our model, we need to specify our parameters. We set them to some initial guess, $\alpha = 0.8$ and $\beta = -80$. We demonstrate how the model can be used with some example data: From a data set of 1 million people, we randomly pick a person with a height of 170cm and a weight of 60kg. The model predicts the weight to be $\alpha x + \beta = 0.8 * 170 - 80 = 56$kg.

To judge how well our linear function models the target data, we choose a cost function: the squared distance

$$C(y, t) = \frac{1}{2}(y - t)^2, \tag{6.6}$$

where y is the model prediction, and t is the true value. A low cost is good, and a high cost is bad.

The cost in the example is then $\frac{1}{2}(y-t)^2 = \frac{1}{2}(56-60)^2 = 8$.

If we want to minimize the cost, we need to change y, and to change y we need to change the parameters of our model, in this case α and β. With the input data set, we write the cost as a function of the model parameters: $C(\alpha, \beta)$. This function has two inputs, and a single number, the cost, as an output, which means we can visualize it as a 3D height map like the one shown in figure 6.2.

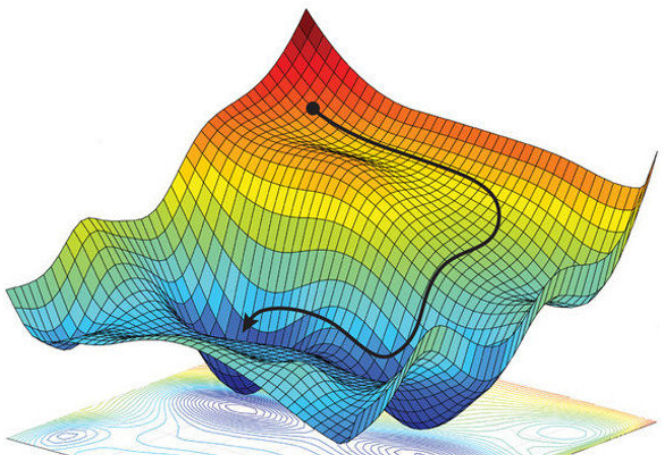

Figure 6.2: TODO

The lowest point on this graph is where the cost is the lowest. The parameters that correspond to this lowest point can be found by evaluating the gradient of the cost function.

First, some intuition: Say you were standing on the surface shown in figure 6.2 blindfolded, at some random point. If you wanted to get to the lowest point, a good strategy would be to move in the downhill direction until you reached some bottom point. Simply feeling your way downwards. Although you could get stuck in a pit which is not at the very lowest point, it's still much better than just moving around randomly. The mathematical way of finding the lowest point of a function is the same. You take the gradient to find which direction the value is decreasing most, then you shift your parameters in that direction. By repeating this operation, you are bound to reach some bottom point (if there is one). The very bottom is called the global minimum, while other "pits in the terrain" are called local minima.

While this procedure will reduce the cost for this single prediction, it might make the model worse for future predictions. We don't want to optimize the model for this single person, we want to optimize it for the entire data set. An alternative is to compute the gradient of our model for every single person in our data set, and use the average gradient to minimize the cost. This is called gradient descent. However, this is computationally expensive, and we can easily get stuck at a local minimum.

Stochastic gradient descent attempts to remedy these problems. While gradient descent takes the gradient of $C(\alpha, \beta)$ with complete data and uses it to minimize the cost, SGD uses the gradient of $C(\alpha, \beta)$ with small set of randomly chosen data. This has two advantages: First, we don't need to compute the gradient with the entire data set, which is computationally expensive. Second, we get a gradient which moves our parameters in mostly the right direction, leading us slowly towards the bottom with randomness that hopefully lets us escape local minima.

Knowing this, we optimize our model with a single step of SGD using our randomly chosen data point. The gradient of our model using this data is found by

$$\begin{aligned}
\frac{\partial C}{\partial y} &= y - t, \\
\frac{\partial C}{\partial \alpha} &= \frac{\partial y}{\partial \alpha}\frac{\partial C}{\partial y} = \alpha * (y - t) = 0.8 * (-4) = -3.2, \\
\frac{\partial C}{\partial \beta} &= \frac{\partial y}{\partial \beta}\frac{\partial C}{\partial y} = 1 * (y - t) = -4.
\end{aligned} \tag{6.7}$$

This suggests that both α and β should be higher (negative gradient gives the direction of decrease in cost). Using SGD, we change our parameters by some small amount proportional to the negative gradient. This proportionality is called the learning rate, and its optimal value depends on the problem and model at hand. After updating our parameters, we choose some new data, compute the gradient, and update our parameters once more. When the gradient gets very small or we have reached our desired number of steps, the SGD process is complete, and we have our final model.

6.3.2 Backpropogation

As the previous section has shown, to optimize our model, we need its gradient. In the previous section this was easy, we only had to chain together a couple derivatives, and we had the gradient. But in an FNN there are many more parameters, and many more operations, which means we need a more systematic way to compute the gradient. This section will show such a way, an algorithm called backpropogation.

First we define a new term $\mathbf{z}^{(l)}$, an intermediate term in the evaluation of a layer

$$\begin{aligned} \mathbf{a}^{(l)} &= \sigma_i(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}) = \sigma_i(\mathbf{z}^{(l)}), \\ \Rightarrow \mathbf{z}^{(l)} &= \mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}. \end{aligned} \tag{6.8}$$

Then we define the cost function we want to take the derivative with respect to, we call it C. We now take the derivatives of this cost function with respect to the different layer parameters. We use the chain rule and our new term $\mathbf{z}^{(l)}$, as this will prove useful.

The following expressions include derivatives with vectors, so we need to take extra care with the shapes of all terms. We choose here to use the unnatural denominator layout, meaning that the first dimension has the length of the denominator and the second dimension has the length of the numerator (the transpose of the natural way). Additionaly, the terms coming from the chain rule are in reverse order (since they have all been transposed and $(AB)^T = B^T A^T$). These are the main results we are after:

$$\begin{aligned} \frac{\partial C}{\partial \mathbf{W}_{ij}^{(l)}} &= \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{W}_{ij}^{(l)}} \frac{\partial C}{\partial \mathbf{z}^{(l)}}, \\ \frac{\partial C}{\partial \mathbf{b}^{(l)}} &= \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{b}^{(l)}} \frac{\partial C}{\partial \mathbf{z}^{(l)}}. \end{aligned} \tag{6.9}$$

We start with simplifying the leftmost factors in the expressions. Following the definition of $\mathbf{z}^{(l)}$.

$$\begin{aligned} \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{b}_i^{(l)}} &= [0, ..., 0, \overset{i}{1}, 0, ..., 0], \\ \Rightarrow \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{b}^{(l)}} &= I, \end{aligned} \tag{6.10}$$

where the floating i indicates the i'th index in the row vector. The derivative of $\mathbf{z}^{(l)}$ wrt. $\mathbf{W}^{(l)}$ is a 3-dimentional tensor, so we index into it to make working with it more manageable. Its elements are

$$\begin{aligned} \frac{\partial \mathbf{z}_k^{(l)}}{\partial \mathbf{W}_{ij}^{(l)}} &= \delta_{ki} \mathbf{a}_j^{(l-1)}, \\ \Rightarrow \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{W}_{ij}^{(l)}} &= [0, ..., 0, \overset{i}{\mathbf{a}_j^{(l-1)}}, 0, ..., 0], \end{aligned} \tag{6.11}$$

where δ_{ki} is the kronecker delta function which equals 1 for $k = i$ and 0 for $k \neq i$.

Recall from equation 6.9 that we take the vector product

$$\frac{\partial C}{\partial \mathbf{W}_{ij}^{(l)}} = \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{W}_{ij}^{(l)}} \frac{\partial C}{\partial \mathbf{z}^{(l)}}. \tag{6.12}$$

Notice that the first vector is a row vector, while the second is a column vector. This means that it is an inner product, and since the first vector only has one non-zero term, as we just showed in equation 6.11, we can make some large simplifications. We now know that $\frac{\partial C}{\partial \mathbf{W}^{(l)}}$ is a matrix where an element in row i and column j is the product of the i'th element of $\frac{\partial C}{\partial \mathbf{z}^{(l)}}$ and $\mathbf{a}_j^{(l-1)}$. This can be written compactly as the outer product

$$\frac{\partial C}{\partial \mathbf{W}^{(l)}} = \frac{\partial C}{\partial \mathbf{z}^{(l)}} (\mathbf{a}^{(l-1)})^T \tag{6.13}$$

The rightmost factors in equation 6.9 are the same for both derivatives, but require some more work.

$$\begin{aligned} \frac{\partial C}{\partial z_i} &= \frac{\partial C}{\partial z_{i+1}} \frac{\partial z_{i+1}}{\partial a_i} \frac{\partial a_i}{\partial z_i} \\ &= \frac{\partial C}{\partial z_{i+1}} W_{i+1} \frac{\partial \sigma_i(z_i)}{\partial z_i}. \end{aligned} \tag{6.14}$$

We again used the definition of $\mathbf{z}_i$ to simplify a partial derivative, and we used the definition of $\mathbf{a}_i$ to replace its derivative with the derivative of $\sigma_i(z_i)$, which can be calculated directly.

The only term left which is not directly known is $\frac{\partial C}{\partial z_{i+1}}$. The keen reader will notice however, that equation 6.14 does in face give us a direct expression for this term, only for all $i \neq L$. For $L = i$ we get

$$\frac{\partial C}{\partial z_L} = \frac{\partial C}{\partial a_L}\frac{\partial a_L}{\partial z_L} = \frac{\partial C}{\partial a_L}\frac{\partial \sigma_L(z_L)}{\partial z_L}. \tag{6.15}$$

where both terms in the final expression can be calculated directly, as a_L is the direct input to C and $\mathbf{z}_L$ is the direct input to σ_i.

Using these equations we can calculate all weight and bias derivatives using the recursive algorithm called backpropogation:

1. Calculate all a_i and z_i in the network (Forward pass)
2. Now, at layer L: Calculate $\frac{\partial C}{\partial z_L}$, use it to calculate $\frac{\partial C}{\partial W_L}$ and $\frac{\partial C}{\partial b_L}$
3. For each layer i from layer $L - 1$ to the first layer:
   (a) Use $\frac{\partial C}{\partial z_{i+1}}$ to calculate $\frac{\partial C}{\partial z_i}$
   (b) Use $\frac{\partial C}{\partial z_i}$ to calculate $\frac{\partial C}{\partial W_i}$ and $\frac{\partial C}{\partial b_i}$

6.3.3 Automatic Differentiation

6.4 Neural Network Wave Function

Chapter 7

Restricted Boltzmann Machines

Part II

Implementation

Chapter 8

The Julia Programming Language

Chapter 9

Feedforward Neural Networks

Our neural network wave function code needs to support four main functions:

- Evaluation of the network
- The derivative of the network wrt. its parameters (backpropogation)
- The derivative of the network wrt. its input (quantum force)
- The double derivative of the network wrt. its input (kinetic energy)

To write well optimized code for all of these functions, we have chosen to implement our own neural network code from the ground up. The following sections describe the design of this code, and the unique approaches used to solve each of the four problems. The final section is a note on why other machine learning libraries were not used instead.

9.1 Fast evaluation

The evaluation of a neural network is very straightforward. As shown in equation 6.2, it is a series of matrix-vector multiplication, vector-vector addition, and element-wise evaluation of activation functions.

All of these operations have fast, in-place implementations in Julia. To make use of this, we preallocate all the memory required when the network is created. Every layer in the network has a vector where it will put its output, and every layer has a reference to the previous layer's output. This means that for the millions of times the network is evaluated, no memory needs to be allocated and garbage-collected, resulting in a speed-up.

The function for evaluating a the output of a linear layer is as follows:

```
function in_place_eval!(layer::Dense)
    (; W, b, input, output) = layer

    mul!(output, W, input)
    output .+= b
end
```

The first line in this function extracts the weight matrix and the bias-, input- and output-vectors of the layer for later use.

The function `mul!()` computes the matrix-vector product of **W** and **input** and places the result in **output** during the calculation. The special Julia syntax `.+=` adds each element of **b** to the corresponding element of **output**, again without any intermediary memory allocation.

Compare this with the similar function `layerEval()`.

```
function simple_eval(input, layer::Dense)
    (; W, b) = layer

    return W * input + b
end
```

This function takes the input to the layer as an input, and performs the matrix-vector multiplication and vector-vector addition with no preallocated output vector used. A benchmark of these two functions is shown in table 9.1

	Time	Allocations
in-place 10x10	117.737 ns	0 allocations: 0 bytes
simple 10x10	194.929 ns	2 allocations: 288 bytes
in-place 100x100	58.800 µs	0 allocations: 0 bytes
simple 100x100	59.500 µs	2 allocations: 1.75 KiB

Table 9.1: In-place and simple layer evaluations benchmarked. The size of the weight matrix is shown for each benchmark. We see that the in-place operation allocates less memory, and is faster, especially for small matrices.

We see that in-place evaluation is faster, and allocates no memory. This effect will be amplified as we make heavy use of small weight matrices, and will be evaluating the network many times during the VMC process. Allocations tend to add up which leads to costly garbage collection.

The activation functions are treated as their own layers and perform similar in-place operations using their input and output vectors. To illustrate, here is the in-place evaluation of the tanh activation function

```
function in_place_eval!(layer::Tanh)
    (; input, output) = layer
    output .= tanh.(input)
end
```

We also have functions for evaluating activation functions without the use of in-place operations. We will come back to the use of these in section 9.4.

9.2 Backpropogation

We compute the derivative of our network parameters using analytical formulas, in contrast to most popular machine learning libraries which use automatic differentiation.

Python packages such as Pytorch and TensorFlow, or Julia packages such as Flux, use automatic differentiation, also known as autodiff, to compute gradients. The use of autodiff is a necessity when working with complicated models such as the ones offered by these libraries. Autodiff gives reliable gradients with very little work required, at a low cost to speed.

When working with only a feedforward neutral network however, the derivatives are simple enough to quickly be implemented by hand. Not having to rely on autodiff makes the computation faster, and it gives more flexibility in how to store the parameters. The autodiff libraries ForwardDiff and ReverseDiff, for instance, requires the parameters to be stored in a single vector, while the autodiff library Zygote is flexible, but often slow.

9.3 Quantum force

Recall from equation TODO that the quantum force (TODO drift force?) of our wavefunction is given by

$$F(\mathbf{r_i}) = \frac{2}{\Psi(\mathbf{r_i})} \nabla \Psi(\mathbf{r_i}). \tag{9.1}$$

This means that we need to compute the derivative of our wavefunction wrt. its inputs $\mathbf{r_i}$.

9.4 Kinetic energy - Forward over Reverse automatic differentiation

In order to avoid computing this derivative by hand, we used automatic differentiation.

Automatic differentiation is a way of computing the derivative of a programmatic function. All functions written in code are built up by simple operations such as addition, multiplication, exponentiation, sin, etc. . By breaking functions into these component parts, a program can use the chain rule to compute the derivative of the entire function.

9.5 Other libraries

Bibliography

[1] Corey Adams et al. "Variational Monte Carlo Calculations of $A \leqslant 4$ Nuclei with an Artificial Neural-Network Correlator Ansatz". In: *Physical Review Letters* 127.2 (July 2021). DOI: 10.1103/physrevlett.127.022502. URL: https://doi.org/10.1103%2Fphysrevlett.127.022502.

[2] Raphaël Feraud and Fabrice Clerot. "A methodology to explain neural network classification". In: *Neural networks : the official journal of the International Neural Network Society* 15 (Apr. 2002), pp. 237–46. DOI: 10.1016/S0893-6080(01)00127-7.

[3] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208.

[4] J.W.T. Keeble and A. Rios. "Machine learning the deuteron". In: *Physics Letters B* 809 (Oct. 2020), p. 135743. DOI: 10.1016/j.physletb.2020.135743. URL: https://doi.org/10.1016%2Fj.physletb.2020.135743.

[5] Hiroki Saito. "Method to Solve Quantum Few-Body Problems with Artificial Neural Networks". In: *Journal of the Physical Society of Japan* 87.7 (2018), p. 074002. DOI: 10.7566/JPSJ.87.074002. eprint: https://doi.org/10.7566/JPSJ.87.074002. URL: https://doi.org/10.7566/JPSJ.87.074002.