January 29-February 2 : Advanced machine learning and data analysis for the physical sciences

Morten Hjorth-Jensen[1,2]

Department of Physics and Center for Computing in Science Education, University of Oslo, Norway[1]

Department of Physics and Astronomy and Facility for Rare Isotope Beams, Michigan State University, East Lansing, Michigan, USA[2]

January 30

Overview of third week

1. Discussion of possible projects
2. Review of neural networks and automatic differentiation
3. Discussion of codes

Mathematics of deep learning

Two recent books online

1. The Modern Mathematics of Deep Learning, by Julius Berner, Philipp Grohs, Gitta Kutyniok, Philipp Petersen, published as Mathematical Aspects of Deep Learning, pp. 1-111. Cambridge University Press, 2022
2. Mathematical Introduction to Deep Learning: Methods, Implementations, and Theory, Arnulf Jentzen, Benno Kuckuck, Philippe von Wurstemberger

Reminder on books with hands-on material and codes

- Sebastian Rashcka et al, Machine learning with Sickit-Learn and PyTorch
- David Foster, Generative Deep Learning with TensorFlow
- Bali and Gavras, Generative AI with Python and TensorFlow 2

All three books have GitHub addresses from where one can download all codes. We will borrow most of the material from these three texts as well as from Goodfellow, Bengio and Courville's text Deep Learning

Reading recommendations

1. Rashkca et al., chapter 11, jupyter-notebook sent separately, from GitHub
2. Goodfellow et al, chapter 6 and 7 contain most of the neural network background.

Mathematics of deep learning and neural networks

Neural networks, in its so-called feed-forward form, where each iterations contains a feed-forward stage and a back-propgagation stage, consist of series of affine matrix-matrix and matrix-vector multiplications. The unknown parameters (the so-called biases and weights which deternine the architecture of a neural network), are uptaded iteratively using the so-called back-propagation algorithm. This algorithm corresponds to the so-called reverse mode of automatic differentation.

Basics of an NN

A neural network consists of a series of hidden layers, in addition to the input and output layers. Each layer l has a set of parameters $\boldsymbol{\Theta}^{(l)} = (\boldsymbol{W}^{(l)}, \boldsymbol{b}^{(l)})$ which are related to the parameters in other layers through a series of affine transformations, for a standard NN these are matrix-matrix and matrix-vector multiplications. For all layers we will simply use a collective variable $\boldsymbol{\Theta}$.
It consist of two basic steps:

1. a feed forward stage which takes a given input and produces a final output which is compared with the target values through our cost/loss function.
2. a back-propagation state where the unknown parameters $\boldsymbol{\Theta}$ are updated through the optimization of the their gradients. The expressions for the gradients are obtained via the chain rule, starting from the derivative of the cost/function.

These two steps make up one iteration. This iterative process is continued till we reach an eventual stopping criterion.

Overarching view of a neural network

The architecture of a neural network defines our model. This model aims at describing some function $f(\boldsymbol{x}$ which represents some final result (outputs or tagrget values) given a specific inpput $\boldsymbol{x}$. Note that here $\boldsymbol{y}$ and $\boldsymbol{x}$ are not limited to be vectors.

The architecture consists of

1. An input and an output layer where the input layer is defined by the inputs $\boldsymbol{x}$. The output layer produces the model ouput $\tilde{\boldsymbol{y}}$ which is compared with the target value $\boldsymbol{y}$
2. A given number of hidden layers and neurons/nodes/units for each layer (this may vary)
3. A given activation function $\sigma(\boldsymbol{z})$ with arguments $\boldsymbol{z}$ to be defined below. The activation functions may differ from layer to layer.
4. The last layer, normally called **output** layer has normally an activation function tailored to the specific problem
5. Finally we define a so-called cost or loss function which is used to gauge the quality of our model.

The optimization problem

The cost function is a function of the unknown parameters $\boldsymbol{\Theta}$ where the latter is a container for all possible parameters needed to define a neural network
If we are dealing with a regression task a typical cost/loss function is the mean squared error

$$C(\boldsymbol{\Theta}) = \frac{1}{n}\left\{\left(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta}\right)^T\left(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta}\right)\right\}.$$

This function represents one of many possible ways to define the so-called cost function. Note that here we have assumed a linear dependence in terms of the paramters $\boldsymbol{\Theta}$. This is in general not the case.

Parameters of neural networks

For neural networks the parameters $\boldsymbol{\Theta}$ are given by the so-called weights and biases (to be defined below).

The weights are given by matrix elements $w_{ij}^{(l)}$ where the superscript indicates the layer number. The biases are typically given by vector elements representing each single node of a given layer, that is $b_j^{(l)}$.

Other ingredients of a neural network

Having defined the architecture of a neural network, the optimization of the cost function with respect to the parameters $\boldsymbol{\Theta}$, involves the calculations of gradients and their optimization. The gradients represent the derivatives of a multidimensional object and are often approximated by various gradient methods, including

1. various quasi-Newton methods,
2. plain gradient descent (GD) with a constant learning rate η,
3. GD with momentum and other approximations to the learning rates such as
    - Adapative gradient (ADAgrad)
    - Root mean-square propagation (RMSprop)
    - Adaptive gradient with momentum (ADAM) and many other
4. Stochastic gradient descent and various families of learning rate approximations

Other parameters

In addition to the above, there are often additional hyperparamaters which are included in the setup of a neural network. These will be discussed below.

Universal approximation theorem

The universal approximation theorem plays a central role in deep learning. Cybenko (1989) showed the following:

Let σ be any continuous sigmoidal function such that

$$\sigma(z) = \begin{cases} 1 & z \to \infty \\ 0 & z \to -\infty \end{cases}$$

Given a continuous and deterministic function $F(\boldsymbol{x})$ on the unit cube in d-dimensions $F \in [0,1]^d$, $x \in [0,1]^d$ and a parameter $\epsilon > 0$, there is a one-layer (hidden) neural network $f(\boldsymbol{x}; \boldsymbol{\Theta})$ with $\boldsymbol{\Theta} = (\boldsymbol{W}, \boldsymbol{b})$ and $\boldsymbol{W} \in \mathbb{R}^{m \times n}$ and $\boldsymbol{b} \in \mathbb{R}^n$, for which

$$|F(\boldsymbol{x}) - f(\boldsymbol{x}; \boldsymbol{\Theta})| < \epsilon \, \forall \boldsymbol{x} \in [0,1]^d.$$

Some parallels from real analysis

For those of you familiar with for example the Stone-Weierstrass theorem for polynomial approximations or the convergence criterion for Fourier series, there are similarities in the derivation of the proof for neural networks.

The approximation theorem in words

Any continuous function $y = F(\boldsymbol{x})$ supported on the unit cube in d-dimensions can be approximated by a one-layer sigmoidal network to arbitrary accuracy.
Hornik (1991) extended the theorem by letting any non-constant, bounded activation function to be included using that the expectation value

$$\mathbb{E}[|F(\boldsymbol{x})|^2] = \int_{\boldsymbol{x}\in D} |F(\boldsymbol{x})|^2 p(\boldsymbol{x}) d\boldsymbol{x} < \infty.$$

Then we have

$$\mathbb{E}[|F(\boldsymbol{x}) - f(\boldsymbol{x};\boldsymbol{\Theta})|^2] = \int_{\boldsymbol{x}\in D} |F(\boldsymbol{x}) - f(\boldsymbol{x};\boldsymbol{\Theta})|^2 p(\boldsymbol{x}) d\boldsymbol{x} < \epsilon.$$

More on the general approximation theorem

None of the proofs give any insight into the relation between the number of of hidden layers and nodes and the approximation error ϵ, nor the magnitudes of $\boldsymbol{W}$ and $\boldsymbol{b}$.
Neural networks (NNs) have what we may call a kind of universality no matter what function we want to compute.

It does not mean that an NN can be used to exactly compute any function. Rather, we get an approximation that is as good as we want.

Class of functions we can approximate

The class of functions that can be approximated are the continuous ones. If the function $F(\boldsymbol{x})$ is discontinuous, it won't in general be possible to approximate it. However, an NN may still give an approximation even if we fail in some points.

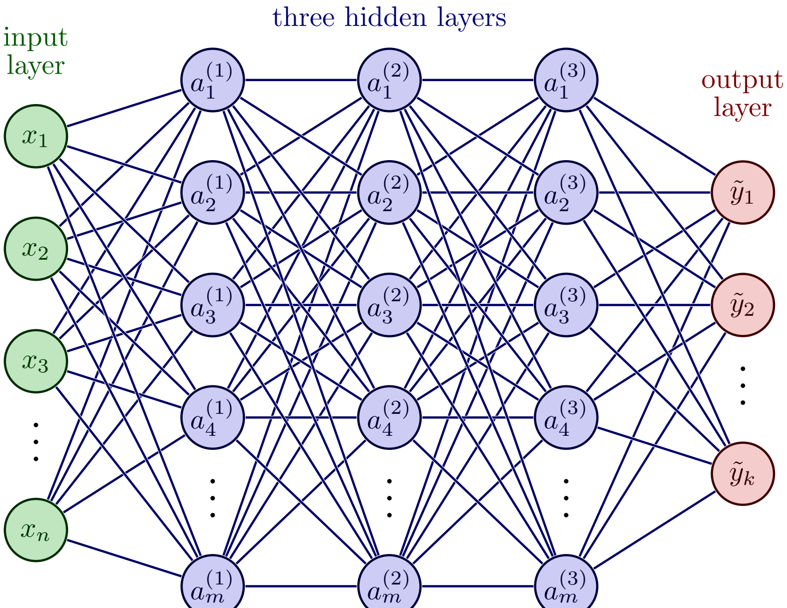
Setting up the equations for a neural network

The questions we want to ask are how do changes in the biases and the weights in our network change the cost function and how can we use the final output to modify the weights and biases?
To derive these equations let us start with a plain regression problem and define our cost function as

$$\mathcal{C}(\boldsymbol{\Theta}) = \frac{1}{2}\sum_{i=1}^{n}\left(y_i - \tilde{y}_i\right)^2,$$

where the y_is are our n targets (the values we want to reproduce), while the outputs of the network after having propagated all inputs $\boldsymbol{x}$ are given by $\boldsymbol{\tilde{y}}_i$.

Layout of a neural network with three hidden layers



Definitions

With our definition of the targets $\boldsymbol{y}$, the outputs of the network $\tilde{\boldsymbol{y}}$ and the inputs $\boldsymbol{x}$ we define now the activation z_j^l of node/neuron/unit j of the l-th layer as a function of the bias, the weights which add up from the previous layer $l-1$ and the forward passes/outputs $\hat{a}^{l-1}$ from the previous layer as

$$z_j^l = \sum_{i=1}^{M_{l-1}} w_{ij}^l a_i^{l-1} + b_j^l,$$

where b_k^l are the biases from layer l. Here M_{l-1} represents the total number of nodes/neurons/units of layer $l-1$. The figure in the whiteboard notes illustrates this equation. We can rewrite this in a more compact form as the matrix-vector products we discussed earlier,

$$\hat{z}^l = \left(\hat{W}^l\right)^T \hat{a}^{l-1} + \hat{b}^l.$$

Inputs to the activation function

With the activation values $\boldsymbol{z}^l$ we can in turn define the output of layer l as $\boldsymbol{a}^l = f(\boldsymbol{z}^l)$ where f is our activation function. In the examples here we will use the sigmoid function discussed in our logistic regression lectures. We will also use the same activation function f for all layers and their nodes. It means we have

$$a_j^l = \sigma(z_j^l) = \frac{1}{1+\exp -(z_j^l)}.$$

Derivatives and the chain rule

From the definition of the activation z_j^l we have

$$\frac{\partial z_j^l}{\partial w_{ij}^l} = a_i^{l-1},$$

and

$$\frac{\partial z_j^l}{\partial a_i^{l-1}} = w_{ji}^l.$$

With our definition of the activation function we have that (note that this function depends only on z_j^l)

$$\frac{\partial a_j^l}{\partial z_j^l} = a_j^l(1-a_j^l) = \sigma(z_j^l)(1-\sigma(z_j^l)).$$

Derivative of the cost function

With these definitions we can now compute the derivative of the cost function in terms of the weights.

Let us specialize to the output layer $l = L$. Our cost function is

$$\mathcal{C}(\boldsymbol{\Theta}^L) = \frac{1}{2}\sum_{i=1}^{n}\left(y_i - \tilde{y}_i\right)^2 = \frac{1}{2}\sum_{i=1}^{n}\left(a_i^L - y_i\right)^2,$$

The derivative of this function with respect to the weights is

$$\frac{\partial \mathcal{C}(\boldsymbol{\Theta}^L)}{\partial w_{jk}^L} = \left(a_j^L - y_j\right)\frac{\partial a_j^L}{\partial w_{jk}^L},$$

The last partial derivative can easily be computed and reads (by applying the chain rule)

$$\frac{\partial a_j^L}{\partial w_{jk}^L} = \frac{\partial a_j^L}{\partial z_j^L}\frac{\partial z_j^L}{\partial w_{jk}^L} = a_j^L(1 - a_j^L)a_k^{L-1}.$$

Simpler examples first, and automatic differentiation

In order to understand the back propagation algorithm and its derivation (an implementation of the chain rule), let us first digress with some simple examples. These examples are also meant to motivate the link with back propagation and automatic differentiation.

Reminder on the chain rule and gradients

If we have a multivariate function $f(x, y)$ where $x = x(t)$ and $y = y(t)$ are functions of a variable t, we have that the gradient of f with respect to t (without the explicit unit vector components)

$$\frac{df}{dt} = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \end{bmatrix} \begin{bmatrix} \frac{\partial x}{\partial t} \\ \frac{\partial y}{\partial t} \end{bmatrix} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial t}.$$

Multivariable functions

If we have a multivariate function $f(x, y)$ where $x = x(t, s)$ and $y = y(t, s)$ are functions of the variables t and s, we have that the partial derivatives

$$\frac{\partial f}{\partial s} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial s} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial s},$$

and

$$\frac{\partial f}{\partial t} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial t}.$$

the gradient of f with respect to t and s (without the explicit unit vector components)

$$\frac{df}{d(s, t)} = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \end{bmatrix} \begin{bmatrix} \frac{\partial x}{\partial s} & \frac{\partial x}{\partial t} \\ \frac{\partial y}{\partial s} & \frac{\partial y}{\partial t} \end{bmatrix}.$$

Automatic differentiation through examples

A great introduction to automatic differentiation is given by Baydin et al., see `https://arxiv.org/abs/1502.05767`.
Automatic differentiation is a represented by a repeated application of the chain rule on well-known functions and allows for the calculation of derivatives to numerical precision. It is not the same as the calculation of symbolic derivatives via for example SymPy, nor does it use approximative formulae based on Taylor-expansions of a function around a given value. The latter are error prone due to truncation errors and values of the step size Δ.

Simple example

Our first example is rather simple,

$$f(x) = \exp x^2,$$

with derivative

$$f'(x) = 2x \exp x^2.$$

We can use SymPy to extract the pertinent lines of Python code through the following simple example

```
from __future__ import division
from sympy import *
x = symbols('x')
expr = exp(x*x)
simplify(expr)
derivative = diff(expr,x)
print(python(expr))
print(python(derivative))
```

Smarter way of evaluating the above function

If we study this function, we note that we can reduce the number of operations by introducing an intermediate variable

$$a = x^2,$$

leading to

$$f(x) = f(a(x)) = b = \exp a.$$

We now assume that all operations can be counted in terms of equal floating point operations. This means that in order to calculate $f(x)$ we need first to square x and then compute the exponential. We have thus two floating point operations only.

Reducing the number of operations

With the introduction of a precalculated quantity a and thereby $f(x)$ we have that the derivative can be written as

$$f'(x) = 2xb,$$

which reduces the number of operations from four in the orginal expression to two. This means that if we need to compute $f(x)$ and its derivative (a common task in optimizations), we have reduced the number of operations from six to four in total.
Note that the usage of a symbolic software like SymPy does not include such simplifications and the calculations of the function and the derivatives yield in general more floating point operations.

Chain rule, forward and reverse modes

In the above example we have introduced the variables a and b, and our function is

$$f(x) = f(a(x)) = b = \exp a,$$

with $a = x^2$. We can decompose the derivative of f with respect to x as

$$\frac{df}{dx} = \frac{df}{db}\frac{db}{da}\frac{da}{dx}.$$

We note that since $b = f(x)$ that

$$\frac{df}{db} = 1,$$

leading to

$$\frac{df}{dx} = \frac{db}{da}\frac{da}{dx} = 2x \exp x^2,$$

as before.

Forward and reverse modes

We have that

$$\frac{df}{dx} = \frac{df}{db}\frac{db}{da}\frac{da}{dx},$$

which we can rewrite either as

$$\frac{df}{dx} = \left[\frac{df}{db}\frac{db}{da}\right]\frac{da}{dx},$$

or

$$\frac{df}{dx} = \frac{df}{db}\left[\frac{db}{da}\frac{da}{dx}\right].$$

The first expression is called reverse mode (or back propagation) since we start by evaluating the derivatives at the end point and then propagate backwards. This is the standard way of evaluating derivatives (gradients) when optimizing the parameters of a neural network). In the context of deep learning this is computationally more efficient since the output of a neural network consists of either one or some few other output variables.
The second equation defines the so-called **forward mode**.

More complicated function

We increase our ambitions and introduce a slightly more complicated function

$$f(x) = \sqrt{x^2 + expx^2},$$

with derivative

$$f'(x) = \frac{x(1 + \exp x^2)}{\sqrt{x^2 + expx^2}}.$$

The corresponding SymPy code reads

```
from __future__ import division
from sympy import *
x = symbols('x')
expr = sqrt(x*x+exp(x*x))
simplify(expr)
derivative = diff(expr,x)
print(python(expr))
print(python(derivative))
```

Counting the number of floating point operations

A simple count of operations shows that we need five operations for the function itself and ten for the derivative. Fifteen operations in total if we wish to proceed with the above codes.
Can we reduce this to say half the number of operations?

Defining intermediate operations

We can indeed reduce the number of operation to half of those listed in the brute force approach above. We define the following quantities

$$a = x^2,$$

and

$$b = \exp x^2 = \exp a,$$

and

$$c = a + b,$$

and

$$d = f(x) = \sqrt{c}.$$