Analyzing protein sequences with Deep Neural Networks

Rasmus Porse Bjørneskov
Roald Frej Vitus Simonsen

June 7, 2019

Abstract

Her skriver vi på dansk.

Contents

1 Introduction

1.1 Motivation

Neurale netværk er rigtigt seje [1]

1.2 Proteins

1.2.1 What is

1.2.2 Secondary Structures

1.2.3 Solvent accessible surface area

1.3 Neural networks

The common perception is that humans and animals process information, i.e. transform perceptional stimuli and physiological conditions into behaviour, by using their brains. This is imprecise however, as the brain as such is only one functioning part of what is called the *nervous system*, which is in turn responsible for the internal workings of human and animal behaviour.

This nervous system is an abstaction over a number of *neurons* interconnected by synapses. Neurons in turn are so-called electrically exitable cells. In a gross over-simplification, this can be translated into the case that each neuron can have different internal states, depending on the internal states of the neurons it is connected to, thus forming a *neural network*.

While obviously interesting within the fields of biology or psychology, this structure has shown to be enourmously interesting in the field of computation, as one can in fact model this very thing and use it to make predictions based on prior observations, for example in regards to the aforementioned structures of proteins folding.

Analogously, or rather, digitally, we can use this model to construct an *artificial neural network*. These

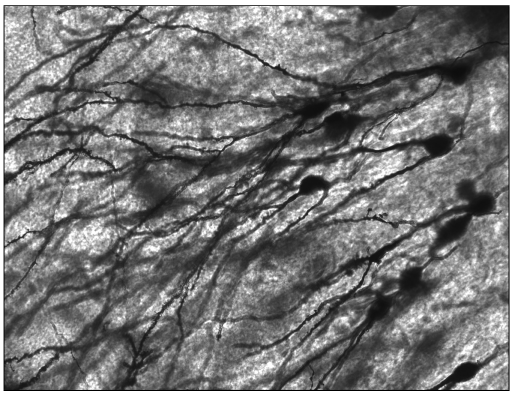

Figure 1: Neurons in the dentate gyrus of an epilepsy patient.

set themselves apart from biological neural networks in a couple of ways: most urgently in that while neurons in biological neural networks are connected to each other via synapses, so that each neuron has an either inhibitory or activating effect on whether or not a connected neuron 'fires', in artificial neural networks neurons are connected by *edges*, each with an associated *weight*, allowing the artificial networks to leave the discrete domain and enter the continuous.

1.3.1 Basics

An artificial neural network is a specific instantiation of a *multi-layer perceptron*. A multi-layer perceptron in turn is a system of nodes arranged into layers and each receiving their value from the value of the nodes in the layer before them (adjusted by some weight) in combination with a bias (a scalar) and an activation function of some sort.
In this way the first layer will be called the *input layer*, the last layer the *output layer* and all layers in between *hidden layers*.
In the case of neural networks, the nodes in the system are referred to as neurons. Thus, if one was to describe it more precisely, the value z_i of neuron layer i with d number of neurons in it, given the matrix of sets of weights w (the biases being the very first row) and the activation function $h()$ can be expressed as follows:

$$z_i = h\left(\sum_{j=0}^{d} w_{ij} \cdot z_{i-1} + w_{i0}\right)$$

Such a model is useful for calculating continuous values as well as for classification purposes. A common example of the latter is the MNIST data set of handwritten digits in 28x28 pixel greyscale images. In this case the light intensity of the 784 (28^2) pixels could fittingly be the input layer, while a layer consisting of 10 nodes (corresponding to digits 0 through 9) could be the output layer, such that the system could be used to predict which digit is written in a given image. A

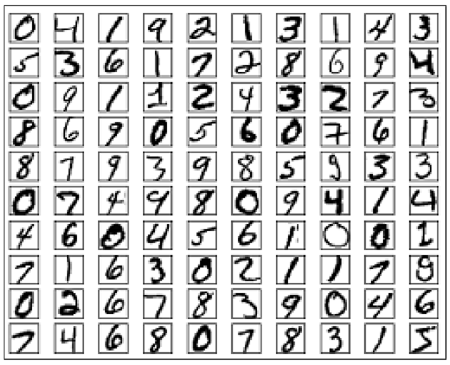

Figure 2: Handwritten digits in the MNIST data set.

common implementation of a neural network to perform this task is to have a single hidden layer of 800 nodes, however the most successful implementations in regards to the MNIST data set have been made with *convolutional neural networks*, but we shall return to these later.

1.3.2 Training

Simply having defined the system of the neural network as above (denoted f_{NN}) naturally does not give us a reliable method for classifying, as the accuracy of the prediction inherently depends on the correctness of the weights and biases of the system (denoted Θ), which must be initialized with random values.
Enter a rather nifty idea named *backpropagation*. Having a set of predicted values ($\hat{y}$) for a data set (x) along with the observed correct values (y), one can establish a *loss* by applying a loss function (that we shall elaborate on later), but which can be as straightforward as a root mean square error. We shall denote this loss function $l(\hat{y}, y)$, so that the total loss of the system when applied to data set x is $l(f_{NN}(\Theta, x), \hat{y})$.
Now backpropagation refers to the practice of essentially taking the derivative of the loss function in regards to the matrix of weights and biases, thereby calculating a *gradient* $\nabla l(f_{NN}(\Theta, x), \hat{y})$ of these values with respect to the accuracy of the system, and then adjusting the weights and biases by the gradient multiplied by a learning rate scalar.
In laymans terms, by doing this one gets a matrix of 'change your weights by this and that much in this and that direction to increase prediction accuracy'.

1.3.3 Convolutional neural networks

Before talking about convolutional neural networks, we have to specify what a convolution is. Again the MNIST data set serves as an obvious example.
One can understand each of the images in the data

set as a matrix of 26 × 26 values in the range of 0 - 255. The formal definition of a convolution is *TODO*, however the more practical explanation is that it is the resulting matrix of the product of a matrix and a filter. For example if one has the matrix a and the filter f:

$$a = \begin{bmatrix} 2 & 3 & 4 \\ 1 & 2 & 3 \\ 0 & 1 & 2 \end{bmatrix} \qquad f = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

the result of convolving a over f would be:

1.3.4 Multitask learning

1.4 Prior research in this field

Skrive noget om den artikel vi tager udgangspunkt i.

2 Methods

2.1 Dataset

2.1.1 Ophav

Hvor har vi dette datasæt fra, og hvem har lavet det?

2.1.2 Features

Skrive noget om hvad vores datasæt er og hvordan det er sat op. Dvs. hvilke egenskaber er der, hvordan er de encoded (one-hot vs. binært (solvent)) og hvorvidt vi bruger dem til noget. Fortæl også om at strukturerne her er encoded som de 8 substrukturer og ikke de 3 grupperinger af strukturer.

2.1.3 Opdeling i træning, test og validering

Fortæl om at vi har rigtigt meget træningsdata og hhv. 255 og 236 værdier i test og validering. Vi træner og backpropagerer over træningssættet, holder øje med valideringssættet (og træffer vores beslutninger ud fra det) og udsætter kun modellen for testsættet til sidst.

2.2 Tools

2.2.1 Model

Being that we are training a convolutional neural network, in particular two sets of functions are important, namely activation and loss functions.

The activation functions we opted to use in this project were the rectifier, softmax and sigmoid functions respectively, whereas we use the Cross Entropy loss in evaluating and training the network.

Rectifier Possibly the simplest of these activation functions is the rectifier, which when implemented in an artificial neural network is referred to as a *rectified linear unit* (ReLU), which, in laymans terms, simply does not allow negative values, and replaces such values with zero.

$$ReLU(x) = x^+ = max(0, x)$$

Sigmoid function Sigmoid functions are a specific variant of logistic functions, and serve to map values in arbitrary ranges to values within a specific range, so that the mapped values over the original values form a sigmoid curve. The value of applying sigmoid functions to outputs from a neural network is that they then enable the model to map its output of arbitrarily big or small values to a probability (i.e. a value $0 \leq x \leq 1$). In the present case, this becomes relevant when predicting relative and absolute solvent accessibilities.

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

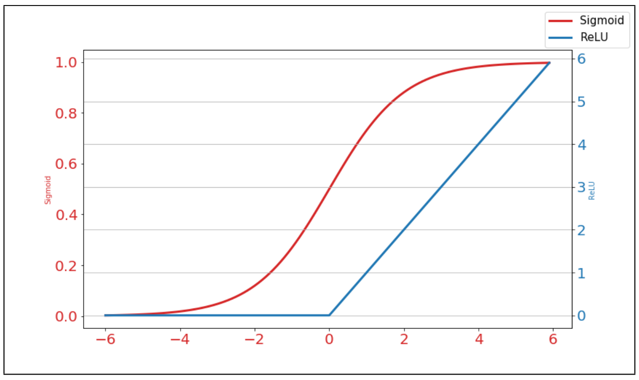

Figure 3: The sigmoid and rectifier activation functions

Softmax The softmax function (σ) does very much the same as the sigmoid, only for a range of values. In other words, a set of non-normalized values (that is: of arbitrary length and spread) can via softmax be mapped to a probability distribution over that set. This means that all the values x_i are in the range $0 \leq x_i \leq 1$, and that they sum to 1.

This is especially relevant in cases where one is attempting to perform classification, such as which is the present case where we are training the model to predict amino acid secondary structures.

Applying a softmax function on a dataset x of j elements would be as follows:

$$\sigma(x_i) = \frac{e^{x_i}}{\sum_{j=1} e^{x_j}}$$

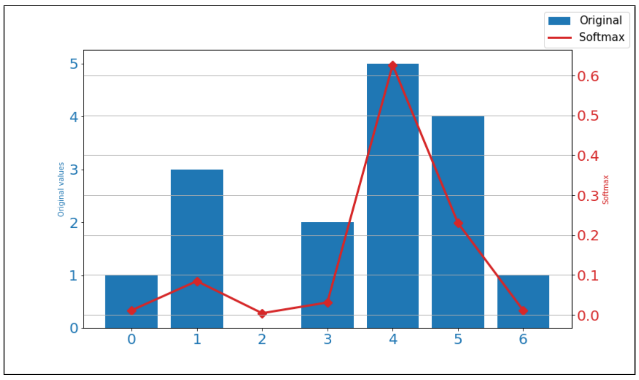

Figure 4: Example data softmaxed

Cross Entropy Loss Between two probability distributions each over the same set of outcomes there exists a certain cross entropy. Thus, given two probability distributions m and n predicting the discrete outcome $\mathcal{Z}$, the formula for calculating the cross entropy is:

$$L(m,n) = -\sum_{z\in\mathcal{Z}} m(z)\log n(z)$$

A specific instance of the cross entropy loss is the *binary cross entropy loss*, which is useful in cases where the possible outcome is binary (i.e. there are only two possible outcomes). In this case the formula becomes:

$$\ell(m,n) = L = \{l_1,\ldots,l_N\}^\top$$

where

$$l_n = -w_n\left[n_n \cdot \log m_n + (1-n_n)\cdot \log\left(1-m_n\right)\right]$$

Seing as the aim of this paper was to train our network to perform classifications first on amino acid secondary structures (of which there are eight) and then relative and absolute solvent accessibility (which are both encoded as either ones or zeros), we have used a Cross Entropy Loss function on the former and a Binary Cross Entropy Loss function on the two latter.

2.2.2 Technological implementation details

In training our network we utilized a machine learning framework for Python called PyTorch, specifically optimized for building deep neural networks.
The PyTorch library is build upon the Torch library, originally a machine learning library and language based on the scripting language Lua.
The strength in using PyTorch stems from several aspects:

GPU support PyTorch provides a strong framework for performing calculations on a graphical processing unit rather than the computer's CPU. While GPUs are valued in video game circles, they are also enourmously useful for performing machine learning, since this is often tasks that involve a high number of calculations that can perfectly fine be performed in parallel. For this, a GPU is preferred over a CPU since they often have a much higher number of cores, and are thus better suited for parallel programming.

Tensors The main data structure used in the PyTorch framework is tensors. In this context, a tensor is to mean a multi-dimensional array much like the ones implemented in numpy, but with the option to place it on the GPU rather than the CPU.

Automatic differentiation In order to utilize the backpropagation that lets neural networks train and improve, the loss must be differentiated in regards to all of the weights and values in the model in order to calculate the gradients. This can be a cumbersome task when performed by hand, but PyTorch provides a powerful tool in for of the `autograd` and `optim` modules. The first of these modules keeps track of which computations were performed on which variables in order to arrive at the final value of a variable, so that it can be retraced backwards in the end to calculate gradients, while the latter automatically adjusts the weights and values in the model, according to the calculated gradient and a supplied learning rate.

These things combined indeed allow a training step for a neural network to be performed in as few steps as shown below:

```
# Setup
LR = 0.0025                # learning rate
optimizer = torch.optim.Adam(cnn.parameters(),
      lr=LR)
loss_func = nn.CrossEntropyLoss()

# One training step
output = cnn(b_x)
loss = loss_func(output, b_y)
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

In terms of performing the calculations, all of our training of the model was done on a Dell PC with a 7th gen 8-cores Intel Core i7 CPU, 16GB of RAM and a Nvidia GeForce GTX 1050TI GPU with 4GB of RAM and 768 cores running Linux.

2.3 Predicting secondary structures alone

Fortælle om hvordan vi byggede vores single-model (convolutionelle lag, ReLU, softmax, BCELoss, padding).

2.3.1 Beregning af præcision

Forklar at det vi tæller er antallet af korrekte forudsigelser over antal af faktiske aminosyrer. Gennemgå herunder matematikken i at vi kollapser fra one-hot til classification, og så kører vores mask-magi på det.

2.4 Implementing multitask learning

2.4.1 Generelt om Multi-task learning

Start med refleksioner over hvordan vi bruger shared-parameters i vores model, og forklar at vi stadig kører udelukkende konvolutionelle lag igennem modellen.

2.4.2 Opbygningen af vores model

Fortæl om opbygningen af vores model - dvs. at vi ReLU'er hele vejen igennem, undtagen sidste lag, og så hvordan vi splitter dataen ud for så at ReLU'e og softmax'e sekundærstrukturerne, medens at vi afrunder solvent-egenskaberne.

2.4.3 Træning

Fortæl hvordan vi udregner loss på vores model, dvs. at vi udregner tre losses; sekundærstruktur, relativ solvency og absolut solvency. Vi bruger BCE på dem alle sammen, så redegør hvorfor vi godt kan det på solvent også.

2.4.4 Beregning af præcision

Forklar hvordan vi til strukturerne bruger samme beregning som ovenfor, mens vi gør næsten det samme for solvent egenskaberne (men hver for sig).

2.5 Hyperparametre

Fortæl at vi i projektet vil prøve at iterere over forskellige hhv. lagstørrelser, antal lag, kernelsizes, og learning rate. Forklar at vi vil starte med at tage udgangspunkt i de værdier som Xi bruger, og så iterere lidt frem og tilbage over dem.

3 Results

3.1 Hyperparameter-optimering

Gentag at vi tog udgangspunkt i værdier fra Xi og andre.

3.1.1 Antal lag

Fortæl om lag, lav en tabel og en graf.

3.1.2 Learning rate

Fortæl om LR, lav en tabel og en graf.

3.1.3 Antal neuroner i hvert lag

Fortæl om det, lav en tabel og en graf.

3.1.4 Kernel size

Fortæl om kernel sizes og om hvordan vi først prøvede én størrelse på dem alle og siden at variere størrelsen (lav evt. en reference til nogen af dem der har trænet på MNIST og deres kernel sizes), lav en tabel og to grafer.

3.2 Final predictive capabilities

Forklar hvordan vi fandt frem til vores hyperparametre på multi-task modellen og så anvendte de samme på single-task. Skriv noget tekst om hvordan vi på test-sættet nåede op på so-and-so meget præcision på hhv. den ene og anden model og hhv. strukturer og solvent egenskaber, samt sammeligning af resultatet på test- og valideringssæt.

3.3 Comparison

Forklar hvordan de to modeller ender med næsten samme præcision, omend multi-task modellen tager langt længere tid om at komme derop. De convergerer (jeg tror ordet er 'predictive ceiling') begge omkring de 68.5%, men for single allerede omkring 10 epoker medens multi skal bruge 25 epoker.

4 Conclusion

Modellen blev ikke bedre, men blev nogenlunde god til både at forudsige struktur og solvent-egenskaber. Vi kan have nogle overvejelser omkring hvad vi kunne have gjort bedre, f.eks:

- Gaussisk filtrering over dataen
- Evt. soft parameter sharing
- Regularization hvis vi tør

- En anden opbygning af multi-modellen hvor der var et enkelt eller flere fully-connected layers til solvent-egenskaberne.

5 Appendix

References

[1] C. Bishop, *Pattern Recognition and Machine Learning.* Information Science and Statistics, Springer, 2006.