Exercise 6: Neural Network Basics

Important: Before asking questions on piazza regarding the input or output specifications, please read the information provided on the exercise sheet and in the student template carefully!

Introduction

For most current problems of modern bioinformatics, neural networks play an important role in solving them. In this exercise we will, hence, have a look at the most basic building block of feed-forward neural networks: the perceptron. A single perceptron calculates the dot product between inputs and trainable weights. If the result of this operation is greater than a certain threshold, the perceptron gives an output. This behaviour is derived from neurons which receive inputs from various other neurons and 'fire' (output) as soon as the incoming signals pass a certain threshold. This very simple architecture can be extended to much more complex architectures. For further explanation, please see the slides.

In the first part of the exercise you will implement some helper functions which will be needed for all subsequent tasks. These functions include: adding a bias term to a given layer of the network and calculating the error (loss) between your prediction and the actual target. In the second part you will train a single perceptron which receives two boolean inputs for predicting the OR gate. This architecture will be implemented without and with an additional bias term. In the third part of the exercise you will stack several perceptrons to a multi-layer neural network. Again, this architecture takes two boolean inputs (and one bias term) but this time you will find a decision border for separating the different states of the XOR gate. This multi-layer architecture will be implemented once without an activation function and once with an activation function.

Helper Functions \((\sum=6P)\)

All neural networks are trained by measuring the difference between the predicted label (or value) and the true label (or value). All weights in the network are adjusted during the training process to minimize this difference. In the first part of the exercise you will implement two different loss
functions which allow you to measure the difference between your predictions and the ground truth. Additionally, you have to implement the derivatives of these loss functions as the weights are not updated based on raw error but on their negative gradient (see gradient descent). Additionally, you will implement an activation function (sigmoid) and a function which allows you to add a bias term for each layer. A bias term is an additional node in a layer which has a constant value (here: 1), meaning it receives no inputs from previous layers.

6.1 (H) Adding a Bias Term

Complete the `_add_bias` function which takes a 1D or 2D numpy array as an input and returns a new array which includes the bias term. For the 1D array, the bias is added as last element while for the 2D array the bias term is added to each row. In case of a 1D numpy array (e.g. [0, 0, 0]) the returned array should look like this: [0, 0, 0, 1]. In case of a 2D numpy array (e.g. [[0,1],[2,3],[4,5]]) the returned array should look like this: [[0,1,1],[2,3,1],[4,5,1]].

6.2 (H) Hinge Loss

Complete the function `_hinge_loss` so that it returns the hinge loss based on a given prediction and the actual label. Keep in mind that this loss requires the classes to be within [-1, 1] instead of [0, 1].

6.3 (H) Gradient of Hinge Loss

Complete the function `_delta_hinge` in such a way that it returns the gradient of the hinge loss function for a given prediction and the corresponding ground truth label. Again, keep in mind that the class labels are within [-1, 1] for the hinge loss instead of [0, 1].

6.4 (H) L2 Loss

Complete the function `_l2_loss` which returns the l2 loss for a given ground truth and the corresponding prediction. Here, a constant factor of 0.5 is multiplied with the loss in order to cancel out the exponent during derivation.

6.5 (H) Gradient of L2 Loss

Complete the function `_delta_l2` which returns the gradient of the l2 loss for a given ground truth and the corresponding prediction. Remember that we've added a constant factor of 0.5 previously to our L2 loss function in order to cancel out the exponent during derivation.

6.6 (H) Sigmoid Activation Function

Complete the function `_sigmoid` which transforms a given input number using the sigmoid function. Return the sigmoid value for the given input.

**Single Layer Perceptrons**

After preparing the required helper functions we can start implementing our first perceptron. For the sake of simplicity we approximate the OR gate:
Your perceptron will receive two boolean inputs $x_1$ and $x_2$ with each input having a weight associated. The sum of the weighted inputs should approximate the corresponding OR state (-1 or 1). Use the hinge loss to measure the difference between your prediction and the ground truth. Use the gradient of the hinge loss to update the weights. Update the weights for each sample. After training for $n$Epochs (given during object initialization), return the weights as one numpy array. Try to understand why this problem is not solvable without adding the bias term.

**6.7 (H) Single Perceptron**

Complete the function `single_perceptron` by implementing the forward and the backward pass for approximating the OR gate by using the backpropagation algorithm. The architecture is depicted in the figure above (Single Perceptron). While training for $n$Epochs, update the weights of your network after every sample was processed. As described above use the hinge loss and its gradient to update the weights. After training your network return its weights as numpy array.

**6.8 (H) Single Perceptron with bias**

Complete the function `single_perceptron_with_bias` by implementing the forward and the backward pass for approximating the OR gate by using the backpropagation algorithm. The difference to the previous architecture is that we will introduce a constant bias term now. The architecture is depicted in the figure above (Single Perceptron with bias). While training for $n$Epochs, update the weights of your network after every sample was processed. As described above use the hinge loss and its gradient to update the weights. After training your network return its weights as numpy array.
Multi Layer Perceptrons

After solving the OR gate problem in the previous tasks, let’s have a look at a slightly more complicated case, the XOR gate:

<table>
<thead>
<tr>
<th>x1</th>
<th>x2</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Please understand that the different states in the OR gate were separable by a single decision border. This is not possible for the XOR gate. Therefore, we need to introduce an additional layer to our network which allows us to find more complex decision borders. In the first version of your multi-layer perceptron you will implement two layers without an activation function. Please understand that this is equivalent to stacking multiple linear transformations which can be replaced by a single linear transformation. This means that we do not gain anything by the added complexity. However, this network will require only small modifications (introduction of a non-linearity during the forward and backward pass) to solve the XOR problem.

In the following we will use the L2 loss for measuring the difference between our predictions and the actual ground truth. As outlined above we will introduce a non-linearity (here: sigmoid) in the last part of the exercise. The range of the sigmoid activation function is [0,1] which is why we will now change our class labels to be also within [0,1]. Again, update the weights for each sample. After training for nEpochs, return the weights from the hidden layer to the output layer as one numpy array. Also remember to add a bias term to every layer as depicted in the figure above.

6.9 (H) Multi-Layer Perceptron

Complete the function `multi_perceptron_with_bias` by implementing the forward and the backward pass for approximating the XOR gate by using the backpropagation algorithm. The architecture is depicted in the figure above (Multi-Layer Perceptron). While training for nEpochs, update the weights of your network after every sample was processed. As described above, please use the L2 loss and its gradient to update the weights now. After training your network return the weights from the hidden to the output layer as a numpy array.

6.10 (H) Multi-Layer Perceptron with bias and non-linearity

Complete the function `multi_perceptron_with_bias_and_nonlinearity` by implementing the forward and the backward pass for approximating the XOR gate by using the backpropagation algorithm. The architecture is depicted in the figure above (Multi-Layer Perceptron). However, this
time you add a sigmoid non-linearity to the hidden-layer. Please note that this also affects the backward pass. While training for $n$ epochs, update the weights of your network after every sample was processed. As described above use the L2 loss and its gradient to update the weights. After training your network return the weights of the hidden to the output layer as a numpy array.