Exercise 7: Neural Networks

Introduction

In the last exercise you have implemented your first neural networks. In this exercise, we will build upon the last to create a neural network which will be trained on classifying residues in a transmembrane helix from all other residues. Before training the network you will also implement the required pre-processing, including one-hot-encoding of inputs, a sliding window approach, the removal of unresolved residues and a split into a training and a test set. After finishing the pre-processing you will gather information on your data set, e.g. number of samples in the data set. Finally, you will train your own network which allows you to experiment with different meta-parameter combinations, s.a. learning rate, number of nodes in the hidden layer.

Pre-processing

As mentioned above, you have to process your data before you can start to train your network. This includes the following steps:

- Read in raw data
- Transform input sequences via one-hot-encoding
- Add neighbouring residues to each sample (sliding window approach)
- Transform class labels to integers
- Split your data set into a training and a test set

7.1 (H) Read in Raw Data

Complete the function _get_raw_data to read in the raw data from a text file. The format is described in the template code. As a recap: Every protein is described by three consecutive lines:

- Header, indicated by a ‘>’ at the beginning of the line
- Sequence, single letter amino acid code (input)
- Transmembrane annotation (output). Labels are explained in more detail in the code template

The data in this textfile has to be parsed and stored in two different dictionaries, one for inputs (sequences) and one for outputs (transmembrane annotation). The unique identifiers given in the header of each sequence are used as keys and the strings describing the sequence or the annotation are stored as values.

7.2 (H) One-Hot-Encoding Lookup (1)
Complete the function `_get_one_hot_encoding` to get a dictionary which maps every character in our alphabet (see static class variable ALPHABET; here: 20 amino acids) to a numpy array which holds the corresponding one-hot-encoding for the given letter in the alphabet. Keep the order of the letters in the alphabet, starting with A being represented by \([1,0,0,...]\). Save the dictionary in the class variable self.AA_2_INT. Hint: Use np.diag.

7.3 (H) One-Hot-Encoding of Inputs (1)
Complete the function `_get_one_hot_inputs` in such a way that it transforms the protein sequence (String) to a numpy array of shape ( length of protein x length of alphabet ). Use the class variable created during the previous task for replacing every amino acid letter by the corresponding one-hot-encoding (numpy array). Again, store the transformed input sequences in a class variable called self.encoded_inputs.

7.4 (H) Sliding Window (1)
Complete the function `_get_sliding_window_view` in order to add neighbouring residues to each sample. Use the window size given during object initialization for this. Use zero-padding to add zero arrays to the start and end of the sequence in order to represent also residues at the start and end of the sequence (see 'same' padding). The size of the zero-padding depends on the window size. Store the transformed input representation again in a dictionary, using the unique protein identifiers as keys and the sliding window view on the one-hot-encoded amino acid sequence as value (numpy array of shape (length of protein x length of alphabet * window size)). Save the dictionary in the class variable self.inputs.

7.5 (H) Output Encoding (1)
Complete the function `_get_integer_outputs` in order to encode the class labels (String) as integers (class). Use the encoding given in the static class variable self.LABEL_2_INT for this. Store the transformed outputs again in a dictionary using the unique protein identifiers as keys and the class labels as values (numpy array of shape (length of protein)). Save the dictionary in the class variable self.outputs. While iterating over the outputs, fill a second dictionary which stores information on which residues were unresolved. As we can not say for those residues whether they are part of a transmembrane helix or not, we will exclude them from our data set. Use a simple flag for each residue, telling whether it is unresolved (0) or not (1). Again, use the unique protein identifiers as keys and numpy arrays as values ( shape: ( length of protein ) ). Save the dictionary in the class variable self.mask.

7.6 (H) Remove Unresolved Residues (1)
Complete the function \_remove\_unresolved to remove all samples from the dataset which were unresolved. Use the class variable self.mask for this. Edit the class variables self.inputs and self.outputs directly. Also return the modified class variables for easier testing.

7.7 (H) Split Train And Test

Complete the function \_get\_train\_and\_test\_split to split your data set in a training and a test set. The size of the test set is defined by the class variable self.test\_percentage. Use a randomly generated number (numpy.random.rand) to define whether a samples goes into the test set (\text{rand < test\_percentage}) or not. Save the data splits in four different dictionaries: test inputs, test outputs, training inputs and training outputs. Again, use class variables for storing the dictionaries to make them easily accessible during training.

Data Analysis \(\sum=3P\)

Before training the network you always have to get familiar with your dataset first. Here, we will perform a very simple data analysis before training the network. It includes retrieving the number of all samples, of all unresolved samples and all positive (transmembrane) or negative samples (all other).

7.8 (H) Number of Samples

Complete the function get_num_samples by simply counting the number of all samples in your data set (after removing unresolved residues).

7.9 (H) Number of Unresolved Samples

Complete the function get_num_masked_out by counting all unresolved samples.

7.10 (H) Number of Positive and Negative Samples

Complete the function get_num_pos_and_neg by counting all positive samples (residue is part of a transmembrane helix) and negative samples (all other residues). Again, provide the numbers after you have removed unresolved residues.
Artificial Neural Network

After performing all required pre-processing steps and the data analysis you are ready to start working on the actual network architecture. First, we will again implement the required loss function, its derivative or gradient and a normalization (softmax) which converts the raw outputs of the network to a probability for each class (sum over probabilities for all classes equals one). After this we will implement the forward and the backward pass. Keep in mind that we will not process every sample separately during this exercise but in batches. Here, proteins represent batches of varying size.

7.11 (H) Softmax Normalization

Complete the function _stable_softmax by implementing a vectorized version of the softmax normalization. This normalization normalizes the raw outputs of the network to class probabilities. As mentioned above, we will use batch-training during this exercise, meaning that we process not single samples but all samples within one batch (here: protein). The raw output scores of your network will have the shape (length of protein x number of classes). Further, subtract a constant (max(predictions)) from the exponents in the softmax formula to make it numerically stable.

7.12 (H) Cross Entropy

Complete the function _cross_entropy by implementing a vectorized version of the cross-entropy loss. Again, the input is one batch (protein), not a single sample. Sum over all losses in the batch.

7.13 (H) Delta Cross Entropy

Complete the function _delta_cross_entropy by implementing a vectorized version of the partial derivative of the cross-entropy loss with a softmax output layer. Again, the input is one batch (protein), not a single sample. Account for varying batch sizes by dividing gradient vectors by the number of samples in the batch.

7.14 (H) Forward Pass

Complete the first part (forward pass, before if-statement) of the function _predict by implementing the inference step (prediction) of a batch. Use the network architecture which was defined during object initialization for this. Please understand that you can change the architecture by adjusting the number of hidden layers (n_hidden). Use a rectified linear unit (ReLU) as a non-linearity between the layers and a softmax activation at the output layer. Use the weights self.w1, self.w2 which were randomly initialized during the object initialization.

7.15 (H) Backward Pass

Complete the second part (backward pass, after if-statement) of the function _predict by implementing the backpropagation of a batch. Use the gradient defined by _delta_cross_entropy, the learning rate defined during object initialization and the output from the previous layer for this. Take the derivative of the non-linearity into account which should be fairly simple for the ReLU.
non-linearity.

7.16 (H) Performance Assessment

Complete the function _test_performance by returning the average accuracy, average loss and confusion matrix for a given data set (either training or test; defined by boolean parameter). The accuracy is calculated as correct samples divided by total number of samples, the loss is averaged over all batches in a data set and the confusion matrix holds true labels (ground truth) in rows and predictions in columns. This helps you to monitor learning progress, overfitting and allows to compare different architectures.