General remarks:

- Code available here with templates for all the problems.
- You can use the internet to check documentation or course material.
- It is strictly forbidden to look for pieces of code on the internet that are then copy-pasted as solutions.
- You should submit your solution by email with the subject@here deadline is 8pm “GSN exam” sent to cygan@mimuw.edu.pl and kuba.swiatkowski@gmail.com.

Problem 1 - Sparse MLP (1.5 points)

Goal: Implement backpropagation for random sparse MLP.

Consider an MLP modified as follows: For each layer L, we fix this layer’s degree d_L. Each neuron of this layer randomly chooses a subset of d_L neurons in the previous layer and only has connections to these neurons - this subset remains the same throughout the learning process.

Your task is to modify the program sparse-mlp.py (which is based on code from the lab assignments), so that it implements this idea. You might consider modifying the following functions: `__init__, feedforward, backprop` and possibly `update_mini_batch`. Do not change the overall structure of the program. In particular, the final program should still run and perform the same task as before. It is ok to perform the computation using loops instead of numpy parallelism.

Your program should accept arbitrary d_L values. However, before submitting, set the layer sizes to `[784,300,10]` and d_L values to `[50,100]`.

Note that you should implement the backpropagation in this setting yourself, in particular:

- you should not use any sparse-matrix calls from numpy or other library,
- your program should not create large matrices, specifically for two subsequent layers with n_1 and n_2 neurons respectively, your program should not create (even temporarily) create matrices of size $\Omega(n_1 n_2)$

Problem 2 - RNN (1.5 points)

Goal: Implement a recurrent layer with dropout regularisation.

In the recurrent layer each step of each sequence is processed using this formula:

$$h_t = \phi(Wx_t + Uh_{t-1}).$$

The hidden state at a time step t is h_t. It is a function of the input at the same time step x_t, modified by a weight matrix W added to the hidden state of the previous time step h_{t-1} multiplied by its own hidden-state-to-hidden-state matrix U, otherwise known as a transition matrix. ϕ is a hyperbolic tangent activation function.

The requested recurrent layer should be implemented as a subclass of the `torch.nn.Module`. **Dropout mask should be sampled once per input sequence**, remain constant during each step of the sequence and be applied only to the hidden states at each step before the matrix U operates on them. You can only use tensor operations: `torch.bernoulli`, `torch.nn.Linear` and activation functions in your implementation. Your input should be of size:

```
(input_size, seqence_length, batch_size),
```

where `input_size` is the size of your single element in a sequence (so it can be a vector of length `input_size`). You have to process the whole batch of sequences at once when you use the `forward`() method of your layer and return a tensor with a hidden size vector for each step in the sequence. Your output has to be the shape of

```
(hidden_size, seqence_length, batch_size).
```

Remember about resetting your hidden state (to any value you want between -1 and 1) at the beginning of each sequence processing. Note that your code does not have to be fully vectorized, you can use loops.

Problem 3 - embedding (1.5 points)

Goal: Implement an embedding layer, that maps integers {0, …, dictionary_size-1} to vectors in $R^{embedding\_size}$

Implement it as a subclass of the torch.nn.Module and using only simple tensor operations (slicing, indexing, etc) and `torch.randn` function. Your layer must take an integer sequence of the shape (with one element index for each step of the sequence):

```
(batch_size, seqence_length)
```

and output a sequence of shape:

```
(embedding_size, seqence_length, batch_size)
```

so it has to match a recurrent layer input (note the order of dimensions). Save your embedding in `self.embeddings` as a torch tensor with rows representing embeddings for each element in the dictionary.

Problem 4 - DQN (1.5 points)

Here you can find a sample implementation of DQN for CartPole, where the goal is keep the pole straight up by steering left and right. You can find code description in this tutorial, however the goal is to do a small modification (a few lines of code), so understanding the whole program is superfluous.

The goal is to change the way the program adds samples to the `ReplayMemory`, we want to execute `memory.push()` only if:

a) there are less than 100 samples in the buffer currently, or
b) The absolute difference between the Q-function for the current state-action pair and its target value is at least 0.5, where the target value is equal to the reward in case the game has finished, and otherwise it is defined by the following formula (where we use the target network's parameters).

$$r + \gamma \max_{a'} Q(s', a'; \theta^-)$$

Note: it is enough to make the requested modification in the program, we do not expect the program to reach a good policy fast.