Translated from Polish to English - www.onlinedoctranslator.com

NEURAL NETWORKS – exercise 4

In exercise 4, we will recreate the already implemented fully connected network architecture using a ready-made solution for building neural networks. Below is an example of learning in Pytorch, other solutions are acceptable (tensorflow, within it keras which is even simpler), provided that they allow you to define your own network architecture.

First of all, we expect from a deep learning framework:

- Implementation of matrix/tensor operations to the extent that we will need to build the network
- Possibility of automatic calculation of gradients after implemented operations
- Efficient dedicated GPU/TPU implementation

And typical functionalities that facilitate building neural networks using the above (cost functions, standard layers, optimizers, etc.).

For torch, the convention is as follows: we use torch.tensor objects to store data, while the network is built from modules inheriting from torch.NN.module. By tensor we mean an n-dimensional array of numbers, analogous to numpy. Basic mathematical operators are overwritten for tensors, and array operations similar to those available in numpy are available (shape changes, transpositions, aggregations such as average and sum, etc.). In the tensor object, the gradient may also be stored next to the actual value. The gradient, if it exists, will always be a tensor of the same dimension as the actual tensor data. It is initialized on the first pass of backpropagation through a given tensor.

The model should inherit from torch.NN.module and define init() - operations performed when creating a class instance and forward() - a function that returns the model's output based on the input data.

The official tutorial on how to train a model describes the training loop this way: https://pytorch.org/tutorials/beginner/introyt/trainingyt.html

```
optimizer =torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)#1

deftrain_one_epoch(epoch_index, tb_writer):
    running_loss =0.
    last_loss =0.

    # Here, we use enumerate(training_loader) instead of #
    iter(training_loader) so that we can track the batch #
    index and do some intra-epoch reporting
    for i, data inenumerate(training_loader):
        # Every data instance is an input + label pair
        inputs,labels = date

        #Zero your gradients for every batch!
        optimizer.zero_grad()#2
```

```
        # Make predictions for this batch
        outputs = model(inputs)#3

        # Compute the loss and its
        gradientsloss=loss_fn(outputs,labels)#4
        loss.backward()#5

        # Adjust learning weights
        optimizer.step()#6

    return loss
```

How and why does it work?

1. The optimizer object is used to optimize a given set of parameters. Parameters are a subclass of tensor specifically considered by modules.

   parameters() of any torch.NN.module object returns the parameters belonging to that class and the parameters of each belonging variable, which is itself a subclass of torch.NN.module. Note: the function will not work on nested variables if any level is not a subclass of torch.NN.module! For example code:

   ```
   __init(self):
        super(TinyModel, self). init()
        self.layer_list   =   [torch.nn.Linear(10,10)   for   and
                              inrange(123)]
   ```

   It will create a list of 123 10x10 linear layers, but since the only variable belonging to the class is the list itself, the model will not correctly account for these 123 layers and their weight matrix in parameters().

2. zero_grad() is needed because the autodifferentiator accumulates rather than overwrites gradients when called multiple times.

3. Calling a model instance like a function (Python call) is equivalent to forward()

4. Cost functions are available as parameterizable classes in torch.nn. This means that we first instantiate them as an object and then call loss_fn() on the appropriate tensors corresponding to the inputs and desired outputs - pay attention to the expected tensor type in the documentation and remember the default formats! (Specifically, torch's default float does NOT match the default in numpy - 32 vs 64bit!)

5. backward() is the entire backward pass in the backpropagation algorithm. How did you manage to simplify it so much? First, each tensor operation available in torch, analogous to the implementation suggested in the previous exercise, has both forward() and backward() functions. Second, operations during forward traversal are appended to the computation graph

containing information about where the entry to a given operation comes from. The existence of a computational graph allows you to iterate through previously called operations backwards and calculate the gradients at each level and accumulate them in parameter objects.

6. `step() changes parameters according to the gradient value and the adopted learning rule. For SGD it is the classic - learning_rate*gradient`

In the exercise, you should build a network with the same parameters as in the previous task, and evaluate its operation on the same data, this time checking the impact:

- selected optimizer (SGD and two others)
- batch size
- learning coefficient values for various optimizers. The

exercise is assessed on a scale of 0-10 points.