Assignment

Computer Vision and Deep Learning

Matthias Fulde

2023

This assignment is optional and not graded!

1 Background

In the lectures, several architectures for artificial neural networks have been introduced, and we have seen how we can train them for some task in a supervised fashion, using stochastic gradient descent optimization with backpropagation.

We want to use these models in practice, but we don't want to implement all the necessary algorithms by ourselves. Instead we want to use some existing and well tested implementations. That is, we want to make use of a machine learning library that fits our needs. In particular, due to the computational complexity of training neural networks, we require that such a library, besides providing the needed abstractions, can execute our programs on the graphics processing unit (GPU) of our computer, which allows for massively parallel computations.

There are a couple of programming libraries for the Python language that satisfy our requirements, most notably the PyTorch and TensorFlow libraries. For this course in general, and this assignment in particular, we highly recommend to use the PyTorch library, which is more widely used in academia and, at least according to our experience, is easier to work with.

2 Installation

The easiest way to work with PyTorch is to use web-based services like Google Colab, where no software has to be installed locally. Similar services are offered by Microsoft and IBM, and possibly other companies. In the case of Google Colab, there is a limited time budget for free GPU usage, which will be enough for smaller tasks. Additional compute time is offered as a paid service, but there might also be student programmes, which allow free GPU access for longer times.

If you have a computer with a dedicated GPU, you can also install PyTorch locally. In this case, we recommend to install PyTorch not directly on the system, but in a virtual environment, as this avoids a lot of potential trouble handling dependencies. We would recommended to use Anaconda, which also contains a package manager that makes it easy to install required libraries. There's a readable documentation on how to manage environments in Anaconda. For instance, a new environment with the name "xaim" can be created with the following command:

```
conda create -n xaim python=3.10
```

Please note that this command gives the version for the Python language explicitly. The default behavior of the command is to always install the latest Python version. However, the PyTorch library currently only supports up to version 3.10, so we have to specify this when we create a new environment meant to be used with this library. On the Get Started page of PyTorch, theres a widget to create the necessary command to install PyTorch with Anaconda. Do this after you activated the environment where you want to install the library in. Extending the example from above, we could do this with the following line:

```
conda activate xaim
```

Depending on the configuration of your system, you might need to install the CUDA toolkit or the ROCm software stack, for GPUs from NVIDIA or AMD, respectively. This should not be done in a conda environment, but directly on the operating system. You can check whether the installation was successful by executing the following code snipped with a Python interpreter:

```python
import torch

print(torch.cuda.is_available())
```

Installing PyTorch locally can be a bit tricky, but for troubleshooting, a web search pasting the received error messages usually yields valuable hints for possible solutions. If this doesn't work, please reach out to us. Besides installing PyTorch, we also recommend to install Jupyter Notebook. This application provides a web interface for interactive documents using Python code, which is very convenient to work with, in particular for smaller projects and experimenting.

A potential issue might be that you don't have a computer with a dedicated GPU available, but you also don't want to give your user data to companies like Google, which would be a valid point. While this might not necessarily be a problem in the group project, where it would be sufficient that one team member has access to such a system, it would prevent you from working on this assignment individually. We're trying to figure out whether it will be possible for you to access some GPU cluster hosted by the university, and if so, how this can be done.

3 Exercises

The goal of this assignment is for you to obtain some familiarity with the PyTorch library, so that you're able to create, adjust, train, and validate neural networks, as this will likely be necessary for you in one way or another to complete your course project. You don't need to become experts in this area, but a basic understanding of what can be done will also help you to assess projects in the future. Since the main focus is on learning, the exercises won't be too specific, so that there is room for you to experiment and try out different things.

3.1 Tutorials

Especially if you don't have any prior experience with PyTorch or other machine learning libraries, we highly recommend that you walk through a couple of tutorials first. On the website of the PyTorch project, there's a huge section of mostly well-written tutorials that you can follow. In particular, the Learn the Basics tutorial will help you to get an idea of the basic building blocks for training neural networks that are offered by the PyTorch programming interface. In addition, we recommend to read the shorter Training a Classifier tutorial, which describes how to load a commonly used dataset and train a simple convolutional neural network for classification. There's also a small exercise at the end of this tutorial that you can do. Finally, you should have a look at the Transfer Learning for Computer Vision tutorial. In practice, you'll often find that you have insufficient data to train a network from scratch. In these cases it's often possible to adapt some pretrained network to the task at hand. Two common approaches for knowledge transfer are described in this tutorial.

3.2 Residual Networks

To get some hands-on experience with the functionality of PyTorch, in this exercise, we want to implement a particular architecture introduced in the lecture, namely, residual networks. However, before we come to model building, we first want to write some reusable code for training a classifier, based on the solutions that are presented in the tutorials linked above.

The first part of the exercise is to write yourself a Python class that contains the boilerplate code for training a model for classification, evaluating the model, and loading and saving model parameters and training state. You could call the class a `Solver`, and its constructor should take as arguments at least a model, a dataset for training and validation, an optimizer, and a loss function. In the constructor, you might also want to define some attributes to store the training history, like number of epochs, losses, and accuracies, allowing you to plot the progress of the training.

You should follow the instructions in Saving and Loading a General Checkpoint to define `save` and `load` methods for your class, which allow you to interrupt and resume training, or to load an already trained model back into your class for inference. If you want, you could also include logic for plotting losses and accuracies into your class, defining a method for this purpose. For plotting, the Matplotlib library could be used, or the Seaborn library, which builds on top of that.

In addition to what we already have, you should define `test` and `train` methods for the class. The `test` method should take a dataset and optionally, a parameter specifying the number of samples to take from the dataset. The torch.utils.data library contains functions that can be used for subsampling a dataset. You might want to call this method once every epoch during training to evaluate your model on equally sized subsets of your training and validation data, in order to track the progress of the training. After training is completed, you can call the method with a test set. The method should compute and return the accuracy for the given dataset.

The `train` method should contain the main training loop. As a parameter, the method should take the number of epochs to train the model. It makes sense to do it like this instead of fixing the number of training epochs ahead of time by passing the number to the constructor of the class. The `train` method basically just contains the logic for training networks described in the tutorials that are linked above. The return value could be a dictionary containing the training history. You can use the tqdm library to create a progress bar for your training, allowing you to track the progress without having to print out results in regular intervals.

To test your implementation and to practice using existing models, you can now try to finetune a pretrained network that you can load directly with PyTorch. The models repository of Torchvision provides a selection of commonly used architectures. For instance, you can use a ResNet18 of moderate size, which was pretrained on the ImageNet dataset. Using Models from Hub provides some examples on how to load such a model. Besides pretrained networks, PyTorch also provides a selection of Datasets for Image Classification. Select one of the datasets and try to finetune the pretrained ResNet as described in the tutorial that is linked above. Can you improve upon the baseline before finetuning?

If you have the time and are willing to dive a bit deeper into the topic, you can try to build a residual network architecture on your own. This is actually not too complicated and only slightly extends on the Definition of a Convolutional Neural Network given in the classifier tutorial linked above. The idea would be to create a new model component by defining a `ResidualBlock` class as a subclass of `torch.nn.Module`.

The constructor of this class would take three parameters, `in_channels`, which is the number of input channels, `out_channels`, which is the number of output channels, and `stride`, which defines the step size for the convolutions. We need to define two branches for the computation, a main branch, which would consist of two convolution layers, both followed by a Batch Normalization layer, and a ReLU activation in between. This could conveniently be implemented using the Sequential container type of PyTorch. The first of the two convolution layers would use the stride provided as an argument, while the second convolution layer uses unit stride. `in_channels` is the number of channels for input to the first of the two layers, and `out_channels` is the number of channels for the output of the first and both the input and output of the second convolution layer. The second branch is the shortcut connection. If the numbers of input and output channels match and stride is 1, we can just use the Identity mapping for the shortcut, since the dimensions won't change. Otherwise we have to map the input to the new output dimensions, for which we can use another convolution layer, with a kernel size of 1 and the same stride as was given as an argument, followed by a batch normalization layer. Again, we can use the `Sequential` class for this.

If you managed to implement that, the hard part is done. Then everything that's left to do is to implement a `forward` method which takes an input, passes it through the main and shortcut branches, adds the results, and finally applies a ReLU activation, returning the final output. This `ResidualBlock` can then be used in the same fashion as single convolution or linear layers when defining a classifier network. Try to define a small residual network using your building block and train it on a dataset of your choice. When designing the network, recall that you can compute the spatial dimensions of the output feature maps of a convolution layer with

$$D_{\text{out}} = \left\lfloor \frac{D_{\text{in}} + 2P - K}{S} \right\rfloor + 1$$

where D is either height or width, P is the amount of padding, S is the stride, and K is the kernel size of the filter. This way you can make sure that input and output sizes between layers match. If you're unsure on how excatly to build your overall network, you can call `print(model)` with the predefined ResNet18 model to see which components are used in this implementation, and take inspiration from this. If you want to further explore the PyTorch library, you can come back to us so that we can give some more suggestions.

A last point that's worth mentioning when designing a network from scratch, is the question of complexity. For convolutional neural network architectures, you can compute the number of parameters in a layer as number of filters times number of input channels times filter size squared, plus a bias for each filter. For linear layers you have a weight matrix whose size is the product of the input and output features, plus a bias vector with a size corresponding to the output features. Normalization layers like the ones used in residual networks add two sets of learnable parameters equal to the number of input channels. However, it's very inconvenient to compute all this from hand, so we can save us some headache by letting PyTorch count the number of parameters in our models. This can be achieved with the following line of code:

```
num_param = sum(p.numel() for p in model.parameters() if p.requires_grad)
```

You can do this both for your own network designs and existing networks from the model hub, to get an idea of your model's size in comparsion to common architectures. Our recommendation would be for this exercise to create networks that do not have more than a million parameters, which, for toy datasets and small scale experiments, should be enough. Depending on the complexity of the chosen dataset, even models with 100 k parameters can produce reasonably good results. A good exercise is to restrict yourself to a fixed model size and then experiment with different hyperparameter settings to get the most out of it under this restriction.

4 Solution

Solutions for this exercise will be presented in the last live meeting of the course on the 5th of December. So, you'll have almost a month to work on this assignment. However, keep in mind that, as mentioned in the very beginning, this is not a graded exercise, so it's up to you if—and to what extend—you work on this.