Computer Vision with Deep Learning: Practical Assignments

VISUM summer school

In this practical assignment you will learn how to implement and train basic neural architectures like multi-layered perceptrons (MLPs), convolutional neural networks (CNNs), and recurrent neural networks (RNNs). The document is organized in a number of sections, each containing its own set of questions and implementation assignments:

- **Section 1**: Multi-layered Perceptrons
- **Section 2**: Convolutional Neural Networks
- **Section 3**: Recurrent Neural Networks

What language to use? Python and PyTorch have a large community of people eager to help other people. If you have coding related questions: (1) read the documentation, (2) search on Google and StackOverflow, and (3) ask the people around.

There are two suggested paths to take for this assignment. The first strategy is to focus on the MLPs and CNNs. One can easily fill a whole afternoon with exploring the many different settings, networks, and hyperparameters for deep networks. Focusing on the first two sections is recommended for people that are relatively new to deep learning and would like to get a good grasp on what it takes to implement, run, and optimize networks. The second strategy is to focus on MLPs and CNNs for the first half and focus on RNNs for the second half. This strategy is recommended for people who are more experienced with PyTorch and want to delve deeper.

1 Multi-layer Perceptrons (MLPs)

The main goal of this part is to make you familiar with PyTorch. PyTorch is a deep learning framework for fast, flexible experimentation. It provides two high-level features:

- Tensor computation (like NumPy) with strong GPU acceleration
- Deep Neural Networks built on a tape-based autodiff system

You can also reuse your favorite python packages such as NumPy, SciPy and Cython to extend PyTorch when needed.

There are several tutorials available for PyTorch:

- Deep Learning with PyTorch: A 60 Minute Blitz
- Learning PyTorch with Examples

In this assignment, you will study and implement a multilayer perceptron (MLP). The task will be to classify tiny images into 10 different categories. Implementing this task will get you familiar with the data processing pipeline as well as the hyper-parameters tuning required to train an MLP.

In general, MLPs are considered as universal approximators. This means they can learn to represent a wide variety of functions. In other words, MLPs are not restricted to images but could also be used for learning text or speech representations for example. Nevertheless, their capacity and their performance might not be optimal, that is why more specialized types of neural networks exist to process images, videos, text, or sound. We will explore in Section 2 how to exploit the spatial structure of images, and in Section 3 how to model sequential data.

Question 1.1

Implement the MLP in `mlp_pytorch.py` file by following the instructions inside the file. The interface is similar to `mlp_numpy.py`. Implement training and testing procedures for your model in `train_mlp_pytorch.py` by following instructions inside the file. You should train your MLP on the CIFAR-10 dataset for which you can use the code in `cifar10_utils.py`.

Before proceeding with this question, convince yourself that your MLP implementation is correct. For this question you need to perform a number of experiments on your MLP to get familiar with several parameters and their effect on training and performance. For example you may want to try different regularization types, run your network for more iterations, add more layers, change the learning rate and other parameters as you like. Your goal is to get the best test accuracy you can. You should be able to get *at least 0.53* accuracy on the CIFAR-10 test set but you are challenged to improve this.

2 Convolutional Neural Networks (CNNs)

At this point you should have already noticed that the accuracy of MLP networks is far from perfect. A more suitable type of architecture to process image data is CNNs. They are now the widely use in both academia and industry for very broad image processing tasks.

CNNs rely on the convolution operation, which has many advantages. It reduces the number of parameters, compared to an MLP, to make CNNs less prone to overfitting. Additionally, this parameter sharing makes the features equivariant to scale and translations.

In this part of the assignment you are going to implement a small version of the popular VGG network, which has won the ImageNet 2014 challenge.

Name	Kernel	Stride	Padding	Channels In/Out
conv1	3×3	1	1	3/64
maxpool1	3×3	2	1	64/64
conv2	3×3	1	1	64/128
maxpool2	3×3	2	1	128/128
conv3_a	3×3	1	1	128/256
conv3_b	3×3	1	1	256/256
maxpool3	3×3	2	1	256/256
conv4_a	3×3	1	1	256/512
conv4_b	3×3	1	1	512/512
maxpool4	3×3	2	1	512/512
conv5_a	3×3	1	1	512/512
conv5_b	3×3	1	1	512/512
maxpool5	3×3	2	1	512/512
avgpool	1×1	1	0	512/512
linear	–	-	-	512/10

Table 1. Specification of ConvNet architecture. All *conv* blocks consist of 2D-convolutional layer, followed by Batch Normalization layer and ReLU layer.

Question 2.1

Implement the ConvNet specified in Table 1 inside `convnet_pytorch.py` file by following the instructions inside the file. Implement training and testing procedures for your model in `train_convnet_pytorch.py` by following instructions inside the file. Use Adam optimizer with default learning rate. Use default PyTorch parameters to initialize convolutional and linear layers. With default parameters you should get around *0.75* accuracy on the test set.

3 Recurrent Neural Networks (RNNs)

In this assignment you will study and implement recurrent neural networks (RNNs). This type of neural network is best suited for sequential processing of data, such as a sequence of characters, words or video frames. Its applications are mostly in neural machine translation, speech analysis and video understanding. These networks are very powerful and have found their way into many production environments. For example **Google's neural machine translation system** relies on Long-Short Term Networks (LSTMs).

Before continuing with the first assignment, it is recommended to each student to read this excellent blogpost by Chris Olah on recurrence neural networks: **Understanding LSTM Networks**.

For the first task, you will implement vanilla Recurrent Neural Networks (RNN). PyTorch has a large amount of building blocks for recurrent neural networks. However, to get you familiar with the concept of recurrent connections, in this first part of this assignment you will implement a vanilla RNN from scratch. The use of high-level operations such as `torch.nn.RNN` and `torch.nn.Linear` is not allowed.

3.1 Toy Problem: Palindrome Numbers

In this first assignment, we will focus on very simple sequential training data for understanding the memorization capability of recurrent neural networks. More specifically, we will study *palindrome* numbers. Palindromes are numbers which read the same backward as forward, such as:

303

4224

175282571

682846747648286

We can use a recurrent neural network to predict the next digit of the palindrome at every timestep. While very simple for short palindromes, this task becomes increasingly difficult for longer palindromes. For example when the network is given the input 68284674764828_ and the task is to predict the digit on the _ position, the network has to remember information from 14 timesteps earlier. If the task is to predict the last digit only, the intermediate digits are irrelevant. However, they may affect the evolution of the dynamic system and possibly erase the internally stored information about the initial values of input. In short, this simple problem enables studying the memorization capability of recurrent networks.

For the coding assignment, in the file `dataset.py`, there is aprepared class `PalindromeDataset` which inherits from `torch.utils.data.Dataset` and contains the function `generate_palindrome` to randomly generate palindrome numbers. You can use this dataset directly in PyTorch and you do not need to modify contents of this file. Note that for short palindromes the number of possible numbers is rather small, but we ignore this sampling collision problem for the purpose of this assignment.

3.2 Vanilla RNN

The vanilla RNN is formalized as follows. Given a sequence of input vectors $\mathbf{x}^{(t)}$ for $t = 1, \dots T$, the network computes a sequence of hidden states $\mathbf{h}^{(t)}$ and a sequence of output vectors $\mathbf{p}^{(t)}$ using the following equations for timesteps $t = 1, \dots, T$:

$$\mathbf{h}^{(t)} = \tanh(\mathbf{W}_{hx}\mathbf{x}^{(t)} + \mathbf{W}_{hh}\mathbf{h}^{(t-1)} + \mathbf{b}_h) \tag{1}$$

$$\mathbf{p}^{(t)} = \mathbf{W}_{ph}\mathbf{h}^{(t)} + \mathbf{b}_p \tag{2}$$

As you can see, there are several trainable weight matrices and bias vectors. $\mathbf{W}_{hx}$ denotes the input-to-hidden weight matrix, $\mathbf{W}_{hh}$ is the hidden-to-hidden (or recurrent) weight matrix, $\mathbf{W}_{ph}$ represents the hidden-to-output weight matrix and the $\mathbf{b}_h$ and $\mathbf{b}_p$ vectors denote the biases. For the first timestep $t = 1$, the expression $\mathbf{h}^{(t-1)} = \mathbf{h}^{(0)}$ is replaced with a special vector $\mathbf{h}_{init}$ that is commonly initialized to a vector of zeros. The output value $\mathbf{p}^{(t)}$ depends on the state of the hidden layer $\mathbf{h}^{(t)}$ which in its turn depends on all previous state of the hidden layer. Therefore, a recurrent neural network can be seen as a (deep) feed-forward network with shared weights.

To optimize the trainable weights, the gradients of the RNN are computed via back-propagation through time (BPTT). The goal is to calculate the gradients of the loss $\mathcal{L}$ with respect to the model parameters $\mathbf{W}_{hx}, \mathbf{W}_{hh}$ and $\mathbf{W}_{ph}$ (biases omitted). Similar to training a feed-forward network, the weights and baises are updated using SGD or one of its variants. Different from feed-forward networks, recurrent networks can give output logits $\hat{\mathbf{y}}^{(t)}$ at every timestep. In this assignment the outputs will be given by the softmax function, *i.e.* $\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{p}^{(t)})$. For the task of predicting the final palindrome number, we compute the standard cross-entropy loss *only* over the last timestep:

$$\mathcal{L} = -\sum_{k=1}^{K} \mathbf{y}_k \log \hat{\mathbf{y}}_k \tag{3}$$

Where k runs over the number of classes ($K = 10$ because we have ten digits). In this expression, $\mathbf{y}$ denotes a one-hot vector of length K containing true labels.

Question 3.1

Implement the vanilla recurrent neural network as specified by the equations above in the file `vanilla_rnn.py`. For the *forward* pass you will need Python's `for`-loop to step through time. You need to initialize the variables and matrix multiplications yourself without using high-level PyTorch building blocks. The weights and biases can be initialized using `torch.nn.Parameter`. The *backward* pass does not need to be implemented by hand, instead you can rely on automatic differentiation and use the RMSProp optimizer for tuning the weights. There is boilerplate code prepared in `train.py` which you should use for implementing the optimization procedure.

Question 3.2

As the recurrent network is implemented, you are ready to experiment with the memorization capability of the vanilla RNN. Given a palindrome of length T, use the first $T - 1$ digits as input and make the network predict the last digit. The network is *successful* if it correctly predicts the last digit and thus was capable of memorizing a small amount of information for T timesteps.

Start with short palindromes ($T = 5$), train the network until convergence and record the accuracy. Repeat this by gradually increasing the sequence length and create a plot that shows the accuracy versus palindrome length. As a sanity check, make sure that you obtain a near-perfect accuracy for $T = 5$ with the default parameters provided in `train.py`.