Assignment 1. MLPs, CNNs and Backpropagation

University of Amsterdam – Deep Learning Course

September 4, 2018

The deadline for this assignment is the 18th of September at 23:59.

In this assignment you will learn how to implement and train basic neural architectures like MLPs and CNNs for classification tasks. Modern deep learning libraries come with sophisticated functionalities like abstracted layer classes, automatic differentiation, optimizers, etc.

- To gain an in-depth understanding we will, however, first focus on a basic implementation of a MLP in numpy in exercise 1. This will require you to understand backpropagation in detail and to derive the necessary equations first.
- In exercise 2 you will implement a MLP in PyTorch and add tune its performance by adding additional layers provided by the library.
- In order to learn how to implement custom operations in PyTorch you will re-implement a batch-normalization layer in exercise 3.
- Exercise 4 aims at implementing a simple CNN in PyTorch.

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, (3) ask your question on StackOverflow or Piazza and finally (4) ask the teaching assistants.

1 MLP backprop and NumPy implementation (50 points)

Consider a generic MLP for classification tasks which is consisting of N layers[1]. We want to train the MLP to learn a mapping from the input space $\mathbb{R}^{d_0}$ to a probability mass function (PMF) over d_N classes given a dateset $\left\{\left(x^{(0),s}, t^s\right)\right\}_{s=1}^{S}$ consisting of S tuples of input vectors and targets. The superscript (0) is added to the input vectors $x^{(0)} \in \mathbb{R}^{d_0}$ as notational convenience for identifying the input as the activation of a 0-th layer. The targets $t \in \left\{ [0,1]^{d_N} \,\middle|\, \sum_{i=1}^{d_N} t_i = 1 \right\}$ could be any pmf but we assume them to be one-hot encoded in the following. Each layer l will first apply an affine mapping

$$\texttt{LinearModule.forward:}\ \left(x^{(l-1)}, W^{(l)}, b^{(l)}\right) \mapsto \tilde{x}^{(l)} := W^{(l)} x^{(l-1)} + b^{(l)}, \quad l = 1, ..., N$$

which is parameterized by weights $W^{(l)} \in \mathbb{R}^{d_l \times d_{l-1}}$ and biases $b^{(l)} \in \mathbb{R}^{d_l}$. Subsequently, nonlinearities are applied to compute activations $x^{(l)}$ from the pre-nonlinearity activations $\tilde{x}^{(l)}$. For all *hidden* layers we choose rectified linear units (ReLU), that is,

$$\texttt{ReLUModule.forward:}\ \tilde{x}^{(l)} \mapsto x^{(l)} := \text{ReLU}\left(\tilde{x}^{(l)}\right) := \max\left(0, \tilde{x}^{(l)}\right), \quad l = 1, ..., N-1.$$

[1] We are counting the output as a layer but not the input.

Since the desired output should be a valid pmf a softmax activation is applied in the *output* layer:

$$\texttt{SoftmaxModule.forward:}\ \tilde{x}^{(N)} \mapsto x^{(N)} := \text{softmax}\left(\tilde{x}^{(N)}\right) := \frac{\exp\left(\tilde{x}^{(N)}\right)}{\sum_{i=1}^{d_N} \exp\left(\tilde{x}^{(N)}\right)_i}$$

Note that both the maximum operation and the exponential function are applied element-wise when acting on a vector. A categorical cross entropy loss between the predicted and target distribution is applied,

$$\begin{aligned}\texttt{CrossEntropyModule.forward:}\ \left(x^{(N)}, t\right) \mapsto L\left(x^{(N)}, t\right) &:= -\sum_i t_i \log x_i^{(N)} \\ &= -\log x_{\arg\max(t)}^{(N)},\end{aligned}$$

where the last step holds for t being one-hot. Here $\tilde{x}_t^{(N)}$ denotes the t-th component of the softmax output.

1.1 Analytical derivation of gradients

For optimizing the network via gradient descent it is necessary to compute the gradients w.r.t. all weights and biases. In the definition of the MLP above we split the forward computation into several *modules*. It turns out that, making use of the chain rule, the gradient calculations can be split in a similar way into *gradients of modules*. For each operation performed in the forward pass, a corresponding gradient computation can be performed to *propagate the gradients back* through the network. You will first compute the partial derivatives of each module w.r.t. its inputs and subsequently put these together to a chain to get the final backpropagation computations. Your answers will be the cornerstone of the MLP NumPy implementation that follows afterwards.

Question 1.1 a) (15 points)

Compute the gradients

$$\frac{\partial L}{\partial x^{(N)}}, \quad \frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}}, \quad \frac{\partial x^{(l<N)}}{\partial \tilde{x}^{(l<N)}}, \quad \frac{\partial \tilde{x}^{(l)}}{\partial x^{(l-1)}}, \quad \frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}} \text{ and } \frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}}$$

of each module.

Hint: You can do this directly by computing vector- or matrix-valued derivatives or by calculating the scalar gradients for all components like for example

$$\left(\frac{\partial L}{\partial x^{(N)}}\right)_i = \left(\frac{\partial L}{\partial x_i^{(N)}}\right), \quad \left(\frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}}\right)_{ij} = \left(\frac{\partial x_i^{(N)}}{\partial \tilde{x}_j^{(N)}}\right) \quad \text{or} \quad \left(\frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}}\right)_{ijk} = \left(\frac{\partial \tilde{x}_i^{(l)}}{\partial W_{jk}^{(l)}}\right).$$

Note that $\frac{\partial L}{\partial x^{(N)}}$ is the backpropagation equation of the last layer, that is, it corresponds to

$$\texttt{CrossEntropyModule.backward:}\ \left(x^{(N)}, t\right) \mapsto \frac{\partial L}{\partial x^{(N)}}.$$

Question 1.1 b) (15 points)

Using the gradients of the modules calculate the gradients

$$\frac{\partial L}{\partial \tilde{x}^{(N)}} = \frac{\partial L}{\partial x^{(N)}} \frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}}, \quad \frac{\partial L}{\partial \tilde{x}^{(l<N)}} = \frac{\partial L}{\partial x^{(l)}} \frac{\partial x^{(l)}}{\partial \tilde{x}^{(l)}}, \quad \frac{\partial L}{\partial x^{(l<N)}} = \frac{\partial L}{\partial \tilde{x}^{(l+1)}} \frac{\partial \tilde{x}^{(l+1)}}{\partial x^{(l)}},$$

$$\frac{\partial L}{\partial W^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}} \quad \text{and} \quad \frac{\partial L}{\partial b^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}}$$

by *propagating back* the gradients of the output of each module to the parameters and inputs of the module. The errors $\frac{\partial L}{\partial [\cdot]}$ on the outputs of each operation occurring *on the right-hand sides* do not have to be expanded in the result since they were computed in the previous step of the backpropagation algorithm. Please give the final result in form of matrix (or in general tensor-) multiplications.

Hint: In index notation the products on the right hand side can be written in components like e.g. $\left(\frac{\partial L}{\partial x^{(l<N)}}\right)_i = \sum_j \left(\frac{\partial L}{\partial \tilde{x}^{(l+1)}}\right)_j \left(\frac{\partial \tilde{x}^{(l+1)}}{\partial x^{(l<N)}}\right)_{ji}$. *Make sure to not confuse the indices which might occur in a transposed form.*

The gradients calculated in the last exercise are the gradients occurring in the backpropagation equations:

$$\texttt{SoftmaxModule.backward}: \quad \left(\frac{\partial L}{\partial x^{(N)}}, \tilde{x}^{(N)}\right) \mapsto \frac{\partial L}{\partial \tilde{x}^{(N)}}$$

$$\texttt{ReLUModule.backward}: \quad \left(\frac{\partial L}{\partial x^{(l<N)}}, \tilde{x}^{(l<N)}\right) \mapsto \frac{\partial L}{\partial \tilde{x}^{(l<N)}}$$

$$\texttt{LinearModule.backward}: \quad \left(\frac{\partial L}{\partial \tilde{x}^{(l)}}, x^{(l-1)}, W^{(l)}\right) \mapsto \left(\frac{\partial L}{\partial W^{(l)}}, \frac{\partial L}{\partial b^{(l)}}, \frac{\partial L}{\partial x^{(l-1)}}\right)$$

The backpropagation algorithm can be seen as a form of dynamic programming since it makes use of previously computed gradients to compute the current gradient. Note that it requires all activations $x^{(l)}$ to be stored in order to propagate the gradients back from $\frac{\partial L}{\partial \tilde{x}^{(l)}}$ to $\frac{\partial L}{\partial W^{(l)}}$. In the case of a MLP, the memory cost of storing the weights exceeds the cost of storing the activations but for CNNs the latter typically make up the largest part of the memory consumption.

So far we only considered single samples being fed into the network. In practice we typically use batches of input samples which are processed by the network in parallel. The total loss

$$L_{\text{total}}\left(\left\{x^{(0),s}, t^s\right\}_{s=1}^{B}\right) := \frac{1}{B}\sum_{s=1}^{B} L_{\text{individual}}\left(x^{(0),s}, t^s\right),$$

is then defined as the mean value of the individual samples' losses. Here $L_{\text{individual}}$ is the cross entropy loss as used before which depends on $x^{(0),s}$ via $x^{(N),s}$. In addition to major computational benefits when running on GPU, performing gradient descent in batches helps to reduce the variance of the gradients.

Question 1.1 c) (5 points)

Argue how the backpropagation equations derived above change if a batchsize $B \neq 1$ is used.

1.2 NumPy implementation

For those who are not familiar with Python and NumPy it is highly recommended to get through the NumPy tutorial.

To simplify implementation and testing we have provided to you an interface to work with CIFAR-10 data in `cifar10_utils.py`. The CIFAR-10 dataset consists of 60000 32×32 color images in 10 classes, with 6000 images per class. The file `cifar10_utils.py` contains utility functions that you can use to read CIFAR-10 data. Read through this file to get familiar with the interface of the `Dataset` class. The main goal of this class is to sample new batches, so you don't need to worry about it. To encode labels we are using an one-hot encoding of labels.

Please do not change anything in this file.

Usage examples:

- Prepare CIFAR10 data:

  ```
  import cifar10_utils
  cifar10 = cifar10_utils.get_cifar10('cifar10/cifar-10-batches-py')
  ```

- Get a new batch with the size of *batch_size* from the train set:

  ```
  x, y = cifar10['train'].next_batch(batch_size)
  ```

  Variables `x` and `y` are numpy arrays. The shape of `x` is $[batch\_size, 3, 32, 32]$, the shape of `y` is $[batch\_size, 10]$.

- Get test images and labels:

  ```
  x, y = cifar10.test.images, cifar10.test.labels
  ```

Note: For multi-layer perceptron you will need to reshape `x` that each sample is represented by a vector.

Question 1.2 (15 points)

Implement a multi-layer perceptron using purely NumPy routines. The network should consist of N linear layers with ReLU activation functions followed by a final linear layer. The number of hidden layers and hidden units in each layer are specified through the command line argument `dnn_hidden_units`. As loss function, use the common cross-entropy loss for classification tasks. To optimize your network you will use the mini-batch stochastic gradient descent algorithm. Implement all modules in the files `modules.py` and `mlp_numpy.py`.

Part of the success of neural networks is the high efficiency on graphical processing units (GPUs) through matrix multiplications. Therefore, all of your code should make use of matrix multiplications rather than iterating over samples in the batch or weight rows/columns. Implementing multiplications by iteration will result in a penalty.

Implement training and testing scripts for the MLP inside `train_mlp_numpy.py`. Using the default parameters provided in this file you should get an accuracy of around 0.46 for the entire *test* set for an MLP with one hidden layer of 100 units. Carefully go through all possible command line parameters and their possible values for running `train_mlp_numpy.py`. You will need to implement each of these into your code. Otherwise we can not test your code. Provide accuracy and loss curves in your report for the default values of parameters.

2 PyTorch MLP (20 points)

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
- PyTorch for former Torch users

Question 2 (20 points)

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. Using the same parameters as in Question 1.2, you should get similar accuracy on the test set.

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 test set but we challenge you to improve this. List modifications that you have tried in the report with the results that you got using them. Explain in the report how you are choosing new modifications to test. Study your best model by plotting accuracy and loss curves.

3 Custom Module: Batch Normalization (20 points)

Deep learning frameworks come with a big palette of preimplemented operations. In research it is, however, often necessary to experiment with new custom operations. As an example you will reimplement the Batch Normalization module as a custom operations in PyTorch. This can be done by either relying on automatic differentiation (Sec. 3.1) or by a manual implementation of the backward pass of the operation (Sec. 3.2).

The batch normalization operation takes as input a minibatch $\{x^s\}_{s=1}^B$ consisting of B samples in $\mathbb{R}^C$ where C denotes the number of channels. It first normalizes each neuron's[2] value over the batch dimension to zero mean and unit variance. In order to allow for

[2] In the case of CNNs, normalization is done for each channel individually with statistics computed over the batch- and spatial dimensions.

different values it subsequently rescales and shifts the normalized values by learnable parameters $\gamma \in \mathbb{R}^C$ and $\beta \in \mathbb{R}^C$. Writing the neuron index out explicitly by a subscript, e.g. $x_i^s, \;\; i = 1 \ldots C$, Batch Normalization can be defined by:

1. compute mean: $\mu_i = \frac{1}{B} \sum\limits_{s=1}^{B} x_i^s$
2. compute variance: $\sigma_i^2 = \frac{1}{B} \sum\limits_{s=1}^{B} \left(x_i^s - \mu_i\right)^2$
3. normalize: $\hat{x}_i^s = \frac{x_i^s - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$, with a constant $\epsilon \ll 1$ to avoid numerical instability.
4. scale and shift: $y_i^s = \gamma_i \hat{x}_i^s + \beta_i$

Note that the notation differs from the one chosen in the original paper where the authors chose to not write out the channel index explicitly.

3.1 Automatic differentiation

The suggested way of joining a series of elementary operations to form a more complex computation in PyTorch is via nn.Modules. Modules implement a method `forward` which, when called, simply executes the elementary operations as specified in this function. The autograd functionality of PyTorch records these operations as usual such that the backpropagation works as expected. The advantage of using modules over standard objects or functions packing together these operations lies in the additional functionality which they provide. For example, modules can be associated with nn.Parameters. All parameters of a module or whole network can be easily accessed via `model.parameters()`, types can be changed via e.g. `model.float()` or parameters can be pushed to the GPU via `model.cuda()`, see the documentation for more information.

Question 3.1 (10 points)

Implement the Batch Normalization operation as a `nn.Module` at the designated position in the file `custom_batchnorm.py`. To do this, register γ and β as `nn.Parameters` in the `__init__` method. In the `forward` method, implement a check of the correctness of the input's shape and perform the forward pass.

3.2 Manual implementation of backward pass

In some cases it is useful or even necessary to implement the backward pass of a custom operation manually. This is done in terms of torch.autograd.Functions. Autograd function objects necessarily implement a `forward` and a `backward` method. A call of a function instance records its usage in the computational graph such that the corresponding gradient computation can be performed during backpropagation. Tensors which are passed as inputs to the function will automatically get their attribute `requires_grad` set to `False` inside the scope of `forward`. This guarantees that the operations performed inside the `forward` method are *not* recorded by the autograd system which is necessary to ensure that the gradient computation is not done twice. Autograd functions are automatically passed a *context* object in the `forward` and `backward` method which can

- store tensors via `ctx.save_for_backward` in the `forward` method
- access stored tensors via `ctx.saved_tensors` in the `backward` method
- store non-tensorial constants as attributes, e.g. `ctx.foo = bar`
- keep track of which inputs require a gradients via `ctx.needs_input_grad`

The `forward` and `backward` methods of a `torch.autograd.Function` object are typically not called manually but via the `apply` method which keeps track of registering the use of the function and creates and passes the context object. For more information you can read Extending PyTorch and Defining new autograd functions.

Since we want to implement the backward pass of the Batch Norm operation manually we first need to compute its gradients.

Question 3.2 a) (6 points)

Compute the backpropagation equations for the batch normalization operation, that is, compute

$$\frac{\partial L}{\partial \gamma} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial \gamma}, \qquad \frac{\partial L}{\partial \beta} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial \beta} \quad \text{and} \quad \frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x}.$$

Hint: The batch normalization operation is not *independent over batches so you have to factor the corresponding index in explicitly. For instance, in terms of components, we have* $\left(\frac{\partial L}{\partial \gamma}\right)_j = \sum_s \sum_i \frac{\partial L}{\partial y_i^s}\frac{\partial y_i^s}{\partial \gamma_j}$ *and* $\left(\frac{\partial L}{\partial x}\right)_j^r = \sum_s \sum_i \frac{\partial L}{\partial y_i^s}\frac{\partial y_i^s}{\partial x_j^r}$.

Having calculated all necessary equations we can implement the forward- and backward pass as a `torch.autograd.Function`. It is very important to validate the correctness of the manually implemented backward computation. This can be easily done via torch.autograd.gradcheck which compares the analytic solution with a finite differences approximation. These checks are recommended to be done in double precision.

Question 3.2 b) (3 points)

Implement the Batch Norm operation as a `torch.autograd.Function`. Make use of the *context* object described above. To save memory do not store tensors which are not needed in the backward operation. Do not perform unnecessary computations, that is, if the gradient w.r.t. an input of the autograd function is not required, return `None` for it.

Hint: If you choose to use `torch.var` for computing the variance be aware that this function uses Bessel's correction by default. Since the variance of the Batch Norm operation is defined without this correction you have to set the option `unbiased=False` as otherwise your gradient check will fail.

Since the Batch Norm operation involves learnable parameters, we need to create a `nn.Module` which registers these as `nn.Parameters` and calls the autograd function in its `forward` method.

Question 3.2 c) (1 points)

Create a `nn.Module` with γ and β as `nn.Parameters` as before. In the forward pass call the autograd function.

4 PyTorch CNN (10 points)

At this point you should have already noticed that the accuracy of MLP networks is far from being perfect. A more suitable type of architecture to process image data is the CNN. The main advantage of it is applying convolutional filters to the input images. In

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.

this part of the assignment you are going to implement a small version of the popular VGG network.

Question 4 (10 points)

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. Study the model by plotting accuracy and loss curves.

Report

We expect each student to write a report answering the questions in this assignment. Please clearly mark each answer by a heading indicating the question number. Use the NIPS LaTeX template as provided here: `https://nips.cc/Conferences/2018/PaperInformation/StyleFiles`.

Deliverables

Create ZIP archive containing your report and all Python code. Please preserve the directory structure as provided in the Github repository for this assignment. Give the ZIP file the following name: `lastname_assignment1.zip` where you insert your lastname. Please submit your deliverable through Canvas. We cannot guarantee a grade for the assignment if the deliverables are not handed in according to these instructions.

The deadline for this assignment is the 18th of September at 23:59.