Practical 1. MLPs, CNNs and Backpropagation

University of Amsterdam – Deep Learning Course

November 2, 2020

The deadline for this assignment is November 11^{th} at 23:59.

In this assignment you will learn how to implement and train basic neural architectures like MLPs and CNNs for classification tasks. Therefore you will make use of modern deep learning libraries which 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 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 layer-normalization layer in exercise 3.
- In exercise 4 you will implement 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 (45 points)

In order to keep track of what needs to be implemented, we will look at backpropagation from a modular perspective. In other words, it will be easier to think of a neural network as a series of functions (with or without adjustable parameters θ) rather than as a network with neurons as nodes. In a traditional sketch of a neural network, it is not as easy to see that within each node, an activation function is being applied to the result of the linear transformation. By making each of these operations a separate module, it will become clear how backpropagation works in the general setting. A simple example of such a modular representation is shown in Figure 1. Note that in the forward pass, certain modules require not only features from the previous layer, but also a set of parameters. *Backpropagation is an algorithm that allows us to update these parameters using gradient descent in order to decrease the loss.*

1.1 Evaluating the Gradients

In the forward pass, some input data is injected into a neural network. The features flow through the neural network blissfully, changing dimensionality along the way. The number of features in a hidden layer corresponds to the number of neurons in the corresponding layer. In the final layer, some sort of output is generated. In the example of a classification problem, one might consider using softmax in the output layer (as in Figure 1). For

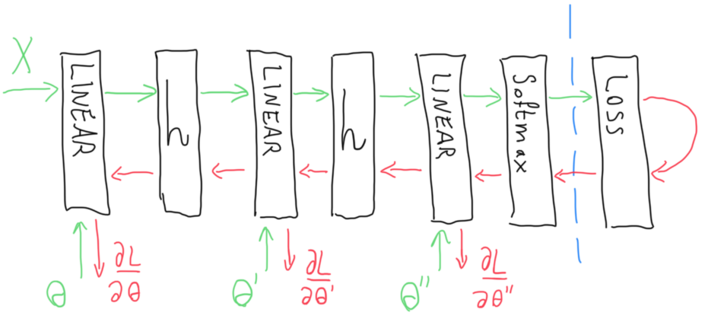

X
LINEAR
h
LINEAR
h
LINEAR
Softmax
LOSS
θ
$\frac{\partial L}{\partial \theta}$
θ'
$\frac{\partial L}{\partial \theta'}$
θ''
$\frac{\partial L}{\partial \theta''}$

Figure 1. Example of an MLP represented using modules

training, we will require a *loss function* L, a measure for how badly the neural network has performed. The lower the loss, the better the performance of our model on that data. Note that our model has parameters θ. In a traditional *linear layer*, the parameters are the weights and biases of a linear transformation. Also note that a conventional activation function (e.g. ReLU) has no parameters that need to be optimized in this fashion.

In general, we do not want to send in one data point at a time, but rather multiple in a *batch*. Let the number of data points in a batch be represented by S and the number of features (or dimensions) in each data point by M. Concatenating all the data points in a single batch as row-vectors, we obtain the feature matrix $\boldsymbol{X} \in \mathbb{R}^{S \times M}$. No matter which operation we wish to perform, the batch dimension should always stay the same.

In a simple linear module, the number of features per data point will usually vary. For example, in a linear transformation from a layer with M neurons to the next layer with N neurons, the number of features goes from M to N. In other words, an input to this linear transformation has M element, and the output has N, which is just like an ordinary matrix multiplication! For one data point $\boldsymbol{z} \in \mathbb{R}^M$ being transformed into $\boldsymbol{v} \in \mathbb{R}^N$ (i.e. batch size of 1) the linear transformation looks like $\boldsymbol{v} = \boldsymbol{W}\boldsymbol{z} + \boldsymbol{p}$, where $\boldsymbol{W} \in \mathbb{R}^{N \times M}$ and $\boldsymbol{p} \in \mathbb{R}^N$. If we transpose this whole equation we get: $\boldsymbol{v}^\top = \boldsymbol{z}^\top \boldsymbol{W}^\top + \boldsymbol{p}^\top$. Note that in programming, the most fundamental array is a list, which is best represented by a row-vector. Instead we rewrite the equation with row-vectors $\boldsymbol{y} = \boldsymbol{v}^\top, \boldsymbol{x} = \boldsymbol{z}^\top, \boldsymbol{b} = \boldsymbol{p}^\top$ and we obtain the much nicer looking: $\boldsymbol{y} = \boldsymbol{x}\boldsymbol{W}^\top + \boldsymbol{b}$. Now we can handle multiple data points at once with input feature matrix $\boldsymbol{X} \in \mathbb{R}^{S \times M}$, the output features are then given by $\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{W}^\top + \boldsymbol{B} \in \mathbb{R}^{S \times N}$. The weight matrix is $\boldsymbol{W}$ and the bias row-vector $\boldsymbol{b} \in \mathbb{R}^{1 \times N}$ is tiled S times into $\boldsymbol{B} \in \mathbb{R}^{S \times N}$. (Note that $B_{ij} = b_j$.)

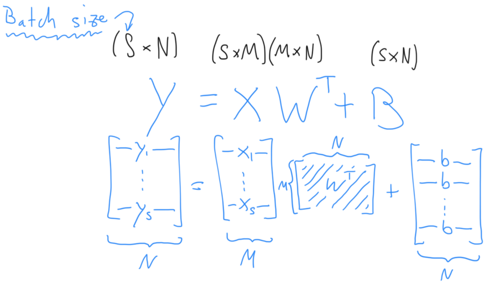


Figure 2. The programming convention used in DL assumes the batch dimension comes first.

For a linear module that receives input features $\boldsymbol{X}$ and has weight and biases given by $\boldsymbol{W}$ and $\boldsymbol{b}$ the forward pass is given by $\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{W}^\top + \boldsymbol{B}$. In the backward pass (backpropagation), the gradient of the loss with respect to the output $\boldsymbol{Y}$ will be supplied to this module by the subsequent module.

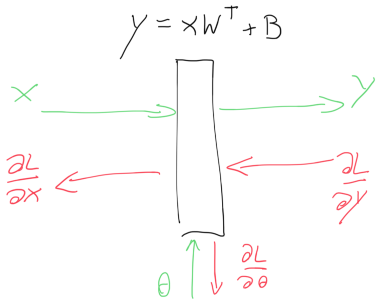

$Y = XW^T + B$
X
Y
$\frac{\partial L}{\partial X}$
$\frac{\partial L}{\partial Y}$
θ
$\frac{\partial L}{\partial \theta}$

Figure 3. A linear module with arrows denoting the forward and backward passes.

Question 1.1 a) Linear Module (15 points)

Consider a linear module as described above. The input and output features are labeled as $\boldsymbol{X}$ and $\boldsymbol{Y}$, respectively. Find closed form expressions for

$$\frac{\partial L}{\partial \boldsymbol{W}}, \frac{\partial L}{\partial \boldsymbol{b}}, \frac{\partial L}{\partial \boldsymbol{X}}$$

in terms of the gradients of the loss with respect to the output features $\frac{\partial L}{\partial \boldsymbol{Y}}$ provided by the next module during backpropagation. Assume the gradients have the same shape as the object with respect to which is being differentiated. E.g. $\frac{\partial L}{\partial \boldsymbol{W}}$ should have the same shape as $\boldsymbol{W}$, $\frac{\partial L}{\partial \boldsymbol{b}}$ should also be a row-vector just like $\boldsymbol{b}$ etc.

In this question, please provide the final answers in the form of matrix and vector operations. (You might find the ones-vector $\mathbf{1}$ to be useful.)

Recall that, in order to apply the chain rule correctly, you must consider partial derivatives with respect to all elements of the matrix in question. For example,

$$\frac{\partial L}{\partial \boldsymbol{W}} \Rightarrow \left[\frac{\partial L}{\partial \boldsymbol{W}}\right]_{ij} = \frac{\partial L}{\partial W_{ij}} = \sum_{m,n} \frac{\partial L}{\partial Y_{mn}} \frac{\partial Y_{mn}}{\partial W_{ij}}.$$

Question 1.1 b) Activation Module (5 points)

Consider an *element-wise* activation function h. The activation module has input and output features labeled by $\boldsymbol{X}$ and $\boldsymbol{Y}$, respectively. I.e. $\boldsymbol{Y} = h(\boldsymbol{X}) \Rightarrow Y_{ij} = h(X_{ij})$. Find a closed form expression for

$$\frac{\partial L}{\partial \boldsymbol{X}}$$

in terms of the gradient of the loss with respect to the output features $\frac{\partial L}{\partial \boldsymbol{Y}}$ provided by the next module. Assume the gradient has the same shape as $\boldsymbol{X}$.

In this question, please provide the final answers in the form of matrix and vector operations.

Hint: You might need to write your answer in terms of a Hadamard product (element-wise product of matrices of identical size). It is defined as follows: $[\boldsymbol{A} \circ \boldsymbol{B}]_{ij} = A_{ij}B_{ij}$.

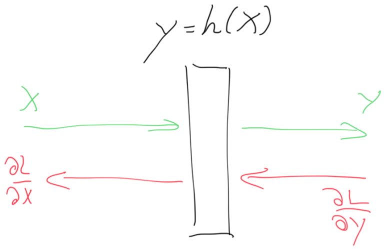

$y = h(x)$
x
y
$\frac{\partial L}{\partial x}$
$\frac{\partial L}{\partial y}$

Figure 4. An activation function module with arrows denoting the forward and backward passes.

The final module before the loss evaluation is responsible for turning the jumbled-up data into predictions for C categories. *Softmax* takes an ordered set of numbers (e.g list or vector) as an input, and returns the same sized set with a corresponding "probability" for each element. Therefore, one must have alread ensured that this module receives data with a number of features equal to the number of categories C. We would like to generalize this to a batch of many such ordered lists (row vectors). The softmax module is defined for feature matrices $\boldsymbol{X}, \boldsymbol{Y} \in \mathbb{R}^{S \times C}$ as follows:

$$Y_{ij} = [\text{softmax}(\boldsymbol{X})]_{ij} := \frac{e^{X_{ij}}}{\sum_k e^{X_{ik}}}.$$

Finally, we must specify a loss function for training in order to compare the outputs from our final module (e.g. softmax) to our *targets* $\boldsymbol{T} \in \mathbb{R}^{S \times C}$, also referred to as *labels*. The rows are the target row-vectors $\boldsymbol{t} \in \mathbb{R}^{1 \times C}$ and are usually one-hot, meaning that all elements are 0 except for the one corresponding to the correct label, which is set to unity. This can be generalized even further such that $\sum_j t_j = \sum_j T_{kj} = 1$, for all samples k. Let us pick the *categorical cross entropy*. The loss of a sample i in the batch is then given by:

$$L_i := -\sum_k T_{ik} \log(X_{ik})$$

The final loss is the mean over all the samples in the batch. Therefore, $L = \frac{1}{S} \sum_i L_i$.

Question 1.1 c) Softmax and Loss Modules (10 points)

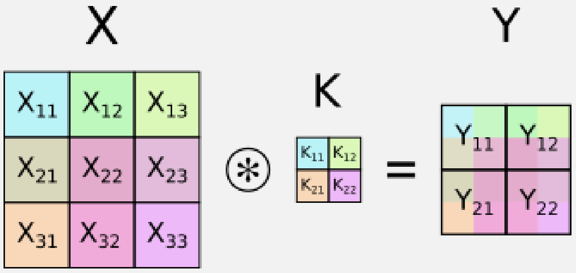
i. Consider a softmax module such that $Y_{ij} = [\text{softmax}(\boldsymbol{X})]_{ij}$, where $\boldsymbol{X}$ is the input and $\boldsymbol{Y}$ is the output of the module. Find an expression for $\frac{\partial L}{\partial \boldsymbol{X}}$ in terms of $\frac{\partial L}{\partial \boldsymbol{Y}}$. You may keep your result in index notation for this sub-task only.

ii. The gradient that kicks the whole backpropagation algorithm off is the one for the loss module itself. The loss module for the categorical cross entropy takes as input $\boldsymbol{X}$ and returns $L = \frac{1}{S} \sum_i L_i = -\frac{1}{S} \sum_{ik} T_{ik} \log(X_{ik})$. Find a closed form expression for $\frac{\partial L}{\partial \boldsymbol{X}}$. Write your answer in terms of matrix operations.

As a bonus question, you can try to derive the gradients for a simple convolution layer. A convolution layer is a special linear layer using a structured weight matrix which shares the same weights (the convolutional kernel) over different spatial positions in the input.

Question 1.1. d) (Bonus) (3 points)

Given the following convolution layer (with no padding and stride 1), where $\mathbf{X}$ is an input image, $\boldsymbol{K}$ the kernel



give

1. The forward propagation equation for Y_{21} (in terms of the input and kernel components, i.e. X_{jk} and K_{lm})
2. The gradient $\frac{\partial \mathcal{L}}{\partial K_{11}}$
3. The gradient $\frac{\partial \mathcal{L}}{\partial X_{12}}$

Assume that the output gradient $\frac{d\mathcal{L}}{\partial \mathbf{Y}}$ is given.

1.2 NumPy implementation

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

Dataset 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.

Note: *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
  ```

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

ELU activation function Neural networks usually alternate linear layers with a non-linear operation h as you have seen in the previous question. The function $h : \mathbb{R} \to \mathbb{R}$

is applied pointwise on the entries of the input $\boldsymbol{X}$, i.e. $[h(\boldsymbol{X})]_{si} := h(X_{si})$. In this assignment, we will use the *Exponential Linear Unit* (ELU), which is defined as:

$$\mathrm{ELU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ e^x - 1 & \text{if } x < 0 \end{cases}$$

Question 1.2 (15 points)

Implement a multi-layer perceptron using purely NumPy routines. The network should consist of N linear layers with ELU 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 0.48 using ELU activation function 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 **(23 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 2.1 (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 activation functions, 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 0.52* 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**.

Question 2.2 (3 points)

Briefly discuss about the benefits and/or the drawbacks of using the function *Tanh* instead of the activation function ELU you used before.

3 Custom Module: Layer Normalization (22 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 Layer 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 layer normalization has been proposed in order to overcome some of the limitations of the batch normalization when the batch size is too small to extract robust statistics about the features. In layer normalization, mean and variance are computed independently for each element of the batch by aggregating over the features dimension. Given the input $\boldsymbol{X} \in \mathbb{R}^{S \times M}$:

1. compute mean: $\mu_s = \frac{1}{M} \sum_{i=1}^{M} X_{si}$
2. compute variance: $\sigma_s^2 = \frac{1}{M} \sum_{i=1}^{M} (X_{si} - \mu_s)^2$
3. normalize: $\hat{X}_{si} = \frac{X_{si} - \mu_s}{\sqrt{\sigma_s^2 + \epsilon}}$, with a constant $\epsilon \ll 1$ to avoid numerical instability.
4. scale and shift: $Y_{si} = \gamma_i \hat{X}_{si} + \beta_i$, where $\gamma_i, \beta_i \in \mathbb{R}$ are learnable parameters

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 (8 points)

Implement the Layer Normalization operation as a `nn.Module` at the designated position in the file `custom_layernorm.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 Layer Norm operation manually we first need to compute its gradients.

Question 3.2 a) (6 points)

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

$$\frac{\partial L}{\partial \gamma}, \quad \frac{\partial L}{\partial \boldsymbol{\beta}} \quad \text{and} \quad \frac{\partial L}{\partial \boldsymbol{X}} \; .$$

Note that $\frac{\partial L}{\partial \mathbf{Y}}$ is the gradient back propagated from the following layer so you do not need to expand it further.

Hint 1: to apply the chain rule correctly, you must consider partial derivatives with respect to all elements of $\boldsymbol{Y}$, i.e.:

$$\frac{\partial L}{\partial \gamma} \Rightarrow \left[\frac{\partial L}{\partial \gamma}\right]_i = \frac{\partial L}{\partial \gamma_i} = \sum_{s,j} \frac{\partial L}{\partial Y_{sj}} \frac{\partial Y_{sj}}{\partial \gamma_i}$$

$$\frac{\partial L}{\partial \boldsymbol{\beta}} \Rightarrow \left[\frac{\partial L}{\partial \boldsymbol{\beta}}\right]_i = \frac{\partial L}{\partial \beta_i} = \sum_{s,j} \frac{\partial L}{\partial Y_{sj}} \frac{\partial Y_{sj}}{\partial \beta_i}$$

$$\frac{\partial L}{\partial \boldsymbol{X}} \Rightarrow \left[\frac{\partial L}{\partial \boldsymbol{X}}\right]_{ri} = \frac{\partial L}{\partial X_{ri}} = \sum_{s,j} \frac{\partial L}{\partial Y_{sj}} \frac{\partial Y_{sj}}{\partial X_{ri}} \, .$$

Hint 2: before computing the backpropagation equations above, compute the partial derivatives $\frac{\partial \boldsymbol{Y}}{\partial \boldsymbol{\gamma}}$, $\frac{\partial \boldsymbol{Y}}{\partial \boldsymbol{\beta}}$ and $\frac{\partial \boldsymbol{Y}}{\partial \boldsymbol{X}}$.

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 Layer 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 Layer Norm operation is defined without this correction you have to set the option `unbiased=False` as otherwise your gradient check will fail.

Since the Layer 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.

Question 3.2 d) (4 points)

Now compare the Layer Normalization approach with the Batch Normalization, explaining the different problem they address and their respective limitations. Finally, think about the different effect of the batch size on both Layer Normalization and Batch Normalization.

Name	Kernel	Stride	Padding	Channels In/Out
conv0	3×3	1	1	3/64
PreAct1	3×3	1	1	64/64
conv1	1×1	1	0	64/128
maxpool1	3×3	2	1	128/128
PreAct2_a	3×3	1	1	128/128
PreAct2_b	3×3	1	1	128/128
conv2	1×1	1	0	128/256
maxpool2	3×3	2	1	256/256
PreAct3_a	3×3	1	1	256/256
PreAct3_b	3×3	1	1	256/256
conv3	1×1	1	0	256/512
maxpool3	3×3	2	1	512/512
PreAct4_a	3×3	1	1	512/512
PreAct4_b	3×3	1	1	512/512
maxpool4	3×3	2	1	512/512
PreAct5_a	3×3	1	1	512/512
PreAct5_b	3×3	1	1	512/512
maxpool5	3×3	2	1	512/512
linear	–	-	-	512/10

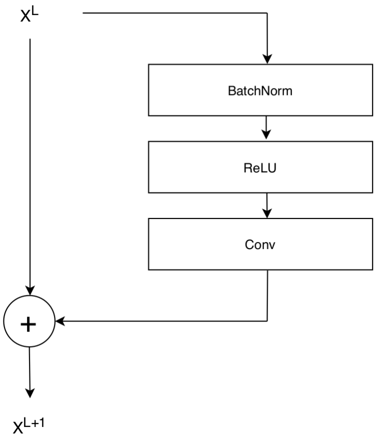


Figure 5. Left: Specification of ConvNet architecture. The *conv* blocks represent a single 2D-convolution without an activation function, and *PreAct* a pre-activation ResNet block depicted on the RHS. A BatchNorm layer and ReLU activation is applied before the final linear layer. **Right:** A simplified pre-activation ResNet block consisting of a BatchNorm layer followed by a ReLU activation function and a 2D-convolutional layer.

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. In this part of the assignment you are going to implement a small version of the popular VGG network with added skip connections in-between layers.

Question 4 a) (10 points)

Implement the ConvNet specified in the table in Figure 5 inside `convnet_pytorch.py` by following the instructions. The architecture resembles the VGG13 net with added pre-activation skip connections you have seen during the notebook tutorial. A pre-activation ResNet block is depicted on the RHS in Figure 5. The 1x1 convolutions before the MaxPool operations are used to upscale the channel dimension.

Implement training and testing procedures for your model in `train_convnet_pytorch.py` by following instructions inside the file. Use the Adam optimizer with the given default learning rate of 1e-4. Use default PyTorch parameters to initialize convolutional and linear layers. With all default parameters you should get around *0.79* accuracy or higher on the test set. Study the model by plotting accuracy and loss curves.

Question 4 b) (Bonus) Transfer Learning (5 points)

Find a PyTorch model trained on ImageNet online. Implement the architecture in your code, load the pre-trained weights and fine tune the model on the CIFAR10 dataset used before for the same task of image classification. Test the model and report your findings. *Hint: Take a look at torchvision's model library for pre-trained deep models on ImageNet.*

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 was provided here: `hhttps://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 November 11$^{\text{th}}$ at 23:59.