Deep Learning Workshop, 27.03 – 30.03.2023

Assignment 03: Residual Neural Networks (ResNet)

The main goals of this third programming session are to:

a) Work with data augmentation tools provided by Torchvision [MR10].

b) Implement a ResNet cell by hand and use our implementation to build a ResNet to classify images containing ants and bees.

c) Use Transfer Learning with a pretrained network to simultaneously reduce the training duration and improve the network's performance.

Short summary of the theoretical session

Nomenclature

Before we start to implement our own ResNet cells, we need to distinguish between two different definitions of Residual Neural Networks. In 2016 He et al. introduced in their original paper "Deep Residual Learning for Image Recognition" [He+16a] the identity mapping framework **before** (**B**) the activation function (canonical form with a one-step skip):

$$y^{\ell+1} := \sigma\left(P_\ell^{\ell+1} y^\ell + W_\ell^{\ell+1} y^\ell + b^\ell\right), \quad \ell = 0, \dots, L-1 \tag{B}$$

where σ is a (nonlinear) activation function (e.g. ReLU) and $(P_\ell^{\ell+1})_{l=0,\dots,L-1}$ are projection operations to match the feature vectors' shapes. In their followup paper "Identity Mappings in Deep Residual Networks" [He+16b] the authors suggested multiple different identity mapping frameworks, e.g. also one with a skip **after** (**A**) the activation function, i.e.

$$y^{\ell+1} := P_\ell^{\ell+1} y^\ell + \sigma\left(W_\ell^{\ell+1} y^\ell + b^\ell\right), \quad \ell = 0, \dots, L-1 \tag{A}$$

which leads to better results and a nice connection to differential equations, as you have seen in the theoretical session and shown e.g. in [RH18]. Unfortunately, in PyTorch all networks still use the framework (**B**) and thus, we will work with both versions during this programming assignment. Fore more details on the PyTorch implementations of ResNet, you can find under PyTorch ResNet the implementation of all available ResNets and PyTorch Training explains the internal training procedure.

Intuition

The following is based on [Zha+21] and serves as a short summary of the theoretical part. In their original paper He at al. introduced ResNets to overcome the *degradation problem*, i.e. with increasing network depths the accuracy saturates and afterwards degrades rapidly [He+16a]. This inability to learn is not caused by overfitting and adding more layers to already deep networks does not improve the training error.

To motivate ResNets and gain some intuitive understanding of the new structure, let us begin by reconsidering our problem. Given a dataset $\mathbf{X} \subset \mathcal{X}, \mathbf{y} \subset \mathcal{Y}$ of training examples, our goal is to determine a feasible prediction function $f : \mathcal{X} \to \mathcal{Y}$, i.e. $f \in \mathcal{H}$ for a suitable function space[1] $\mathcal{H}$, which minimizes a given loss function L

$$\min_{f \in \mathcal{H}} L(\mathbf{X}, \mathbf{y}, f). \tag{1}$$

It is reasonable to assume that using a more powerful architecture $\mathcal{H}'$ with $\mathcal{H} \subsetneq \mathcal{H}'$ we can hope for a better function approximator $f \in \mathcal{H}'$ in the sense of

$$\min_{f \in \mathcal{H}} L(\mathbf{X}, \mathbf{y}, f) \geq \min_{f \in \mathcal{H}'} L(\mathbf{X}, \mathbf{y}, f). \tag{2}$$

He et al. implemented this idea by allowing identity skips of the form (**B**) and (**A**).

Thus, they created a bigger set of feasible function approximators while not introducing any additional parameters and simplifying the learning of identity mappings by enabling residual learning. Additionally, we approach the problem of vanishing and exploding gradients caused by the chain rule by adding an identity term.

Data Augmentation

Data augmentation is a technique used in machine learning to artificially increase the size of a dataset by applying various transformations to the existing data. This can help to improve the performance of a model by providing it with more diverse and robust training data. Some common techniques used for data augmentation include:

- Random cropping: Randomly cropping a portion of an image and using that as a new sample.
- Random flipping: Flipping an image horizontally or vertically to create a new sample.
- Random rotation: Rotating an image by a random angle to create a new sample.
- Color jittering: Randomly changing the brightness, contrast, and/or saturation of an image to create a new sample.
- Gaussian noise: Adding random noise sampled from a Gaussian distribution to an image or feature to create a new sample.
- Random scaling: Randomly scaling an image by a certain factor to create a new sample.
- Elastic transformations: Applying a non-rigid transformation to an image, such as a random deformation, to create a new sample.
- Cutout: Randomly masking out some portion of the image, to make the model robust to occlusion and learn more robust features.

[1] Usually we pick a parametrized model and the function space is the corresponding parameter space. Thus, we transform a (potentially) infinite-dimensional optimization problem to a very high-, but finite-dimensional optimization problem.

The idea behind data augmentation is to artificially increase the diversity of the data, so that the model is more robust to changes in the data distribution, and can generalize better to new unseen data. Keep in mind, the appropriate data augmentation techniques to use will depend on the specific task and dataset, and it's always a good idea to try multiple techniques and compare their effectiveness.

Programming

In this task we will implement a classifier to predict whether an image contains ants or bees. Every image shows either ants or bees, but never both simultaneously.

1) Data Preparation: ·Data Augmentation using PyTorch

In this exercise we will work with the file `resnet.py`. Open the folder `./data`, you will find two subfolders named `./train` and `./test` each containing two subfolders with images of ants and bees.[2] In this task we will learn how we can automatically extract a dataset from a subfolder named after the label and filled with images. Familiarize yourself with the data.

a) [DL] Implement the constructor of the `DataLoaderResNet` class by using the `datasets.ImageFolder` function. Split the training data again into training and validation samples.

b) [DL] *Only for the training and validation data:* Take a look at the list of available transformes in Torchvision Transformer Torchvision and the provided examples Transformer Examples. Choose one or two transformers to augment your training.

c) [DL] *For the entire dataset:* The pretrained network that we will use in the last task internally normalizes the data. In order to compare our network with the pretrained network, we also normalize the data by using the `transforms.Normalize` function with the parameters:

```
transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]),
```

The parameters correspond to the mean respectively the standard deviation of the ImageNet[3] [Den+09] dataset from which our images are taken.

d) [DL] *For the entire dataset:* Ensure that all of your tensors have a size of `224`. You can either automatically implement a resize operation in one of the data augmentation functions or use `transforms.Resize`.

2) Custom ResNet: Connection to your former self.

Open the file `resnet.py`, we will complete the template of the `ResNetCell` class and the `ResNet` class. We will implement version (**A**), recall the formula:

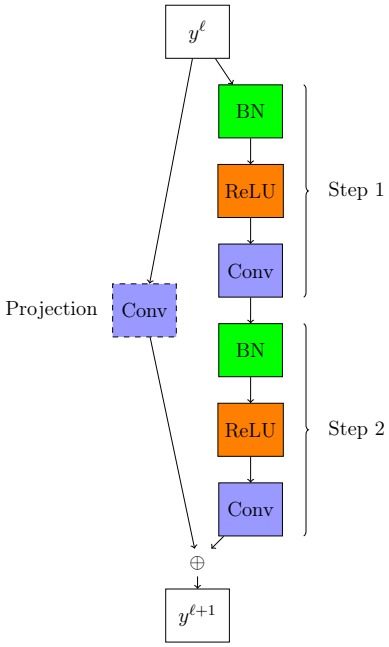
$$y^{\ell+1} := P_\ell^{\ell+1} y^\ell + \sigma\left(W_\ell^{\ell+1} y^\ell + b^\ell\right), \quad \ell = 0, \dots, L-1 \tag{\textbf{A}}$$

but we will slightly modify the order of weight and activation layers to match the architecture which performed best in the followup paper [He+16b]. *Don't worry*, we will guide you through the architecture step-by-step.

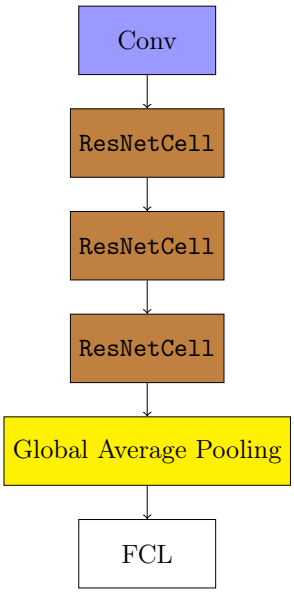
a) [N] First, we implement a generic ResNet cell with a two-step skip. As this is a relatively tricky task, we divide the implementation into multiple subtasks which are also visualized in Figure 1a:

- Implement Step 1 shown in Figure 1a. Use a convolutional operation with `in_channels` as the number of input channels and `out_channels` as the number of output channels. As in the original paper, we use a kernel size of $(3, 3)$ and a stride of 2. To avoid problems with the identity connection, use a padding of 1.

[2] The dataset is taken from the example Transfer Learning in PyTorch.
[3] https://www.image-net.org/



(a) Architecture of our custom ResNet-cell. BN is a batch normalization layer, ReLU is a layer with a ReLU activation function and Conv is a convolutional layer.



(b) Architecture of our custom ResNet-cell. Conv is an abstract convolutional layer (including batch normalization, pooling and an activation function), `ResNetCell` is a custom ResNet-cell of Task 2 and FCL is a fully connected layer.

Figure 1: ResNet cell and ResNet architecture we will implement in the second exercise.

- Implement Step 2 shown in Figure 1a. Use a convolutional operation with `out_channels` as the number of input channels and `out_channels` as the number of output channels. As in the original paper, we use a kernel size of $(3, 3)$ and a stride of 2. To avoid problems with the identity connection, use a padding of 1.
- Implement the residual connection shown in Figure 1a. How do you have to adjust your code if the number of input and output channels of your `ResNetCell` differ (see Projection step)? *Hint:* Projection operations are implemented by using a 2D convolutional operation with a kernel size of 1. How do you have to adjust the stride of the operation to match the output size of the first two steps?
- Implement the `forward` method of your custom `ResNetCell`. Remember the order of operations in equation (**A**).

b) [N] Next, we will implement a (shallow) Residual Neural Network by completing the template of the `ResNet` class. Because a ResNet consists of several different parts, we also divide this exercise into smaller subtasks, again the architecture of our network is visualized in Figure 1b:

- Implement the first convolutional layer, again the link Convolution arithmetic might be helpful if you want to visually see the effect of the different parameters. As in the original architecture, use a kernel size of 7, a stride of 2 and a padding of 3. Choose the number of input channels based on the number of input channels of our images. Add a max pooling layer with a kernel size of 3, a stride of 2 and a padding of 1. Afterwards implement a batch normalization layer [IS15] and the last step should be a ReLU activation function. For the number of output channels choose a relatively small number (e.g. 8). If your computational power can ensure a moderate training time, feel free to increase the number of channels.

Again, `nn.Sequential` might be your best friend.

- Add three cells of your custom `ResNetCell` and increase the number of output channels for each of them (e.g. 16, 32, 64).
- Subsequently, append one global average pooling layer by using the PyTorch command `nn.AdaptiveAvgPool2d((1, 1))` and flatten the output afterwards.
- Finally, add the fully connected layer to your network. What is the output's dimension?
- Execute the steps in the `forward` function of the class.

We want to reuse our `NetworkTrainer` framework for both, our custom ResNet and the pretrained ResNet of the next task. As both of them are technically inherited by the same Baseclass, but practically implemented in different ways (cf. next task) we have implemented a factory class named `ResNetFactory` which wraps both of the constructors into a uniform interface that can be called by the `NetworkTrainer`. *Don't worry* about the factory method, the constructor works exactly like the constructor of your `ResNet` class, you can use the same network parameters.

c) [NT] (~ 30 s, 10 epochs) Train your network using the `NetworkTrainer`. Again, use **Adam** [KB14] as optimizer, the cross entropy as loss function and your custom data loader `DataLoaderResNet` of task one.
Note: The only difference is that you have to pass the class `ResNetFactory` into the trainer instead of the `ResNet` class.

d) [NU] Use the `NetworkUser` to evaluate your network's performance on the testing data and visualize the results.

3) Transfer Learning: Let someone else do the work.[4]

Based on the number of feature channels you used in the previous exercise, the chances are high that your network completely overfitted the data. This is reasonable, as we provided far too few images to convincingly train a ResNet. In practice, we face two types of problems: a) we usually do not have enough data and b) do not have the computational power to train large classifiers with architectures like **ResNet152** [He+16a] from scratch.

A powerful method to benefit from pretrained networks is called **Transfer Learning**: Instead of training a classifier fully from scratch, we use a pretrained network for feature extraction and only learn the compression and classification layer by ourselves. Thus, we have to (optionally) interchange the first layer (to match our own input data) and (necessarily) the last layer (to customize the feature selection and classification as well as to fit our number of labels). In this task, we just have to implement the latter, as PyTorch's pretrained networks were already trained on the ImageNet dataset and therefore have the same input dimensions.

Disclaimer: As mentioned in the beginning, PyTorch's networks use the ResNet architecture (**B**) and thus, we are now using *two different ResNet structures*.[5] If you are interested, you can find all implementation details about ResNets in PyTorch here. For this task it is sufficient to know that PyTorch ResNets use version (**B**) while we implemented version (**A**).

a) [N] Open the page PyTorch pretrained ResNets and pick a model of your choice (the solution proposal uses **ResNet18**). In your file `resnet.py` you should find a class named `ResNetPretrained` with an empty `__new__` function. Complete this function by loading the pretrained network of

[4] This task is inspired by the excellent examples https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html and https://pytorch.org/tutorials/beginner/finetuning_torchvision_models_tutorial.html provided by the amazing PyTorch community and contributors.

[5] If you need a pretrained network with architecture (**A**), you can use TensorFlow [Mar+15] and pick a model from here. Credit belongs to Evelyn, thank you very much for pointing this out.

your choice and use a parameter to load precomputed weights (check available options in the network's description).

b) [N] Next, change the last layer of the network by adding the commands

```
    _set_parameter_requires_grad(model, feature_extract)
    num_features = model.fc.in_features
    model.fc = nn.Linear(num_features, num_classes)
return model
```

to a learnable fully connected layer. The first line of the code snippet above is really important: If we want to use the pretrained network only for *feature extraction*, we can turn off the backward pass for all layers except for the last one with `feature_extract = True`. Freezing the parameters of the pretrained model tremendously accelerates the training. If you want to *fine-tune* the pretrained network even further, i.e. enable the training for all parameters, use the flag `feature_extract = False`.
Note: We do not use a standard constructor in this subtask, as the `__init__` function in Python does not have a return value by default. Because we want to embed our pretrained model into the `ResNetFactory` class, we need a return value.

c) [NT] ($\sim$ 90 s, 10 epochs) Train your network by using the `NetworkTrainer`. Use the flag `"pretrained": True` inside your network parameters to obtain the pretrained model by the `ResNetFactory` class. Why does the training of the pretrained network with frozen layers take longer than the training of our own ResNet implementation?

d) [NU] Visualize and compare the network's performance with our own ResNet by using the `NetworkUser`.

4) **BONUS: Visualizing Kernels and Advanced Ideas.**
We suggest to skim all the exercises first, spend some time to think about the discussion questions (i.e. part a) might be a good start) and afterwards pick a task that you want to implement.

a) As in the previous two exercises: Implement a method to compute the correct predictions in percentage points. Think closely about the number of labels we have, can you simplify the function used in Assignment 02?

b) In this subtask, we will try to follow the idea of [ZF13] to visualize the weights of the convolutional kernels and the corresponding feature maps after forwarding an image.[6] Thus, we will see what the network internally sees. For the sake of simplicity, we will just consider the plain convolutional layers and ignore ReLU layers, Batch normalization layers and residual connections.

As this task is a bit tricky, we will guide you closely through the process. To get better results, we will work with the pretrained network. We work inside the `03_main.py` file of the ResNet assignment. You will just implement helper functions that are called inside the `bonus_visualization` function in the right order. If you are interested, feel free to check the docstring of the function and the implementation.

i. We start by extracting the convolutional layers of our (pretrained) network. Complete the template of the function `extract_conv_layers` inside the `03_main.py` file. Iterate through the layers of the network and save the weights of all convolutional layers inside a list. Save the layers themselves in another list and return both lists. We will use the list of layers to perform a simplified forward pass later on and the list of weights to visualize the network's kernels.

[6] Some of the steps are inspired by the tutorial Visualization Tutorial.

ii. Next, implement the function `vis_conv_kernels`. This function should take the model's weights and visualize the weights of the *first layer* as an image. Save that image to the folder `./bonus_vis/kernels`.

iii. Complete the template of the `forward_image` function. This function takes our list of convolutional layers and an input image as parameters and performs the *simplified forward pass*. Pass the image into the first layer and afterwards use the output of the previous layer as the input of the current layer. Save all intermediate results in a list, i.e. call `append` in every step of the iteration. Return the list of results.

iv. Finally, implement the `vis_feature_map` function which takes the results of the simplified forward pass as a parameter and shows the features maps of each convolutional layer.
Hint: As each layer consists of 64 or more channels, we only focus on the first 64 layers and ignore the remaining layers. Create an 8×8 grid containing a small image for each channel and save it into the folder `./bonus_vis/feature_maps`.

v. Uncomment the last line of code

```
bonus_visualization(ResNetFactory(True, num_classes=2), data_loader)
```

inside the `__main__` function of your `03_main.py` file, comment out the training part of the previous exercises and run your code. What do you observe? Interpret what the network sees.

Note: The code also visualizes the image which is forwarded. *Don't worry* if the colors might look strange: We use our custom Data Loader which still includes the global normalization layer.

*c) [N] Make the number of kernels in our `ResNet` class generic and implement a rule to increase the number of kernels with each layer (e.g. by doubling).

*d) [DL] So far, the `__call__` function of all of our data loaders has been the same. Implement an abstract class `DataLoaderAbstract` which generically implements the `__call__` function and let your custom data loader inherit that method. Think about advantages and disadvantages of using an abstract base class.

*e) [NT] Modify the `NetworkTrainer` class such that it saves all models and/or only the best one instead of the model of the last training cycle. You can decide whether it means best in terms of training or testing data.
Hint: Cache the error and implement a condition-based saving method.

*f) [NT] Take a closer look at the implementation of the `NetworkTrainer` class, focus especially on similarities between the training and validation loop. Implement a simplified version by using an additional loop and a condition-based `with` statement. Discuss (dis-)advantages of the modified code.
Hint: `with torch.set_grad_enabled(phase == 'train')` might be helpful.

*g) [N] Go back to Assignment 02: Implement a pretrained ResNet as the CNN part with shared weights. How does a pretrained feature extractor affect your network's performance and the training duration? What happens if you also enable the pretrained network to learn (i.e. turn on the gradients)?

*h) [N] Read the documentation of `nn.LazyLinear` and `nn.LazyConv2D`. What is the difference compared to the layers we used so far? Reimplement your own ResNet using the lazy layers. Do you see any (dis-)advantages?

References

[Den+09] Jia Deng et al. "ImageNet: A large-scale hierarchical image database". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[He+16a] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[He+16b] Kaiming He et al. "Identity Mappings in Deep Residual Networks". In: *CoRR* abs/1603.05027 (2016). arXiv: 1603.05027. URL: http://arxiv.org/abs/1603.05027.

[IS15] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *CoRR* abs/1502.03167 (2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

[KB14] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[Mar+15] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[MR10] Sébastien Marcel and Yann Rodriguez. "Torchvision the Machine-Vision Package of Torch". In: *Proceedings of the 18th ACM International Conference on Multimedia*. MM '10. Firenze, Italy: Association for Computing Machinery, 2010, pp. 1485–1488. ISBN: 9781605589336. DOI: 10.1145/1873951.1874254. URL: https://doi.org/10.1145/1873951.1874254.

[Pas+19] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[RH18] Lars Ruthotto and Eldad Haber. "Deep Neural Networks motivated by Partial Differential Equations". In: *CoRR* abs/1804.04272 (2018). arXiv: 1804.04272. URL: http://arxiv.org/abs/1804.04272.

[ZF13] Matthew D Zeiler and Rob Fergus. *Visualizing and Understanding Convolutional Networks*. 2013. DOI: 10.48550/ARXIV.1311.2901. URL: https://arxiv.org/abs/1311.2901.

[Zha+21] Aston Zhang et al. "Dive into Deep Learning". In: *arXiv preprint arXiv:2106.11342* (2021).