Deep Learning Workshop, 27.03 – 30.03.2023

Assignment 02: Convolutional Neural Networks (CNN)

The main goal of this second programming session is to build a fraud detection system with the help of a Siamese Convoluational Neural Network (SNN). In two steps, we will:

a) Develop and implement a custom data loader based on a given data set.

b) Implement a Siamese Convolutional Neural Network.

Disclaimer: This task is in general one of the most challenging tasks in biometrics and forensics and is still an open research question. The main goal of this exercise is to get familiar with custom data loaders in PyTorch and gain practical experience with CNNs. First, we focus on the general implementation without concerning ourselves with the computational time. Later, in the bonus exercise and the next assignment, we investigate techniques to improve our results and simultaneously obtain major speed-ups. As before, we use the abbreviations

[N] Network

[DL] Data Loader

[NT] `NetworkTrainer`

[NU] `NetworkUser`

in front of each subtask to indicate which parts of our structure we use or implement. As usual, whenever we train a network, the estimated training time is given in brackets.[1]

Introduction and Theory

In this assignment we will work on the task of signature verification and use parts of the dataset provided by the "Signature Verification Competition" [Liw+11] which took place in Beijing, China in 2011. In order to speed up the training process, we will not work with the whole dataset[2]. The general objective is to compare two signature observations (questioned and reference signature) and we want to check for two cases:

(C1) The reference and the questioned signature are both written by the same writer,

(C2) The questioned signature is written by a writer other than the reference writer.

In particular, we want our network to return a **probability** for the signatures being written by **different writers**, i.e. $\mathbb{P}(\text{author}_1 \neq \text{author}_2)$. Consider the following example shown in Figure 1. The example also illustrates our training process: The dataset consists of signature pairs of (real) people and corresponding fake signatures (so-called skilled forgeries) created by professionals which are designed to imitate the real signature. We will train our network with combinations of the form (orignal, original), (original, skilled forgery) and (original, unrelated) where unrelated represents a signature of a totally different person.

(a) (C1), (original, original): The reference and the questioned signature match exactly. The result of the network should be $\mathbb{P} = 0$. The difficulty for the network should be medium.

(b) (C2), (original, skilled forgery): The questioned signature is a skilled forgery of the reference signature. The result of the network should be $\mathbb{P} = 1$. This case is by far the hardest one.

(c) (C2), (original, unrelated): The questioned signature is not related to the reference signature. The result of the network should be $\mathbb{P} = 1$. This case should be easy to detect.

Figure 1: An overview of the three different training cases. The upper image is the reference signature and the lower image is the questioned signature.

In our training process, the images will be scaled to the same size and a lower resolution to accelerate the training. For more details and exemplary outputs consider Figure 1.

So this issue is **not pure classification** as we are not interested in probability of authorship, but instead just decide whether the reference author and the questioned author match.[3]

Originally invented in the early 1990s by Bromly and LeCun [Bro+93] to solve the signature verification problem, Siamese Networks are a widely used framework. We will use a simplified combination of [KZS+15] and [Dey+17] in which the method of Bromly and LeCun is slightly modified and enhanced.

The structure of a SNN is simple but powerful: a Siamese Network consists of two identical twin CNNs (shared weights) that accept two distinct inputs (in our case the questioned and the reference signature) which are merged into a fully connected layer and a joint loss function. Figure 2 shows the structure of the SNN we will implement in this assignment.

A Siamese approach has three major benefits:

a) As we use shared weights for both heads, Siamese Networks need in general fewer parameters and are consequently able to achieve similar results with less images. As shown in [KZS+15], SNNs are more robust to class imbalances.

b) When working on a classification problem, one usually has to change the network's structure and perform additional retraining steps if more classes are added to the dataset. Siamese Networks on the other hand are not subjected to this problem, since only a similarity measure is trained. Therefore a new author's signature can be easily checked, without the need for a new class, i.e. a different output layer, and additional training.

c) Each head produces features of it's input which are then compared in a contrastive manner. Thus, we can focus on learning similarities which is naturally related to the problem's symmetric structure. Furthermore, the symmetric structure of the network ensures that the reference and the questioned signature are interchangeable which is reasonable for this task.

[1] The networks were trained on a Mac M1 Pro with 16 GB of RAM.

[2] All samples can be found under http://www.iapr-tc11.org/mediawiki/index.php/ICDAR_2011_Signature_Verification_Competition_(SigComp2011).

[3] Explanation based on https://forensic.to/webhome/afha/SigComp.html.

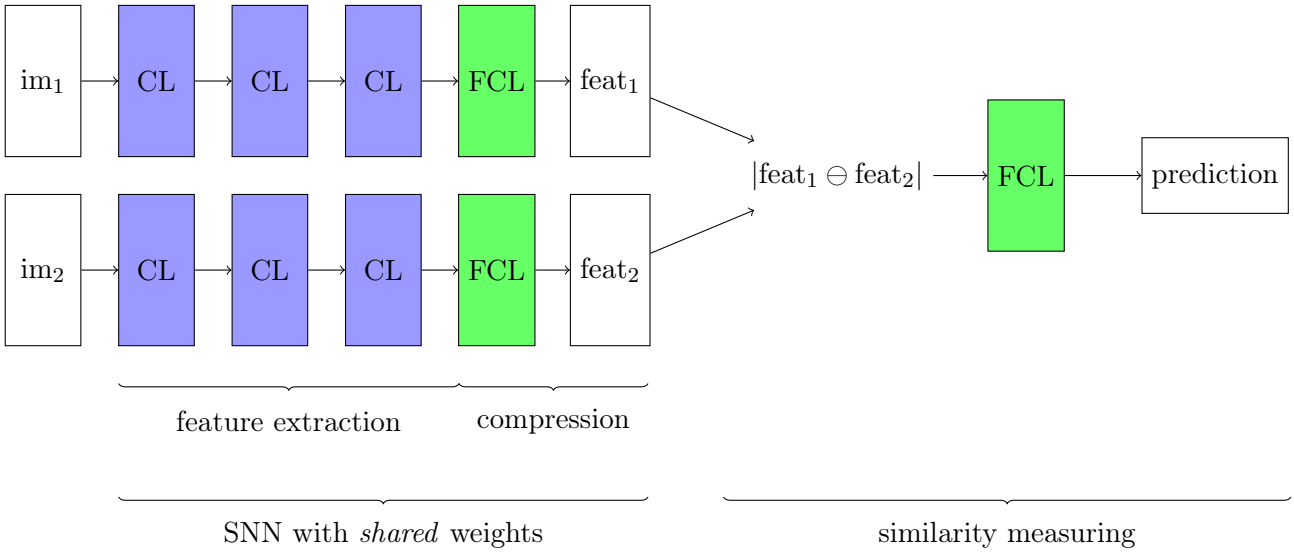

Figure 2: Structure of our Siamese Network. CL is an abbreviation for a Convolutional Layer and FCL for a Fully Connected Layer (both already including additional steps like e.g. the activation function).

On the other hand, we face two major downsides when using Siamese Networks:

a) Because Siamese Networks require pairwise data points, we need in general more training time.

b) A Siamese Network is not a classifier, instead we can just make a statement about the distance between two inputs. In particular, if the signatures are identified as different, we cannot make a statement about the forger.

Programming

1) **Writing a custom data loader: Feed the network.**
In this task, we will work with the file `cnn.py`.

Take a look at the `./data` folder: you should see two `.csv` tables and two subfolders named `test_data` and `train_data`. Each of the subfolder contains multiple subfolders named by the writer IDs and the suffix `_forg` for the corresponding forged signatures. For instance: The folder `049` contains 12 valid signature of the person with ID 49 and `49_forg` contains 12 skilled forgeries for the same signature. The `.csv` tables include the filenames and the corresponding labels. A value of 1 represents (C2) and a value of 0 represents (C1), see Figure 1. The first column always represents the filename of a signature by the reference writer, column two the filename of the questioned signature and the third column contains the label.

a) [DL] We start by implementing the missing methods of the class `SignatureDataset`. The constructor should take a path to the `.csv` table, a path to the data folder and an (optional) argument to transform the images (we will need to rescale and transform all images). The class is inherited by the PyTorch `Dataset` class which behaves like a generator, thus we also have to implement a `__get_item__` method.

- Implement the constructor of the class. Read the `.csv` file and store the `pandas.DataFrame` as a class attribute.
- Implement the `__get_item__` method which takes a generic index (i.e. a row index of the table) and returns the references image, the questioned image and the corresponding label given by that row.

Hint: As the result of our network will be a floating point number (probability score between 0.0 and 1.0), we have to return the label as a float. Otherwise some PyTorch loss functions might throw an exception.

b) [DL] Complete the template of the `DataLoaderSNN` class. The constructor should generate three datasets: Train data and validation data (created by dividing the raw train data with the given splitting ratio) and the test data. To enable shuffling on the training data, set `shuffle = True`. *Hint:* You can get inspiration from the `DataLoaderDNN` class of the previous programming session.

2) Implementing a SNN: Using shared weights to benefit from symmetry.
In this task, we work with the files `cnn.py` and `02_main.py`. We start by implementing the network shown in Figure 2. We will use the `nn.BCELoss`, given by

$$L_{\text{BCE}}(x, y) := \frac{1}{N} \sum_{i=1}^{N} y_i \cdot \log(x_i) + (1 - y_i) \cdot \log(1 - x_i), \tag{BCE}$$

as a loss function (i.e. we treat our problem as a binary classification problem for the sake of simplicity). We will consider a non-standard loss function in one of the Bonus tasks.

a) [N] Open the file `cnn.py`. Implement the CNN part of the Siamese network which represents the feature extraction (see Figure 2). Use three layers, each consisting of a `nn.Conv2d` layer, `nn.ReLU` as an activation function and a `nn.MaxPool2d` pooling layer. For each layer increase the number of output channels and decrease the kernel size, this allows our network to learn smaller and more features. Start with a small number of channels in each layer. Under the link Convolution arithmetic [DV16] you can find an intuitive visualization of the convolution operation and the effect of different parameters.
Hint: Again, the command `nn.Sequential` might be helpful.

b) [N] Now, add a fully connected layer after the CNN-part representing the compression part of the network (see Figure 2).
Hint: Take a look at the shape of the previous layer's output. Perform a reshape operation by using the command `x.view()`.

c) [N] Finally, add the similarity measure part (see Figure 2), by computing the absolute value of the difference between the two outputs and afterwards passing the difference into one fully connected layer with `nn.Sigmoid` as an activation function. Which shape should the final output have?

d) [NT] ($\sim$ 25 min, 5 epochs) Switch to the file `02_main.py`. Add the following transformers to your code

```
transform = transforms.Compose(
    [
        transforms.ToTensor(),
        transforms.Resize((100, 100)),
    ]
)
```

to obtain tensors and reduce the image sizes to reduce the training duration. Instantiate your dataloader, load the BCE-loss, define an optimizer (e.g. `torch.optim.Adam`), use a scheduler and train your network with the `NetworkTrainer`. We recommend using a second terminal, so that you can work on the remaining questions in the meantime.

e) [NU] Evaluate your network by using the `NetworkUser`. Compare your results to your neighbor's results but do not expect exceptionally good results.

3) BONUS: Advanced loss functions and feature embedding metrics.
We suggest to skim all the exercises first, spend some time to think about the discussion questions and afterwards pick a task that you want to implement.

a) Can you think of an evaluation strategy to show the network's accuracy in percentage points? Pass it into the `NetworkTrainer` via the `sum_correct_preds` function.

*b) Can you think of a method to evaluate the model's performance besides the value of the loss function?
Hint: Think about *true positives* and *true negatives* or take a look at [Dey+17].

*c) What happens, if you change the

- number of convolutional layers
- size and or the stride of the convolutional kernels
- number of linear layers after the CNN part

in your network or the

- image sizes

of our input data? Feel free to download the full dataset from the link above and see how a bigger dataset changes the network's performance.

*d) For CNNs a classical approach to reduce overfitting by enforcing more generalizations is to include so-called **Dropout layers** [Sri+14]. The idea of this regularization method is to randomly freeze units (both hidden and visible) and all associated weights during the training process (e.g. 30% of the involved Neurons). Use the function `torch.nn.Dropout` to add dropout layer(s) (e.g. after the activation function of a linear layer) to your SNN.

*e) In our first approach we used the ansatz of propagating the absolute value of the difference between the outputs into the similarity measuring part. Can you think of different ideas for the last part of our SNN?
Hint:

- [N] Instead of computing the difference, we could also pass the concatenated feature vectors into the last part of the SNN. This allows are network to freely choose between all available features and weight them adaptively. Can you implement this approach by using `torch.cat`?
- [NT] We could also use a different loss function. In the paper [Bro+93] the authors suggested a more complex loss function, the so-called **Contrastive Loss** given by

$$L_{\mathrm{CL}}(x_1, x_2, y) := \alpha y D_w(x_1, x_2)^2 + \beta(1-y)\max(0, m - D_w(x_1, x_2))^2, \qquad \text{(CL)}$$

with the distance $D_w(x_1, x_2) := \|f(x_1; w_1) - f(x_1; w_2)\|_2$, a margin[4] m and a boolean $y \in \{0, 1\}$ indicating whether the samples x_1, x_2 belong to the same class or not ($y = 1$ means same class).

Interpretation of the loss function (cf. [Dey+17]): D_w is the Euclidian distance computed in the embedded feature space, f is the embedding function with weights w_1, w_2 that maps our signature images to a real vector through the CNN-part (i.e. we have $w_1 = w_2$ due to the Siamese structure). By properly adjusting the margin, we can enforce our SNN to bring output feature vectors closer for similar input pairs (i.e. questioned signature is an authentic signature (C1)) and push feature vectors away if the input pairs are dissimilar (i.e. (C2)). This ensures that images belonging to the same class (C1) will be closer to each other than images of different classes (C2).

[4] In [Dey+17] the authors used the Contrastive Loss with a margin of $m = 1$.

There is no implementation of the Contrastive Loss in PyTorch. Implement the Contrastive Loss as your own loss function, use the template

```
class ContrastiveLoss(nn.Module):
    def __init__(self, margin: float = 1.0):
        super(ContrastiveLoss, self).__init__()
        pass

    def forward(
        self,
        input1: torch.tensor,
        input2: torch.tensor,
        target: torch.tensor
        ):
        pass
```

and start with $\alpha = \beta = 1$. How do you need to adjust definition CL given a batch of data points and the labels in our table? Introduce a decision threshold to decide between (C1) and (C2) using the results of the contrastive loss.

*f) [N] In the AlexNet architecture [KSH12], the developers introduced a non-learnable normalization method called **Local Response Normalization** with the goal of normalizing the data channel-wise. Local Response Normalization addresses the problem of increasing output layer values when working with unbounded activation functions such as `ReLU` or `ELU`. Take a look at PyTorch's version of `torch.nn.LocalResponseNorm` and add it to your network. How is the performance influenced?

g) Assume you are working in the safety department of a large bank and your boss asks you to implement a signature verification model for handwritten transfers. What challenges occur in reality and how could you solve them? Think about the training process as well as the evaluation.

References

[Bro+93] Jane Bromley et al. "Signature verification using a 'siamese' time delay neural network". In: *Advances in neural information processing systems* 6 (1993).

[Dey+17] Sounak Dey et al. "Signet: Convolutional siamese network for writer independent offline signature verification". In: *arXiv preprint arXiv:1707.02131* (2017).

[DV16] Vincent Dumoulin and Francesco Visin. "A guide to convolution arithmetic for deep learning". In: *ArXiv e-prints* (Mar. 2016). eprint: 1603.07285.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[KZS+15] Gregory Koch, Richard Zemel, Ruslan Salakhutdinov, et al. "Siamese neural networks for one-shot image recognition". In: *ICML deep learning workshop*. Vol. 2. Lille. 2015, p. 0.

[Liw+11] Marcus Liwicki et al. "Signature Verification Competition for Online and Offline Skilled Forgeries (SigComp2011)". In: *2011 International Conference on Document Analysis and Recognition* (2011), pp. 1480–1484.

[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.

[Sri+14] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.