Project Machine Learning
Task 1: Deep Neural Networks

Philipp Liznerski[1], Saurabh Varshneya[2], and Tobias Michels[3]

[1]liznerski@cs.uni-kl.de, `@liznerski`
[2]varshneya@cs.uni-kl.de, `@varshneya`
[3]t_michels15@cs.uni-kl.de, `@tmichels`

October 18, 2021
Deadline: 29/11/2021

Important: Please read the documentation on the OLAT page and the README files in your repository carefully. They contain detailed instructions and hints on how to perform and submit your tasks. If you have any further questions, don't hesitate to contact us on Mattermost.

Introduction

In *Machine Learning* our goal is to learn or extract meaningful patterns from given *data*. A datum in the so called *supervised* setting is a pair $(\mathbf{x}, y) \in \mathcal{X} \times \mathcal{Y}$, where $\mathbf{x}$ represents the *input* (e.g., an image) and y represents the *label* (e.g., a word like "dog"). A collection of such pairs $D := \{(\mathbf{x}_i, y_i)\}_{i=1}^n \subseteq \mathcal{X} \times \mathcal{Y}$ is called a *dataset* and can also be represented as a vector of inputs $X = \begin{bmatrix}\mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_n\end{bmatrix}^\mathsf{T} \in \mathcal{X}^n$ and a vector of labels $\mathbf{y} = \begin{bmatrix}y_1 & y_2 & \cdots & y_n\end{bmatrix}^\mathsf{T} \in \mathcal{Y}^n$. Our goal is to find a function f based on the given dataset D, such that $f\colon \mathcal{X} \to \mathcal{Y}$ with $f(\mathbf{x}) \approx y$ for all $(\mathbf{x}, y) \in \mathcal{X} \times \mathcal{Y}$.

We call the setting where $\mathcal{Y}$ is finite *classification*, and the elements of $\mathcal{Y}$ are then *classes*. For instance, assume that we have a bunch of 512×512 pixel RGB images showing either cats or dogs and we, as human experts, have already determined for each image whether it shows a cat or a dog. We have $\mathcal{X} = \{0, 1, \ldots, 255\}^{3\times512\times512}, \mathcal{Y} = \{\text{"cat"}, \text{" dog"}\}$ and try to find a classifier f that determines if a given image shows a cat or a dog. Ideally, f should also make correct decisions on new, unseen images, if they are reasonably similar to the already seen images.

1 K Nearest Neighbor Classification

KNN Definition The classical k-nearest neighbor (KNN) algorithm is perhaps one of the simplest approaches to perform classification. Given a training dataset $D_{\text{tr}} := \{(\mathbf{x}_i, y_i)\}_{i=1}^n \subseteq \mathcal{X} \times \mathcal{Y}$, we assign a label to an unseen test point $\mathbf{x}_{\text{te}} \in \mathcal{X}$ by finding the $k \in \mathbb{N}$ closest training points to it and choosing the most frequent label from those points.

Let $\mathcal{N}(\mathbf{x}_{\text{te}})$ be the set of k nearest neighbors for a test point $\mathbf{x}_{\text{te}}$, where

$$\mathcal{N}(\mathbf{x}_{\text{te}}) = \underset{\{(\mathbf{x}_i, y_i) : i \in \{i_0, \ldots, i_k\}\}}{\arg\min} \sum_{i \in \{i_0, \ldots, i_k\}} d(\mathbf{x}_{\text{te}}, \mathbf{x}_i) \quad \text{s.t.} \quad \forall \alpha, \beta \in \{i_0, \ldots, i_k\}^2 : \alpha \neq \beta.$$

Here, $d \colon \mathcal{X} \times \mathcal{X} \to \mathbb{R}_0^+$ is a distance function, quantifying the closeness of datapoints in $\mathcal{X}$. This allows us to predict the label of $\mathbf{x}_{\text{te}}$ as

$$y_{\text{te}} = f(\mathbf{x}_{\text{te}}) := \underset{y' \in \mathcal{Y}}{\arg\max} \left| \left\{ y' = y : (\mathbf{x}, y) \in \mathcal{N}(\mathbf{x}_{\text{te}}) \right\} \right|.$$

Note that $\mathcal{N}(\mathbf{x}_{\text{te}})$ and y_{te} might not be well-defined since the min and max are not necessarily unique. In order to fix this, we must employ some kind of tie breaker strategy, a simple approach would be to just randomly select one of the min/max elements. You now have the adequate information to carry out Tasks a) and b) listed below.

Evaluation Techniques To rate the performance of a model, we need to define an evaluation metric. The most common performance metric is *accuracy*: Given some dataset D and a classifier $f \colon \mathcal{X} \to \mathcal{Y}$, for example our KNN classifier, we define the accuracy of f on the dataset D as

$$\text{Acc}(f, D) := \frac{1}{|D|} \sum_{(\mathbf{x}, y) \in D} \mathbb{I}_{f(\mathbf{x}) = y},$$

where the *indicator function* $\mathbb{I}_{\mathtt{p}}$ returns 1 if $\mathtt{p}$ is true, otherwise it returns 0. In words, the accuracy is the fraction of datapoints in D that are correctly classified by f. We also need to find a proper test set D, which should be distinct of the training data and hence contain unseen samples. However, data is often a rare and precious resource in machine learning and access to fresh datapoints is not always available. One possible way to overcome this dilemma is called m-fold *cross validation*:

- Partition the dataset D *randomly* into m equally sized *folds*.
- Choose one fold as the test set and train your classifier on the union of the remaining ones. Evaluate the performance of the classifier on the chosen test set.
- Repeat the training/evaluation process, while choosing each fold as the test set exactly once. Report the average of the performance metrics over all m runs.

You now have the adequate information to carry out tasks c), d) and e) listed below.

Feature Engineering We described the KNN algorithm using a distance function directly on the input space (e.g., space of images). Using raw images' pixel intensities to compute distances for the KNN algorithm will tend to give poor results. One of the reasons is that slight geometric variations such as rotation, translation, or scaling cause a drastic change in the pixel intensities although the semantic information contained in the image stays the same. To remedy this, we can compute distance functions on some *feature* representations of the images that are less sensitive to geometric variations. Typically, in computer vision, such feature representations are obtained by convolving the image with a *filter*. The output of a convolutional operation is a two-dimensional array (i.e., an image) and often referred to as the *feature map*. For example, there are filters that blur or sharpen the image and filters that extract edges, corners, or blobs. Let the image be $I \in \mathbb{R}^{H \times W \times C}$ and our filter be $K \in \mathbb{R}^{F \times F \times C}$. The output of the convolutional operation at location (i, j) is:

$$O_{i,j} = \sum_{l=1}^{F} \sum_{n=1}^{F} \sum_{n=1}^{C} I_{i-(l-1),j-(m-1),n} K_{l,m,n}$$

You now have adequate information to carry out Tasks f), g), h), and i).

1.1 Tasks

a) The template in your git-repository includes utilities for downloading the *Strange Symbols* dataset. It consists of 28×28 pixel grayscale images divided into 15 different classes. Create a plot containing a few images from each class in the dataset and comment about what you think they represent.

b) Implement a generic version of the KNN algorithm—as explained above—that takes a distance function d and a number k as parameters.

c) Before we can use our KNN classifier on the Strange Symbols dataset, we need to find a suitable function d that quantifies the distance between two images. To keep the things simple, we can interpret a 28×28 image as a vector in $\mathbb{R}^{784}$ and then use the Euclidean distance between the vectors, i.e., for two vectors $\mathbf{x}, \mathbf{x}' \in \mathbb{R}^{784}$ we have

$$d(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|_2 = \sqrt{\langle \mathbf{x} - \mathbf{x}', \mathbf{x} - \mathbf{x}' \rangle} = \sqrt{\sum_{j=1}^{784} (x_j - x'_j)^2}.$$

Estimate the performance of KNN on the Strange Symbols dataset using 5-fold cross validation for all $k \in \{1, \ldots, 10\}$ and the Euclidean distance function. Do not employ a library for this, implement cross validation by yourselves instead. Finally, plot your results in a single figure.

d) Naturally, you would choose k as the value that maximized the accuracy estimated in item c). Do you think that the corresponding accuracy value is a good estimate for the performance of the classifier on fresh, unseen data? Or does it under/overestimate the true performance? Explain your answer.

e) The Euclidean distance used in item c) is not the only possible distance measure between two images from the Strange Symbols dataset and probably not even the most suitable for the task. Find (at least) two more applicable distance measures, run 5-fold cross validation, and compare the results. You may check this paper on different distance measures [3]. For this task and all following, use the k found in item d).

f) Implement the convolutional operation using Python and NumPy only.

g) Obtain feature representations for the images by applying the convolutional operation with at least two different filters. For example, one filter could be to blur the image and another one to detect edges. Apply the KNN algorithm on the features separately as well as the concatenation of all features. Run 5-fold cross validation using at least one distance measure and compare the results. You can check your implementation of the convolution by comparing your results with any existing implementation.

h) So far, our KNN algorithm predicts the most frequent label in the nearest neighbor set with all elements in the set being of equal importance. Extend your implementation of the KNN algorithm so that each neighbor is assigned a weight that is inversely proportional to the distance from a given test point. Run 5-fold cross validation using at least one distance measure on the images and compare the results.

i) For at least three different distance measures and features obtained from two different filters, plot the nearest neighbors to five random misclassified samples. Indicate the ground truth along with the nearest neighbor image samples.

j) Write a small conclusion comparing everything you've done and include a table of all results. Feel free to add more experiments. For instance, you can try to combine different tasks or try to find the best k when using features or different distance measures.

1.2 Competition (10 Points)

The competition will be about who can *speed up the testing method by the largest margin.* Either way, your tasks are as follows:

1. Make your implementation of KNN deterministic. Specifically, if there are two training points with the same distance to the testing point, consider only the one

with a lower index. If there is more than one possible choice when determining the class label from the k nearest neighbors, choose the lowest class number.

2. Perform any changes to the implementation that you deem necessary to make the testing procedure faster. Please don't use any approximate KNN algorithms for that and use the Euclidean distance on images as described in item c).

3. Don't use any additional libraries for your implementation, except for Numpy, PyTorch and their dependencies. If you absolutely need another library for a really good reason, contact us on Mattermost.

4. The template contains a driver program `knn_challenge.py` that will run our evaluation procedure. You should make the necessary changes to this file and run it to test your implementation.

In our evaluation procedure, we will basically just run the `knn_challenge.py` script with $k = 5$, 100 repetitions and use the reported results. The code should be pretty self-explanatory, but if you have any questions regarding it, please contact us on Mattermost. Some general info:

- While your implementation should be as fast as possible, the predictions that it outputs should still be the same as that of any slower KNN implementation. Therefore, we will compare the predictions of your code to those from our reference implementation. If there are less than 95% matches between the two, we must assume that you did not implement KNN correctly and you will be disqualified from this competition.
- We won't run the script using the provided `test_data.pt` as the testing data file. Instead, we will use a different file that contains the same amount of data points.

2 Deep Neural Networks

Deep Learning In Deep Learning (DL), the classifier function f is represented by a *Neural Network* (NN), usually also called *Deep Neural Network* (DNN). It is of the form

$$f(\mathbf{x}) = \text{softmax}(f_l(\sigma(f_{l-1}(\dots(\sigma(f_1(\mathbf{x})))\dots)))),$$

where $f_i(\mathbf{x}) = W_i\mathbf{x} + \mathbf{b}_i$, $W_i \in \mathbb{R}^{d_i \times d_{i-1}}$, $\mathbf{b}_i \in \mathbb{R}^{d_i}$, σ is the activation function and

$$\text{softmax}(\mathbf{x}) = \left(\frac{e^{\mathbf{x}_i}}{\sum_j e^{\mathbf{x}_j}}\right)_{i=1}^{n}, \text{ here } \mathbf{x}_i \text{ is the i-th element of the vector } \mathbf{x}.$$

First, you should do some basic research and try to figure out what neural networks are and how they can be trained. Get yourself familiar with different terms used in deep learning and try out the task a) below.

PyTorch At the same time, you will also need to familiarize yourself with *PyTorch* [1], the deep learning framework we will be using for this project. A good place to start with both tasks is the PyTorch Blitz Tutorial [2]. The book *Deep Learning* [1] by Goodfellow et al.—especially chapters 5 and 6—is a great resource for learning the more advanced concepts. It is freely available as an e-book on the authors' website [3], but it might rely a bit too much on the mathematical background that was introduced in the first four chapters for your taste. In that case, you can find less mathematically centred, but still excellent explanations of most concepts in one of the numerous blog posts that pop up when you enter the keywords "Machine Learning", "Deep Learning", or "Neural Networks" in a search engine of your choice. You have now the adequate information to carry out tasks c), d) and e) listed below.

Batch Normalization Normalization of the input data improves the training of machine learning algorithms by bringing elements of the input to the same scale. Otherwise elements with large values tend to have more effect on the prediction, which does not necessarily correlate with the element's usefulness. Batch Normalization (BN) is a technique in DL that normalizes the data at intermediate layers of the neural network, which also mitigates some other problems in DL such as covariate shift. For more details on batch normalization and the implementation read the original paper [2].

We will implement BN for two-dimensional feature maps as a differentiable layer so that PyTorch automatically computes the gradients via the backpropagation algorithm. The input to the batch normalization operation is a batch of feature maps $\mathbf{a}^{\mathcal{B}}$ obtained from a convolutional layer, where $\mathbf{a}^{\mathcal{B}} \in \mathbb{R}^{N\times C\times H\times W}$ and N is the number of training samples in a batch $\mathcal{B}$, C is the number of feature maps, and H, W are the height and width of the feature maps. During training, the BN operation normalizes the feature maps as:

$$\left(\bar{a}^{\mathcal{B}}_{n,c}\right)_{h,w} \leftarrow \frac{\left(a^{\mathcal{B}}_{n,c}\right)_{h,w} - \mu_c}{S_c} \quad \forall n, c, h, w.$$

Here $\left(a^{\mathcal{B}}_{n,c}\right)$ is the c-th feature map of the n-th sample, $\left(a^{\mathcal{B}}_{n,c}\right)_{h,w}$ denotes the spatial location on a feature map (basically a "pixel"), and μ_c and S_c are the mean and standard deviation of $\left(a^{\mathcal{B}}_{n,c}\right)$:

$$\mu_c = \frac{1}{N\cdot H\cdot W}\sum_{n,h,w}\left(a^{\mathcal{B}}_{n,c}\right)_{h,w}, \qquad S_c = \frac{1}{N\cdot H\cdot W}\sqrt{\sum_{n,h,w}\left(\left(a^{\mathcal{B}}_{n,c}\right)_{h,w} - \mu_c\right)^2}.$$

In a second step, BN transforms the normalized activations $(\bar{a}^{\mathcal{B}}_{n,c})_{h,w}$ linearly with learnable parameters γ_c and β_c:

$$\left(o^{\mathcal{B}}_{n,c}\right)_{h,w} \leftarrow \gamma_c \ \left(\bar{a}^{\mathcal{B}}_{n,c}\right)_{h,w} + \beta_c \quad \forall n, c, h, w \ .$$

[1] `https://pytorch.org`
[2] `https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html`
[3] `https://www.deeplearningbook.org`

During testing, batches of data may not be available. Therefore, calculating statistics over the batches as per above equations becomes unfeasible. As a solution, instead, each activation map of an input is normalized using the memorized running mean and variance from training, which we denote as $(\hat{\mu}_c, \hat{S}_c)$. The BN equation becomes:

$$\left(o^{\mathcal{B}}_{n,c}\right)_{h,w} \leftarrow \gamma_c \frac{\left(a^{\mathcal{B}}_{n,c}\right)_{h,w} - \hat{\mu}_c}{\hat{S}_c} + \beta_c \quad \forall n, c, h, w.$$

You now have the adequate information to carry out the remaining task f).

2.1 Tasks

a) In your own words, write a short paragraph of two or three sentences each, explaining the following terms: *activation function*, *loss function*, *hyperparameters*, *optimizer*, *epoch*, *underfitting*, *overfitting*, *training set*, *test set*, *validation set*. Give an example, wherever this seems reasonable.

b) Implement a deep neural network using PyTorch and apply it to the Strange Symbols dataset. We will provide a template for that with some boilerplate code as part of your git repository. We advise you to use three fully connected layers for your initial setup, with 512, 256, and 15 neurons each and ReLU as the activation function. Since we are tackling a classification problem, *Cross Entropy* should be a good choice for the loss function. That network should quickly achieve an accuracy of at least 90% on the training set.

c) Experiment with different architectures, loss functions, optimizers, hyperparameters, etc. Incorporate a few convolutional layers in your architecture and get yourself familiar with Convolutional Neural Networks (CNNs). Train your network on the Strange Symbols dataset and record your progress in the form of a plot that shows the number of epochs on the x-axis and the training loss on the y-axis. Make a similar plot for the accuracy. Include a detailed description of your best architecture and hyperparameter choice in your report.

d) To better understand the behaviour of the classifier you trained in item c), it is often helpful to examine its output. To that end, compute and interpret the *confusion matrix* for your classifier. Additionally, for each class, plot the ten images for which the network predicts the label correctly and is most confident, and the ten images for which it classifies incorrectly and is least confident in predicting the correct label. In the latter case, indicate the class the model actually predicts. Try to come up with an explanation for those results.

e) Remove the last layer of your classifier (i.e., the one with 15 neurons). The output of your network now contains features instead of classification scores. Take your trained CNN and collect features for all data points. Use these features as an input to the KNN of the task above. Compare the results to the CNN and the KNN versions of Task 1.

f) Implement a 2D Batch Normalization layer in PyTorch; i.e., implement a class inheriting from "torch.nn.Module". Use this layer in your architecture for every convolutional layer. Train the model and compare the results with the ones from previous networks. You can check your implementation by comparing with PyTorch's *BatchNorm2d* layer.

Competition (15 Points): The code template for downloading the Strange Symbols dataset also provides a function `get_strange_symbols_test_data()` that returns a test set of unlabeled images. Use your best classifier to predict a class for each of the images. Save the output of your network's last layer, which should be a 15-dimensional vector for each image, in a CSV-file called `test_predictions.csv`. It should follow the format of `example.csv`, which you can find in your git repository.

We will compare your predictions with the true labels for those images and rank the groups according to the accuracy that their network has achieved.

Include a brief description of the model you've chosen in your report.

Hints:

- If you find it hard to keep track of your experiments, i.e., which combination of architecture and hyperparameters has achieved which performance, have a look at the *sacred* [4] library. It is really useful in such cases!
- To estimate the performance of your classifier on unseen data, you can use, for example, cross validation, as discussed in section 1.
- Make sure that your network is not overfitting too much on the training data. There exist several techniques to prevent overfitting in neural networks, which you can find using the keyword *regularization*.

References

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: `http://www.deeplearningbook.org`.

[2] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *International conference on machine learning*. PMLR. 2015, pp. 448–456.

[3] Nuno Vasconcelos and Andrew Lippman. "A unifying view of image similarity". In: *Proceedings 15th International Conference on Pattern Recognition. ICPR-2000*. Vol. 1. IEEE. 2000, pp. 38–41.

[4] `https://sacred.readthedocs.io/en/stable/`