

Convolutional neural networks

Exercise H6.1: Convolutional neural network (homework, 10 points)

The goal of this exercise is to get familiar with a deep learning framework of your choice, e.g., TensorFlow, Keras or PyTorch. The hints below are specific to TensorFlow¹ or Keras in Python but you are free to choose any other framework/library you want.

The Data:

In this exercise we will work with the popular and publicly available dataset MNIST to train a multi-class classification model. The MNIST dataset contains grayscale images of handwritten digits from 0 to 9. Each image consists of 28×28 pixels. MNIST is comprised of a training and a separate pre-defined hold-out set such that the performance of different models can be compared on the exact same hold-out data (i.e. benchmarking).

In the lecture we use the name “validation” for the hold-out set. This corresponds to the “test” set in the data loaded by TensorFlow and other libraries. The important thing here is that generalization performance is measured on a portion of the data that was **not** used for training.

Many machine learning libraries make loading the MNIST data easy. For example, in TensorFlow², the Python command required to access this data is given by:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```

In Keras:

```
from keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

Most machine learning libraries require a one-hot encoding for the class labels in order to perform multi-class classification. In the case of MNIST you get a 10-dimensional sparse label vector (one element for each of the 10 digit classes in MNIST). The entry 1 is assigned to a single element in that vector to indicate which class the observation is assigned to (e.g. an observation of the digit 3 has a 1 at the 4th index $\underline{y}_T = (0, 0, 0, 1, 0, 0, 0, 0, 0, 0)^\top$ (the first index is for the digit zero). Furthermore, each image of size 28×28 pixels needs to be flattened to a 784-dimensional vector \underline{x} .

Building models for recognizing digits:

1. (1 point) As a start implement a linear model ($\underline{h} = \underline{W} \underline{x} - \underline{\theta}$). Initialize $\underline{W} \in \mathbb{R}^{10,784}$ and $\underline{\theta} \in \mathbb{R}^{10}$ with zeros.
 - (a) Use the *softmax* operation to normalize the predictions $\underline{y}(\underline{h})$.
 - (b) Use the *cross-entropy* as the cost function (often called “loss” instead of cost).

¹We have only confirmed that the hints apply to version 1.12 of TensorFlow and not for Tensorflow ver. 2.0, but you are still highly encouraged to experiment with the latest version.

²at least for version 1.12 of TensorFlow

- (c) To train the model, use a gradient descent algorithm with a constant learning rate $\eta = 0.5$.
 - (d) Process the data in mini-batches³. Use one mini-batch for each gradient descent step (one iteration). For every model in this exercise, use a mini-batch size of 100.
 - (e) Perform 10000 iterations for this linear model. This can lead to multiple runs over the data (i.e. multiple “epochs”, where one epoch refers to having visited every training sample exactly once).
 - (f) Keep track of the model’s performance over multiple training iterations (e.g. after every 100 iterations). Measure the performance by calculating the model’s accuracy which is the fraction of correctly classified digits. Compute it on the training set as well as the hold-out set⁴.
2. (2 point) Implement a second model: A Multi-layer perceptron (MLP) with 3 hidden layers with 1500 hidden neurons in each hidden layer.
- (a) Initialize the elements of the weight matrices with (small) normally distributed random values with mean 0 and standard deviation 0.01. Initial weight values that exceed two times the standard deviation should be recalculated⁵. The biases should be initialized with a constant of **-0.1**⁶. Use this weight and bias initialization **in this and all following models**.
 - (b) Use *Rectified Linear Units (ReLU)* as the activation function for the hidden neurons.
 - (c) As in the linear model, use *softmax* as output non-linearity, and *cross-entropy* for the loss function⁷.
Hint: softmax only acts as a normalization and does not change the size order of its argument components. Although required during training, one does not necessarily have to apply the *softmax* operation when making predictions in order to measure the network’s performance. The predicted class can be determined by the maximal value of the logits in the output layer \mathbf{h}^L . The *softmax* operation does not change the maximal value or its argument.
 - (d) Train your MLP, using the *Adam* algorithm, which is often used in deep neural networks in place of gradient descent. Use the following parameters: $\eta = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 1 * 10^{-8}$. On ISIS you can find a text comparing the different variants of gradient descent: “Sebastian Ruder - An overview of gradient descent optimization algorithms”
 - (e) Train the network for 20000 iterations.
3. (2 point) For the third model, take the above MLP while adding *dropout* regularization during training. Apply a dropout rate of 0.5 to each of the hidden layers.

³In most deep learning frameworks mini-batches are simply referred to as batches.

⁴In the lecture we use the name “validation” for the hold-out set. This corresponds to the “test” set in the data loaded by TensorFlow. The important thing here is that accuracy is measured on a portion of the data that was **not** used in training.

⁵For TensorFlow see: `truncated_normal`

⁶The sign of the constant depends on how you compute the total inputs in a layer. If $\mathbf{h}^v = \mathbf{W}\mathbf{x} - \mathbf{b}$ initialize b_i with **-0.1**. If you are computing $\mathbf{h}^v = \mathbf{W}\mathbf{x} + \mathbf{b}$, initialize b_i with **+0.1**

⁷During training consider using a combined softmax and cross-entropy operation instead of computing them successively to avoid numerical inconsistencies. For TensorFlow see `softmax_cross_entropy_with_logits_v2`

4. (3 point) As a fourth and final model, instead of using fully connected layers combined with dropout regularization, you will use convolutional layers, which are well suited for finding discriminant features for image classification tasks.

(a) We define our convolution operation as follows:

- Use a convolutional kernel of size 5×5 pixels and apply it to each image (2D convolution).
- Apply the convolutional kernel to every pixel in the image (i.e. set the stride to 1 in both dimensions).
- Use zero-padding to preserve the size of the images after applying the convolution (In TensorFlow: `padding='same'`).

We define our downsampling operation as follows:

- Use *max-pooling* with a size (window) of 2×2 pixels (2D *max-pooling*).
- Each pixel of the image should only be used in a single *max-pooling* operation. Each pixel should be part of only one 2×2 *max-pooling* neighborhood (i.e. set the pooling stride to 2 in every spatial dimension).

(b) The **architecture** of the network should be as follows:

1. First feed the input into a convolutional layer which produces 32 feature maps (in TensorFlow: 32 output channels). Add a bias to every feature map and apply a *ReLU* activation.
2. Apply downsampling as defined above to each feature map.
3. The second convolutional layer should produce 64 features. Proceed as in the first convolutional layer.
4. Again apply downsampling as defined above.
5. Feed the downsampled features into a fully connected layer. The nodes of this layer will be the network's output nodes.

(c) **Training:**

- Use the same loss function (*cross-entropy*) and training algorithm (*Adam*) as used for the MLP before.
- Again use 20000 iterations to train the model.

Comparing Models:

- a) (1 point) Compare the training and validation accuracies of the models and plot them over the training iterations (showing every 100th iteration should be a sufficient resolution).
Hint: You have two options for plotting the training and validation accuracies over time:

1. Using TensorFlow's `summary.FileWriter` and `summary.scalar` (you need one of each for training and validation). Combining these with TensorBoard enables you to view the accuracy traces from logs in different directories produced by multiple runs. It is sufficient to include a screenshot of the plot into your jupyter notebook.

OR

2. Evaluate the accuracy values explicitly using `sess.run([accuracy])` and appending the result to some list.

For Keras you can use `history` which is returned by `fit(...)`.

- b) (1 point) Can you detect any under- or overfitting and explain why? What can you say about the use of regularization vs. convolutional layers with downsampling with respect to the overall performance of the models and their ability to prevent overfitting? Consider for example the approximate number of parameters that have to be optimized for each model.

Total 10 points.