Convolutional neural networks

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

The goal of this exercise is two-fold:

1. *experimenting with deep learning architectures and techniques,*
2. *getting familiar with a deep learning framework of your choice, e.g., TensorFlow, Keras or PyTorch. Most of the hints below are specific to TensorFlow[1] or the more high-level API Keras in Python but you are free to choose any other framework.*

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 word "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 generlization performance is measured on a portion of the data that was **not** used for training or hyperparamter optimization.*

Many machine learning libraries make loading MNIST easy. For example, in TensorFlow[2], 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 tensorflow import keras
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
```

One-hot encoding: 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.
Example: An observation of the digit 3 has a 1 at the 4^{th} index $\underline{\mathbf{y}}_{\text{True}} = (0,0,0,1,0,0,0,0,0,0)^\top$ (the first index is for the digit zero). Furthermore, each image of size 28×28 pixels may need to be flattened to a 784-dimensional vector $\underline{\mathbf{x}}$.

[1] We have only confirmed that the hints apply to version 1.12 of TensorFlow and not for Tensorflow version 2.0 or later, but you are highly encouraged to experiment with the latest version and find inspiration from examples online.

[2] at least for ver. 1.12 of TensorFlow

Building models for recognizing digits:

1. (1 point) As a start implement a linear model ($\underline{\mathbf{h}} = \underline{\mathbf{W}}\,\underline{\mathbf{x}} - \underline{\boldsymbol{\theta}}$). Initialize $\underline{\mathbf{W}} \in \mathbb{R}^{10,784}$ and $\underline{\boldsymbol{\theta}} \in \mathbb{R}^{10}$ with zeros.
   (a) Use the *softmax* operation to normalize the predictions $\underline{\mathbf{y}}(\underline{\mathbf{h}})$.
   (b) Use the *cross-entropy* as the cost function (often called "loss" instead of cost).
   (c) To train the model, use a gradient descent algorithm with a constant learning rate $\eta = 0.5$.
   (d) Process the data in mini-batches[3]. 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 10,000 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 on the hold-out set.
2. (2 point) Implement a second model: A fully connected 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 recalculated[4]. The biases should be initialized with a constant of **-0.1**[5]. Use this weight and bias initialization scheme **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* for your output non-linearity, and *cross-entropy* for the loss function [6].
   Hint: softmax only acts as a normalization and does not change the size order of its argument components. Although required during training, there is no added value in applying the *softmax* operation when you only want to make predictions in order to measure the network's performance on the hold-out data (unless you are interested in interpreting the output as a posterior). The predicted class can be determined by the maximal value of the logits in the output layer $\underline{\mathbf{h}}^L$. The *softmax* operation does not change which element has the maximal value.
   (d) Train your MLP, using the *Adam*[7] algorithm, which is often used in deep neural networks in place of gradient descent. Use the following parameters: $\eta = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 1 * 10^{-8}$.

[3] In most deep learning frameworks mini-batches are simply referred to as batches.
[4] For TensorFlow see: `truncated_normal`
[5] The sign of the constant depends on how you compute the total inputs in a layer. If $\underline{\mathbf{h}}^v = \underline{\mathbf{W}}\,\underline{\mathbf{x}} - \underline{\mathbf{b}}$ initialize b_i with -0.1. If you are computing $\underline{\mathbf{h}}^v = \underline{\mathbf{W}}\,\underline{\mathbf{x}} + \underline{\mathbf{b}}$, initialize b_i with $+0.1$
[6] During training, consider using a combined softmax and cross-entropy operation instead of computing them successively to avoid numerical inconsistencies. For TensorFlow ver 1.* see `softmax_cross_entropy_with_logits_v2` This may not be necessary for all frameworks or more recent versions of TensorFlow
[7] On ISIS you can find a text comparing the different variants of gradient descent: "Sebastian Ruder - An overview of gradient descent optimization algorithms" for furhter reading.

(e) Train the network for 20,000 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. Watch out for what happens with dropout when you're making predictions on the hold-out set.

   Disclaimer: An MLP with 3 layers of 1500 neurons each is not the ideal architecture for highlighting the effect of dropout on MNIST. The regularization effect of dropout is too subtle in this case. It is possible to see a more pronounced effect if we used less data or if we increased the number hidden neurons to 10,000 or more per layer, but this could drastically increase the computational resources for training the models fast enough.

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 window size 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 after they have been normalized using a *softmax* operation.

   (c) **Training**:

   - Use the same loss function (*cross-entropy*) and training algorithm (*Adam*) as used for the MLP before.
   - Again use 20,000 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 100^{th} 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 two 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 explicity 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? 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.