Exercise 5.0 — Deep Learning

Artificial Intelligence for Robotics

Renaud Dubé, Fadri Furrer, Mark Pfeiffer

Spring Semester 2017

1 Deep learning basics

Deep learning is a machine learning approach where multiple layers of nodes are stacked on top of each other, which results in a so called neural network. The simplest form of neural network are fully connected (FC) neural nets which mainly consist of matrix multiplications and non-linear activation functions. Both FC-networks and the slightly more complicated convolutional neural networks will be covered in this exercise. More details about neural networks can be found in the lecture notes. For a profound theoretical background we refer to the literature, e.g. in Ian Goodfellow and Yoshua Bengio's book about deep learning[1].

1.1 Fully connected networks

As mentioned above, FC neural networks consist of multiple neural layers stacked on top of each other. Each layer consists of multiple nodes. Nodes of one layer are only connected to the nodes in the previous and succeding layer (layer index increases from input to output). There is no connection among the nodes of one layer. A simple version of a FC neural network is provided in Figure 1. Figure 2 visualizes typical activation functions which can be applied to the layers in order to introduce non-linearities to the network.

We want to use neural networks to get an output $\mathbf{y}$ using a given input $\mathbf{x}$. The network can be seen as a big non-linear function

$$\mathbf{y} = \mathcal{F}_{\Theta}(\mathbf{x}). \tag{1}$$

The goal is to adapt the network parameters Θ in a way that the network output $\mathbf{y}_{target}$ matches the desired output $\mathbf{y}$ as closely as possible.

[1] `http://www.deeplearningbook.org`

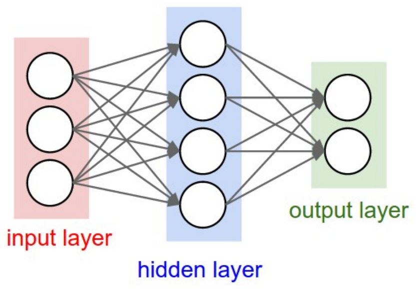

Figure 1: Basic FC neural network with an input layer ($\mathbf{x}$), one hidden layer ($\mathbf{h}_1$) and the output layer ($\mathbf{y}$). For simplicity, the activation functions ($\mathbf{a}$) are not visualized. Image source: `http://cs231n.github.io/assets/nn1/neural_net.jpeg`

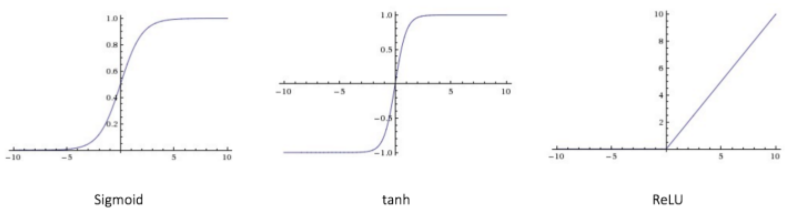

Figure 2: Most common activation functions for neural networks. Image source: `https://ujwlkarn.files.wordpress.com/2016/08/screen-shot-2016-08-08-at-11-53-41-am.png?w=748`

1.1.1 Feedforward computations

The forward computations of a neural network (forward pass) are composed of an affine transformation (matrix multiplication and adding a bias) and the application of activation functions. For the basic example provided in Figure 1, the full model $\mathcal{F}_\Theta(\mathbf{x})$ can be split up into the following operations:

$$\mathbf{h}_1 = \mathbf{a}_1\big(\underbrace{\mathbf{x} \cdot w_1 + b_1}_{\gamma_1}\big) \tag{2}$$

$$\begin{aligned}\mathbf{y} &= \mathbf{a}_{out}\big(\underbrace{\mathbf{h}_1 \cdot w_{out} + b_{out}}_{\gamma_{out}}\big)\\ &= \mathbf{a}_{out}\Big(\mathbf{a}_1\big(\mathbf{x} \cdot w_1 + b_1\big) \cdot w_{out} + b_{out}\Big)\end{aligned} \tag{3}$$

where (w_1, b_1) the weights and biases of the hidden layer $\mathbf{h}$ and (w_{out}, b_{out}) the weights and biases of the output layer. The affine output of layer i is denoted by γ_i, which is the input to the activation function $\mathbf{a}_i$. The shapes of the weight and bias matrizes of each layer are determined by the previous and succeeding layer dimensions. For a simple FC network each weight matrix w_i has the shape $[layersize_{i-1} \times layersize_{i+1}]$ and each bias matrix has the shape $[1 \times layersize_{i+1}]$, assuming that the input $\mathbf{x}$ and all the layer outputs are row vectors.

Given a loss function $\mathbf{L}(\mathbf{y}, \mathbf{y}_{target})$, the quality of the network predictions can be assessed numerically. A loss function calculates the difference of two input vectors, $\mathbf{y}$ and $\mathbf{y}_{target}$ in this case, and ouputs one scalar value. During model training, such a loss function will be used to find the optimal network parameters.

1.1.2 Gradient computations

In order to optimize the parameters of the network, gradient-based optimization can be applied, e.g. the classical gradient descent algorithm. Therefore, the cost derivative w.r.t. all parameters (weights and biases) is required. Due to the stacked structure of the network layers, the gradients are propagated through the network via the layers that connect the output layer with the layer of interest. This mechanism is called *backpropagation*, it consists of repeatedly applying the chain rule.

Gradient computation is not straightforward for every network. General information about backpropagation can be found in the literature[2]. For the example network used in this exercise and shown in Figure 1 and Equations (2) and (3) the gradient computations look like the following:

Output layer

$$\frac{\partial \mathbf{L}}{\partial w_{out}} = \mathbf{h}_1^T \cdot \underbrace{\frac{\partial \mathbf{L}}{\partial \mathbf{y}} \circ \mathbf{a}'_{out}(\gamma_{out})}_{\delta_{out}} \tag{4}$$

$$\frac{\partial \mathbf{L}}{\partial b_{out}} = \delta_{out} \tag{5}$$

[2]`http://www.deeplearningbook.org`

Hidden layer

$$\frac{\partial \mathbf{L}}{\partial w_1} = \mathbf{x}^T \cdot \underbrace{\delta_{out} \cdot w_{out}^T \circ \mathbf{a}_1'(\gamma_1)}_{\delta_1} \tag{6}$$

$$\frac{\partial \mathbf{L}}{\partial b_1} = \delta_1 \tag{7}$$

As the activation function (e.g. Sigmoid, ReLU, tanh, ...) is applied on each node, element wise multiplication (Hadamard product) is applied. Also be aware of the correct positioning and transposing of the matrizes. This matrix multiplication computation only works for this example with the row vector $\mathbf{x}$ as network input and row vector $\mathbf{y}$ as output. For the general case, indexing notation needs to be used.

As you can see in the equations above (and it would continue like this if there were more FC layers), the error is propagated through the layers. For each layer, the helper variable δ_i has to be updated and multiplied with the transposed input of this layer (from the left) and the activation function derivative (from the right).

The input layer contains solely the input to the network, so no weights or activation functions are applied to the input before the input layer.

1.1.3 Gradient update

There are various optimization techniques that can be used for neural network training (e.g. batch gradient descent, stochastic gradient descent, RMSprop[3], ADAM[4], ...). The simplest one and the one used during this exercise is the batch gradient descent algorithm. The gradient of the loss function w.r.t. all weights (w) and biases (b) is computed for all samples among the batch (of a certain *batch size*).

The gradients among the batch are summed up and applied to the gradient update rule

$$w_{i,\text{new}} = w_{i,\text{old}} - \alpha \cdot \frac{\partial \mathbf{L}}{\partial w_i}, \tag{8}$$

[3] http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[4] https://arxiv.org/pdf/1412.6980.pdf

where α is the *learning rate* of the algorithm. The optimization is conducted in batches in order to reduce the noise in the gradient estimation by averaging it out.

1.2 Convolutional neural networks

A Convolutional Neural Network (CNN) is a type of Neural Network (NN) which is particularly suitable to capture spatial patterns in data. Recent development in 2D CNNs showed impressive results for for image-based recognition tasks. A pioneer CNN which was proposed for the task of hand-written digit recognition is illustrated in Fig. 3.

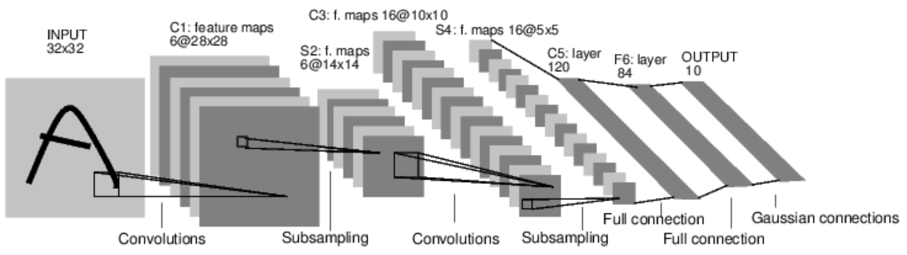

Figure 3: Illustration of LeNet-5: A back-propagation 2D convolutional network for hand written digit recognition [1]. Note the convolutional, the subsampling, and the fully connected layers.

In a CNN, each convolutional layer is made of multiple filters, each composed of different neurons. Taking the example of recognizing objects in RGB images, the first convolutional layer of a 2D CNN could be composed of 32 filters, each of dimension 5x5x3 where the last dimension represents the three colors. There would then be 75 neurons per filter, resulting in 2400 neurons in this first layer. Convolving these filters on the images would result in a stack of 32 new images. Convolutional layers are composable with each subsequent layer taking as an input the stack of filtered images generated by the previous layer. One or more fully connected layers would typically be used to make the final decision regarding the object classes. As described in Section 1.1.2, the values of the convolution filters and the weights of the fully connected layers can be learned through back-propagation.

In order to reduce the dimension of the data, pooling layers are typically inserted between the convolutional layers. This method of sub-sampling is illustrated in Fig. 3. A common choice is to use max-pooling filters which simply return the highest value over a window size. These max-pooling filters are defined by their window size and by the stride, i.e. the number of pixels by which the filter should be moved between each operation. In addition to reducing the dimensionality, pooling makes the network less sensitive to small translational and rotational noise in the original image.

The dimensionality reduction offered by pooling layers allows subsequent layers to work on a larger section of the data. As illustrated in Fig. 4, different layers extract features of different scales in the image. In this example, the first layer learned local features like edges, corners, textures and contrasts between colors whereas the second layer learned slightly larger features like noses, eyes and ears. The final convolutional layer, thanks to the dimensionality reduction offered by the max-pooling layer, could work on sub-sampled and filtered versions of the original images in order to extract global face features.

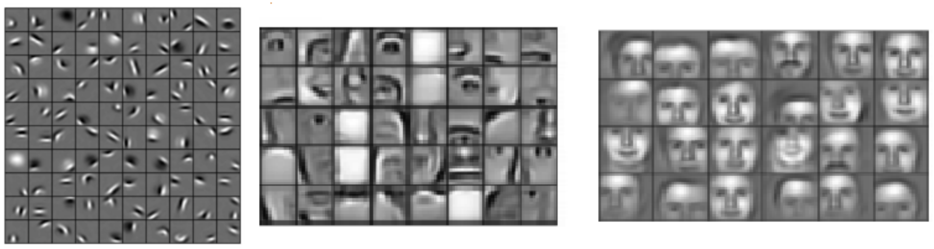

Figure 4: Examples of features which can be learned at different layers of a CNN for a visual recognition task [1]. Note that the first layer (on the left) learns local properties of the images whereas latter layers work on larger but sub-sampled sections of the image.

1.3 Deep reinforcement learning

1.3.1 What is RL?

Reinforcement Learning (RL) originates from behaviorist psychology, and is a type of learning that tries to maximize an overall reward that is given for individual actions. In the case of robotics we are typically looking at an agent that can perform a set of actions A in an environment. By performing an action a_t, at a discrete point in time t it gets an immediate reward r_t, which depends on the agent's and the environment's state s_t, and their future states s_{t+1}

$$r_t = f(a_t, s_t, s_{t+1}). \tag{9}$$

The goal is to find a policy π that maximizes the long-term reward

$$R = \sum_{t=0}^{N-1} r_{t+1}, \tag{10}$$

where N denotes the time when a terminal state is reached. If there is no terminal state rewards are usually discounted by a discount-factor γ

$$R = \sum_{t=0}^{\infty} \gamma^t r_{t+1}. \tag{11}$$

The environment in RL problems is typically represented as a Markov Decision Process (MDP) or a Partially Observable Markov Decision Process (POMDP). To the notations above, we additionally denote the probability that an action a taken at t transitions the system from state s to s' at $t+1$ as

$$P_a(s, s') = \Pr(s_{t+1} | s_t = s, a_t = a). \tag{12}$$

If we know these probabilities, and the state and action set is finite we can apply linear or dynamic programming to find the optimal policy π^*, by iterating over the following two steps

$$\pi(s) := \arg\max_a \left\{ \sum_{s'} P_a(s, s') \left(R_a(s, s') + \gamma V(s') \right) \right\}, \tag{13}$$

$$V(s) := \sum_{s'} P_{\pi(s)}(s, s') \left(R_{\pi(s)}(s, s') + \gamma V(s') \right), \tag{14}$$

where $V(s)$ is the value function.

In RL the state transition probabilities $P_a(s, s')$ are usually not known and it might also be infeasible to keep track of all the actions and/or states.

Therefore, one approach is to learn an action-value for a state-action pair (s, a) under the policy π

$$Q^{\pi}(s, a) = E[R|s, a, \pi], \tag{15}$$

where R is the reward for first taking action a from state s and then following policy pi. We can learn this function iteratively

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha_t}_{\substack{\text{learning}\\ \text{rate}}} \cdot \left(\overbrace{\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\substack{\text{discount}\\ \text{factor}}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\substack{\text{optimal future}\\ \text{value estimate}}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right). \tag{16}$$

One important consideration is when to exploit and when to explore. A very simple and often used method is the ϵ-greedy method in which the agent chooses greedily (exploit) the best action according to its current policy with probability $1 - \epsilon$, and uniformly samples a random action, otherwise.

1.3.2 When do we use it?

RL can be used a huge variety of problems but typically it is very well suited for problems where

- the agent has to reason about the long term consequences of its actions, including taking actions with negative immediate reward.
- a model of the environment is unknown (the knowledge of an MDP is not required, but can be approximated).
- the only way to collect information from the environment is to interact with it.
- correct input/output pairs are unknown.

2 Exercises

2.1 Task 1 (40P)

This task should improve your understanding about neural networks in general. You will have to deal with FC neural networks and implement basic functions. Of course, ready to use deep learning libraries already contain these functionalities (and most likely you will simply use the libraries in the end), however it is also important to understand the relations and connections in a neural network.

We provided you with our own basic deep learning framework. As specified in the following tasks, you have to implement the missing pieces in the code.

Usage

Once everything is up and running, you can use the `make` command to train (`make train`) the model. The unit test(s) can be run by typing `make run_tests` in the directory of this exercise. In the end, running the unit tests should not result in an error anymore. Feel free to adjust the unit tests in a more suitable way, if you like.

2.1.1 Implement layer evaluation (5P)

File: `FCLayer.py`

Take into account Equations (2) and (3) in order to fill in the missing lines of code. Remember that the layer output is the activation function applied on the affine part (γ) of the layer.

2.1.2 Implement forward pass (5P)

File: `FCNetwork.py`

Implement the `evaluateLayer` method. This will allow you to evaluate any layer (index 0 is the input layer).

2.1.3 Implement backpropagation (10P)

File: `FCNetwork.py`

Have a look at the manual gradient computation in the unit test and adjust this such that the gradient can be computed for an arbitrary number of layers.

2.1.4 Implement weight update rule (5P)

File: `GradientDescentOptimizer.py`

Implement the gradient update rule for all network parameters. Therefore, make use of the operators specified for the `Variables` class.

2.1.5 Train and save the model (15P)

File: `train_classification.py`

You can adjust the network architecture by adapting the hidden layer specifications. Train the network and save it with the filename `trained_network.pkl`. You can also think about validating your network by splitting the data.

2.1.6 Submission

We will test your method implementations and test your model on a different dataset from the same distribution. The number of points you will get for this task scales with your classification score on our test dataset.

Make sure that you upload all your source code and the trained network. We will not re-train your network but take the version you saved under the name `trained_network.pkl`.

2.2 Task 2 (40P)

The goal of this exercise is to build a CNN for the task of object classification in gray scale images. In order to train and evaluate your network we provide the files `data/deers_and_trucks` and `data/deers_and_trucks_test` which are in the python `pickle` format. These files respectively contain 7000 and 2000 images of deers and trucks which are taken from the CIFAR-10 dataset [5] and converted to gray scale images in order to reduce the training time required to achieve good performances.

A sub-goal of this and the following exercise is to learn to use Google's Tensorflow deep learning framework. This framework can be installed from the following link: `https://www.tensorflow.org/install/`

2.2.1 Build a convolutional neural network (15P)

For this exercise, we provide a Tensorflow graph in `cnn_model.py` which currently only represents a fully connected network. This network can be trained by simply running the `train_cnn.py` script.

Your first task is to modify the `cnn_model.py` file in order to add at least one convolutional and one max-pooling layer to the network. To this end, we made available an helper functions in the `tf_helpers.py` file. One can also consider using the Tensorflow's contrib.layers API[6]

Creating a CNN model which can be trained and that contains at least one max-pooling layer gives the full score for this tasks.

2.2.2 Train, save, load and test the network (15P)

Your second task is to adapt and train the model in order to improve it's performances. To this end you can modify the model and the `train_cnn.py` script. At the end of the script we added a function for saving the model to a folder. You can then use the `test_cnn.py` script to load the model and evaluate it on separate data.

For evaluation, we ask you to submit all your python files and the trained model. For grading we will evaluate the model on a separate dataset and will compare the performances to the fully connected version. Please make sure

[5] `https://www.cs.toronto.edu/~kriz/cifar.html`

[6] `https://www.tensorflow.org/api_guides/python/contrib.layers`

that your model can correctly be loaded and evaluated using the `test_cnn.py` script.

2.3 Task 3 (20P)

In this task you are using RL to stabilize a pole attached to a cart by an unactuated joint, see Figure 5. The cart moves along a frictionless track and is controlled by applying a force of +1 or −1 to it. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center[7].

To get started, you should install the OpenAI Gym framework:

```
pip install gym
```

You can test the installation by for instance executing the following command that prints your Gym version:

```
python -c 'import gym; print(gym.__version__)'
```

2.3.1 Setup the RL problem (5P)

In this sub-task you should finish some parts of the RL problem. Specifically you should

- Add an additional abort criterion
- Compute the probability of the chosen action
- Conduct a simulation iteration

2.3.2 Setup the NN(5P)

- Define the appropriate sizes of your hidden layers

[7]Description taken from: `https://gym.openai.com/envs/CartPole-v0`.

- Define the probability of taking the wrong action

  $$P[\bar{a}] = \bar{a} * (\bar{a} - P[a]) + (1 - \bar{a}) * (\bar{a} + P[a])), \tag{17}$$

  where $P[a]$ is the network output (`action_probability`), $\bar{a}$ is the inverse of the taken action.

2.3.3 Train your network (10P)

We will evaluate your network based on the average performance across several episodes.

Note: The maximum amount of timesteps you can achieve is 200.

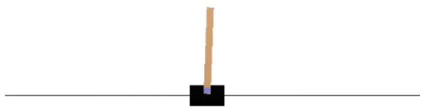

Figure 5: The OpenAI Gym ‘cart-pole-v0’ environment. Image source: `https://gym.openai.com/envs/CartPole-v0`

2.3.4 Submission RL

The final version of the trained model will be saved by the Tensorflow saver. Please alos make sure that you save and submit your version of the `NNModel.py` file such that we are able to reconstruct your model.

3 Submission

To hand in the exercise you have to zip the whole folder (`5_0_dl`) with your code and result files and upload it on moodle. The zip file you upload should have the name *aifr_lastname_ex5* (replace *lastname* with your last name). Make sure that you created and stored the required files and data in the given tasks. Which files are required per tasks can be found at the end of each task description.

References

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.