Assignment: Deep Learning

2ID90, edition 2018-2019 Q3

1 *Overview*

In this assignment you are going to gain practical experience with basic aspects of DEEP LEARNING, in particular with deep artifical neural networks. You will be given a library containing skeleton implementations for handling neural networks. You will gradually learn to use the library and extend it with additional functionality. Finally, you will design and tune a *convolutional neural network* for image classification.

The tasks marked ♣ have to be reported on.

- Make sure that you describe HOW you realized a task,
- report on your RESULTS,
- and supply EVIDENCE (algorithmic details, formulas, tables, figures, etcetera).

2 *Installation*

A MAVEN project[1], called DeepLearning, is provided. It contains the source code for our deep learning library. To install the project

1. Download the assignment, see section 10;
2. Unzip it in a suitable folder of your own choice;
3. Open the project DeepLearning in your IDE;
4. Compile the project: this step will download several libraries from online maven repositories.

If the project compiles, your installation is completed successfully.

[1] This project has been developed and tested in NetBeans 8.2. It should also work in other IDE's that support maven. However, your mileage may vary.

3 *Regression: Three valued function*

Copy the following empty experiment to folder `src/main/java/-experiments`.

```
package experiments;
import nl.tue.s2id90.dl.experiment.Experiment;
import java.io.IOException;

public class FunctionExperiment extends Experiment {
  // (hyper)parameters
  // ...
  public void go() throws IOException {
    // you are going to add code here
  }
  public static void main(String[] args) throws IOException {
          new FunctionExperiment().go();
  }
}
```

In a first step we are going to read some data. The data is going to be processed in so-called batches[2]. Each batch consists of `batchSize` labeled input records. Add `batchSize`[3] to the hyperparameters[4], add the following lines to the `go()` method, and use your IDE to fill in missing imports[5].

[2] Sometimes also called mini-batches.

[3] `int batchSize = 32;`

[4] Hyperparameters are parameters of the learning process that are not learned.

[5] In Netbeans, `Right-Mouse-Click menu/Fix imports`.

```
// read input and print some information on the data
InputReader reader = GenerateFunctionData.THREE_VALUED_FUNCTION
    (batchSize);
System.out.println("Reader info:\n" + reader.toString());
```

Running the resulting program should give you an output similar to the following[6].

[6] To print 10 sample records: `reader.getValidationData(10).forEach(System.out::println);`

```
reader class         : GenerateFunctionData
batch size           : 16
#batches             : 62
#training pairs      : 1000
#validation pairs    : 100
input shape          : (1,1)
output shape         : (1,3)
headers              : [x, x, x^2, exp(x)]
```

From the printed headers you can see that each record has 4 fields: a feature vector with 1 input field ("x") and three result fields ("x", "x^2", and "exp(x)"). The feature or input vector has SHAPE (1,1): that is, it is one vector with 1 element; the output has SHAPE (1,3): one vector with 3 elements[7]. In our program we can pick up these values as follows.

[7] See section 7 for more information about shapes.

```
int inputs  = reader.getInputShape() .getNeuronCount();
int outputs = reader.getOutputShape().getNeuronCount();
```

Next we construct a neural network with `inputs` input neurons and `outputs` output neurons. We do so in a separate method `createModel` that is called from the method `go()`;

```
Model model = createModel(inputs, outputs);
...
Model createModel( int inputs, int outputs ) {
  Model model = new Model(new InputLayer("In", new TensorShape(inputs), true));
  model.addLayer(new SimpleOutput("Out", new TensorShape(inputs), outputs, new MSE(), true));
  return model;
}
```

The created model[8] has an input layer with *`inputs`* neurons and is FULLY CONNECTED to an output layer that has *`inputs`* connections coming in to each of its *`outputs`* neurons. The output layer has LOSS FUNCTION `MSE`, mean squared error. The last step in creation of a model consists of initializing its weights[9]. After that we have a fully initialized model, but we still need to train it. For training we need two additional hyperparameters[10]:

[8] `System.out.println(model);` prints a summary of the model.

[9] model.initialize(new Gaussian());

[10] `int epochs=10; double learningRate = 0.01;`

- `epochs`: The parameter `epochs` is the number of epochs that a training takes. In an epoch all the training samples are presented once to the neural network.

- `learningRate`: Parameter for the gradient descent optimization method.

Add the following code to your experiment's `go()` method to first create the SGD[11] optimizer and then call your experiment's training method.

[11] SGD stands for Stochastic Gradient Descent, a method for optimization of the network weights.

```
// Training: create and configure SGD && train model
Optimizer sgd = SGD.builder()
    .model(model)
    .validator(new Regression())
    .learningRate(learningRate)
    .build();

trainModel(sgd, reader, epochs, 0);
```

After compiling and running[12] your experiment, its output should be similar to the following, ignoring the individual batches.

[12] To appreciate what the `trainingModel` method does, read its source code. In netbeans you can jump to the source code by `Ctrl-Left` click on the method name.

```
Validation after epoch  1: (batch:  31; validation: 0.36179352)
Validation after epoch  2: (batch:  62; validation: 0.19996504)
Validation after epoch  3: (batch:  93; validation: 0.13828962)
Validation after epoch  4: (batch: 124; validation: 0.13278994)
Validation after epoch  5: (batch: 155; validation: 0.15740058)
Validation after epoch  6: (batch: 186; validation: 0.19607270)
Validation after epoch  7: (batch: 217; validation: 0.23963349)
Validation after epoch  8: (batch: 248; validation: 0.28260258)
Validation after epoch  9: (batch: 279; validation: 0.32205325)
Validation after epoch 10: (batch: 310; validation: 0.35727578)
```

Changing the topology of the netwerk, may change its behaviour. To do so, an additional layer may be added to the network, as shown below, where the integers m and n are the number of incoming connections and the number of neurons of the new layer, respectively. The rest of the network needs to be adapted accordingly.

```
model.addLayer(new FullyConnected("fc1", new TensorShape(m), n, new RELU()));
```

- ♣ **Design** a network that fits the data[13], choosing appropriate values for the hyperparameters *epochs*, *batchSize*, and *learningRate*. Given a fitting network experiment with different combinations of *epochs* and *batchSize*.

[13] Letting class `FunctionExperiment` inherit from `GUIExperiment`, instead of `Experiment`, will show plots of the generated stats.

4 *Classification: fashion-MNIST*

We are now going to look at the fashion-MNIST dataset; see Figure 1. This dataset contains labeled grayscale images of 28x28 pixels. There are 60,000 training images and 10,000 validation images.

The 10 labels are: 0: T-shirt/top, 1: Trouser, 2: Pullover, 3: Dress, 4: Coat, 5: Sandal, 6: Shirt, 7: Sneaker, 8: Bag, and 9: Ankle boot.

- Create a new experiment `ZalandoExperiment with learning rate 0.01, batch size 32, and epochs 5.`
- Read the Zalando data.

```
// read input and print some information on the data
InputReader reader = MNISTReader.fashion(batchSize);
System.out.println("Reader info:\n" + reader.toString());
```

- Print the values of a record to get some idea of the data:

```
// print a record
reader.getValidationData(1).forEach(System.out::println);
```

- Show a few images to get more acquinted with the dataset:

```
// add two fields to your Experiment class
String[] labels= {
   "T-shirt/top","Trouser","Pullover","Dress","Coat",
    "Sandal","Shirt","Sneaker","Bag","Ankle boot"
};
ShowCase showCase = new ShowCase(i -> labels[i]);
...
// add these lines to the method go():
FXGUI.getSingleton().addTab("show case", showCase.getNode());
showCase.setItems(reader.getValidationData(100));
...
// add an additional method
public void onEpochFinished(Optimizer sgd, int epoch){
    super.onEpochFinished(sgd, epoch);
    showCase.update(sgd.getModel());
}
```

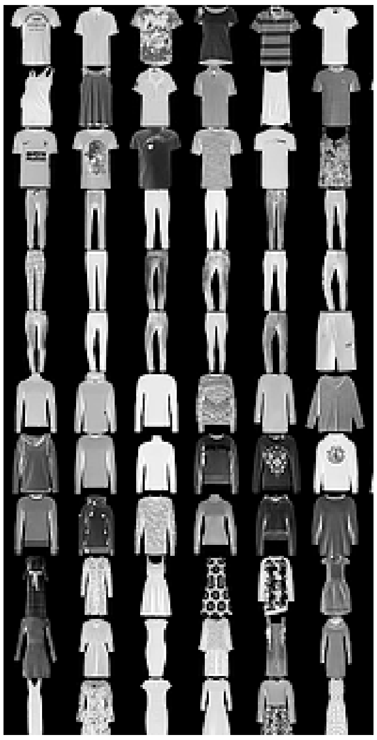


Figure 1: some images from the fashion MNIST dataset, made available by Zalando research.

- Build a fully connected network, having an input layer, a flatten layer, and an output layer. The flatten layer flattens the shape[14] of an input image into a linear shape $(1, m)$[15]. For the output layer, use a softmax activation function that produces a probability distribution over the output classes. Use CrossEntropy as the loss function. See the implementation below.

[14] See Section 7.

[15] For the output layer, the integers m and n are the number of incoming connections and the number of neurons of the layer, respectively.

```
// add flatten layer after input layer
model.addLayer(new Flatten("Flatten",new TensorShape(...));
// add output layer
model.addLayer(new OutputSoftmax("Out",
      new TensorShape(m), n, new CrossEntropy())
);
```

- Finally, create a suitable optimizer. We set a validator[16] that is suitable for classification.

```
Optimizer sgd = SGD.builder()
    .model(model)
    .learningRate(learningRate)
    .validator(new Classification())
    .build();
trainModel(model, reader, sgd, epochs, 0);
```

[16] A validator computes a validation of the network, which is used to indicate the performance of the network. The validator is used after training with a batch using the data in that batch, and after an epoch with the full validation set. Here, the `Classification` validator computes the fraction of correct predictions. So, for this validator a higher value is better.

- ♣ **Optimize** the hyperparameters learningRate and batchSize.
- ♣ Sometimes it is possible to improve your results by preprocessing the data. Try to **implement** MEAN SUBTRACTION. See course notes CS231n part 2 for background on data preprocessing. See Section 8 for details and use the following code skeleton to realize this.

```
public class MeanSubtraction implements DataTransform {
    Double mean;
    @Override public void fit(List<TensorPair> data) {
      if (data.isEmpty()) {
        throw new IllegalArgumentException("Empty dataset");
      }
      for(TensorPair pair: data) {
        ...
      }
      ...
   }
   @Override public void transform(List<TensorPair> data) {
        // To do
    }
}
```

- ♣ Finally, several variations of gradient descent exist. GRADIENT DESCENT WITH MOMENTUM is one of them; See course notes CS231n part 3 on SGD. **Implement** gradient descent with momentum. See Section 9 for details on adding your own gradient descent variant. For gradient descent with momentum you need to memorize the previous update during the gradient descent. This update is of the same dimension as your gradient. A safe way to construct one is as follows.

```
INDArray update;
void update(...) {
    // on the first call of this method, create update vector.
    if (update==null) update=gradient.dup('f').assign(0);
    ...
}
```

5 *Classification: Square, Circle, or Triangle?*

- ♣ Read the SCT dataset[17] with 28x28 grayscale images containing either a square, a circle or a triangle. **Apply** your Zalando network to this dataset and **tune** your hyperparameters.

[17] The SCT dataset is generated on the spot, if you choose big data sizes, its generation is slow.

```
InputReader reader = MNISTReader.primitives(batchSize);
```

- ♣ To improve the performance of the network, **design** a network using convolutional layers, possibly combined with pooling layers. Use the following layer constructors to create these layers.

```
/* Convolution2D layer constructor */
public Convolution2D(String layerName, TensorShape inputShape,
   int kernelSize, int noFilters, Activation activation
);

/* Pooling layer constructor */
public PoolMax2D(String layerName, TensorShape inputShape,
   int stride
);
```

Our convolutional layers have stride 1, and zero-padding such that width and height of output images equal those of the input images. Note that this limits the choice of the kernel size. For the pooling layer the stride has to be a divisor of the image size.

- ♣ **Implement** L2 weight decay. See CS231n course notes. Make a custom `UpdateFunction` that combines a gradient descent update with L2 weight decay. Do not apply weight decay for the bias parameters. Use the following as a code skeleton for a weight decay class.

```
public class L2Decay implements UpdateFunction {
   double decay;
   UpdateFunction f;
   public L2Decay(Supplier<UpdateFunction> supplier,double
    decay){
        this.decay = decay;
        this.f = supplier.get();
   }

   @Override
   public void update(...) {
      ...
   }
}
```

This class can then be applied as follows in building an optimizer.

```
// while building an optimizer
Optimizer sgd = SGD.builder()
...
.updateFunction(
    () -> new L2Decay(GradientDescentWithMomentum::new,0.0001f)
  ).build();
```

- ♣ **Implement** Adadelta , a gradient descent alternative. See section 10 for a paper describing this method. The main advantage of AdaDelta is that it automatically chooses the learning rate.

6 *Convolution: CIFAR10*

- Read the Cifar10 dataset, a dataset containing 28x28 RGB images with 10 different labels: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck; see Figure 2. This dataset contains 50,000 training images and 10,000 test images. For this exercise we only use 5 out of the 10 available labels.

```
Cifar10Reader reader = new Cifar10Reader( batchSize, 5);
```

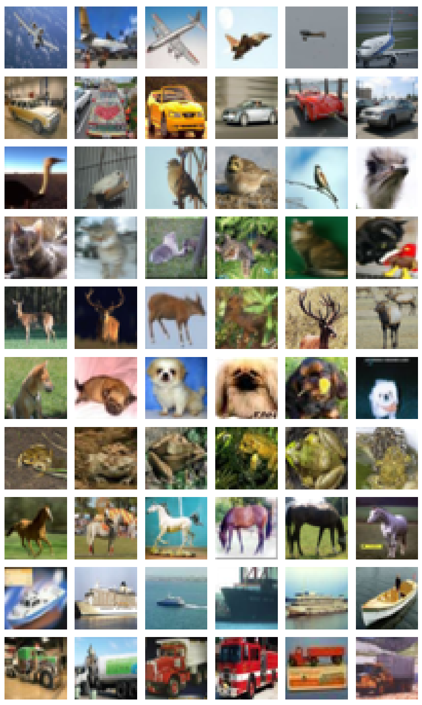


Figure 2: Some Cifar10 images.

The second parameter in the above constructor is the number of labels; so, in the above case only images classified with the first 5 labels are read.

- Inspect the data.
- Normalize the data.
- ♣ **Design** a network for this dataset.
- ♣ **Tune**: parameters, #layers, layer sizes, etcetera; targeting 70+ % accuracy.
- ♣ Optionally, **implement**
  - new layers: DropOut; see CS231n notes;
  - Batch Normalization (hard);
  - ...

7 *Appendix: Shape*

A shape is a tuple that describes how many numbers an array has in each dimension. For instance, shape $(3,5)$ stands for a 2-dimensional array with 3 rows and 5 columns; see Figure 3.

When describing the input (or output) of a neural network there often is an additional 1 at the beginning of the shape. This is due to the fact that the network can actually handle, for efficiency reasons, multiple vectors simultaneously. So, for example if the input of a network has shape $(1,3,5)$ it actually accepts, for instance 2 vectors of shape (3,5) as input, but only when they are combined in an array of shape $(2,3,5)$; see Figure 4.

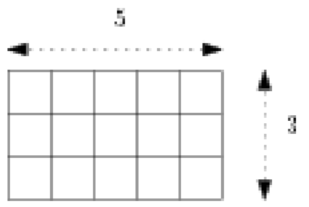
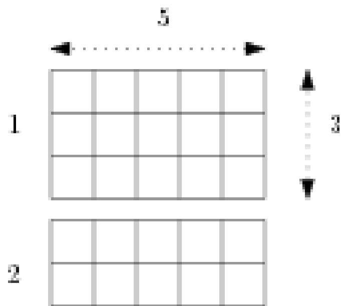


Figure 3: Array with shape $(3,5)$.



When defining the layers of a network we typically need to specify the shape of its input. For that we use the class `TensorShape`. We then write, for instance, new TensorShape(10) to specify that a layer has 10 inputs, or new TensorShape(20,25,3) to specify that the input is a 20x25 image with 3 layers, r, g, and b. There are two peculiarities with this class:

- For the constructor of the class `TensorShape`, the leading 1, as described in the previous paragraph, is left out. So, `new TensorShape (10)`, results in shape $(1, 10)$.
- The order of the arguments of the TensorShape class differs from the above introduced shapes. For example, `new TensorShape(20,25,3)`, gives shape $(1, 3, 20, 25)$.

8 *Appendix: Data preprocessing*

Data transformation in our deep learning library is done by implementing the `DataTransform` interface.

```
public interface DataTransform {
    /** computes statistics for the dataset consisting of input-output pairs, these statistics
     * are used in the transform method. Note that the stats are only based on the input.
     * @param pairs dataset **/
    void fit(List<TensorPair> pairs);

    /** transforms the dataset, using the statistics calculated by the fit method.
     * @param pairs dataset **/
    void transform(List<TensorPair> pairs);
}
```

Data is then, for instance, transformed as follows:

```
DataTransform dt = new MyDataTransform();
dt.fit(myTrainingData);
dt.transform(myTrainingData);
dt.transform(myValidationData);
```

Each data element is a tensor pair, consisting of an input tensor and an output tensor. The data transformation only operates on the input tensor. The data[18] inside a tensor is stored in an n-dimensional array `INDArray`. Such arrays and operations on it are implemented in a highly optimized library, called N-DIMENSIONAL ARRAYS FOR JAVA, or in short ND4J[19].

[18] To get the INDArray in a tensor: Tensor t; ...; INDArray a = t.getValues();

[19] For a quick introduction see the user guide and syntax at https://nd4j.org/. For the full api of ND4j's `INDArray`, see its javadoc.

There are typically two types of operations on an `INDArray nda`; those that return their result in a newly created array, and those that store their results 'in place' in `nda` itself. The latter operations typically have a suffix 'i'. For instance, `nda.addi(1)` adds 1 to each element of `nda`. The statement `nda.add(1)` on the otherhand does not change `nda` at all, but returns a newly created array.

9 *Appendix: Gradient Descent variants*

For your own gradient descent variant you need to create a class that implements the `UpdateFunction interface and inform`[20] `the SGD optimizer to use it.` As a result the optimizer will automatically create two separate `UpdateFunction` objects for each

[20] The optimizer actually needs a parameter-less method that can create an UpdateFunction object. Two ways to realize that are: 1) a reference to a constructor of a class, e.g. `MyGradientDescent::new`; or, a lambda function, e.g. `() -> new MyGradientDescentVariant()`.

layer in the network: One for updating the weights and one for updating the bias.

```
class MyGradientDescentVariant implements UpdateFunction {...}
....
Optimizer sgd = SGD.builder()
        ...
        .updateFunction(MyGradientDescentVariant::new)
        .build();
```

Below the interface `UpdateFunction is` given.

```
public interface UpdateFunction {
  /** A typical implementation of this interface does a gradient descent step, like
   *      array <-- array - (learningRate/batchSize) * gradient.
   * Other implementations may decide to ignore e.g. the learningRate.
   * As a side effect the method update makes all components of the gradient vector zero.
   * Typically, this is done by the call: gradient.assign(0).
   * @param array         array that is to be updated, could contain either weights or biases.
   * @param isBias        true if and only if array represents bias values, as opposed to weights.
   * @param learningRate  learning rate for gradient descent
   * @param batchSize     number of samples whose resulting gradients are accumulated in gradient
   * @param gradient      accumulated gradient **/
  void update(INDArray array, boolean isBias, double learningRate, int batchSize, INDArray gradient)
}
```

The default implementation of this interface is given below. The Nd4j method called there is a generalized vector addition: $\vec{v} \leftarrow \vec{v} + \mu\vec{w}$, implemented in a native BLAS[21] library.

[21] Basic Linear Algebra Subprograms

```
public class GradientDescent implements UpdateFunction {
  /* Does a gradient descent step with factor 'minus learningRate' and corrected for batchSize. */
  @Override public
  void update(INDArray array, boolean isBias, double learningRate, int batchSize, INDArray gradient){
      double factor = -(learningRate/batchSize);
      // array <-- array + factor * gradient
      Nd4j.getBlasWrapper().level1().axpy( array.length(), factor, gradient, array );
  }
}
```

10 *Appendix: Resources*

- Assignment text and programming resources are available from the 2018-Q3 2ID90 canvas website.
- Course notes: CS231n Convolutional Neural Networks for Visual Recognition part 1, part 2, part 3; Andrej Karpathy, Stanford Computer Science.
- ADADELTA: An Adaptive Learning Rate Method; Matthew D. Zeiler.
- ND4j overview
- INDArray javadoc