Assignment 4
Artificial Neural Networks
for
Regression & Classification

Mats Gustafsson & Mohammed Al-Jaff
October 7, 2021

Contents

1 Introduction

Help, Inspiration and Hints

- https://www.youtube.com/watch?v=O2wJ3tkc-TU

This assignment assumes you have already gained a reasonable understanding about what is a feedforward artificial neural network (FANN) and the surrounding theory and terminology. The overarching aims here are to learn about

- how single artificial neurons work in practice by first computing weighted sums and then applying a non-linear activation function.
- how softmax layers work in practice by transforming an input vector into an output vector with values that always sum to one.
- the Cross Entropy cost function for use in classification problems
- the power of Stochastic Gradient Descent using *mini-batches*, compared to conventional *epoch* based Gradient Descent
- how common data like signals and images are represented as one/multi-dimensional numerical arrays and tensors in order to be useful for FANN training.
- how sets of training, test and validation examples must be represented and organized in order to use them in the PyTorch framework and similar frameworks.
- how the PyTorch framework works when it comes to implement and train FANNs.
- how to solve large-scale real world classification problems using FANNs in PyTorch.

2 Some general theory and tips

2.1 Single artificial neurons

In the context of FANNs, the output y from a single artificial neuron is usually determined as $y = \varphi(v)$ where $v = w_0 + w_1x_1 + w_2x_2 + ... + w_dx_d$ where $\varphi()$ is a nonlinear *activation function* defined for example as $\varphi(v) = \max\{0, v\}$ and $\varphi(v) = \tanh(v)$.

Problem 3: A single artificial neuron
(a) Make a plot using Python that shows $\varphi(v) = \max\{0, v\}$ and $\varphi(v) = \tanh(v)$ on the interval $[-5, 5]$.
(b) How many adjustable parameters does a single neuron have when its has d inputs?

2.2 Multilayer FANNs - also known as Multilayer Perceptons

Fig. 1 illustrated a FANN with two inputs, 5 artificial neurons in the first hidden layer, 2 artificial neurons in the second layer, and one single output neuron. This particular kind of networks are often called Multilayer Perceptrons. Notably, such networks do not have any direct connections to the output neuron(s) from the input or any of the hidden layers other than the final one. Having this kind of connections is indeed a possibility, but not considered any further here.

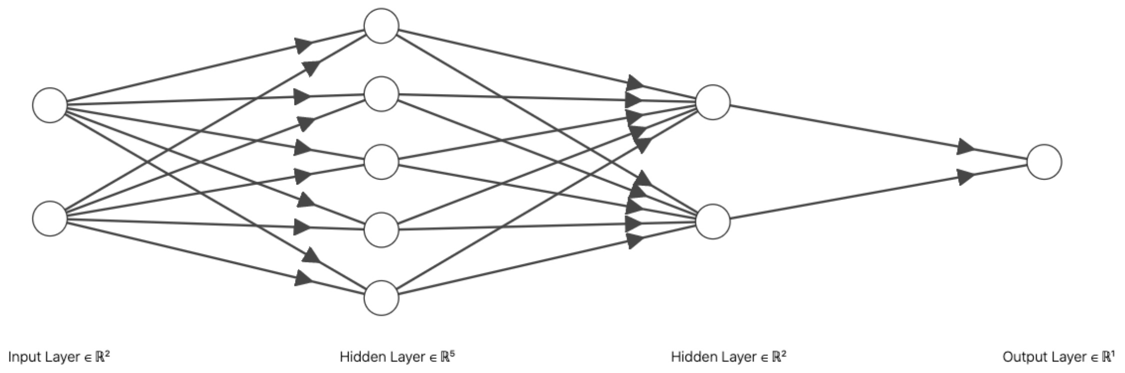


Figure 1: A FANN with two hidden layers of neurons. Notably, this network architecture does not have any direct connections to the output node(s) from the input or any hidden layer except for the last one.

Problem 4: Multilayer FANN
(a) Assuming that each neuron works as described in the previous subsection above (meaning a weighted sum plus a bias, followed by a nonlinearity), how many tunable parameters does the network have?

(b) Let the weight from input i to neuron j in the first hidden layer be denoted by U_{ji} and let its bias term be denoted $U_{j,0}$. Similarly, let the weight from output i of the first hidden layer to neuron j in the second layer be denoted V_{ji} with bias term V_{j0}. Finally, let the weights from output i of the second hidden layer to the output neuron be denoted $w_{\text{out},i}$ with bias term $w_{\text{out},0}$. Using this notation, assuming that all nodes are using the same activation function $\varphi()$, write down a mathematical expression (formula) for the network output $y(x_1, x_2)$ where x_1 and x_2 denote the network input.

Tip: For part (b) you may start as follows. $y(x_1, x_2) = \varphi(w_{\text{out},0} + w_{\text{out},1} \cdot z_1(x_1, x_2) + w_{\text{out},2} \cdot z_2(x_1, x_2))$ where $z_1(x_1, x_2)$ and $z_2(x_1, x_2)$ are the two outputs from the second hidden layer. The final mathematical expression should be a function of (x_1, x_2) as well as of all weights and biases in the network

2.3 CNNs - Convolutional ANNs

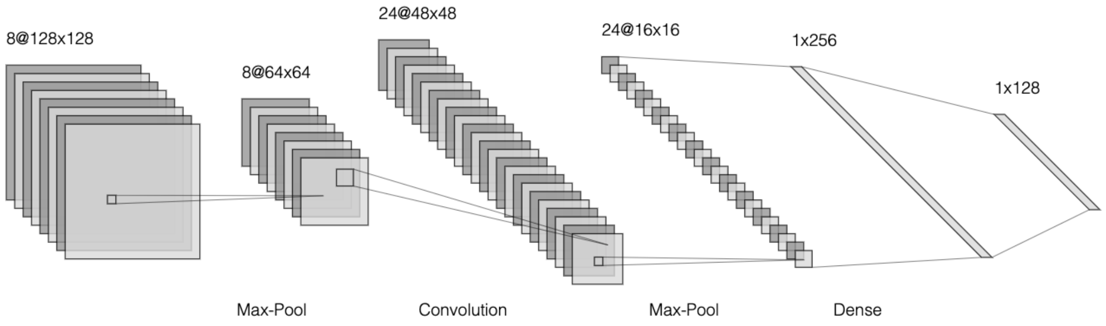


Figure 2: An example of a convolutional ANN.

Although fully connected FANNs can be quite powerful for both regression and classification problems, when it comes to processing and analysing images and temporal data, there are multiple reasons for using convolutional ANNs (CNNs). An example CNN is presented in Fig. 2. Three of the most important reasons for using a CNN instead of a fully connected FANN in for this kind of data are:

- Much fewer total number of parameters to tune for the same number of neurons.
- Much faster computations because they are mainly mathematical convolutions that can be rapidly calculated using dedicated hardware and the Fast Fourier Transform algorithm.
- Prior knowledge/assumptions are build into the FANN network about the problem task. In particular, the networks are often designed to create a hierarchy of so called "features maps" where each feature map is designed to pick up particular input features that may occur at any location in the image, and that are nontrivial to discover/engineer manually.

Problem 5: CNN
Consider a single CNN features map produced directly from an input image. Assume that the size of the kernel is $m \times m$.
(a) How many adjustable parameters does such a feature map have?
(b) Assume your CNN has 2 feature maps produced directly from the input image, all with kernel size $m \times m$. How many adjustable parameters are there in this case? Why does the number of adjustable parameters not depend on the size of the input image?

2.4 The Softmax layer

When designing and training a FANN for classification, the standard operating procedure is to have a so-called *Softmax* layer as the final one of the network, instead of a conventional neuron layer where each neuron works independently of the others.

If there are K classes the Softmax layer consists of K different outputs nodes/neurons, where node k gets the input v_{kn} from the previous layer when the current network input is $\mathbf{x}_n$. The output y_k for node k can be written as

$$y_{kn} = \frac{e^{v_{kn}}}{\sum_{k=1}^{K} e^{v_{kn}}}$$

Problem 6: A Softmax layer
Assume you have a 3-class problem and that the current input $\mathbf{x}_n$ gives $v_{1n} = 1$, $v_{2n} = 2$, $v_{3n} = 4$.

(a) Determine the corresponding three outputs y_{kn}.

(b) Determine the corresponding three outputs y_{kn} when instead $v_{1n} = 1$, $v_{2n} = 3$, $v_{3n} = 9$.

(c) Based on the results in (a) and (b), try to explain the name “Softmax”.

2.5 Cross Entropy for classification

While the mean squared error is the standard cost function used to train ANNs when the response values are real numbers (regression), the standard objective function used for classification problems is the Cross Entropy. For two discrete probability distributions $\mathbf{p}$ and $\mathbf{q}$ specifying the probabilities of d different possible outcomes, the Cross Entropy is defined as

$$H(\mathbf{p}, \mathbf{q}) = -\sum_{i=1}^{d} p_i \log(q_i) = \sum_{i=1}^{d} p_i \log(1/q_i)$$

For ANN learning, $\mathbf{q}$ is typically the d-dimensional output from a final Softmax layer (see previous section) while $\mathbf{p}$ is a “one-hot-encoding” target vector that contains one at a single unique position with the remaining elements being zero. For example, if $d = 4$ and the true class for an examples $\mathbf{x}_n$ is class 3, then $\mathbf{q}(n) = [0, 0, 1, 0]^T$.

Since $0 \cdot \log(0) = 0$ by definition, this means that for the example $\mathbf{x}_n$, the Cross Entropy for this example will be $-\log(q_3)$, meaning no cost if $q_3 = 1$ and a very big Cross Entropy when $q_3 \approx 0$. For a problem with N training examples the objective function $J()$ used for training of an ANN with its parameters collected in $\boldsymbol{\theta}$ is simply the average Cross Entropy across all examples:

$$J(\boldsymbol{\theta}) = -\frac{1}{N}\sum_{n=1}^{N}\sum_{i=1}^{d} p_{in} \log(q_{in}) = \frac{1}{N}\sum_{n=1}^{N}\sum_{i=1}^{d} p_{in} \log(1/q_{in})$$

Problem 7: Average Cross Entropy
Assume you have a a 3-class classification problem where your ANN model produces the output $\mathbf{p}_1 = [0.01, 0.9, , 0.09]$ for input example $\mathbf{x}_1$ and the output $\mathbf{p}_2 = [0.8, 0.15, 0.05]$ for input example $\mathbf{x}_2$. Determine the average Cross Entropy for this result if $\mathbf{x}_1$ belongs to class 2 and $\mathbf{x}_2$ belongs to class 1.

Tip: This means that the desired output distributions for the two inputs $\mathbf{x}_1$ and $\mathbf{x}_2$ are $\mathbf{q}_1 = [0, 1, 0]$ and $\mathbf{q}_1 = [1, 0, 0]$, respectively.

2.6 The Root Mean Squared Error

Often when training ANNs for regression problems we use the mean squared error as the objective function to be minimized. However, a quantity which is more easily interpreted is the square root of this performance measure. This is known as the Root Mean Squared Error (RMSE) and defined as

$$J_{RMSE} = \sqrt{\frac{1}{N}\sum_{n=1}^{N}(y_n - \hat{y}_n)^2}$$

where y_n is the desired (target) value and $\hat{y}_n$ is the corresponding network prediction for example n. In practice this quantity is usually not used for training but only during final evaluation of the prediction model built. This is because it is much easier for a human to have an practical interpretation of the size of an RMSE vale, compared to the corresponding mean square error. This is because the RMSE is expressed in the same unit as the response y_n.

Problem 8: RMSE
Assume y_n has the unit kg. Then what will be the unit of the mean squared error and the RMSE, respectively?

2.7 Stochastic Gradient Descent, Mini-Batches and Epochs

If the problem is to minimize the objective function

$$J(a, b) = (y_1 - a \cdot x_1 - b)^2 + (y_2 - a \cdot x_3 - b)^2 + (y_3 + a \cdot x_3 - b)^2$$

using Gradient Descent, we should move in the opposite direction of the gradient $\mathbf{g} = [g_1, g_2]^T$. Here the partial derivatives of the gradient are:

$$g_1 = \frac{\partial J}{\partial a} = -2(y_1 - a \cdot x_1 - b) \cdot x_1 - 2(y_2 - a \cdot x_2 - b) \cdot x_2 - 2(y_3 - a \cdot x_3 - b) \cdot x_3$$

$$g_2 = \frac{\partial J}{\partial b} = -2(y_1 - a \cdot x_1 - b) - 2(y_2 - a \cdot x_2 - b) - 2(y_3 - a \cdot x_3 - b)$$

A *stochastic approximation* of the gradient is obtained if we use only one or two of these terms to approximate the partial derivatives, for example:

$$g_1 \approx \hat{g}_1 = -2(y_2 - a \cdot x_2 - b) \cdot x_2$$

$$g_2 \approx \hat{g}_2 = -2(y_2 - a \cdot x_2 - b)$$

Moving in the opposite direction of $\hat{\mathbf{g}} = [\hat{g}_1, \hat{g}_2]^T$ determined by such an approximation is known as Stochastic Gradient Descent and comes with at least two key advantages related to speed and sensitivity to getting trapped in local minima.

A mini-batch: In the simple example above the stochastic approximation of the total sum was based on using only one of the terms each time you would like to make an approximate Gradient Descent step. Another alternative is to use a subset of terms for each update. Such a subset is known as a *mini-batch*.

Epoch: In this context the term *epoch* is also often employed. This refers to a complete cycle through the whole set of N training examples. Thus if the number of training examples is 100 and the mini batch size is 10, then one epoch is completed after 10 consecutive mini batches have been used to take 10 approximate Gradient Descent steps.

Adam - an adaptive stochastic gradient method: Many different algorithms have been proposed to automatically adjust the size and direction of the steps taken during Stochastic Gradient Descent search. The main problems are to avoid oscillations in the search direction from one iteration to the next and to ensure convergence close to a local minimum without just hovering around it. One of the most popular algorithms proposed in 2015 to solve this problem is called *Adam* and is available in both PyTorch and Tensorflow2. Remark: The name Adam is derived from adaptive moment estimation, which has to do with estimating how much the search direction is oscillating.

Problem 9: Advantages of Stochastic Gradient Descent
What do you think are the main advantages of Stochastic Gradient Descent in terms of computational savings if the number N of training examples is very large, and risks of getting stuck in small/minor local minima (when they exist)?

Tip: Why does conventional Gradient Descent becomes very slow when the number of training examples becomes extremely large? Why is conventional Gradient Descent not successful in terms of finding the global optimum when the objective function has a lot of shallow local minima?

3 Building and training Neural Networks in PyTorch

In the following we will go step by step though how to arrive at a general workflow for training an ANN on a dataset using mini-batch Stochastic Gradient Decent with performance monitoring during the training. This general workflow applies to training ANNs for both regression tasks and classification tasks. For a first brief overview, see the following list of steps:

1. Converting raw data into convenient PyTorch `Dataset` objects or downloading standard common datasets.
2. Splitting the origal data into a 'Train-', 'Validation-' and 'Test-' set.
3. Using PyTorch `DataLoader` to make life easier when doing mini-batch Stochastic Gradient Decent.
4. Defining, designing and initiating ANN architectures using the `nn.Sequential(..)` from the `torch.nn` submodule which contains useful predefined ANN functionality and building blocks.
5. Picking a suitable PyTorch cost. NOTE: In PyTorch there is no difference in semantics between loss and cost function, so be a bit careful when picking your cost function for your regression or classification problem.
6. Specifying and using PyTorch built-in optimizers that automatically take care of the Stochastic Gradient Decent update step for all parameters in our model. This is where we provide this optimizer object with the parameters of our model and specify the learning rate.
7. Define and implement a *model performance evaluation* criterion. This means choosing which prediction/classification performance metrics to use for quantifying the performance of your model.
8. Implement the training loop where the actual iterative training/update of your ANN model happens. This step corresponds exactly to the Gradient Decent update loop you are familiar with from assignments 1 and 3, but this time using Stochastic Gradient Descent. Here we also specify the number of epochs (total number of iterations through the whole dataset), as well as how often we will perform model evaluation using all examples present in both the training and validation datasets. More specifically, this step consists of a double loop:
   - Inner loop: For each epoch, iterate through a sequence of single steps of mini-batch Stochastic Gradient Decent parameter updates. This practically boils down to doing a single ordinary Gradient Decent update of the model parameters while using a stochastic cost defined using only a small randomly selected subset of the training examples, meaning the examples in the current mini-batch.
   - Outer loop: Embed the inner loop in an outer loop iterating over the desired/maximum number of epochs specified. After each block of a pre-defined number of completed epochs, make a model evaluation using all examples present in both the training and the validation datasets
9. Evaluate the final trained model on the independent external test dataset to arrive at an estimate of its performance in terms of our chosen metrics.

In the following pages, will go through how to implement each of the above steps in PyTorch. In order for you to have a preliminary mental model of how all the above steps translate into code and relate to each-other, the next page contains the corresponding pseudo-code (still without deails - to be filled in as we proceed).

```python
# CODE TEMPLATE FOR TRAINING NEURAL NETWORKS IN PYTORCH - Conceptual / No-code
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset,
from torchvision import datasets
from torchvision.transforms import ToTensor,

# 1. & 2. Split or get  Data into train, validation and test sets

# 3. Make a DataLoader object to take of 'serving' you mini-batches of your data.

# 4. Define and construct your neural network architecture.

# 5. Define and Pick a suitable cost function depending on your problem type
# regression or classification problem

# 6. Define and Set an Optimizer object that takes care of
# the Stochastic Gradient Decent update step.
# Specify also the model parameters and learning rate

# 7. Define how you evaluate your model performance as a function.

# 8. Set number of epochs to train for
#    Set the epoch interval  model performance evaluation, ie evaluate model every k'th epoch.
#    NN Training
#
#Outer loop: Iterate through whole dataset nr_epochs times and evaluate repeatedly
for epoch_i in range(nr_epochs):
    # Model Performance evaluation block
    if epoch_i%eval_every_kth ==0: # Eval model very k'th epoch.
        # set model into evaluate mode #
        # EVALUATE Model 'performance' on whole train and  validation dataset
        # reset model into train mode.
        # track performance measures from both train and val sets

        # Optional stopping criteria based on validation and/or
        # train set performance/cost to ovoid overfitting.

    # Inner loop: Model Parameter update using mini-batch SGD  blocks
    for X_batch, y_batch in my_dataloader:  # Iterate through  mini-batches
        # Make Prediction on mini-batch

        # Compute cost for mini-batch

        # Set the grads of all model params to zero.

        # compute and populate gradients of model params

        # Instruct optimizer to take one update step (using SGD)

# 9. Evaluate Model 'performance' on whole train and Validation AND TEST dataset
```

3.1 The PyTorch Dataset object does "tensoring" and pre-processing

Raw collected data can be stored inside a computer file and storage system in very many ways and in very many formats. However, when we want to feed our inputs $\mathbf{x_n}$ into our ANN models built in PyTorch, then these inputs must be already in the form of torch tensors. Here the PyTorch object `Dataset` will be explained, as can help the user with performing the required transformation into tensors ("tensoring") and possibly additional pre-processing. For a starter you are expected to read the following:

`https://pytorch.org/tutorials/beginner/basics/data_tutorial.html#datasets-dataloaders`

3.1.1 Already prepared/preloaded PyTorch Datasets

Many image datasets commonly used for education and competitive comparisons of different algorithms are available for automatic downloading via the sub-module `datasets` inside the package `torchvision`. This package is a dedicated ANN Library in PyTorch for everything related to image processing and computer vision, such as functionality for getting image/video datasets, torch mathematical operations common for images etc. In this assignment we will later on use the already provided FashionMNIST dataset, which is a large collection of 60000 28x28 pixel images of clothing items, each with a label from 10 different classes of clothing. Here follows a code snippet that accesses the FashionMNIST dataset using the PyTorch framework.

```python
## Code to access the FashionMNIST
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets

from torchvision.transforms import ToTensor, Lambda, Compose
import matplotlib.pyplot as plt


# Download training data from open datasets.
training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
)

# Download test data from open datasets.
test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor(),
)
```

The `download` argument indicates if we want to download the dataset to our local storage or not. If we have already downloaded this dataset before, then we set this to False, else to True. If set to True, then in the `root` argument we specify the relative path and folder name to download the dataset into, in this case in the folder "data", which PyTorch will create for us. Some datasets come with already pre-selected train and test datasets. Moreover, the `train` argument simply specifies which dataset we want for our dataset object. In the above code snippet, we create two dataset objects, one for the training set and one for the test set (which contains 10000 images together with their labels). Finally, the `transform` argument is used to specify that we want all the data to be returned to us in torch tensors. For images we specify the function `ToTensor()` that transforms images into proper torch tensors.

Remark.
For a more through explanation of what each argument means and represents, see the official PyTorch tutorial: `https://pytorch.org/tutorials/beginner/basics/data_tutorial.html`.

3.2 Splitting your data into ’train-’, ’validation-’ and ’test-’ -sets

The PyTorch call
`torch.utils.data.random_split(dataset, [train_size, val_size, test_size])`
helps to split the original dataset into 3 parts being the training-, validation-, and test-set, respectively. As suggested by the names, these sets have different roles:

- *The training set* is used for parameter fitting, meaning minimizing an objective function.
- *The validation set* is used to tune hyperparameters, for example the initial learning rate η and the penalty factor used when the objective function is extended with a regularization term. It is also used for model familiy selection, for example between alternative ANN architectures. A third use is for early stopping, meaning another type of termination criteria for the iterative numerical optimization that makes the parameter fitting. NOTE: When K-fold cross-validation is used, then there is a different training and validation set in each fold.
- *The test set* is used to obtain an independent and therefore unbiased performance estimate for the final prediction model selected based on using both the training and validation sets.

The following code snippet shows an example of how to use this functionality to construct the three sets from your original dataset object.

```python
# TEMPLATE: SPLITTING YOUR DATA INTO 'TRAIN-', 'VAL-' AND 'TEST-' sets.
original_data = # Init your PyTorch Dataset object
size_of_original_data = len(original_data)

# Specify split fractions: !Must sum to 1!
train_fraction = 0.70
val_fraction = 0.10
test_fraction = 0.20 # for show purposes

# Determine size of each set
train_dataset_size = int(train_fraction * size_of_original_data)
val_dataset_size = int(val_fraction * size_of_original_data)
test_dataset_size = int(size_of_original_data - train_dataset_size - val_dataset_size)

# Split whole original data into train, val and test datsets
train_dataset, val_dataset, test_dataset =  torch.utils.data.random_split(original_data,
                                                               [train_dataset_size,
                                                               val_dataset_size,
                                                               test_dataset_size])
# Sanity checking
print(f" Train set Size: {len(train_dataset)}")
print(f" Val set Size: {len(val_dataset)}")
print(f" Test set Size: {len(test_dataset)}")
```

Remark.
For a more detailed exploration of the need for validation and test sets and its relation to the concepts of ***over-fitting*** and ***generalization***, see Section 5.5.3 (p. 131) in the freely available book *Deep learning with PyTorch* by Stevens, Antiga and Viehmann:
`https://pytorch.org/assets/deep-learning/Deep-Learning-with-PyTorch.pdf`

3.3 Define and construct fully connected FANN models in PyTorch

In this section you will learn how to define ***custom architectures*** of ANNs in PyTorch. By architecture, one simply means the number of inputs, the number of hidden layers, the number of output neurons, the number of neurons/nodes in each layer, the type of activation functions, etc. In other words, the architecture defines every module and operation involved in processing an input $\mathbf{x}_n$ into an output $\hat{y}_n$.

There is two ways to specify and build your ANN architecture, either by creating a PyTorch specific class for your network, or by using the more simpler but less flexible `nn.Sequential(..)` method. In this assignment we will only need to be able to construct our ANNs using the latter approach. However, for the task associated with the higher grades 4/5, you will need to construct your network with the more flexible sub-classing method that is described well in one of the official PyTorch tutorials on model building:
`https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html`

Simplified specificatin of your ANN architecture: The general procedure of defining your ANN architecture using `nn.Sequential(..)` is by specifying all the components that your data will flow through inside the `torch.nn.Sequential(....)` model constructor. These components are layers, activation functions and other common pre-defined alternative ANN architecture components.

Weight initialization: Every time you initialize a new ANN model in PyTorch, that model will have its parameters (weights and biases) initialized using a set of fresh randomly selected small values.

Example 1: Linear Regression model viewed as a single neuron with a linear (no) activation function. In this example we want to create an ANN that is nothing more than a linear parametric model that takes in 10-dimensional input vectors $\mathbf{x}_n$ and returns a scalar value that is a weighted sum of the elements in input vector $\mathbf{x}_n$ defined by parameter vector $\mathbf{w}$, also with an bias w_0 added. In other words, we want to build an ANN corresponding to the following mathematical model:

$$y(\mathbf{x};\mathbf{w}) = w_0 + w_1x_1 + w_2x_2 + ... + w_10x_10 = w_0 + \sum_{i=0}^{10} w_ix_i \tag{1}$$

Visually, the network diagram in Fig. 3 corresponds to the above model where the input layer of 10 nodes represents the ***input layer***, meaning the input $\mathbf{x}$ and the connections represent the elements w_i of $\mathbf{w}$. The single node to the right is the output node representing the output value of the network $\hat{y}$. Notably, the addition of the term w_0 is not visible in this figure.

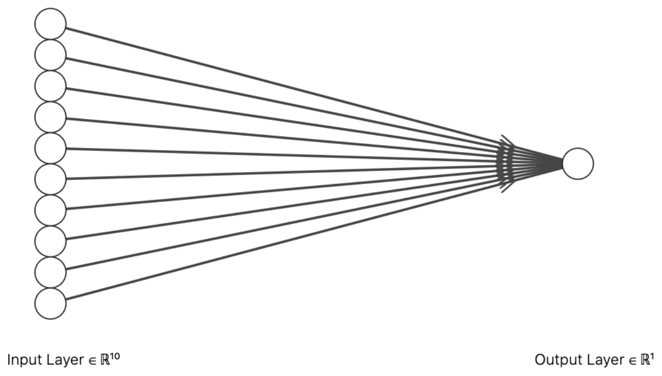


Figure 3: A single artificial neuron with 10 inputs having no non-linear activation function. Such a simple ANN model is equivalent to a linear regression model. Note: The additive bias term is not visible in this network diagram.

The code needed to create this network in PyTorch is presented here:

```
# Defining a single neuron having 10 inputs
model = nn.Sequential(
    nn.Linear(in_features=10, out_features=1)
)
```

Example 2: Building a FANN with 2 inputs, 2 hidden layers and 1 output
For an illustration, see Fig.4. The ReLU activation function should be used for the hidden layer neurons and a tanh activation function should be used at the output so that network output values are restricted to the interval $[-1, 1]$.

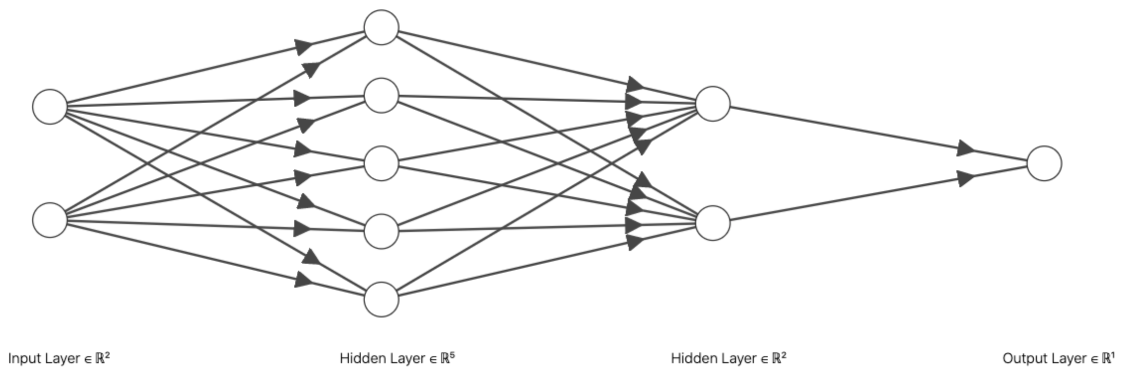


Figure 4: A FANN with 2 inputs, 2 hidden layers and 1 output.

The code needed to create this network in PyTorch is presented here:

```
# Simple way to define a Fully connected feed forward ANN with two inputs, 5
# neurons in the first hidden layer, 2 neurons in the second hidden layer
# and a single output neuron. NOTE: Different types of activation functions
# in each layer
model = nn.Sequential(
    nn.Linear(in_features=2, out_features=5),
    nn.ReLU(),
    nn.Linear(in_features=5, out_features=2),
    nn.ReLU(),
    nn.Linear(in_features=2, out_features=1)
    nn.Tanh()
)
```

Example 3: A FANN with 3 inputs, 2 equally large hidden layers and one output. For an illustration, see Fig.5. Here the ReLU activation function should be used for all hidden neurons and no activation function should be employed for the output neuron.

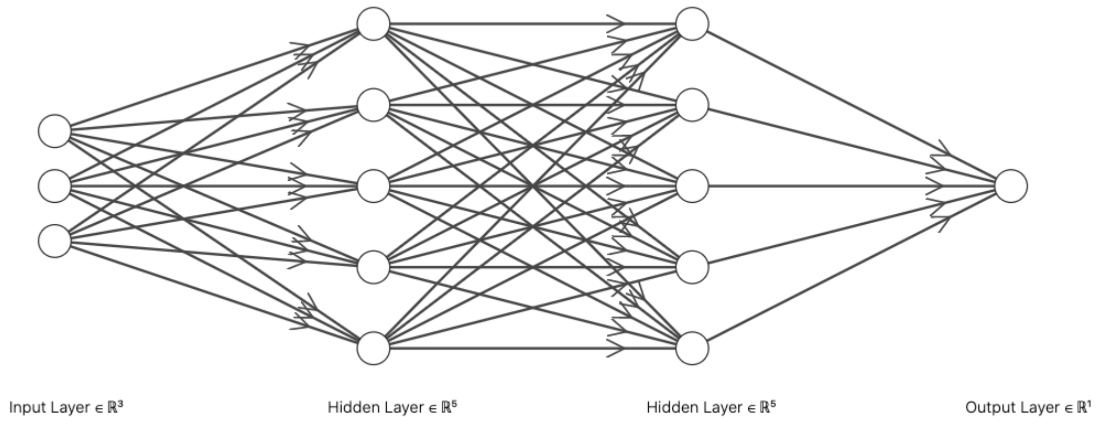


Figure 5: A FANN with 3 inputs, 2 equally large hidden layers and one output.

The code needed to create this network in PyTorch is presented here:

```python
# Simple way to define a Fully connected feed forward ANN
model = nn.Sequential(
    nn.Linear(in_features=3, out_features=5),
    nn.ReLU(),
    nn.Linear(in_features=5, out_features=5),
    nn.ReLU(),
    nn.Linear(in_features=5, out_features=1)
)
```

3.4 How to feed data into your ANN models to make predictions

Once an ANN model has been specified and has been instantiated with some parameter values, it is ready to be used to make predictions using old and new data. In PyTorch this is easily done as follows, feeding in either a single data point $\mathbf{x}_n$, or a set of points (provided that each data point corresponds to one row in a two-dimensional tensor) is as simple as shown in the following code snippet.

```python
model = nn.Sequential(
    # Model architecture components
):

X = # tensor object where each 'row' (first index of tensor) corresponds to a single data-point
y_preds = model(X)
```

Taking data from our `Datset` object in order to feed it into our ANN model during training. is done with PyTorch's `DataLoader` object, which will be the topic of the next section.

3.5 Simplified managing of your datasets in PyTorch using `DataLoader`

PyTorch makes it easy to perform Stochastic Gradient Descent based training of any FANN architecture without having to go into the same kind of details you have seen in previous assignments. Here follows a code snippet illustrating how the management of your datasets can be simplified using `DataLoader`.

```
X_data = # all your x_n's in tensor format
y_data = # all your y_n's in tensor format

# making A PyTorch dataset object for your data
my_dataset = TensorDataset(X_data, y_data)

# making a PyTorch dataloader object
my_dataloader = DataLoader(my_dataset,batch_size=50, shuffle=True, )

for epoch_i in range(nr_epochs): # epoch counter
    for X_batch, y_batch in my_dataloader:
        # Train network on a single mini_batch
        # ie. perform gradient decent on a single data mini batch
        # Every iteration of this inner loop will give you a
        #  mini-batch subset of the original data, presented in X_batch and y_batch
```

3.6 Simplified updating of ANN parameters in PyTorch using `optimizer`

As mentioned above, PyTorch makes it easy to perform Stochastic Gradient Descent based training of any FANN architecture without having to go into the same kind of details you have seen in previous assignments. For the parameter updating, there is an object `optimizer` that takes care of updating each and every parameter in our model. We only need to feed into this optimizer with all parameters defining our ANN architecture, and also specify some other optimization specific arguments like the value of the learning-rate η discussed in previous assignments.

```
# Mini-batch SGD with PyTorch optimizer in the train loop
model = # Define and construct your model

# Pick a Cost function
cost_function = # Pick your PyTorch cost/loss function from torch.nn

# Define and Set an optimizer object.
# Input as arguments the collection of parameters of your model
# and your desired learning rate aka eta..
optim = torch.optim.SGD(model.parameters(), lr=...  )#

for epoch_i in range(nr_epochs): # epoch counter
    for X_batch, y_batch in my_dataloader:
        y_preds = model(X) # Make Prediction on batch
        cost = cost_function(y_preds, y_batch)# Compute cost
        optim.zero_grad() # zero the grads of all model params
        cost.backward() # compute J gradient of all model params
        optim.step() # take one update step for all model params
```

As you can see in the code snippet above, in PyTorch there is a tight relationship between your

ANN model (and its parameters), your optimizer and the cost function. This is especially the case when you perform auto-differentiation by calling `.backward()`. A good mental model to have regarding how these three things relate to each other is explained for example by Stevens, Antiga and Viehmann (section 5.5.2, p.128, see link:

`https://pytorch.org/assets/deep-learning/Deep-Learning-with-PyTorch.pdf`)

Parts of their explanation and the associated key figure is shown in Fig. 6.

Every optimizer constructor takes a list of parameters (aka PyTorch tensors, typically with `requires_grad` set to `True`) as the first input. All parameters passed to the optimizer are retained inside the optimizer object so the optimizer can update their values and access their `grad` attribute, as represented in figure 5.11.

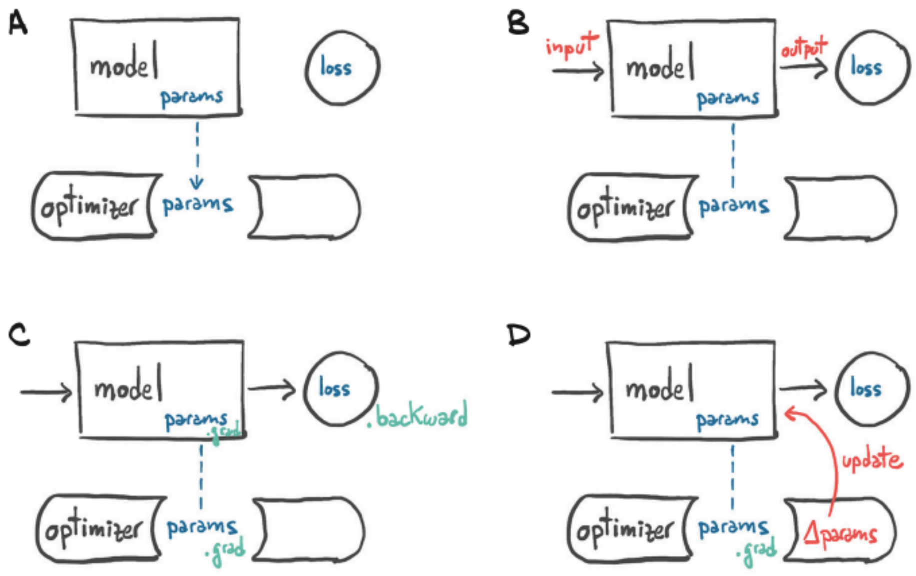


Figure 5.11 (A) Conceptual representation of how an optimizer holds a reference to parameters. (B) After a loss is computed from inputs, (C) a call to `.backward` leads to `.grad` being populated on parameters. (D) At that point, the optimizer can access `.grad` and compute the parameter updates.

Each optimizer exposes two methods: `zero_grad` and `step`. `zero_grad` zeroes the `grad` attribute of all the parameters passed to the optimizer upon construction. `step` updates the value of those parameters according to the optimization strategy implemented by the specific optimizer.

Figure 6: Page 128 from section 5.5.2 of *Deep Learning with PyTorch* by Stevens, Antiga and Viehmann

3.7 Model performance evaluation - during and after training

The following code snippet shows an example of how you can implement a model performance evaluation that takes in a model and a dataset and outputs an performance estimate of the classification accuracy of the model. Note: When we want to obtain a performance estimate, we should not perform any Gradient calculations and we should have only one batch, the whole dataset that we want to evaluate model performance on.

```
 1  from sklearn.metrics import accuracy_score
 2
 3  def evaluate_model_performance(dataset, model):
 4      # Function that takes in a model and a dataset
 5      # and outputs an performance estimate of the classification
 6      # accuracy of the model.
 7
 8      # Make a Dataloader for the dataset.
 9      # Note, we are not performing any SGD here, so our batch
10      # size is the whole # dataset we want to evaluate model performance on.
11      d_loader = DataLoader(dataset = dataset, batch_size=len(dataset))
12
13      cost_function = nn.CrossEntropyLoss() # For classification evaluation
14      model.eval()
15      # Make predictions for the eval dataset
16      with torch.no_grad():
17          for X, y in d_loader:
18              raw_y_preds = model(X)
19
20          y_class_preds = raw_y_preds.argmax(dim=1)
21          eval_cost = cost_function(raw_y_preds, y).item()
22      model.train()
23
24      # compare predictions with true labels and compute performance metric
25      # performance metric in this example is classification accuracy
26      eval_acc = accuracy_score(y_pred = y_class_preds, y_true = y)
27
28      return eval_cost, eval_acc
```

Problem:
Assume you have a single input $x = [x_1, x_2] \in \mathbb{R}^2$ and you want to classify it into one of two classes using the code snippet.
(a) What would your model output vector look like in terms of shape?
(b) What is happening in line 20 in the code snippet? Be very explicit because you need this understanding to have a good mental model of classification accuracy. In particular, give an example of how the variables `raw_y_preds` and `y_class_preds` look in vector/matrix form.

3.8 Putting it all together: The flow of ANN training

Recall that in a previous assignment (The art of fitting parametric predictive models), we established the general workflow for fitting parametric models to simple one-dimensioal datasets. There we used Gradient Decent to minimize a cost function defined using the whole dataset. The overall methodology is still more or less the same when we now want instead to fit the parameters of ANNs to much more complex types of datasets. The difference between what we did then and what we do in this assignment is that we are now using much more flexible/powerful type of model families, ANNs, and we replace conventional Gradient Descent by mini-batch Stochastic Gradient Decent when fitting our our model family to the data. Note: Instead of calling it *parameter fitting*, in the context of ANNs this procedure is called *network training*. Here follows an example of a complete code snippet where all pieces are put together to get a workflow for ANN training.

```python
# CODE TEMPLATE FOR TRAINING NEURAL NETWORKS IN PYTORCH
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset,
from torchvision import datasets
from torchvision.transforms import ToTensor,

# Split or get  Data into train, validation and test sets
train_dataset, val_dataset, test_dataset  # Split or Get these three datasets

# Make a DataLoader object to take of of mini-batches for data.
train_dataloader =  Dataloader(dataset = train_dataset, batch_size = ..., shuffle = True )

model = # Define and construct your neural network architecture.

# Pick a Cost function
cost_function = # Pick your PyTorch cost/loss function from torch.nn

# Define and Set an Optimizer object
optim = # Some choice of optimizer and specify model params and lr

# Define how you evaluate your model performance
def evaluate_model_performance(model, dataset, ...):
    # Function that computes performance measures of your model
    # NOTE: Don not keep track of grads when evaluating model.
    return metric_1, metric_2, ...

for epoch_i in range(nr_epochs): # Iterate through whole dataset nr_epochs times

    if epoch_i%eval_every_kth ==0: # Eval model very k'th epoch.
        model.eval() # set model into evaluate mode #
        # EVALUATE Model 'performance' on whole train and  validation dataset
        train_cost, train_acc, ...= evaluate_model_performance(model=model, dataset = train_set)
        val_cost, val_acc, ... = evaluate_model_performance(model=model, dataset = val_set)
        model.train() # reset model into train mode.
        # track performance measures from both train and val sets

        # Optional stopping criteria based on validation and/or
        # train set performance/cost to ovoid overfitting.

    for X_batch, y_batch in my_dataloader:  # Iterate through a batch
        y_preds = model(X) # Make Prediction on mini-batch

        cost = cost_function(y_preds, y_batch)# Compute cost for mini-batch

        optim.zero_grad() # Set the grads of all model params to zero.

        cost.backward() # compute and populate gradients of model params

        optim.step() # instruct optimizer to take one update step (using SGD)

# EVALUATE Model 'performance' on whole train and Validation AND TEST dataset
test_cost, test_acc, ...= evaluate_model_performance(model=model, dataset = test_set)
```

3.9 Using your trained model for predictions

Using your model after training, does not require calculations of gradients anymore. Therefore, in order to make predictions and store them in a variable `y_preds`, we simply call our model inside a `with torch.no_grad():` block, as shown below.

```
with torch.no_grad():
    y_preds = model(X)
```

4 Task A - 1D Linear and non-linear regression - using simple FANNs

In this first task, we will revisit datasets from a previous assignment (Assignment 3: The Art of fitting parametric predictive models to data) with the aim of fitting some small/simple FANNs to these. The aim is to understand the potential of training these networks to produce predictive models with similar performances as the much simpler models used in the previous assignment.

Task Overview: The task is to train the simplest possible ANN to linearly related but noisy data-points of the kind we studied already in assignment 3. Your job is to define a FANN that consists of a single input for x_n and a single output neuron to predict the desired value y_n. You will use this to fit your own ***synthetic*** dataset coming from an linear relationship between x and y, with all y-values embedded in zero mean gaussian additive noise. The valus of the two parameters (slope and intercpt) defining the linear relationship, as well as the value of standard deviation of the additive noise, you should choose yourself. Thus you should begin by generating something similar to the left subfigure in Fig. 7 and then train your simple FANN using your generated data such that it can perform predictions like the orange curve/line in the right subfigure. All the work in between will more or less a simpler version of the workflow detailed in Chapter 3.

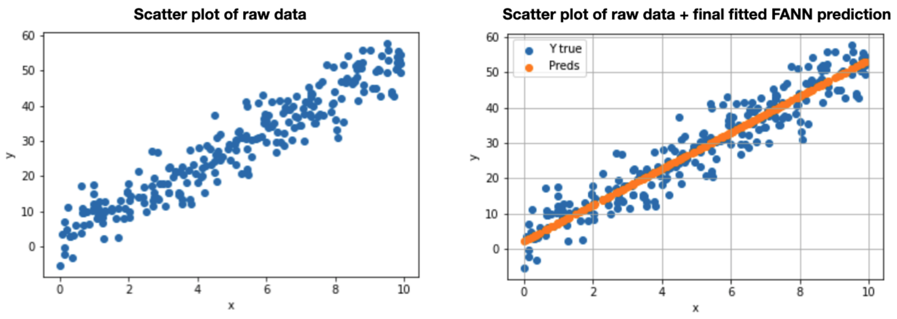


Figure 7: Left: First you should create our own synthetic dataset where you will know exactly what is the true functional relationship between the values x_n and y_n generated. You should also embed the generated response values y_n in zero mean additive noise. Right: Same data with predictions illustrated in orange.

(a) Creating your own Synthetic data: First you should create our own synthetic dataset where you will know exactly what is the true functional relationship between the values x_n and y_n generated.

```
1 a_true = 5.0  #  choose your own value here
2 b_true = 3.0  # choose your own value here
3 sigma_epsilon = 5.0 #choose your own noise level here
4
5 X = np.random.uniform(0, 10, 250) # sample a bunch of #x_n in [0,10]
6 y_obs = a_true*X + b_true + sigma_epsilon*np.random.randn(len(X)) # linear relationship embedded i
7
8 X = torch.tensor(X).reshape(len(X),1).float() # comment about each row being a data-pair
9 y_obs = torch.tensor(y_obs).reshape(len(y_obs),1).float()
```

Pay close attention to lines 8 and 9 where we need to reshape our data so that each row is a single x_n. This is because when we feed our X into the model, the model spits out a prediction for each row, meaning a column vector of the same shape as y_{obs}.

(b) Make a PyTorch dataset object for your synthetic data using `TensorDataset(X, y_obs)`. The use it create a PyTorch `DataLoader` object that will automate your mini-batch selection. This means you need to fill in the following code snippet:

```
my_dataset = TensorDataset(.....)
my_dataloader = DataLoader( .... )
```

You should choose the mini-batch size to be somewhere between 1% to 50% of your data and do not forget to make sure that you are reshuffling (reordering) your mini-batches for every new epoch.

No validation set in this task: In this task, we will not deal with validation or test sets in order to simplify your first FANN training. Thus you should only fit the parameters of your ANN to the training set, like you did in assignment 3. In the following tasks, the training and performance evaluation workflow will be extended.

(c) Create the following network in PyTorch corresponding to the one-dimensional linear predictive model

$$y(x, \mathbf{w}) = y(x, [w_0, w_1]) = w_0 + w_1 x \tag{2}$$

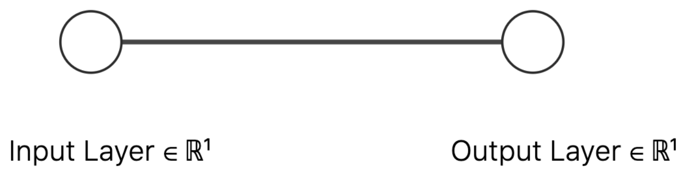


Figure 8: One-dimensional linear parametric model in the form of an ANN.

(d) Is this a regression problem or a classification problem? This will determine what type of cost function you should choose. Thus select the most naturally suitable cost function among the following alternatives.

```
# Alternative loss functions
cost_function = nn.MSELoss()
cost_function = nn.L1Loss()
cost_function = nn.CrossEntropyLoss
```

(e) Choose you optimizer: As your `optimizer` object, choose the `torch.optim.SGD(...)` optimizer and feed in as arguments your model parameters, and choose some appropriate initial learning rate. Note: You might need to adjust the learning rate if your training does not seem to converge to a local minimum.

(f) Train your model using mini-batch Stochastic Gradient Decent by implementing the main training loop. Keep track of the cost of each mini-batch associated with each parameter update.

```
training_minibatch_Js =[]
nr_epochs = # Try
for epoch_i in range(nr_epochs):
    for X_batch, y_batch in my_dataloader:
        # perform gradient decent on mini-batch
        # like you have done many times before,
        # but this time with your optimizer taking care of the actual updates
        # of your model parameters.

# Plot your cost
plt.figure(figsize=[10,5])
plt.plot(training_minibatch_Js)
plt.xlabel('update step i on mini-batch')
plt.ylabel('Cost')
plt.title('Cost during training on train-set (per mini-batch)')
plt.grid()
```

You should arrive at a training cost curve similar to the one in Fig. 9. **You should report your final cost value and your plot!**

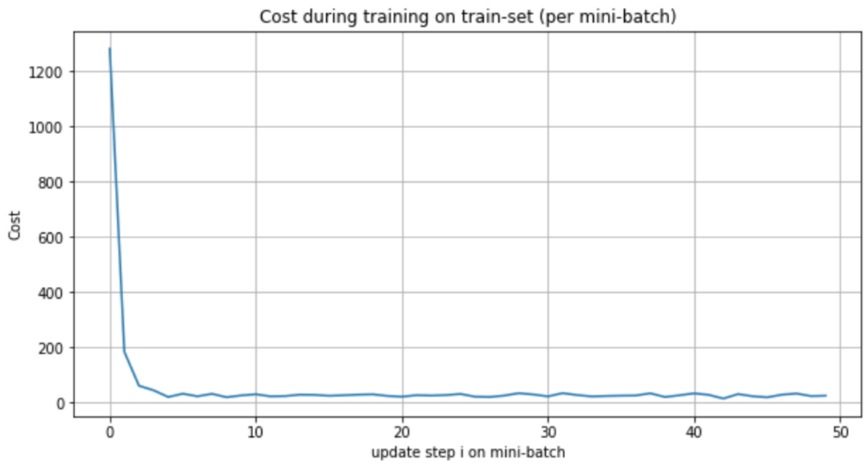


Figure 9: An example of how the cost based on the training set is decreasing as a function of the number of mini-bach updates performed during trianing.

(g) Use your fitted model to make predictions based on the inputs x_n. Plot both your data-points and your model predictions like in the right sub-figure in Fig. 7. Using your model outside of training you should not spend time and energy computing gradients anymore. Therefore you should turn the gradient calculations off by means of `with torch.no_grad():` as shown below.

```
with torch.no_grad():
    y_preds = model(X)
```

(h) Determine and look at your final model parameters. This makes sense in this small scale example. Report on your parameter values and show if your training resulted in parameter values that agree with the true parameters you used to generate the training examples.

```
# Print all parameters in your neural network model
for p in model.named_parameters():
    print(p)
```

(i) Repeat the above steps (b)-(h) for the MarvinMinsky.csv dataset that you worked with in assignment 3. Step (a) for this dataset is done by the following code snippet to get the data into a PyTorch dataset:

```
from numpy import genfromtxt # we use this to load data from a csv into a numpy array
my_data = genfromtxt('dataset_Marvin_Minsky.csv', delimiter=',', skip_header=1) both x and y are i

X = my_data[:,0] #Extract the x_n
y_obs = my_data[:,1] #Extrac the y_n

# comment about each row being a data-pair # reshape to make each row in X a data-pint x_n
X = torch.tensor(X).reshape(len(X),1).float()
# turn to column vector, each row corresponds to each y_n
y_obs = torch.tensor(y_obs).reshape(len(y_obs),1).float()

my_dataset = TensorDataset(X, y_obs) # create your dataset by turning tensors into datasets object
my_dataloader = DataLoader(my_dataset, batch_size=...., shuffle=.....) # create your dataloader fr
```

Here you will need to use a more flexible/powerful ANN architecture. Try some of the architectures shown in the examples in chapter 3 and increase the number of hidden nodes if needed. The goal of this sub-task is to arrive at a trained model that makes predictions simular/better than the ones in Fig. 10.

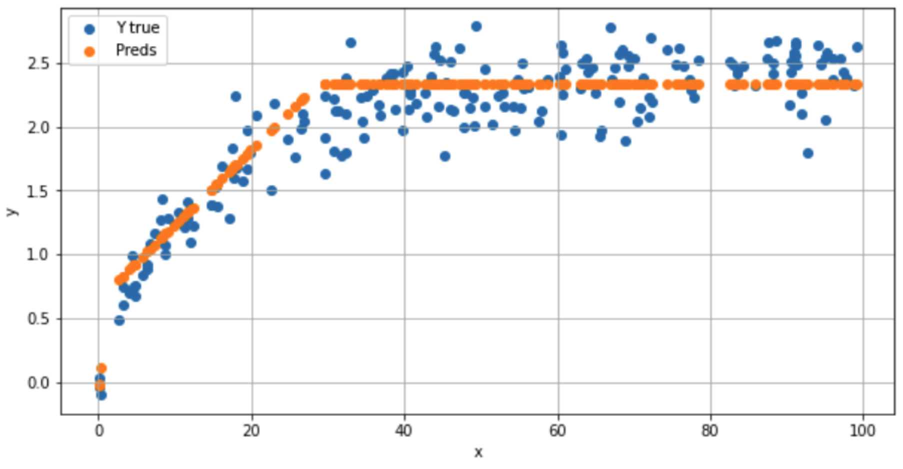


Figure 10: Example of trained ANN predicting the Marvin Minsky data from a previous assignment.

You should report a plot showing the cost curve as a function of iterations. You should also report the final value of the cost function (at the end of the training), which should be below 0.1.

Tip: The internet is your friend. Look for solutions but don't forget to motivate all your choices.

5 Task B: 2-variable regression - using FANNs to fit data drawn from a bell curve.

Here we will continue with fairly simple and small FANN models but for a slightly more challenging 2-variable regression problem, meaning $\mathbf{x}_n \in \mathbb{R}^2$. The specific task is to fit an ANN to a dataset that when plotted looks to come from a bell shaped surface (with noise), for an example see Fig. 11.

More specifically, the is task to fit the parameters of a number of FANN architectures to a dataset $D = \{(\mathbf{x}_n, y_n)\}$ where each 2-dimensional input $\mathbf{x}_n$ is drawn from a uniform distribution on the square $[-5, 5] \times [-5, 5]$ and the response values (our y_n) are determined as

$$y(\mathbf{x}) = e^{-\frac{\mathbf{x}^T\mathbf{x}}{4}} = e^{-||\mathbf{x}||^2/4} \tag{3}$$

Thus the task to train a FANN so that for a new input $\mathbf{x}_n = [x_1, x_2]$, it will output a prediction of $\hat{y}_n$ very close to the desired value.

No validation and test set: Like in the previous task, we will not yet consider splitting our training set into train-, validation and test sets for model performance. The idea again is that you should first get i good at just training networks to any training dataset.

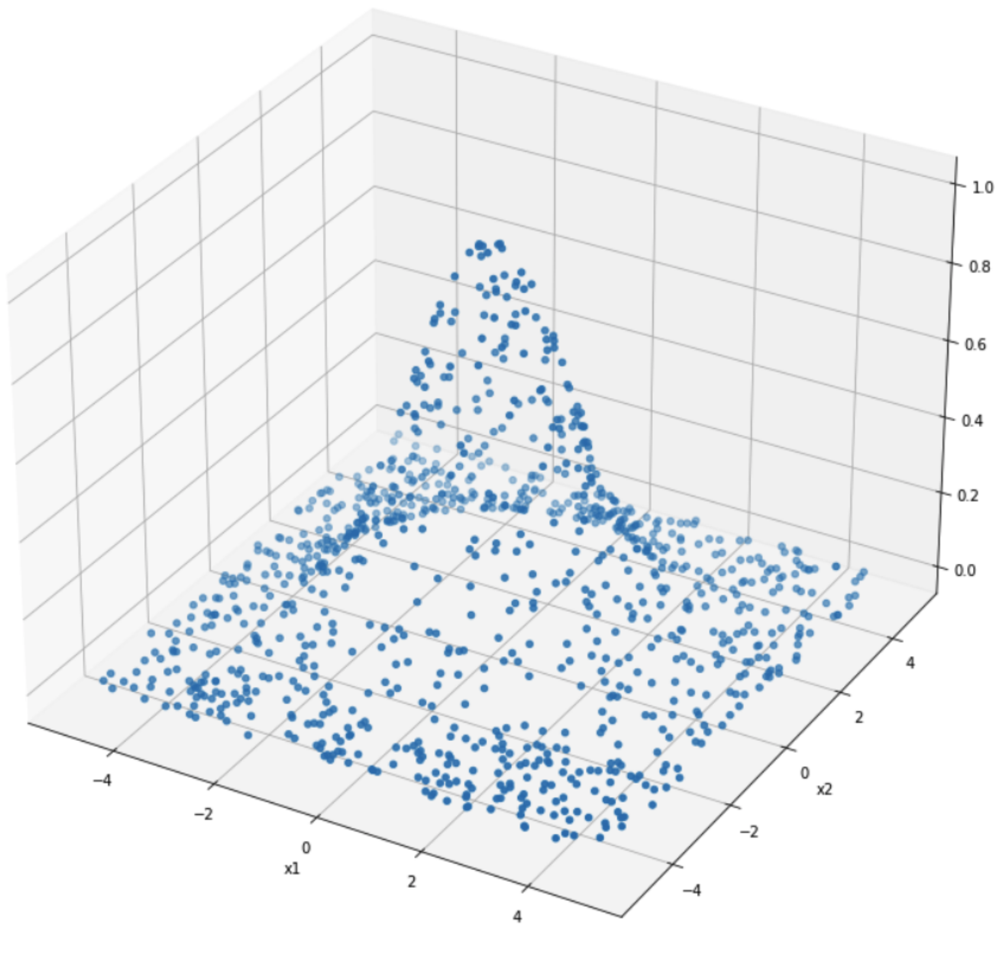


Figure 11: 3D scatter plot of the data-points from the bell shaped function $y(\mathbf{x}) == e^{-||\mathbf{x}||^2}$.

(a) Yous should generate 1000 synthetic data-points from the above bell-curve function with the following code. The idea is then to create a dataset object and a data-loader object ready to be used for the desired ANN training.

```
N = ..... # Nr of data-points to sample and generate
x = 10*torch.rand(size=[N,2]) - 5
mean_true = torch.tensor([0.0, 0.0])
y = torch.zeros(N).reshape(-1,1)
for i in range(len(x)):
    y[i] = torch.exp(-((x[i,:]-mean_true).T@(x[i,:]-mean_true))/4) + 0.04*torch.randn(1)

# Plotting
fig = plt.figure()
fig.set_figheight(10)
fig.set_figwidth(10)
ax = Axes3D(fig)
ax.scatter(x[:,0], x[:,1], y)
ax.set_xlabel('x1')
ax.set_ylabel('x2')

my_dataset = TensorDataset( ....... ) # do
batch_size = .... # do
my_dataloader = DataLoader( .....)  # do...
```

Before continuing, you should make sure that your plot of your synthetically generated datapoints look like in Fig. 11.

(b) Consider the following five different feed forward neural network architectures, the number of layers and the number of nodes in each layer written on the format $I - H_1 - H_2 - \ldots - H_M - O$ where I denotes number of inputs, H_i the number of nodes in the i^{th} hidden layer, and O dentotes the number of nodes in the output layer.

Architecture I: 2-10-1 with ReLU activation for all hidden neurons
Architecture II: 2-50-1 with ReLU activation for all hidden neurons
Architecture III: 2-300-1 with ReLU activation for all hidden neurons
Architecture IV: 2-100-20-5-1 with ReLU activation for all hidden neurons
Architecture V: 2-300-100-20-1 with ReLU activation for all hidden neurons

For each of the above ANN architectures, calculate the number of parameters (weights and biases).

Tip: If needed have a look at your course material or at
`https://towardsdatascience.com/number-of-parameters-in-a-feed-forward-neural-network-4e4e33a53655`

(c) Hopefully you can confirm the fact that the more neurons and the more hidden layers a network has, the more flexible the network is at fitting a given target function.

For each of the above ANN architectures, build these in PyTorch and train each of them to the best of your abilities in order to achieve the smallest cost. You may have to play around with a couple of hyper-parameters to get good results:

- The learning rate η should be adjusted according to the architecture used. If you find your training diverging towards infinity, try with a smaller value.
- Put the maximum number of epochs to a couple hundred for a starter. But this too should be tweaked and changed if you find that your training cost curve is still going down. Do not restrict your imagination when trying around with these.

In all cases, you must report:
(i) how you build the network using Python/PyTorch.
(ii) training cost curves across the number of mini-batches.
(iii) the final cost function value.
(iv) your chosen learning rate.
(v) and any other relevant piece of information that would enable a reader to replicate your runs if they had your data.
(vi) a 3D plot showing the surface generated by the trained ANN when you feed into it the same domain region $[-5, 5] \times [-5, 5]$, meaning similar to Fig.12. The code snippet below will be useful for this task.

```python
x1 = torch.linspace(start=-5, end=5, steps=100)
x2 = torch.linspace(start=-5, end=5, steps=100)
X, Y = torch.meshgrid(x1, x2)

s = torch.stack([X.ravel(), Y.ravel()]).T # all grid coordinates for region.

with torch.no_grad():
    y_preds = model(s) # making predictions on surface grid to get surface values.

# 3D surface plotting
fig = plt.figure()
fig.set_figwidth(10)
fig.set_figheight(10)
```

```
ax = Axes3D(fig)

ax.scatter(s[:,0], s[:,1], y_preds, label='Fitted FANN') #
ax.scatter(x[:,0], x[:,1], y, label='datapoints') # plotting data points
ax.legend()
ax.set_xlabel('x1')
ax.set_ylabel('x2')
```

(f) Compute the cost (mean squared error) for each trained model for a separate test set consisting of $M = 10000$ test examples generated in the same way as the training examples in (a). Which of the trained networks perform best?

(g) Consider the smallest and largest networks:
Architecture I: 2-10-1 with ReLU activation for all hidden neurons
Architecture V: 2-300-100-20-1 with ReLU activation for all hidden neurons

In this sub-task, provide results/answers to the following: (i) train both of the above two network on a synthetic dataset, exactly like you did above, but that only consists of 50 (!) data-points instead.

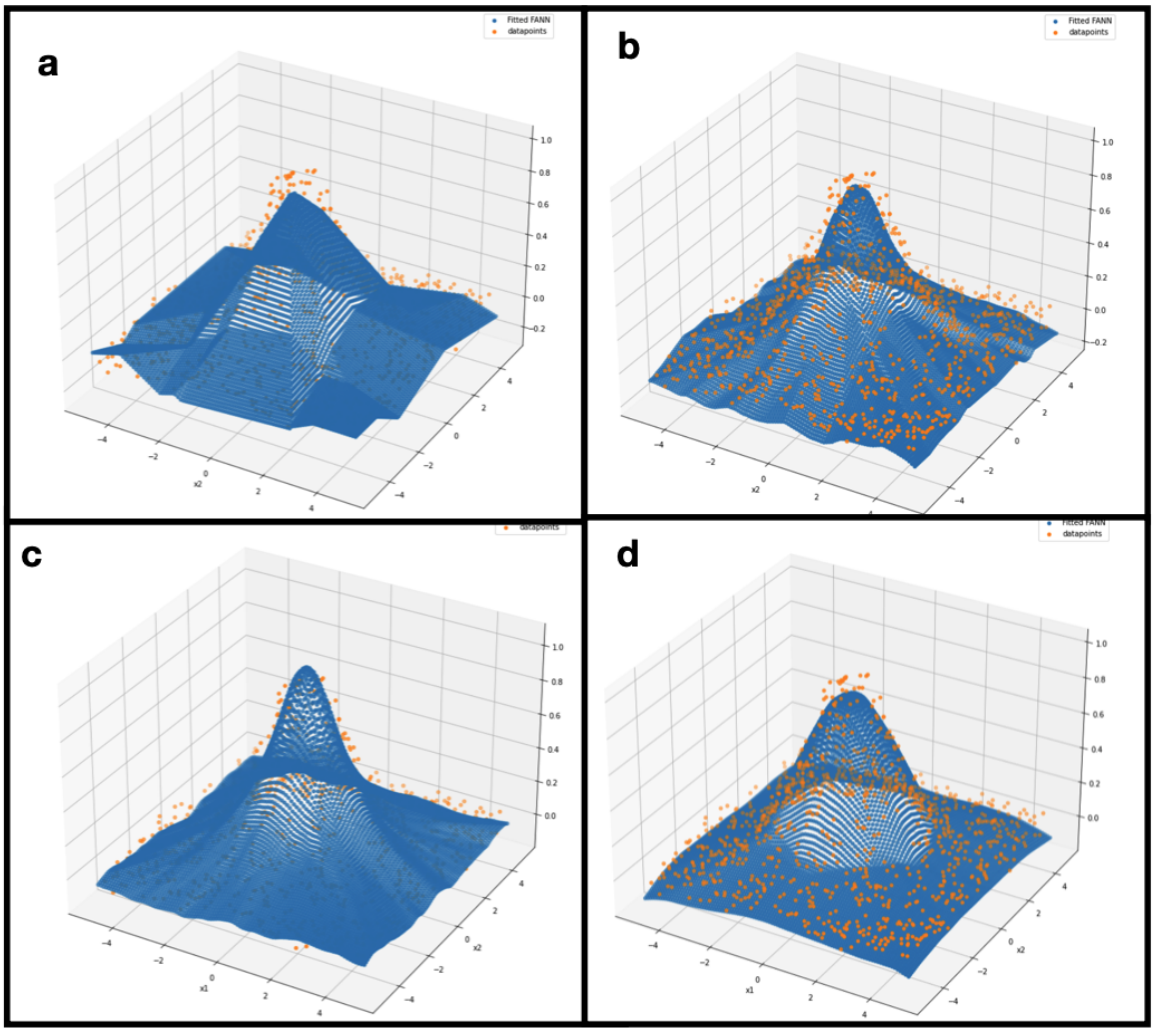


Figure 12: Some example results where different FANNs have been fitted to data generated using a bell-shaped function.

(ii) After you have trained both models to the best of your abilities to achieve the smallest possible cost you can, then compute the cost associated with each model on an independent generated test-set of size $M = 10000$ data points.
(iii) You should report, as usual, the training plots, hyper-parameter values and training plots. You should also visualize the surface of each one of these models together with the the data-points, like in Fig. 12.
(iv) Report also on each models cost on the test set.
(v) Which model would you choose knowing the true underlying function of the data?
(vii) Which model would you choose if you did not know the underlying function that generated the data? Motivate and discuss the problem.

6 Task C: Signal Classification with FANNs

Perhaps many of us think of 1D signals as time series data and similar, like the ones to be studied in this subtask. However, it is important to note that also many other types of measurement data are also 1D signals, without being a function of a time index. For example spectra of different types (IR, NMR, Mass) are also 1D but in these cases the axis corresponding to time reflects other quantities such as wavelength, chemical shift, and mass/charge ratio. Anyway, for most practical purposes we can think of any 1D signal simply as a long vector.

Classification of temporal signals is often a challenging task with many potential applications, for example in speech/word recognition, music/song recognition, monitoring the mechanical sounds from industrial machines, and clinical diagnosis of the functionality of a human hart. Compared to the previous exercises, here there are two big differences. One is the fact that we are now switching from regression to classification. The other is that we are also switching from low-dimensional inputs to significantly higher dimensions.

6.1 Description of the task

Overall description: In this task, you are given a dataset containing a fairly large amount ”signal-data”, each signal being labelled either as “normal” (numeric label 0) or “abnormal” (numeric label 1). You assignment is to construct a FANN that is able to achieve a high classification accuracy when trying to classify a signal as normal or abnormal. This mmakes this task a 2-class classification problem. In Fig. 13 you can see 3 pairs of examples of normal and abnormal data.

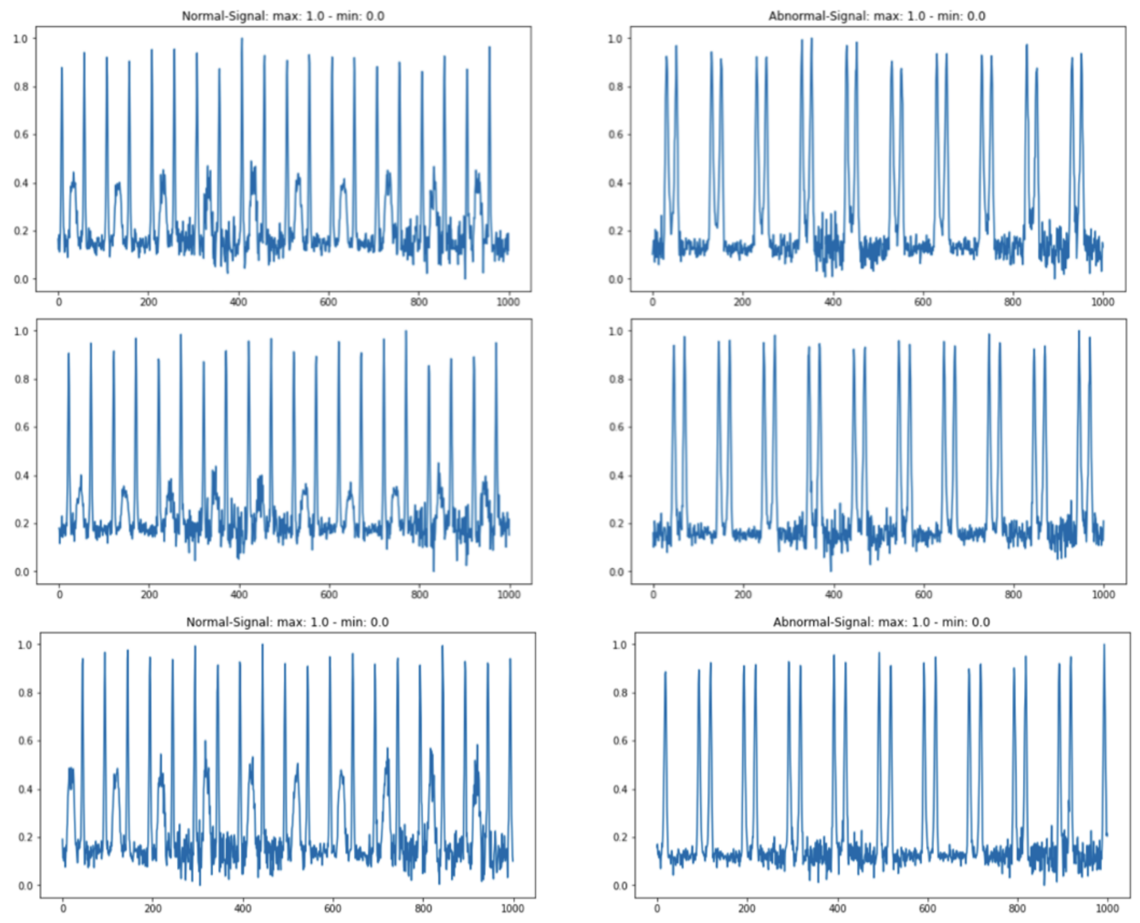


Figure 13: Left Column: ’Normal’ class signals. Right Column: ’Abnormal’ Signals

Accessing provided data and code: You are provided with a zip-file on the relevant assignment page in Studium. At the same page you downloaded this document as a pdf. There is also a python file python `assignment_nn.py` containing some code that you import into your notebook in order to get access to the signal dataset as a PyTorch `Dataset` object. Just make sure to have the folder and the python file in the same folder, next to each other. The following code shows how you can use `assignment_nn.py`.

```python
import assignment_ann as a4
og_data = a4.SignalDataset() # Provides back a PyTorch Dataset
size_of_og_data = len(og_data) # Print size of whole dataset
```

(a) How large is the original dataset? In other words, how many signals do you have?

(b) How long is each signal vector? In other words, what is the shape each tensor $\mathbf{x}_n$? To do this, find out how to access single data-points and there labels from your own internet search.

(c) Plot a couple of the signals with there labels (normal / abnormal) in the plot title. These should look like the ones in Fig. 13 with one difference. What is this difference?

(d) Split the original dataset into train-, validation- and test-sets. Choose some reasonable split values. You have to have enough data-points so that your network can learn as much and easily as possible while at the same time you also want enough data-points in your test set so that your estimate of the cost/accuracy on unseen data becomes as certain as possible. For example you may choose 70% for training, 20% for validation and 10% for test.

(e): Choose a first preliminary ANN architecture that seems suitable for this 2-class classification task. You will not need more than two hidden layers with with ReLU activation functions in all hidden nodes. Make sure that you have the correct number of input nodes (the argument `in_feature` in your first linear layer) and 2 final output nodes (the argument `out_feature` in your last linear layer).

IMPORTANT: Quite unexpectedly, you should not add any `nn.SoftMax()` layer as your last layer, because `nn.CrossEntropyLoss()` actually employs the desired softmax function itself to produce the network output for you during the loss/cost calculation! This means that the network outputs do not sum to one after training as when a softmax layer is used. However, if you want to predict the actual class labels for a new input image, still you simply find the largest network output for each row and the corresponding class index. In terms of Python/PyTorch, you may apply `torch.argmax()` to the network output across the columns to get the index of the largest class prediction score for each row (data-point). Below you find a code snippet that helps you with this.

```python
with torch.no_grad():
    raw_y_preds = model(X) # predict model output for X batch

y_class_preds = raw_y_preds.argmax(dim=1) #compute class labels for each x_n
```

If a tensor `X` with the shape `[batch_size, input_dim]` contains set of signals as rows, then in the code above, `raw_y_preds` will be a tensor of shape `[batch_size, nr_classes]` where `nr_classes` denotes the number of classes. In our case `X` is a `[batch_size,500]` tensor and `raw_y_preds` is a `[batch_size,2]` tensor. The last row takes the raw outputs, and makes them into a `[batch_size,1]` tensor, where each row now is the predicted class label for the corresponding signal. Notably, this is a general method for getting the predicted class labels from a classification ANN, and it works for arbitrary number of classes.

(e) Select and instantiate the loss function and the SGD optimizer with some suitable preliminary learning rate (the values 0.05 or 0.005 often work).

(f) Implement a very basic training loop consisting of only the mini-batch SGD inner-loop, meaning without the evaluation procedure. Moreover, use only the train-set to check if you can get your model to reduce the cost/loss. This type of "test runs" of the training loop is always recommended as a sanity-check, to ensure that everthing (model, batches and batch sizes) seem to work as expected. At this stage you just want to check if you can make the cost/loss on the mini-batches go down. This means you should monitor the mini-batch cost as a function of the number of mini-batches completed, aiming for a run that results in a graph similar to the one in Fig. 14.

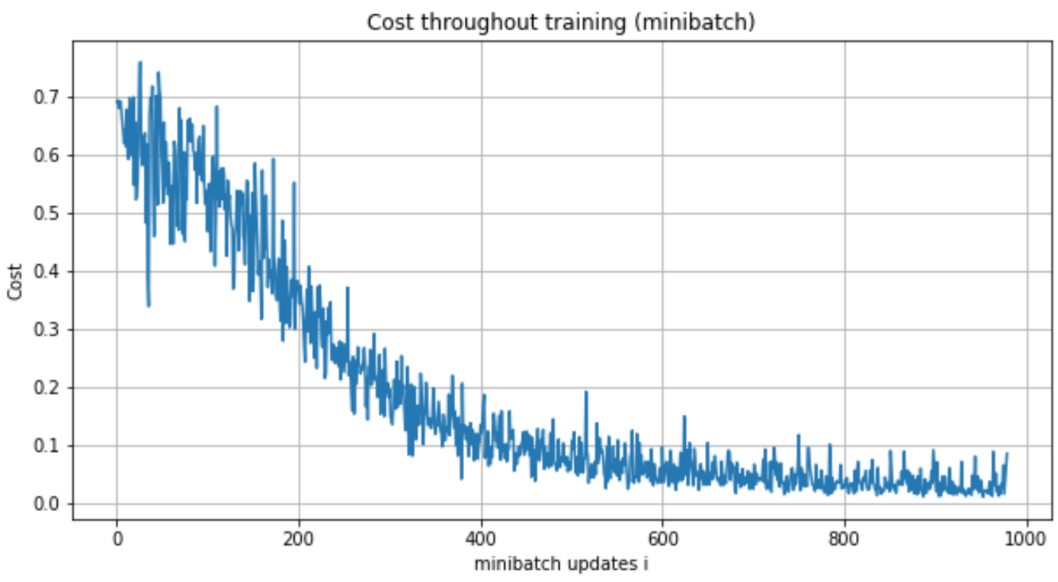


Figure 14: Example of how the cost is decreasing as a function of the number of mini-batches performed.

If you can do this, then you are in good progress. Then you can go on and try to implement a proper version of the complete training loop, including model performance evaluation also using the training and validation sets.

(g) Define the function for model performance evaluation based on the template in chapter 3. Remember, this is classification task, and as such we should be interested in the accuracy of our model.

(h) Integrate your model performance evaluation function into your training loop and perform model evaluation on both validation and train sets every epoch before the inner SGD training loop. Keep track and store these performance metrics during your whole training session, because you will need to plot and report these values for your fully trained network. Your training should result in plots looking like those in Fig. 15 and Fig. 16.

Your training runs should look something similar to Fig. 17 if you print the performance metrics accordingly.

Examination: To fully accomplish this task, you need to report all plots and results, including hyper-parameter choices.

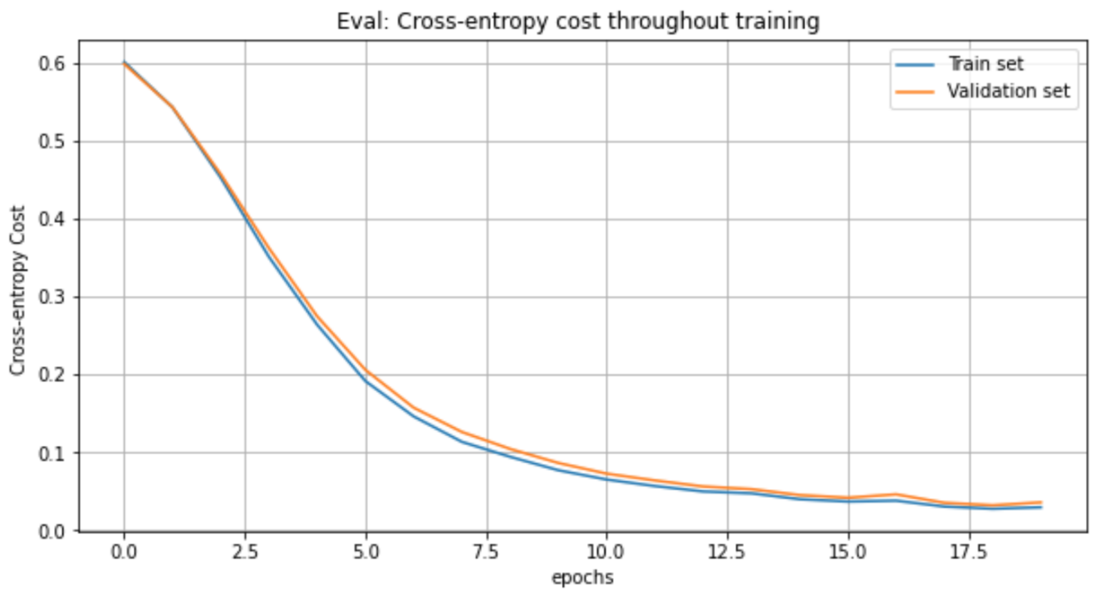


Figure 15: Example of how the average Cross Entropy is developing as a function of the number of mini-batches performed.

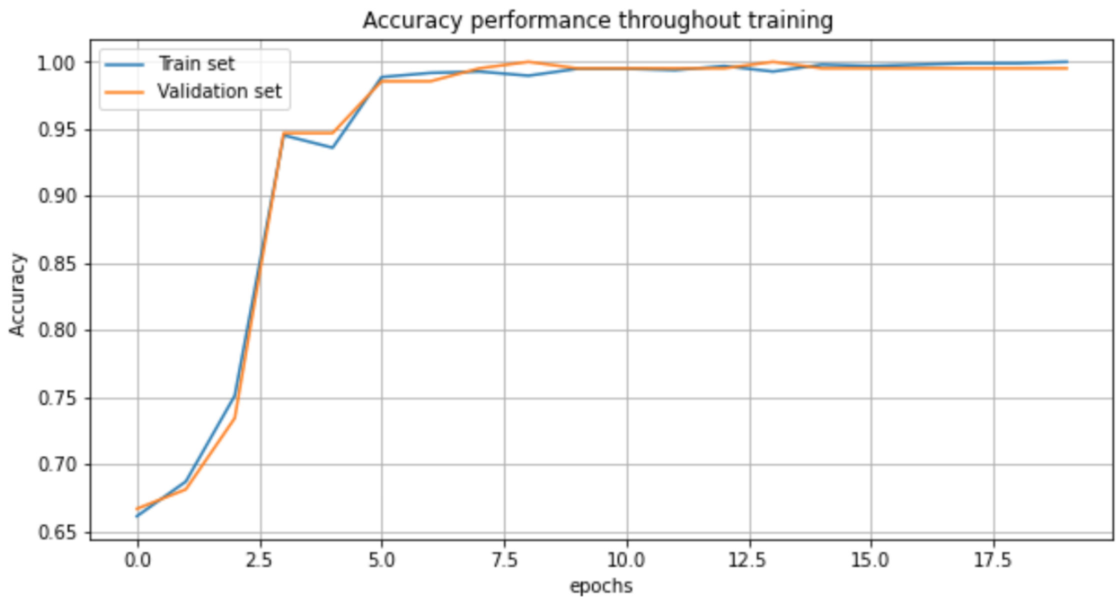
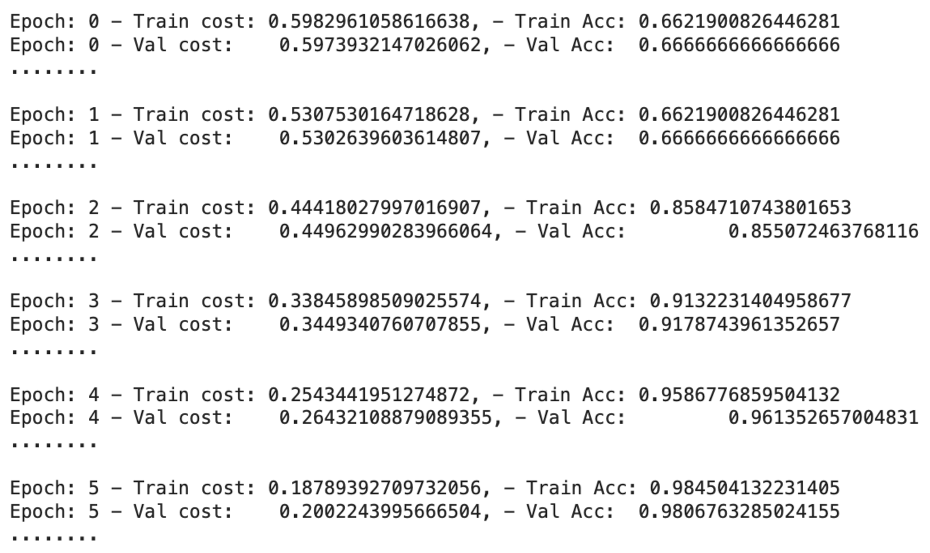


Figure 16: Example of how the accuracy is developing as a function of the number of epochs completed.



```
Epoch: 0 - Train cost: 0.5982961058616638, - Train Acc: 0.6621900826446281
Epoch: 0 - Val cost:    0.5973932147026062, - Val Acc:  0.6666666666666666
........

Epoch: 1 - Train cost: 0.5307530164718628, - Train Acc: 0.6621900826446281
Epoch: 1 - Val cost:    0.5302639603614807, - Val Acc:  0.6666666666666666
........

Epoch: 2 - Train cost: 0.44418027997016907, - Train Acc: 0.8584710743801653
Epoch: 2 - Val cost:    0.44962990283966064, - Val Acc:        0.855072463768116
........

Epoch: 3 - Train cost: 0.33845898509025574, - Train Acc: 0.9132231404958677
Epoch: 3 - Val cost:    0.3449340760707855, - Val Acc:  0.9178743961352657
........

Epoch: 4 - Train cost: 0.2543441951274872, - Train Acc: 0.9586776859504132
Epoch: 4 - Val cost:    0.26432108879089355, - Val Acc:        0.961352657004831
........

Epoch: 5 - Train cost: 0.18789392709732056, - Train Acc: 0.984504132231405
Epoch: 5 - Val cost:    0.2002243995666504, - Val Acc:  0.9806763285024155
........
```

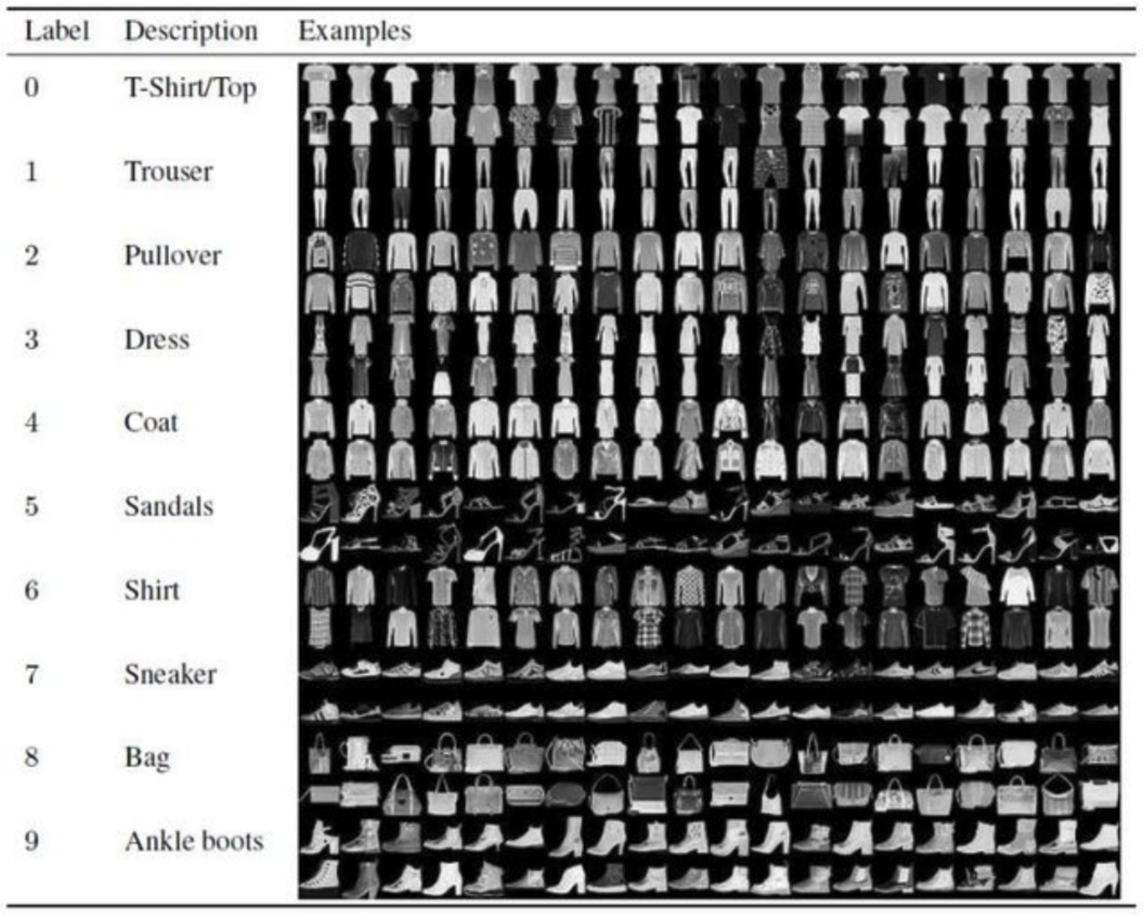
Figure 17: Example of performance metrics printouts during a training session.

7 ANN Task 4: Image Classification of FashionMNIST using FANNs

In this task we are switching from classification of 1D signals, as in the previous task, to classification of 2D images. Here you will train a FANN to classify images of clothing items, into 10 clothing categories. The steps involved here are more or less identical to the signal classification task previously. What differs is the dataset (2d images instead of 1d signals), the number of classes (10 instead of 2), and the FANN architecture we need to consider to accommodate these changes.

7.1 The dataset

The dataset: Fashion-MNIST[1] is a dataset of Zalando's article images. It consists of 70000 images of clothing items in gray-scale: a training set of 60000 examples and a test set of 10000 examples. Each example is a 28×28 grayscale image and associated with a label from one out of 10 classes.



Label	Description	Examples
0	T-Shirt/Top	
1	Trouser	
2	Pullover	
3	Dress	
4	Coat	
5	Sandals	
6	Shirt	
7	Sneaker	
8	Bag	
9	Ankle boots	

Figure 18: Illustration of the FasionMNIST dataset showing samples from each of the 10 clothing classes.

[1]Zalando intends Fashion-MNIST to serve as a direct replacement for the original MNIST dataset for benchmarking machine learning algorithms. It shares the same image size and structure of training and testing splits. The original MNIST dataset contains handwritten digits and is very popular within the AI/ML/Data Science community, where it is often used as a benchmark to validate and compare different algorithms.

Labels: Each training and test example is assigned to one of the following 10 labels:

0-T-shirt/top
1-Trouser
2-Pullover
3-Dress
4-Coat
5-Sandal
6-Shirt
7-Sneaker
8-Bag
9-Ankle boot

Remark.
For more details, see the following links.
`https://www.kaggle.com/zalando-research/fashionmnist`
`https://github.com/zalandoresearch/fashion-mnist`

As shown below, downloading the dataset is conveniently handled by PyTorch since this is an stablished standard dataset. As a matter of fact, we have already shown this before in chapter 3;

```python
## Code to access the FashionMNIST
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets

from torchvision.transforms import ToTensor, Lambda, Compose
import matplotlib.pyplot as plt

# Download training data from open datasets.
training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
)

# Download test data from open datasets.
test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor(),
)
```

Questions to consider and Deliverables & Results to produce

(a) Repeat the procedure you performed when designing and testing the signal classifiers in the previous task, this time using the FashionMNIST dataset. The main difference here is that you now have:

1. Images that need to be flattened into vectors. (see ”But what is a neural network? — Chapter 1, Deep learning” from 3Blue1Brown from the Extra materials.). How we deal with this is by placing a `nn.Flatten()` component at the beginning of our architecture specification in `nn.Sequential(...)`.
2. A 10-class problem instead of a 2-class problem.

(b) Based on your own interests and perhaps some internet search: Propose at least two concrete potential areas of industrial or research applications where you personally think image classification of this kind can be (or already is) of practical and commercial importance.

8 ANN Task 5*: CNN classification of FashionMNIST

Note: This task is for Grades 4/5 only.

Description of the task

In this subtask you are going to perform the same kind of classification of FashionMNIST images as before but this time using a Convolutional neural network (CNN). A good starting point is therefore to use the already available ANN called "LeNet", and described in Wikipedia: `https://en.wikipedia.org/wiki/LeNet`. Using this network as a starting guess for the architecture you will need will save both time and computational resources.

Thus your task is to build a LeNet5-style network, train it on FashionMNIST dataset until you reach a very good classification performance on your validation set. We advise you to use the 'other' more flexible way of building neural networks in PyTorch, where you efine your neural network by subclassing the so called `nn.Module`: Read about them here: `https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html` In this task you will not be guided as much as in previous tasks. This means you need to find out more details yourself regarding how to design the network, get the training to work etc.

Questions to consider and results to produce
(a) What is the size of the images in FashinMNIST dataset? What does this mean in terms of reasonable sizes of the kernel to be used in the first layer of the CNN? Hint: It is not reasonable to have a kernel size that is equal to or larger than the image itself. The kernel size should cover only a relatively small fraction of the image.
(b) What is a LeNet5? How do you alter this architectures to fit out FashionMNIST case?
(c) Repeat the task to classify the FashionMNIST dataset as in Task 4.
(d) Compare your results in terms of accuracy on the external test set with the results obtained in Task 4.

Examination: Your performance on this task will be evaluated based on your ability to convincingly perform proper training and performance evaluation of a Convolutional Neural Network on the specified dataset according to the steps provided in the code template in chapter 3. Your report slides should contain final model performance values on all datasets, training plots, hyper-parameter value choices, and CNN architecture design choices.

Tip: If you want to know more about the convolution operator. have a look at the 20 minutes video provided via this link `https://youtu.be/B-M5q51U8SM`. This is to understand more about how convolution works in 2D in terms of a reversed (rotated) kernel, and in the context of ANN.

9 Extra Study Material, Tips, Etc

9.1 PyTorch Help, Troubleshooting and Guidance

1. Probably the most gentle introduction to training ANNs In PyTorch: Training a classification model on MNIST with PyTorch `https://www.youtube.com/watch?v=OMDn66kM9Qc`

2. Certain bugs and oddities are more common than others. Here is a link to a 20 minutes long video that gives you 10 great practical tips when you are about to start training your PyTorch models `https://www.youtube.com/watch?v=O2wJ3tkc-TU`

9.2 Neural Networks, what they actually do and how they do it

1. "But what is a neural network? — Chapter 1, Deep learning" from 3Blue1Brown - `https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=1`

2. Chapter 6 "Neural networks and deep learning" in the book *MACHINE LEARNING: A First Course for Engineers and Scientists* by Andreas Lindholm, Niklas Wahlström, Fredrik Lindsten Thomas B. Schön. Availible for free at `http://smlbook.org/book/sml-book-draft-latest.pdf`

9.3 Gradient Decent and Stochastic Gradient Decent

1. "Gradient descent, how neural networks learn — Chapter 2, Deep learning" from 3Blue1Brown - `https://www.youtube.com/watch?v=IHZwWFHWa-w&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=2`

2. "Stochastic Gradient Descent, Clearly Explained!!!" by StatQuest - `https://www.youtube.com/watch?v=vMh0zPT0tLI`

3. "25. Stochastic Gradient Descent" Suvrit Srafrom from MIT 18.065 Matrix Methods in Data Analysis, Signal Processing, and Machine Learning, Spring 2018 - `https://www.youtube.com/watch?v=k3AiUhwHQ28`

4. "Lecture 17.2 — Large Scale Machine Learning — Stochastic Gradient Descent" by Andrew Ng - `https://www.youtube.com/watch?v=W9iWNJNFzQI`

5. "The Stochastic Gradient Descent Algorithm" by Nathan Kutz `https://www.youtube.com/watch?v=_adhFSH66jc`

9.4 Backpropagation to achieve auto-differentiation

1. "What is backpropagation really doing? — Chapter 3, Deep learning" from 3Blue1Brown - `https://www.youtube.com/watch?v=Ilg3gGewQ5U&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=3`

2. "Backpropagation calculus — Chapter 4, Deep learning" from 3Blue1Brown - `https://www.youtube.com/watch?v=tIeHLnjs5U8&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=4`

9.5 Convolutional Neural Networks

1. ”Lecture 7: Convolutional Neural Networks” by Andrej Karpathy from Stanford’s CS231n Winter 2016 [de facto canonical lecture on CNNs] - `https://www.youtube.com/watch?v=LxfUGhug-iQ`
2. ”A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way” by Sumit Saha - `https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53`

Assignment Soundtrack

- Bob Dylan - The Times They Are A-Changin'
- Jon Hopkins - Everything Connected
- Frank Ocean feat André 3000 - Pink Matter
- Bon Iver - Re: Stacks
- SOHN - Lessons
- The Jackson 5 - ABC
- Predicting Song - (To the tune of Can't Stop the Feeling)
- The Beatles - Help!
- Hans Zimmer - Detach
- Moderat - A New Error
- Drake & Yebba - Yebba's Heartbreak
- Donna Summer - No More Tears (Enough is Enough)
- Wu-Tang Clan ft. Redman, Inspectah Deck - Lesson Learn'd
- Lenny Williams - Problem Solver
- Kanye West - Selah
- The Flaming Lips "We Can't Predict The Future"
- Martin Garrix feat. Tove Lo - Pressure
- Selena Gomez The Scene - Naturally
- WhoMadeWho - Sooner
- SUNMI - Heroine
- Marin Marais - Le Badinage, IV.87 (Marais) (performed by Jordi Savall)
- J.S Bach - Bwv No. 142 "Uns Ist Ein Kind Geboren" recomposed by Hanan Townshend
- Lisa Gerrard - Now We Are Free