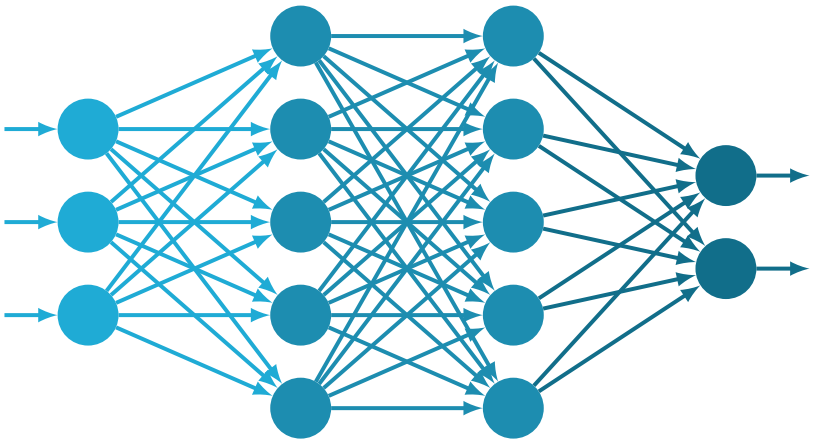
KU LEUVEN

DATA MINING AND NEURAL NETWORKS

Assignments

Prof. dr. ir. Johan A. K. Suykens[1]

Assistants
Henri De Plaen[2]
Arun Pandey[3]
Yingyi Chen[4]
Francesco Tonin[5]



October 2020

[1] johan.suykens@esat.kuleuven.be
[2] henri.deplaen@esat.kuleuven.be
[3] arun.pandey@esat.kuleuven.be
[4] yingyi.chen@esat.kuleuven.be
[5] francesco.tonin@esat.kuleuven.be

General guidelines

This course is evaluated by means of a set of assignments for which the student is required to make an individual report. In order to help you getting through these assignments, a number of guided sessions are organized.

Due to this year's exceptional context due to the COVID-19 pandemic, these will happen online through Blackbox Collaborate, a tool directly available in Toledo: after a general introduction, you will be split into small groups where you can discuss your answers and difficulties. Assistants will hop from group to group and provide help and explanations. Therefore, we ask you to prepare your exercises before the corresponding session.

When answering the questions, try to be complete, but also try to avoid redundancy. For every exercise, a maximum page number (figures included) is indicated: this is the number of pages we estimate necessary and sufficient for this exercise. Answers should be precise, concise and complete. Pay also attention to the language, structure, style, typesetting, as well as the readability and illustrative power of the plots and tables.

No specific language is required for the execution of the assignments. When we provide base code, it will however be in MATLAB for the three first assignments and Python for the fourth one. Still, you are free to translate it in the language you want or use an equivalent code, as long as it performs the same task. In case of any question, feel free to send us an email.

Good luck and we hope you will take pleasure exploring the wonderful world of neural networks and data mining!

ASSIGNMENT 1

Training Algorithms and Generalization

This assignment will cover the use of neural networks as general models for function estimation and classification. From its simplest form, the perceptron, to more complex structures. The goal is to gain a sufficient understanding of the mathematics of the model, its power and its limitations, its estimation and its training as well as the precise influence of the model parameters. For all exercises and even if not asked explicitly, you should always pay attention to speed, generalization and overfitting; this will help you to understand the motivations, consequences and limitations of each model or learning scheme. As a general tip: always try to get an insight of the bigger picture.

MATLAB provides a vast set of tools for treating with neural networks. These are regrouped in the *Deep Learning Toolbox*[1], which we will be using throughout this assignment and the following ones.

1.1 THE PERCEPTRON AND BEYOND

The perceptron is the simplest one-layer network. It consists of R inputs connected to 1 neuron arranged in a single layer via interconnection weights. These neurons have the hardlim transfer function, thus the output values are only 0 and 1, where the inputs can take on any value. The perceptron is used as a simple classification tool.

Introduction

In order to create such a network, we can use the command:

```
net = newp(P,T,TF,LF);
```

where `P` and `T` are input and target vectors e.g. `P=[2 1 -2 -1; 2 -2 2 1], T=[0 1 0 1]`. `TF` is the transfer function (typically `'hardlim'`), `LF` represents the perceptron learning rule (for instance `'learnp'`). In this case the number of neurons is set automatically.

The weights of the connections and the bias of the neurons are initially set to zero, which can be checked and changed with the following commands:

`net.IW{1,1}`	Returns the weights of the neuron(s) in the first layer.
`net.b{1,1}`	Returns the bias of the neuron(s) in the first layer.
`net.IW{1,1} = rand(1,2);`	Assigns random weights in $[0, 1]$.
`net.b{1,1} = rands(1);`	Assigns random bias in $[-1, 1]$.

[1] In Python, neural networks can be easily used with, e.g. *PyTorch*.

One can initialize the network with a single command by entering:

```
net = init(net);
```

all weights and biases of the perceptron will be initialized to zero.

We can teach a perceptron to perform certain tasks which are defined by pairs of inputs and outputs. A single perceptron can learn only linearly separable tasks: inputs belonging to different classes are separated by a hyperplane in the input space (decision boundary). In two dimensions the decision boundary is a line. The default learning function is the perceptron learning rule `'learnp'`.

We can use the function train to let the perceptron perform the classification task. In this case the learning occurs in batch mode:

```
[net,tr_descr] = train(net,P,T);
```

here the second argument is a description of the learning process. To set the number of iterations or epochs, we can use the command `net.trainParam.epochs = 20;`. After training we can simulate the network on new data with the function `sim`:

```
sim(net,Pnew)
```

with `Pnew` an input vector. For our example Pnew has to be a (column) vector of length 2, with elements between -2 and 2, e.g. `Pnew = [1;-0.3]`. Multiple input vectors can be fed at once to the network by putting them together in an array.

Demos

The following demos can be run from the MATLAB prompt:

`nnd4db`	decision boundary (2d input);
`nnd4pr`	perceptron learning rule (2d input);
`demop1`	classification with 2d input perceptron;
`demop4`	classification with outlier (2d input);
`demop5`	classification with outlier using normalized perceptron learning rule (2d input);
`demop6`	linearly non-separable input vectors (2d input).

Exercises

1. In the case of a linear regression, the relation between an d-dimensional input and a univariate output datapoint is modelled by a linear relation of the form $y_i = w_1 x_{1,i} + w_2 x_{2,i} + \ldots + w_d x_{d,i} + \beta$. The parameters $\{w_j\}_{j=1}^{d}$ and β are found by minimizing the sum of squared errors $\sum_{i=1}^{N} \left(y_i - (w_1 x_{1,i} + w_2 x_{2,i} + \ldots + w_d x_{d,i} + \beta)\right)^2$. How can you perform a linear regression with a perceptron? Describe the link between those two models.
2. Consider a dataset $\{(x_i, y_i)\}_{i=1}^{N}$ where $\{x_i\}_{i=1}^{N}$ and $\{y_i\}_{i=1}^{N}$ are respectively the input and output set. Generate one corresponding to $y_i = -\sin(0.8\pi x_i)$:

```
x = linspace(0,1,21); % creates 21 datapoints ...
   uniformily distributed in the interval [0,1].
y = -sin(.8*pi*x);     % computes the image of these 21 ...
   datapoints in a new vector y.
```

   Plot the function. Will a linear model adequately capture the relationship between the input and output data? Can you relate this to over- or underfitting?

3. Now train a neural network with one hidden layer containing two neurons on the data-set of the previous question. You can use the following code to train your neural network:

```
net = fitnet(2);
net = configure(net,x,y);
net.inputs{1}.processFcns = {};
net.outputs{2}.processFcns = {};
[net, tr] = train(net,x,y);
```

in which `x` and `y` contain the input and output points respectively. Lines two, three and four prevent the network from rescaling the inputs and outputs.

1.2 BACKPROPAGATION IN FEEDFORWARD MULTI-LAYER NETWORKS

A general feedforward network consists of at least one layer, and it can also contain an arbitrary number of hidden layers. Neurons in a given layer can be defined by any transfer function. In the hidden layers usually nonlinear functions are used, e.g. `tansig` or `logsig`, and in the output layer `purelin`. The only important condition is that there is no feedback in the network, neither delay.

In MATLAB one can create such a network object by e.g. the following command:

```
net = feedforwardnet(numN,trainAlg);
```

This will create a network of one hidden layer with corresponding `numN` neurons, which will use the trainAlg algorithm for training (e.g. `traingd`). This network can be trained using the `train` function:

```
net = train(net,P,T);
```

Finally the network can be simulated in two ways:

```
sim(net,P);
```

or,

```
Y = net(P);
```

Some available training algorithms are:

`traingd`	gradient descent;
`traingda`	gradient descent with adaptive learning rate;
`traincgf`	Fletcher-Reeves conjugate gradient algorithm;
`traincgp`	Polak-Ribiere conjugate gradient algorithm;
`trainbfg`	BFGS quasi Newton algorithm (quasi Newton);
`trainlm`	Levenberg-Marquardt algorithm (adaptive mixture of Newton and steepest descent algorithms).

To analyze the efficiency of training one can use the function `postreg` which calculates and visualizes regression between targets and outputs. For a network `net` trained with a sequence of examples `P` and targets `T` we have:

```
a=sim(net,P);
[m,b,r]=postreg(a,T);
```

where `m` and `b` are the slope and the y-intercept of the best linear regression respectively. `r` is a correlation between targets `T` and outputs `a`.

Demos

`nnd11nf`	network function;
`nnd11bc`	backpropagation calculation;
`nnd11fa`	function approximation;
`nnd12sd1`	steepest descent backpropagation;
`nnd12sd2`	steepest descent backpropagation with various learning rates;
`nnd12mo`	steepest descent with momentum;
`nnd12vl`	steepest descent with variable learning rate;
`nnd12cg`	conjugate gradient backpropagation;
`nnd9mc`	comparison between steepest descent and conjugate gradient.

Exercises

1. **Backpropagation.** Consider the neural network at Figure 1.1. Each hidden hidden neuron uses a sigmoid activation function $\sigma(t) = \frac{1}{1+\exp(-t)}$. The loss is the mean square error and the datapoints are $\{(x_1, x_2, y)_i\}_{i=1,2,3} = \{(1,1,2),(2,-1,4),(3,0,3)\}$. Input units are green, biases are yellow, hidden units blue and output units red. The weights can be found on the corresponding arrows. Compute the gradient through backpropagation analytically and don't forget to give the corresponding development. After one step of (simple) gradient descent, how does the updated network look like?
   Hint: $\frac{\mathrm{d}(\sigma(t))}{\mathrm{d}t} = \sigma(t)(1-\sigma(t))$. No need of a computer here.

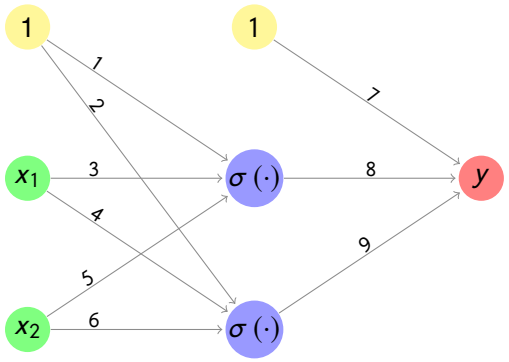


FIGURE 1.1: Neural network to perform backpropagation on.

2. **Function approximation: comparison of various algorithms.** Download the `algorithms.mlx` live script on Toledo and try out different training algorithms. How does gradient descent perform compared to other training algorithms? What are the advantages or disadvantages of the algorithms yo compare? What is the influence of noise of the dataset size? Discuss the difference between epochs and timing to assess the speed of the algorithms.
   Hint: by playing with the script, yo will generate a lot of plot, please try to figure out what is relevant to be reported.

1.3 PERSONAL REGRESSION EXAMPLE

In this problem, the objective is to approximate a nonlinear function using a feedforward artificial neural network. The nonlinear function is unknown, but you are given a set of 13 600 datapoints uniformly sampled from it. Everybody has to use a different dataset based on his or her student number. It will have to be built by yourself from 5 existing nonlinear functions $f_1, f_2, \ldots, f_5$. To do this, you will have to find variables `X1`, `X2`, `T1`, `T2`, `T3`, `T4` and `T5` in the given MATLAB datafile. The two vectors `X1` and `X2` contain the input variables (in the domain $[0,1] \times [0,1]$). The five vectors `T1` to `T5` are the 5 independent nonlinear functions evaluated at the corresponding points from (`X1`, `X2`). In other words, f_1(`X1(i)`, `X2(i)`) = `T1(i)`, f_2(`X1(i)`, `X2(i)`) = `T2(i)`, ..., f_5(`X1(i)`, `X2(i)`) = `T5(i)`. The datapoints are noise free (the evaluation is exact). To construct your personal and unique data set, you have to build a new target `Tnew`, which represents an individual nonlinear function to be approximated by your neural network. For this, consider the largest 5 digits of your student number in descending order, represented by d_1, d_2, d_3, d_4, d_5 (with d_1 the largest digit). `Tnew` is build as follows:

$$\texttt{Tnew} = \frac{d_1\texttt{T1} + d_2\texttt{T2} + d_3\texttt{T3} + d_4\texttt{T4} + d_5\texttt{T5}}{d_1 + d_2 + d_3 + d_4 + d_5}$$

For example, if your student number is m0224908, then your list of largest digits in descending order is 9, 8, 4, 2, 2 and therefore your target is `Tnew=(9*T1+8*T2+4*T3+2*T4+2*T5)/(9+8+4+2+2)`.

1. **Define your datasets.** Your dataset consists now of `X1`, `X2` and `Tnew`: draw 3 (independent) samples of 1000 points each. Use them as the training set, validation set, and test set, respectively. Motivate the choice of the datasets. Plot the surface of your training set.
2. **Build and train your feedforward neural network.** Use the training and validation sets. Build the neural network with 2 inputs and 1 output. Select a suitable model for the problem (number of hidden layers, number of neurons on each hidden layer). Select the learning algorithm and the transfer function that may work best for this problem. Motivate your decisions. How do you validate your model?
3. **Performance assessment.** Evaluate the performance of your selected network on the test set. Plot the surface of the test set and the approximation given by the network. Show and comment the training, validation and test loss curves. Be sure that you cannot train further. Give the final RMSE on the test set, what else could you do to improve the performance of your network?

1.4 BAYESIAN INFERENCE

The case of network weights

For simplicity of exposition, we begin by considering the training of a network for which the architecture is fixed in advance. More precisely we focus on the case of a one-neuron network. In the absence of any data, the distribution over weight values is described by a prior distribution denoted $p(\mathbf{w})$, where $\mathbf{w}$ is the vector of adaptive weights (normally also biases, but in our example we will consider the case without bias). We also denote by $\mathcal{D}$ the available dataset. Once we observe the data, we can write the expression for the posterior probability distribution of the weights using Bayes' theorem:

$$p(\mathbf{w}|\mathcal{D}) = \frac{p(\mathcal{D}|\mathbf{w})p(\mathbf{w})}{p(\mathcal{D})},$$

where $p(\mathcal{D})$ is a normalization factor ensuring that $p(\mathbf{w}|\mathcal{D})$ gives unity when integrated over the whole weight space. $p(\mathcal{D}|\mathbf{w})$ corresponds to the likelihood function used in maximum likelihood techniques. Since in the beginning we do not know anything about the data, the prior distribution of weights is set to (for example) a Gaussian distribution. The expression of the prior is then:

$$p(\mathbf{w}) = \left(\frac{\alpha}{2\pi}\right)^{\frac{W}{2}} \exp\left(-\frac{\alpha}{2}||\mathbf{w}||_2^2\right),$$

with W the number of weights, and α is the inverse of the variance. For simplicity we will choose $\alpha = 1$. This choice of the prior distribution encourages weights to be small rather than large, which is a requirement for achieving smooth network mappings. So when $||\mathbf{w}||_2$ is large, the parameter of the exponential is large, and thus $p(\mathbf{w})$ is small (small probability that this is the correct choice of weights. Things are reversed for small $||\mathbf{w}||_2$).

A concrete example: binary classification

We consider the case where input vectors are 2-dimensional $\mathbf{x} = (x_1, x_2)$, and we have four data points in our dataset as in Figure 1.2. We take a network of a single neuron (compare to the perceptron of the first exercise session), so there is only a single layer of weights, and choose the logistic function as transfer function:

$$y(\mathbf{x}; \mathbf{w}) = \frac{1}{1 + \exp(-\mathbf{w}^T\mathbf{x})}. \tag{1.1}$$

The weight vector $\mathbf{w} = (w_1, w_2)$ is two-dimensional and there is no bias parameter. We choose a Gaussian prior distribution for the weights (with $\alpha = 1$). This prior distribution is plotted in Figure 1.3.
The data points can belong to one of the two classes (cross or circle), the output y giving the membership to one of these classes. The likelihood function $p(\mathcal{D}|\mathbf{w})$ in Bayes' theorem will be given by a product of factors, one for each data point, where each factor is either y or $(1 - y)$ according to whether the data point belongs to the first or the second class.

First we consider just the points labeled (1) and (2). Then we consider all four points and recompute the posterior distribution of the weights. Note: For an example of this case, run demo `bayes.mlx`. After training with only the first two data points, we see that the network function is a sigmoidal ridge (w_1 and w_2 control the orientation and the slope of this sigmoid). Weight vectors from approximately half of the weight space will have probabilities close to zero, as they represent decision boundaries with wrong orientation. When using all 4 data points, there is no boundary to classify all 4 points correctly. The most probable solution is the one corresponding to the peak point of the sigmoid, the others having quite low probabilities (so the posterior distribution of the weights is relatively narrow).

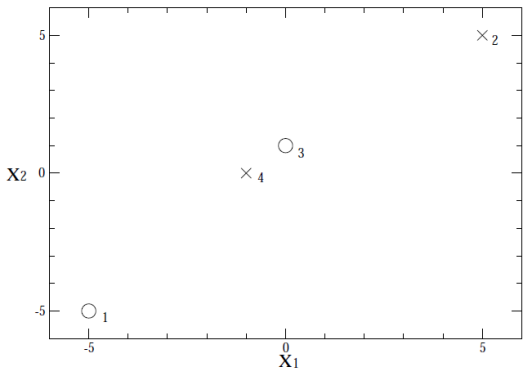


FIGURE 1.2: The four numbered data points in the dataset.

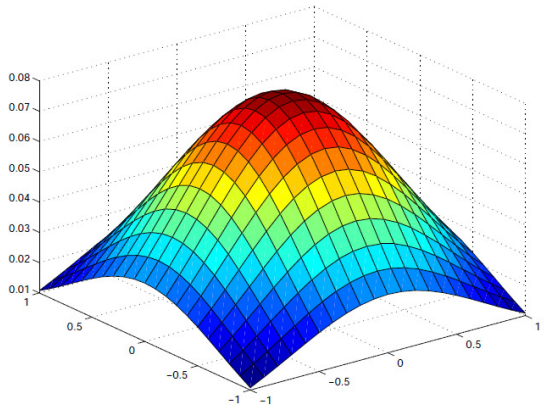


FIGURE 1.3: The prior distribution: a Gaussian.

Hyperparameters

From an optimization point of view the training is performed by iteratively adjusting $\mathbf{w}$ so as to minimize an objective function $M(\mathbf{w})$. In a maximum likelihood framework, M is taken to be the error function $E_D(\mathbf{w})$ that depends on the dataset $\mathcal{D}$. A common prescription to avoid overfitting is to include in M a regularization term $E_W(\mathbf{w})$ (*a.k.a.* weight decay) to favor small values of the parameter vectors, as discussed above. In particular, if we let

$$M(\mathbf{w}) = \beta E_D(\mathbf{w}) + \alpha E_W(\mathbf{w}), \tag{1.2}$$

we can immediately give to this a Bayesian interpretation. In fact it is not difficult to show that βE_D can be understood as minus the log likelihood for a noise model whereas αE_W can be understood as minus the log prior on the parameter vector. Correspondingly, the process of finding the optimal weight vector minimizing M can be interpreted as a *maximum a posteriori* (MAP) estimation. Notice that we have implicitly considered α and β fixed here.

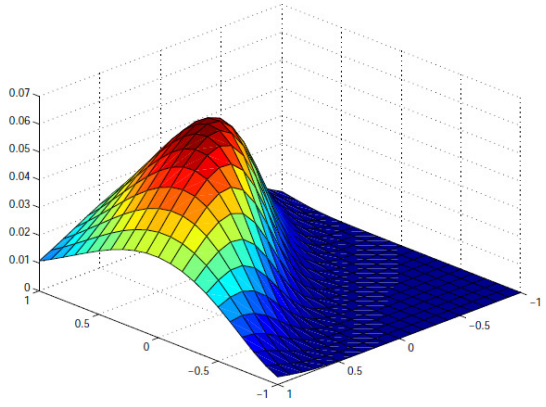


FIGURE 1.4: The distribution after presenting all four data points. Half of the weight space has become very improbable.

However these are hyperparameters that control the complexity of the model. Once more within a Bayesian framework, one can apply the rules of probability and find these parameters accordingly. In MATLAB and for a general feedforward neural network, this can be accomplished by means of `trainbr`. This training function updates the weight according to Levenberg-Marquardt optimization. For equation 1.2, the minimization is done with respect to **w** and determines at the same time the correct combination of the two terms that produce a network generalizing well.

Exercises

1. Similarly as you did in part 1.2, use `trainbr` and compare it with other relevant training functions seen. Try with and without noise. Consider overparametrized networks (many neurons): do you see any improvement with `trainbr`? Discuss.
2. How does this differ from regularization before? How did you achieve it previously?

1.5 REPORT

For this assignment, you should write a report for each of the four sections. Answer only the questions mentioned in each Exercise subsection; explanations outside should not be explicitly commented on. Similarly, the MATLAB demos are only meant to help you getting used to necessary tools. Running the demos is thus optional and should certainly not be part of the report. The maximum page division is as follows:

- **1.1 The Perceptron and Beyond** maximum 1 pages;
- **1.2 Backpropagation in Feedforward Multi-layer Networks** maximum 2 pages;
- **1.3 Personal Regression Example** maximum 2 pages;
- **1.4 Bayesian Inference** maximum 1 page;

ASSIGNMENT 2

Applications: Time-series Prediction and Classification

This assignment focuses on applying the feedforward neural networks to typical machine learning tasks, such as time-series prediction and classification. For time-series prediction, we first investigate one benchmark data set, namely, the Santa Fe dataset. Then, we analyze a global temperature data set. For binary classification, we turn to a benchmark medical data set, that is, the Breast Cancer Wisconsin (Diagnostic) Data Set from the UCI machine learning repository. Last but not least, we consider applying automatic relevance determination on the breast cancer data set so as to find the most relevant inputs.

2.1 TIME-SERIES PREDICTION

In this section, we will investigate two time-series data sets, which are the Santa Fe data and another data set on global temperature prediction.

Introduction: Time-series Prediction

A time series is a sequence of observations, ordered in time. Forecasting involves training a model on historical data and using them to predict future observations. A simple example is a linear auto-regressive model. The linear auto-regressive (AR) model of a time-series Z_t with $t = 1, 2, \ldots, \infty$ is given by

$$\hat{z}_t = a_1 z_{t-1} + a_2 z_{t-2} + \cdots + a_p z_{t-p}$$

with $a_i \in \mathbb{R}$ for $t = 1, \ldots, p$ and p being the model lag. The prediction for a certain time t is equal to a weighted sum of the previous values up to a certain lag p. At the same time, the nonlinear variant (NAR) is described as

$$\hat{z}_t = f(z_{t-1}, z_{t-2}, \ldots, z_{t-p}).$$

A depiction of these processes can be found in Figure 2.1. Remark that in this way, the time-series identification can be written as a classical black-box regression modeling problem

$$\hat{y}_t = f(\mathbf{x}_t)$$

where $y_t = z_t$ and $\mathbf{x}_t = [z_{t-1}, z_{t-2}, \ldots, z_{t-p}]$.

Santa Fe Dataset

The Santa Fe dataset is obtained from a chaotic laser which can be described as a nonlinear dynamical system. Given are 1000 training data points. The aim is to predict the next 100 points (it is forbidden to include these points in the training set!). The training data stored in `lasertrain.dat` and the test data contained in `laserpred.dat` are shown in Figure 2.2.

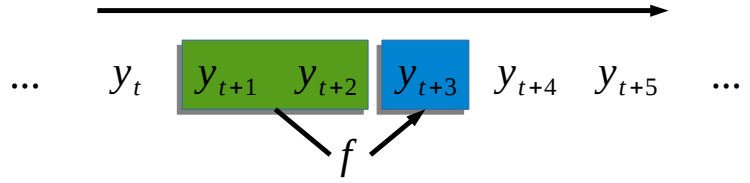


FIGURE 2.1: Schematic representation of the nonlinear auto-regressive model.

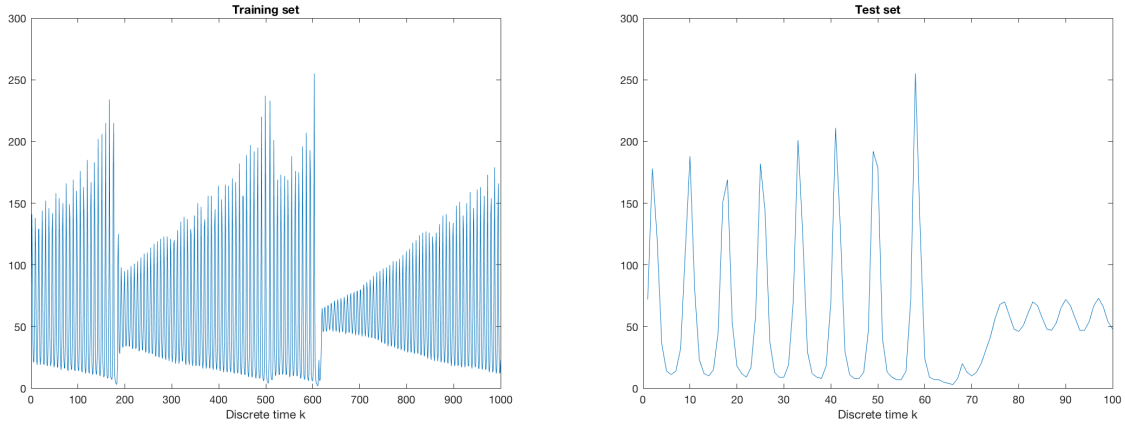


FIGURE 2.2: Visualization of the training and test sets of the Santa Fe data set.

Exercises

Train an MLP with one hidden layer after pre-processing the data set. The training is done in feedforward mode:

$$\hat{y}_{k+1} = \mathbf{w}^T \tanh(\mathbf{V}[y_k; y_{k-1}; \ldots, y_{k-p}] + \beta).$$

In order to make predictions, the trained network is used in an iterative way as a recurrent network:

$$\hat{y}_{k+1} = \mathbf{w}^T \tanh(\mathbf{V}[\hat{y}_k; \hat{y}_{k-1}; \ldots, \hat{y}_{k-p}] + \beta).$$

To format the data you can use the provided function `getTimeSeriesTrainData.m`. Make sure you understand what the function does by trying it out on a small self-made toy example.

1. Investigate the model performance with different lags and number of neurons so as to select the most appropriate hyperparameters. Note that the performances can be assessed by the mean squared error (MSE) or root-mean-square error (RMSE).

2. Would it be sensible to use the performance of this recurrent prediction on the validation set to optimize hyperparameters? If it is, please conduct the hyperparameters selection on the validation set. How is the performance of your feedforward neural network with the chosen hyperparameters on test set? (Hint: You can find how to do validation on time-series data set in the codes of next exercise, i.e. `global_temperature_train.m` and `global_temperature_test.m`)

Climate Change

Climate change is one of the biggest threats of our age. In this section, we work on a global climate change prediction data set using time-series methods. Specifically, we will use the non-linear auto-regression (NAR) for this case. The data can be downloaded from [1]. Note that this dataset is repackaged from a newer compilation put together by the Berkeley Earth, which is affiliated with Lawrence Berkeley National Laboratory. The Berkeley Earth Surface Temperature Study combines 1.6 billion temperature reports from 16 pre-existing archives. In this dataset, we only use the global land temperature in `GlobalLandTemperaturesByCity.csv`, which includes:

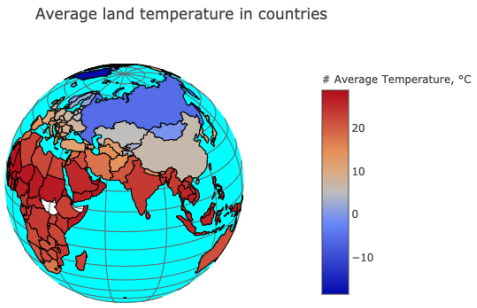


FIGURE 2.3: Visualization of the global land temperature in countries.

- `Date`: starts in 1750 for average land temperature and 1850 for max and min land temperatures;
- `LandAverageTemperature`: global average land temperature in celsius;

Note that in the following exercises, we will only use `LandAverageTemperature`.

Exercises

In this exercise, we use `GlobalLandTemperaturesByCity.csv`. Please select at least two cities (either from the same or different countries), and conduct the following analyses on each of the city temperature data sets:

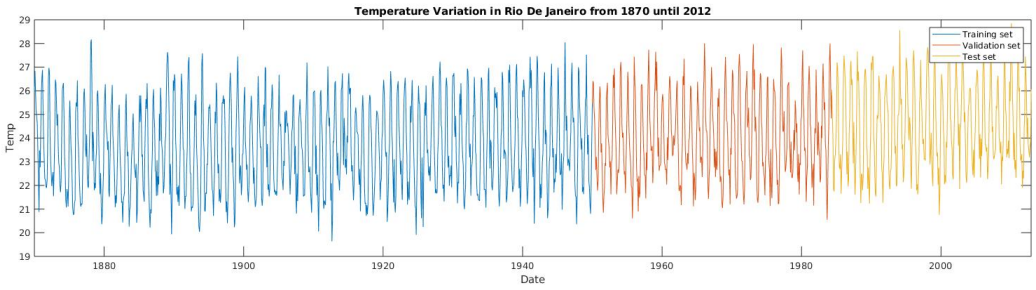


FIGURE 2.4: Temperature data of Rio De Janeiro from 1870 until the end of 2012.

1. Build and train your feedforward neural network for the city temperature data set. Since different city data sets may have different sizes, you can specify an appropriate time interval, e.g. from 1850 to 2012. For this real-world data, we consider the first 80% data as the training set and the last 20% as test set. Moreover, the training set should be further partitioned into a sub-training set consisting of the first 80% data in the training set, and the last 20% is the validation set. To be specific, we should tune the hyperparameters, such as time lags, number of hidden layers, number of neurons on each hidden layer, based on their corresponding validation errors. An example of the temperature data of Rio De Janeiro from 1870 until the end of 2012 is provided in Figure 2.4. Please refer to `global_temperature_train.m` for further details.
2. Performance Assessment: evaluate the performance of your selected network on the test set. Please refer to `global_temperature_test.m`. What else could you do to improve the performance of your network?

3. What conclusion can you derive based on the prediction results of different cities?

2.2 CLASSIFICATION

This section provides an opportunity to apply feedforward multi-layer neural networks investigated in exercise session 1 to real data sets. Specifically, we consider a benchmark data set, that is, the Breast Cancer Wisconsin (Diagnostic) Data Set [2] (`breast.mat`) from the UCI machine learning repository [3].

Exercises

1. Visualize the breast cancer data set. (Hint: you may try PCA, t-SNE, etc.)
2. Train an MLP on the `breast.mat` with tuned hyperparameters, *e.g.* appropriate number of hidden layers, neurons and epochs, for this binary classification problem. Moreover, we can also consider MLP with different training algorithms, such as `traingd`, `trainlm`, `trainbr` mentioned in exercise session 1. Note that cross-validation is recommended when the computation is not so heavy.

2.3 AUTOMATIC RELEVANCE DETERMINATION

The Automatic Relevance Determination (ARD) uses the Bayesian framework to select the inputs that are most relevant to the outputs. In order to apply ARD, you can download the Netlab software from `http://www.ncrg.aston.ac.uk/netlab/` or use the following link to get to the download's page `http://cl.ly/RwOP`. The toolbox is also available on Toledo. Note that you can run the Netlab programs within Matlab.

Exercises

1. We first consider a simple toy example of a three dimensional input selection task contained in the Netlab, which is `demard.m`. To have a better insight into the Bayesian framework contained in the ARD, we try another demo `demev1.m` which illustrates the application of Bayesian re-estimation to determine the hyperparameters in a simple regression problem.
2. We once again consider the the Breast Cancer Wisconsin (Diagnostic) Data Set with 400 training data and 30 inputs. As is similar as in Section 2.2, this binary classification problem can be solved as a nonlinear regression problem with targets ± 1. When implementing `demard.m` on `breast.mat`, which inputs are most relevant? Now, extract the input variables of `breast.mat` that are most relevant, train a MLP once again for classification on this data set and evaluate the classification results on the test set. How is its performance when compared with the results of `breast.mat` on test data in Section 2.2? Discuss the obtained results.

2.4 REPORT

For this assignment, you should write a report for each of the three sections. Answer only the questions mentioned in each Exercise subsection; explanations outside should not be explicitly commented on. Similarly, the MATLAB demos are only meant to help you getting used to necessary tools. Running the demos is thus optional and should certainly not be part of the report. The maximum page division is as follows:

- **2.1 Time-series Prediction** maximum 3 pages;
- **2.2 Classification** maximum 2 pages;
- **2.3 Automatic Relevance Determination** maximum 1 page;

References

[1]. Climate Change: Earth Surface Temperature Data
(https://www.kaggle.com/berkeleyearth/climate-change-earth-surface-temperature-data)

[2]. Breast Cancer Wisconsin (Diagnostic) Data Set
(https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29)

[3]. UCI Machine Learning Repository
(https://archive.ics.uci.edu/ml/index.php)

ASSIGNMENT 3

Unsupervised Learning and Data Visualization

This assignment investigates unsupervised learning tasks, including self-organizing map, principal component analysis, autoencoder neural networks and stacked autoencoder neural networks. Specifically, self-organizing map trained based on prototype data can be helpful when dealing with large-scale data sets. Principal Component Analysis (PCA) is one classical method for dimensionality reduction. Autoencoder can be viewed as a modified deep-version of the linear PCA, which can be used to conduct image reconstruction. Stacked autoencoder includes autoencoders as basic blocks and can be used for classification and other machine learning tasks.

3.1 SELF-ORGANIZING MAP

Self-organizing map (SOM) is a type of artificial neural network (ANN) that is trained using unsupervised learning to produce a low-dimensional (typically two-dimensional), discretized representation of the input space of the training samples, called a map, and is therefore a method to do dimensionality reduction. Self-organizing maps go back to the 1980s, and the credit for introducing them goes to Teuvo Kohonen, which leads to another name, the Kohonen map.

Like most artificial neural networks, SOMs operate in two modes: training and mapping. "Training" builds the map using input examples, while "mapping" automatically clusters a new input vector. Note that vector quantization (VQ) is utilized in the training procedure. This reflects that SOM shares a similar idea as the VQ with respect to the fact that a set of prototypes are chosen to represent the whole original data set. However, one property of SOM that makes it more advanced is that it can visualize high dimensional data by mapping them to a finite two-dimensional region, *i.e.* map space.

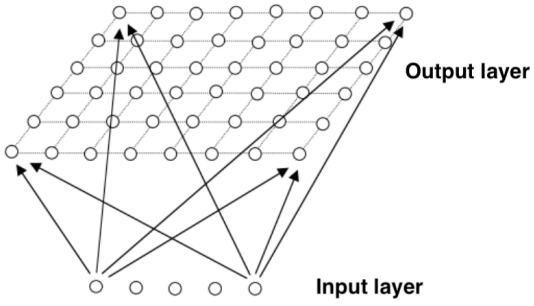


FIGURE 3.1: Network structure of SOM.

As is shown in Figure 3.1, the neurons in the output layer of the SOM network are arranged in a matrix in a two-dimensional space. Each neuron represents a prototype in the input space and its associated weight vector is exactly its position in the input space. After receiving one input vector, the network will determine

the winning prototypes and thus the winning neurons in the output layer. The training goal of SOM is to find an appropriate weight vector for each neuron so as to maintain the topology of the data.

The training procedure of SOM is simple: after receiving a training sample, each neuron in the output layer will compute the distance between this sample and its associated weight vector. The neuron with the shortest distance is chosen as the winner and it is called the best matching unit (BMU). Then the weight vectors of this BMU and its neighbor neurons will be updated so as to further shorten their distances from the training sample. The above recursive manner will stop until the algorithm converges.

Exercises

1. We provide `SOM_concentric_cylinders.m` as a template for generating data uniformly distributed within simple shapes in the plane or in the 3-D space. Then as in SOM concentric cylinders initialize multiple SOMs playing with the different topology and distance functions. Based on this template, conduct SOM on the synthetic data set `banana.mat`. Check how the prototypes distribute in the feature space before training. Then execute training and observe how the competitive rule determines the final placement according to the data distribution.

2. We now consider a real large-scale data set. The *Covertype Data Set* (`covtype.mat`) from the UCI machine learning repository aims at predicting forest cover type from cartographic variables only (no remotely sensed data). The actual forest cover type for a given observation (30×30 meter cell) was determined from US Forest Service (USFS) Region 2 Resource Information System (RIS) data. This data sets contains is of 54 dimensions and $581,012$ samples.

   Since SOM is often utilized as a clustering method, we conduct SOM on this `covtype.mat` data set. You can get an idea by running the script `example_SOM.m`. Try a few different values for the grid size, topology and number of epochs. How can you evaluate the performance (hint: Adjusted Rand Index (ARI))? To see what is happening, you might also take a look at the plots: `SOM ... Topology`, `SOM Sample Hits` and `SOM Weight Positions`, etc., on the `nntraintool`, which can be treated as data visualization. Moreover, how can you perform clustering according to `SOM Neighbor Distances`?

3.2 PRINCIPAL COMPONENT ANALYSIS

Introduction: Principal Component Analysis

Principal Component Analysis (PCA) involves projecting onto the eigenvectors of the covariance matrix. The basic idea behind PCA is to map a vector $\mathbf{x} = (x_1, x_2, \ldots, x_p)$ of a p dimensional space to a lower-dimensional vector $\mathbf{z} = (z_1, z_2, \ldots, z_q)$ in a q dimensional space (where $q < p$). We are going to only consider linear mappings, these are ones where $\mathbf{z}_d = \mathbf{e}_d^T \mathbf{x}$ for some unknown column vectors $\mathbf{e}_d$ and T denotes vector (and later on, matrix) transposition. In other words, $\mathbf{z}$ can be obtained from $\mathbf{x}$ by simple matrix multiplication:

$$\mathbf{z} = \mathbf{E}^T \mathbf{x} \tag{3.1}$$

where $\mathbf{E}$ is a p by q matrix whose column are $\mathbf{e}_d$: $\mathbf{E} = [\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_q]$. Our goal is then to reconstruct as well as possible the original vectors by using another matrix $\mathbf{F}$ so that

$$\hat{\mathbf{x}} = \mathbf{F}\mathbf{z} = \mathbf{F}\mathbf{E}^T \mathbf{x} \tag{3.2}$$

- Zero-mean the data by subtracting the mean of the dataset from each datapoint.

- Calculate the $p \times p$ dimensional covariance matrix of the zero-mean dataset.
- Calculate the eigenvectors and eigenvalues of this covariance matrix.
- Determine the dimension q of the reduced dataset by looking at the largest eigenvalues. The quality of the reduction depends on how close the sum of the largest q eigenvalues is to the sum of all p eigenvalues.
- Create the $q \times p$ projection matrix $\mathbf{E}^T$ from the eigenvectors corresponding to the q largest eigenvalues, and reduce the dataset by multiplying it with this matrix.
- To obtain the corresponding p-dimensional datapoints multiply the new data with the transpose of the projection matrix. Notice that this corresponds to choosing $\mathbf{F}$ in (3.2) such that $\mathbf{F} = \mathbf{E}$. If q is well chosen these regenerated datapoints should be fairly similar to the original datapoints, thus capturing most of the information in the dataset. Remember to add the mean again when comparing with the original data instead of the zero-mean data.

Redundancy and Random Data

The idea of this subsection is to implement the PCA algorithm in Matlab and apply this to different datasets. The above algorithm can be easily programmed in Matlab with the following functions:

- `cov(x)` calculates the average of the dataset `x`, subtracts it from each datapoint, and returns the covariance matrix of the result. For matrix input `x`, it assumes that each row is an observation, and each column is a variable.
- `[v,d]=eig(x)` returns the eigenvectors (`v`) and eigenvalues (`d`) of the square matrix `x` as square matrices.
- `diag(x)` returns the diagonal of the square matrix `x` as a column vector.
- `[v,d]=eigs(x,k)` returns the k largest eigenvalues and corresponding eigenvectors of the square matrix `x`.
- `transpose(x)` returns the transpose of the matrix `x`. The command `x'` has the same effect.
- `help` command will show the available documentation on command.

Exercises

1. Generate a 50×500 matrix of Gaussian random numbers (`randn(50,500)`) and try to reduce dimensions with PCA (interpret this as 500 datapoints of dimension 50). Examine different reduced datasets for different dimensions. Try to reconstruct the original matrix. Estimate the error, *e.g.* by calculating the root mean square difference between the reconstructed and the original data (`sqrt(mean(mean((x-xhat).^2)))`).
2. Do the same using the data file `choles_all` (standard in Matlab, can be loaded with `load ... choles_all`). For the exercise, use only the p component, which is a 21×264 matrix (so the dimension of the data equals 21). How does the reduction of random data compare to the reduction of highly correlated data?
3. Now experiment with the functions `mapstd(x)` and `processpca(x,maxfrac)` (try for example `maxfrac = 0.001`), and see whether you can obtain similar results. Note that `mapstd` must be used before `processpca` for the normalization of the matrix and that you can use `processpca('reverse',z,PS)` to get the reconstructed dataset.

3.3 AUTOENCODER

An autoencoder neural network is an unsupervised learning algorithm where the target vectors are the same as the input vectors. *i.e.*, it uses $y(i) = x(i), \forall i = 1, \ldots, N$. Figure 3.2 shows an example of an autoencoder. The autoencoder tries to learn a parametric function $\hat{x} := h_{\{W,b\}}(x) \approx x$, where the parameters are updated using backpropogation. In other words, it is trying to learn an approximation to the identity function, so as to output $\hat{x}$ that is similar to x. The identity function seems a particularly trivial function to be trying to learn; but by placing constraints on the network, such as by limiting the number of hidden units (called a sparse autoencoder), we can discover interesting structure about the data. In fact, this simple autoencoder often ends up performing a non-linear dimensionality reduction, thereby overcoming certain limitations of linear PCA. The network can be trained by minimizing the appropriate reconstruction error. An example of the use of an autoencoder can be found in [1].

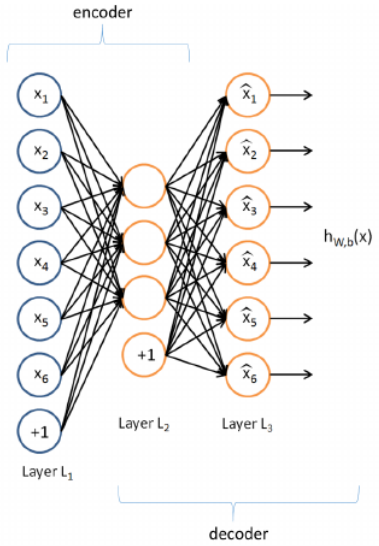


FIGURE 3.2: An example of Autoencoder. The part from the input layer to the hidden layer is called the encoder and the part from hidden layer to the output layer is called the decoder.

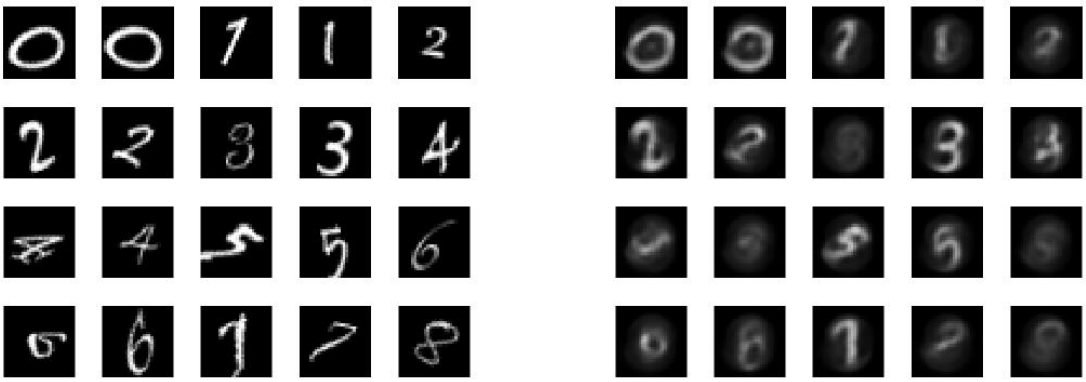


FIGURE 3.3: Visualization of the original handwritten digits (left) and their corresponding reconstruction images (right) by the autoencoder.

Exercises

1. Conduct image reconstruction on synthetic handwritten digits dataset using an autoencoder. The training data is a 1-by-5000 cell array, where each cell containing a 28-by-28 matrix representing a synthetic image of a handwritten digit. Note that you can try to tune the number of neurons in the hidden layer of the autoencoder and also the number of training epochs so as to obtain good reconstruction results. An example of the reconstruction results (not have been well tuned) can be

found in Figure 3.3. How is your own performance and how can you improve your reconstruction results? Please refer to `ae_reconstruction.m` for further details.

3.4 STACKED AUTOENCODERS

A *stacked autoencoder* is a neural network consisting of multiple layers of sparse autoencoders in which the outputs of each layer become the inputs of the successive layers. The information of interest is contained within the deepest layer of hidden units. This vector gives us a representation of the input in terms of higher-order features. For the use of classification, the common practice is to discard the "decoding" layers of the stacked autoencoder and link the last hidden layer to the a classifier, as depicted in Figure 3.4.

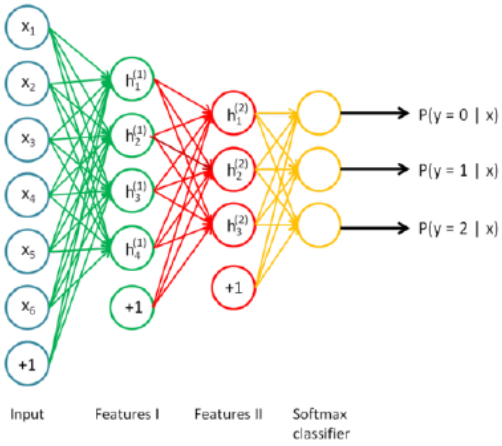


FIGURE 3.4: An example of a stacked autoencoder used for classification.

A good way to obtain parameters for a stacked autoencoder is to use *greedy layer-wise training*. We first train the autoencoder on the raw input to obtain the weights and biases from the input to the first hidden layer. Then we use this hidden layer as input to train the second autoencoder to obtain the weights and biases from the first to the second hidden layer. Repeat for subsequent layers, using the output of each layer as input for the subsequent layer. This method trains the parameters of each layer individually while freezing parameters for the remainder of the model. To produce better results, after this phase of training is complete, fine-tuning using backpropagation can be used to improve the results by updating the parameters of all layers at the same time.

Exercises

Run the script `StackedAutoEncodersDigitClassification.m` [2] from Toledo, try to understand training procedure. Use the script `DigitClassification.m` from Toledo to investigate the different parameters (MaxEpochs, number of hidden units in each layer, number of layers) and to compare the performance of the Stacked Autoencoder to a normal multilayer neural network. Do not forget to comment the command `rng('default')` when you want to average the results over multiple runs.

1. Are you able to obtain a better result with different parameters (wrt to the default ones)? How many layers do you need to achieve a better performance than with a normal neural network? What can you tell about the effect of finetuning?

3.5 REPORT

For this assignment, you should write a report for each of the four sections. Answer only the questions mentioned in each Exercise subsection; explanations outside should not be explicitly commented on. Similarly, the MATLAB demos are only meant to help you getting used to necessary tools. Running the demos is thus optional and should certainly not be part of the report. The maximum page division is as follows:

- **3.1 Self-organizing Map** maximum 2 pages;
- **3.2 Principal Component Analysis** maximum 1 page;
- **3.3 Autoencoder & 3.4 Stacked Autoencoder** maximum 3 pages.

References

[1]. SOM
(https://en.wikipedia.org/wiki/Self-organizing_map)

[2]. UFLDL Tutorial: Autoencoders
(http://ufldl.stanford.edu/tutorial/unsupervised/Autoencoders/)

[3]. MathWorks documentation: Train Stacked Autoencoders for Image Classification
(https://ww2.mathworks.cn/help/deeplearning/ug/train-stacked-autoencoders-for-image-classification.html)

ASSIGNMENT 4

Variational Auto-Encoders and Convolutional Neural Networks

In this assignment we will get hands-on practice with slightly more complicated neural networks and strategies for training them. In particular, we will see the role of weight initialization in training neural networks with a common technique called Batch-Normalization. Then, we will look at two powerful feature extractor models: Variational Autoencoders and Convolutional Neural Networks to understand how these are used to perform various machine learning tasks. The aim is to understand how such architectures are trained and the corresponding arithmetic involved.

4.1 WEIGHT INITIALIZATION & BATCH-NORMALIZATION

Weight initialization The intent of training a neural network is to obtain *good* values of parameters of the network i.e. weights and biases. When the input data is standardized (zero mean and unit variance), it may be reasonable to assume that approximately half of the weights will be positive and half of them will be negative. This in turn might indicate to set all the initial weights to zero. But this may stagnate the training, because for the same weight every neuron will compute the same output, and hence the same gradients during back-propagation and undergo the exact same updates. In other words, there is no source of asymmetry between neurons if their weights are initialized to be the same.
Another idea could be to initialize weights randomly with zero mean and small variance.

One issue with the above idea is that the distribution of the outputs from a randomly initialized neuron has a variance that grows with the input dimension. For instance, consider the variance of the raw output (before initialization) of neuron with randomly initialized weight

$$
\begin{aligned}
\operatorname{Var}\left(\sum_{i=1}^{d} w_i x_i\right) &= \sum_{i=1}^{d} \operatorname{Var}(w_i x_i) && \text{(IID random variables)} \\
&= \sum_{i=1}^{d} [\mathbb{E}(w_i)]^2 \operatorname{Var}(x_i) + [\mathbb{E}(x_i)]^2 \operatorname{Var}(w_i) + \operatorname{Var}(x_i)\operatorname{Var}(w_i) && \text{(product of IID variables)} \\
&= \sum_{i=1}^{d} \operatorname{Var}(x_i)\operatorname{Var}(w_i) && \text{(since } \mathbb{E}(x_i) = \mathbb{E}(w_i) = 0 \text{ from our assumption)} \\
&= d \operatorname{Var}(x)\operatorname{Var}(w) && \text{(IID random variables)}
\end{aligned}
$$

This indicates that we can normalize the variance of each neuron's output to 1 by scaling its weight vector by the square root of its input-dimension. That is, the recommended heuristic is to initialize each neuron's

weight vector as: $\frac{w}{\sqrt{d}}$, where d is the input dimension. This ensures that all neurons in the network initially have approximately the same output distribution and empirically improves the rate of convergence.

Note that the above analysis doesn't take into account the initialization of bias vectors and effects of the changes in output distribution of neurons as the training progresses. To this end, the following method has become the de-facto technique for neural networks training.

Batch Normalization A technique developed by Ioffe and Szegedy [2] called Batch Normalization alleviates a lot of concerns with properly initializing neural networks. It does so by explicitly forcing the activations throughout a network to take on a unit Gaussian distribution. The core observation is that this is possible because normalization is a simple differentiable operation. This method draws its strength from making normalization a part of the model architecture and performing the normalization for each training mini-batch. That is, for m samples in a minibatch, we have

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mu}{\sqrt{\sigma^2 + \epsilon}}, \qquad \forall k \in \{1, \ldots, m\}$$

where the mean $\mu = \frac{1}{m}\sum_{i=1}^{m} x_i$, variance $\sigma^2 = \frac{1}{m}\sum_{i}^{m}(x_i - \mu)^2$ are the mini-batch statistics and $\epsilon \in \mathbb{R}$ is a very small number for numerical stability. Note that simply normalizing each input of a layer may change what the layer can represent. For instance, normalizing the inputs of a sigmoid would constrain them to the linear regime of the non-linearity. To address this, a pair of *learnable* parameters $\gamma^{(k)}, \beta^{(k)}$ are introduced, that scale and shift the normalized value. That is

$$y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)}, \qquad \forall k \in \{1, \ldots, m\}.$$

Batch Normalization allows us to use much higher learning rates and be less careful about initialization. It also acts as a regularizer, in some cases eliminating the need for Dropout.

Exercises

Run the python script `BatchNormalization.py` and try to understand the procedure. The dataset is CIFAR10 which consists of $32 \times 32 \times 3$ pixels color images of natural objects from 10 classes. The intent of this exercise is to analyze the influence of Batch Normalization on the classification task. From the results of the experiment, answer the following questions.

1. How does the scale of weight initialization affect models with/without batch normalization differently, and why?
2. What does this imply about the relationship between batch normalization and batch size? Why is this relationship observed?

4.2 VARIATIONAL AUTOENCODERS

Variational autoencoder is a type of latent variable graphical model, introduced in the paper 'Autoencoding Variational Bayes, 2014'. Many statistical models make use of latent variables to describe a probability distribution over observables. Usually, the latent variables have a simple distribution, often a separable distribution. Thus when learning a latent variable model, we are finding a description of the data in terms of *independent components*. Bottleneck vectors can be seen as latent variables in context of VAE. We will make it precise in a moment.

In principal, a variational autoencoder is a model to estimate the variational lower bound on the marginal likelihood estimate of datapoints. When neural-networks are used to infer the distribution over latent variables given the observables and also as a generator given the latent variables, it renders the form of a VAE. The architecture is similar to a stacked autoencoder with the difference that instead of mapping an

input to a fixed vector, now the input is mapped to a *distribution*. The bottleneck vector in stacked autoencoders is now replaced by two different vectors representing the mean and the standard deviation of the distribution.

Consider a probabilistic model with observations $\boldsymbol{x}$, latent variable $\boldsymbol{z}$ over which we must marginalize, and a parametric probability density function $p_\theta(\boldsymbol{x})$. We introduce an approximate posterior distribution for the latent variables $q_\phi(\boldsymbol{z}|\boldsymbol{x})$ and follow the variational principle (Jordan et al., 1999) to obtain a lower-bound on the log-likelihood:

$$
\begin{aligned}
\log p_\theta(\boldsymbol{x}) &= \log \int p_\theta(\boldsymbol{x}|\boldsymbol{z}) p(\boldsymbol{z}) d\boldsymbol{z} \\
&= \log \int \frac{q_\phi(\boldsymbol{z}|\boldsymbol{x})}{q_\phi(\boldsymbol{z}|\boldsymbol{x})} p_\theta(\boldsymbol{x}|\boldsymbol{z}) p(\boldsymbol{z}) d\boldsymbol{z} \\
&= \log \left(\mathbb{E}_{q_\phi(\boldsymbol{z}|\boldsymbol{x})} \left[\frac{p(\boldsymbol{z})}{q_\phi(\boldsymbol{z}|\boldsymbol{x})} p_\theta(\boldsymbol{x}|\boldsymbol{z}) \right] \right) \\
&\geq \mathbb{E}_{q_\phi(\boldsymbol{z}|\boldsymbol{x})} \left[\log \left(\frac{p(\boldsymbol{z})}{q_\phi(\boldsymbol{z}|\boldsymbol{x})} p_\theta(\boldsymbol{x}|\boldsymbol{z}) \right) \right] \\
&= \mathbb{E}_{q_\phi(\boldsymbol{z}|\boldsymbol{x})} \left[\log \left(\frac{p(\boldsymbol{z})}{q_\phi(\boldsymbol{z}|\boldsymbol{x})} \right) + \log \left(p_\theta(\boldsymbol{x}|\boldsymbol{z}) \right) \right] \\
&= -\mathbb{D}_{\mathrm{KL}}[q_\phi(\boldsymbol{z}|\boldsymbol{x}) \| p(\boldsymbol{z})] + \mathbb{E}_{q_\phi(\boldsymbol{z}|\boldsymbol{x})}[\log p_\theta(\boldsymbol{x}|\boldsymbol{z})],
\end{aligned}
$$

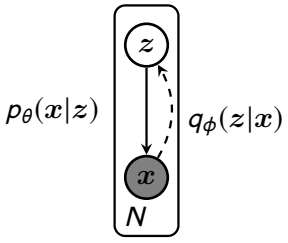


FIGURE 4.1: Solid line denotes the generative model $p_\theta(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})$ and dashed line denotes the variational approximation $q_\phi(\boldsymbol{z}|\boldsymbol{x})$ to the posterior $p_\theta(\boldsymbol{z}|\boldsymbol{x})$.

where we have used Jensen's inequality ($f(\mathbb{E}[\boldsymbol{x}]) \geq \mathbb{E}[f(\boldsymbol{x})]$) when $f(\cdot)$ is concave, $p_\theta(\boldsymbol{x}|\boldsymbol{z})$ is a likelihood function and $p(\boldsymbol{z})$ is a prior over the latent variables. This bound is often referred to as the negative free energy $\mathcal{F}$ or as the Evidence Lower Bound (ELBO). It consists of two terms: the first is the Kulback-Liebler divergence between the approximate posterior and the prior distribution (which acts as a regularizer), and the second is an expected negative reconstruction error. This bound provides a unified objective function for optimization of both the parameters θ and ϕ of the model and variational approximation respectively.

To sample from posterior $\boldsymbol{z}^i \sim q_\phi(\boldsymbol{z}|\boldsymbol{x}^i)$, we employ the *re-parameterization trick*. Note that random sampling cannot be used here, because back-propagation through such operation is not possible. This re-parametrization trick moves the random sampling operation to an auxiliary variable $\boldsymbol{\epsilon}$, which is then shifted by the mean and scaled by the standard deviation. It is often possible to express the random variable $\boldsymbol{z}$ as the deterministic variable $\boldsymbol{z} = g_\phi(\boldsymbol{\epsilon}, \boldsymbol{x})$, where $\boldsymbol{\epsilon}$ is an auxiliary variable with independent marginal $p(\boldsymbol{\epsilon})$, and $g_\phi(\cdot)$ is some vector-valued function parameterized by ϕ. For further details, the reader may refer to Section 2.4 of the original paper: 'Autoencoding Variational Bayes, 2014'.

In the demo, we let the variational approximate posterior to be a multivariate Gaussian

$$\log q_\phi(\boldsymbol{z}|\boldsymbol{x}^i) = \log \mathcal{N}(\boldsymbol{z}; \boldsymbol{\mu}^i, \sigma^{2(i)}\boldsymbol{I})$$

and the prior $p(\boldsymbol{z}) = \mathcal{N}(\boldsymbol{z}; 0, \boldsymbol{I})$. Hence, as explained above the sampling from posterior is done using $\boldsymbol{z}^i = g_\phi(\boldsymbol{x}^i, \boldsymbol{\epsilon}) = \boldsymbol{\mu}^i + \boldsymbol{\sigma}^i \odot \boldsymbol{\epsilon}$, where $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \boldsymbol{I})$ and $\odot$ is the element-wise product. For an in-depth review and a survey of related models, reader may to refer [1].

Exercises

Run the python script `vae_pytorch.py` **or** `vae_tensorflow.py` and try to understand the training procedure. In this demo we are not using the pre-trained model, instead we would train the model from scratch.

1. In particular, what similarities and differences do you see when compared with stacked autoencoder from the previous assignment? What is the metric for reconstruction error in each case?
2. Which optimizer is used in the demo file of stacked autoencoders and in variational autoencoders? Write a short note on each method and compare the pros and cons.

3. (Optional Exercise) Pick a dataset of your own choice and train a VAE on it. Feel free to modify the encoder-decoder architecture and the latent-space dimension. Be careful about the choice of activation function which might cause vanishing/exploding gradient problems (ref Backpropogation exercise in Assignment 1). Describe your choices and show the randomly generated results. (Note: Be mindful of the ethical concerns related to generative models and avoid training the model on human subjects.)

4.3 CONVOLUTIONAL NEURAL NETWORKS

Convolutional Neural Networks (CNN) is a deep learning technique that uses the concept of local connectivity. In a normal multilayer neural network all nodes from subsequent layers are connected, we call these models fully connected. Instead, here the idea is that in a lot of datasets, points that are close to each other are likely to be more related than points that are further away. For example, in images where the data points represent pixels. Pixels that are close are likely to represent the same part of the image, while pixels that are further away can represent different parts. Based on this idea, it is reasonable to assume that the parameters learned at some local position of the image would be useful at other location. This is known as *parameter sharing*, which is exploited in CNNs to dramatically reduce the number of parameters needed for training. For a detailed explanation, reader may refer to this Stanford's tutorial [3].

Convolution Arithmetic: A Toy Example

Consider an input vector $x =$ | 2 | 5 | 4 | 1 | 3 | 7 |, which we want to convolve with a 1D kernel $k =$ | 1 | 0 | 1 |. The kernel slides across the input vector at each step the product between each element of the kernel and the input element it overlaps is computed and the results are summed up to obtain the output. This step-size is formally called as *stride*. With no-padding and unit stride, this will produce the output $y =$ | 6 | 6 | 7 | 8 |. For extensions to 2D cases with various strides and padding, please refer to MILA's guide [6].

Transposed-Convolutional Neural Networks

Transposed convolutions can be seen as the opposite transformations of the convolution operations. Their need typically arises when the practitioner wants to *up-sample* from a low-dimensional space. Examples of some classical handcrafted methods are bi-linear interpolation, bi-cubic interpolation, nearest-neighbour interpolation etc. Instead of such manual feature-engineering, we could define a network that *up-sample* the low-dimensional image in an appropriate way and learn its parameters while training. For example, in the decoding layers of a convolutional-autoencoder, the transposed convolution is used to reconstruct the images back from the latent space. Sometimes in the literature, one may call such layers *deconvolution* layers. Since in mathematics, deconvolution is defined as the *inverse* of a convolution, which is different from transposed convolution, we avoid the term deconvolution here. Similar to the example above, transposed-convolution can be performed with matrix-multiplication [6].

Exercises

1. With reference to the above toy-example, discuss the following questions.

   i. Instead of multiplying the kernel with input vector iteratively, the convolution operation could be written as matrix multiplication. Infact this is how convolution operation is implemented efficiently in machine learning libraries. That is, $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$, where $\boldsymbol{y} \in \mathbb{R}^4$, $\boldsymbol{x} \in \mathbb{R}^6$ are column-vectors, and $\boldsymbol{A} \in \mathbb{R}^{4\times6}$ is the *Toeplitz matrix* whose elements are arranged in special order using the elements of kernel $\boldsymbol{k}$. As an exercise, construct the matrix $\boldsymbol{A}$.

ii. Does simply transposing the matrix $\boldsymbol{A}$ and multiplying with $\boldsymbol{y}$ would recover $\boldsymbol{x}$? Explain your observations.

2. Run the script `CNNex.m` [4] from Toledo, try to understand how different filters are connected to various channels in previous layers and the corresponding convolution arithmetic[1].

   Take a look at the layers of the downloaded CNN and answer the following questions:

   i. Take a look at the dimension of the weights `(size(convnet.Layers(2).Weights))` of the first convolutional layer (layer 2). If you think back at what you saw in class and/or in [4], what do these weights represent?

   ii. Inspect layers 1 to 5. If you know that a ReLU and a Cross Channel Normalization layer do not affect the dimension of the input, what is the dimension of the input at the start of layer 6 and why?

   iii. What is the final dimension of the problem (*i.e.* the number of neurons used for the final classification task)? How does this compare with the initial dimension?

3. The script `CNNDigits.m` [5] runs a small CNN on the handwritten digits dataset. Use this script to investigate some CNN architectures. Try out some different amount of layers, combinations of different kinds of layers, dimensions of the weights, etc. Discuss your results. Be aware that some architectures will take a long time to train!

4.4 REPORT

Write a report of maximum 6 pages (including text and figures) to discuss

- the answers to the exercise questions of Section 4.1, Section 4.2 and Section 4.3.

References

[1]. Diederik P. Kingma, Max Welling. An Introduction to Variational Autoencoders (`https://arxiv.org/pdf/1906.02691.pdf`)

[2]. Sergey Ioffe, Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, 2015 (`https://arxiv.org/pdf/1502.03167.pdf`)

[3]. UFLDL Tutorial: Convolutional Neural Network (`http://ufldl.stanford.edu/tutorial/supervised/ConvolutionalNeuralNetwork`)

[4]. MathWorks documentation: Image Category Classification Using Deep Learning (`https://nl.mathworks.com/help/vision/examples/image-category-classification-using-deep-learning.html`)

[5]. MathWorks documentation: Create Simple Deep Learning Network for Classification (`https://nl.mathworks.com/help/deeplearning/examples/create-simple-deep-learning-`

[1] Since we do not have access to a CUDA-capable GPU on the computers, we can not train a large CNN in class. If you are interested in CNNs and have access to a GPU for computing you may try the full demo [4] at home (note: this is not necessary for the report).

network-for-classification.html)

[6]. A Guide to Convolution Arithmetic for Deep Learning, Dumoulin, V. and Visin, F. (2018). (arXiv: 1603.07285v2)