Learning Regularization Parameters of Inverse Problems via Deep Neural Networks

A simplified Introduction

Guided Independet Study, Nagoya University, Spring 2023

Dominik Strutz

July 2023

Contents

1 Introduction **2**

1.1 The Problem . . . 2

1.2 Invertability of $A^TA + \lambda^2 I$ . . . 4

1.3 Optimal Space of ϵ . . . 4

1.4 Ill-posed Problems . . . 4

1.5 Error Decomposition . . . 5

2 Algorithm **5**

3 Offline Phase **6**

3.1 Model, Noise, and Test Data . . . 6

3.1.1 Code . . . 7

3.2 Generation of Training Data . . . 7

3.2.1 Code . . . 7

3.3 Computation of the Optimal Regularization Parameter . . . 8

3.3.1 Tikhonov Regularization . . . 8

3.3.2 Finding the Optimal Regularization Parameter . . . 8

3.3.3 Code . . . 8

3.4 Setting up the Neural Network . . . 10

3.4.1 Code . . . 12

3.5 Train the Neural Network . . . 13

3.5.1 Code . . . 14

4 Online Phase **15**

4.1 How to set up the Forward Propagation . . . 15

4.2 Computation of the Estimates for the Regularization Parameter . . . 15

4.3 Computation of the Inverse Solution . . . 15

4.4 Code . . . 15

5 Discussion **16**

5.1 Evaluation . . . 16

5.2 Future Work . . . 16

A Packages **18**

1 Introduction

In this work, we discuss a method of learning regularization parameters of inverse problems via deep neural networks. It is based on the reference written by Afkham et al. (2021). We explain the main idea of this work and give an implementation for an explicit case of the Algorithm 1 discussed there.

The generation of a usable data set is based on a vector of observations $b \in \mathbb{R}^m$. These observations are often measurements from the real world, e.g., in the context of inverse problem theory in weather forecasting, oceanography, hydrology, and petroleum engineering. An exact measurement however is almost impossible. Therefore, we introduce a vector of additive noise $\epsilon \in \mathbb{R}^m$, which accounts for each individual measurement's error.

The desired solution we are trying to find in the context of inverse problems are the design points $x_{\text{true}} \in \mathbb{R}^n$ of the respective observation b. Within this context we assume, that the process that maps a design point onto an observation b is known. We call this process the model A. In this introduction we restrict the model A to be a linear map given by

$$\begin{aligned} \mathbb{R}^n &\longrightarrow \mathbb{R}^m \\ x &\longmapsto Ax, \end{aligned} \tag{1}$$

with $A \in \mathbb{R}^{m \times n}$, which results in following relation

$$b = Ax_{\text{true}} + \epsilon. \tag{2}$$

To solve the introduced problem with an deep learning approach for a different data sets of the same context, it is efficient to split construction and training of a deep neural network (DNN) from the computation of the solution. Therefore, we first describe the construction and training in the offline phase and then the computation of an approximation $\widehat{x}_{\text{true}} \in \mathbb{R}^n$ of the desired solution x_{true}.

1.1 The Problem

More precisely, we want to find the best possible prediction for the design point given through

$$\widehat{x}(\lambda) = \arg\min_x \mathcal{J}(x, b) + \mathcal{R}(x, \lambda) \tag{3}$$

with the regularization parameter $\lambda \in \mathbb{R}$ letting its optimal value be

$$\lambda_{\text{opt}} = \arg\min_\lambda \|\widehat{x}(\lambda) - x_{\text{true}}\|_2 \tag{4}$$

and

$$\mathcal{J}(x, b) = \|Ax - b\|_2^2\,, \tag{5}$$

$$\mathcal{R}(x, \lambda) = \lambda^2 \|x\|_2^2\,, \tag{6}$$

which then defines the Tikhonov regularization (Hilt et al., 1977)

$$\widehat{x}(\lambda) = \arg\min_x \|Ax - b\|_2^2 + \lambda^2 \|x\|_2^2 = \arg\min_x \left\| \begin{pmatrix} A \\ \lambda I \end{pmatrix} x - \begin{pmatrix} b \\ 0 \end{pmatrix} \right\|_2^2. \tag{7}$$

Assuming A is invertible the $\arg\min_{\widehat{x}} \left\|A\widehat{x}^{(j)}(\lambda) - b^{(j)}\right\|_2^2$ for every tuple in the data set $\left\{x_{\text{true}}^{(j)}, b^{(j)}\right\}_{j=1}^J$

is solved by $A^{-1}b^{(j)} = \widehat{x}^{(j)}(\lambda)$ and the regularization parameter λ to be 0. But for inverse problems, we generally cannot expect the model to be intervitable. Regularization is a general method for solving ill-posed and ill-conditioned problems. Hence, we need to solve an alternated problem starting with a single tuple of the data.

Now, we alter the solution further by first, rewriting the norm

$$\begin{aligned} \|Ax - b\|_2^2 &= (Ax-b)^T(Ax-b) \\ &= (Ax)^T(Ax) - (Ax)^T b - b^T Ax + b^T b \\ &= x^T A^T A x - 2x^T A^T b + b^T b \end{aligned} \tag{8}$$

and deriving the least square solution via vector analysis

$$\begin{aligned} \frac{\partial}{\partial x}(x^T A^T A x - 2x^T A^T b + b^T b) &= 2(A^T A x - A^T b) \\ 0 &= 2(A^T A x - A^T b) \\ \Leftrightarrow x &= (A^T A)^{-1} A^T b. \end{aligned} \tag{9}$$

Remark 1.1. Only the first derivative is needed to find the only existing extrema of the term, since the norm is convex.

Second, we set

$$\begin{aligned} \widetilde{A} &= \begin{pmatrix} A \\ \lambda I \end{pmatrix} \\ \widetilde{b} &= \begin{pmatrix} b \\ 0 \end{pmatrix}, \end{aligned} \tag{10}$$

and observe following simplifications through substitution

$$\begin{aligned} x &= (\widetilde{A}^T \widetilde{A})^{-1} \widetilde{A}^T \widetilde{b} \\ &= \left(\begin{pmatrix} A^T & \lambda I \end{pmatrix} \begin{pmatrix} A \\ \lambda I \end{pmatrix} \right)^{-1} \begin{pmatrix} A \\ \lambda I \end{pmatrix}^T \begin{pmatrix} b \\ 0 \end{pmatrix} \\ &= \left(\begin{pmatrix} A^T & \lambda I \end{pmatrix} \begin{pmatrix} A \\ \lambda I \end{pmatrix} \right)^{-1} \begin{pmatrix} A^T & \lambda I \end{pmatrix} \begin{pmatrix} b \\ 0 \end{pmatrix} \\ &= (A^T A + \lambda^2 I)^{-1}(A^T b + 0) \\ &= (A^T A + \lambda^2 I)^{-1} A^T b. \end{aligned} \tag{11}$$

By using this result for λ_{opt}

$$\begin{aligned} \lambda_{\text{opt}} &= \arg\min_{\lambda} \|\widehat{x}(\lambda) - x_{\text{true}}\|_2 \\ &= \arg\min_{\lambda} \left\| (A^T A + \lambda^2 I)^{-1} A^T b - x_{\text{true}} \right\|_2 \\ &= \arg\min_{\lambda} \left\| (A^T A + \lambda^2 I)^{-1} A^T (A x_{\text{true}} + \epsilon) - x_{\text{true}} \right\|_2 \\ &= \arg\min_{\lambda} \left\| (A^T A + \lambda^2 I)^{-1} (A^T A x_{\text{true}} + A^T \epsilon) - x_{\text{true}} \right\|_2 \end{aligned} \tag{12}$$

A natural question is now what we should do in the case of multiple x_i. As a first attempt, we consider $\frac{1}{2J}\sum_{i=1}^{J} \left\| A\widehat{x}^{(j)}(\lambda) - b^{(j)} \right\|_2^2$. Since every summand is a norm, it is easy to see that the sum is minimal if and only if each summand is minimal. For each summand we get an optimal λ, λ_{opt}. However, the problem can be stated equivalently as $\|AX_{\text{true}}^T - B\|$ with $X_{\text{true}} \in \mathbb{R}^{J\times n}$, $B \in \mathbb{R}^{m\times J}$ and be considered in different matrix instead of vector norms where some commonly used are:

(i) Row-sum norm: $\|A\|_\infty = \max_{1\le i\le m} \sum_{i=1}^{n} |a_{i,j}|$

(ii) Column-sum norm: $\|A\|_1 = \max_{1\le i\le n} \sum_{i=1}^{m} |a_{i,j}|$

(iii) Spectral norm: $\|A\|_2 = (\max_i \lambda_i(A^TA))^{\frac{1}{2}} = \max_i \sigma_i(A)$

(iv) Frobenius norm: $\|A\|_F = (\mathrm{tr}(A^TA))^{\frac{1}{2}} = (\sum_{i=1}^{\min\{m,n\}} \sigma_i^2(A))^{\frac{1}{2}}$

(v) Nuclear norm: $\|A\|_* = \sum_{i=1}^{\min\{m,n\}} \sigma_i(A)$

where λ is the eigenvalue and σ is the singular value of A.

1.2 Invertability of $A^TA + \lambda^2 I$

The product A^TA of any real valued matrix $A \in \mathbb{R}^{m\times n}$ is called gram matrix.

Theorem 1.1. *Every eigenvalue λ_i of a real valued gram matrix A^TA is non-negative.*

Proof 1.1. *Let $\langle\cdot,\ \cdot\rangle$ denote the inner product, λ_i an eigenvalue of A, and x its respective eigenvector, then*
$\lambda\,\|x\|^2 = \langle A^TAx,\ x\rangle = \|Ax\|^2$
Hence, λ_i is non-negative.

Adding positive multiple of the identity matrix of respective dimension $I_n \in \mathbb{R}^{n\times n}$ results in a positive definite matrix, i.e., all eigenvalues are greater than 0. Hence, the matrix $A^TA + \lambda^2 I$ is regular.

1.3 Optimal Space of ϵ

The solution of Equation (12) is the minimum of a norm. From the properties of norms follows, that we have a convex behaviour with an overall minimum of 0. This is true if and only if the argument of the norm is equal to 0. Here, the question arise of which values of noise ϵ are feasible, such that we are still able to find a lambda which solves $f(\lambda) = (A^TA + \lambda^2 I)^{-1}(A^TAx_{\text{true}} + A^T\epsilon) - x_{\text{true}}$. We can achieve $f(\lambda) = 0$ in the case $A^T\epsilon = \lambda^2 x_{\text{true}}$.

1.4 Ill-posed Problems

Definition 1.1. The problem $b = A(x_{\text{true}})$ is well posed, if

(i) for all admissible data b exists an x_{true} with $b = A(x_{\text{true}})$,

(ii) the solution x_{true} is uniquely determined by the data,

(iii) the solution depends continuously on the data.

If one of the above conditions is violated, the problem $b = A(x_{\text{true}})$ is said to be ill posed.

Regularization in the form of prior knowledge on the distribution of x_{true} must be included to compute reasonable solutions. A prior probability distribution of an uncertain quantity, often simply called the prior, is its assumed probability distribution before some evidence is taken into account.

1.5 Error Decomposition

To elaborate why incorporating prior knowledge is important to access current predictive limitations, we state the well known Bayes' theorem.

Theorem 1.2 (Bayes' theorem)**.** *We have*

$$\mathrm{P}(\theta|\boldsymbol{D}) = \mathrm{P}(\theta)\frac{\mathrm{P}(\boldsymbol{D}|\theta)}{\mathrm{P}(\boldsymbol{D})}\,, \tag{13}$$

where the data are represented by $\boldsymbol{D}$ and parameters are represented by θ. $\mathrm{P}(\theta)$ is the prior probability, $\mathrm{P}(\boldsymbol{D}|\theta)$ the likelihood, and $\mathrm{P}(\boldsymbol{D})$ the evidence. The solution $\mathrm{P}(\theta|\boldsymbol{D})$ is called the posterior probability.

Another important effect, which occurs while applying regularization is the bias-variance tradeoff. The bias–variance tradeoff is the property of a model that the variance σ^2 of the parameter estimated across samples can be reduced by increasing the bias in the estimated parameters (Neal, 2019). The conflict lies in trying to simultaneously minimize these two sources of error (variance and bias) that prevent the model to generalizing beyond their training set. This property is involved in the prediction of the regularization parameter λ since the neural network uses a quadratic loss function, we can decompose its error in a reducible and irreducible part.

$$\underbrace{\mathrm{E}_{b,\lambda_{\mathrm{opt}},D}\left[\left(\theta(b)-\lambda_{\mathrm{opt}}\right)^2\right]}_{\text{Expected Error}} = \underbrace{\mathrm{E}_{b,D}\left[\left(\theta(b)-\overline{\theta(b)}\right)^2\right]}_{\text{Variance}} + \underbrace{\mathrm{E}_{b,\lambda_{\mathrm{opt}}}\left[\left(\overline{\theta(b)}-\overline{\lambda_{\mathrm{opt}}}\right)^2\right]}_{\text{Bias}^2} + \underbrace{\mathrm{E}_{b,\lambda_{\mathrm{opt}}}\left[\left(\lambda_{\mathrm{opt}}-\overline{\lambda_{\mathrm{opt}}}\right)^2\right]}_{\text{Noise}} \tag{14}$$

Here, $\bar{\cdot}$ is a average over respective values and $\theta(b)$ the observation b propagated through the network's parameter θ.

2 Algorithm

The overall algorithm will be discussed and implemented with regard to an example and is given through:

Algorithm 1 Learning regularization parameters via DNNs

```
 1: Offline phase
 2:     Define model $A$, noise model $\epsilon$, and $x^{(j)}_{\mathrm{true}}$
 3:     Generate appropriate training signals $b^{(j)} = Ax^{(j)}_{\mathrm{true}} + \epsilon^{(j)}$, for $j = 1, \dots, J$
 4:     Obtain $\lambda^{(j)}_{\mathrm{opt}}$
 5:     Set up DNN $\widehat{\Phi}$
 6:     Use training data $\left\{ b^{(j)}, \boldsymbol{\lambda}^{(j)}_{\mathrm{opt}} \right\}_{j=1}^{J}$ to compute network parameters $\widehat{\theta}$
 7: Online phase
 8:     Obtain new data $\mathbf{b}$
 9:     Propagate $b$ through the learned network to get $\widehat{\lambda} = \widehat{\Phi}(b;\ \widehat{\theta})$
10:     Compute inverse solution $\widehat{x}(\widehat{\lambda})$
```

3 Offline Phase

In this section, we will describe in 5 steps how to construct and train a DNN to predict the regularization parameter of the Tikhonov regularization. Since the training process of a neural network requires the highest computational effort within the Algorithm 1, we advise to split the training and all its precursors.

3.1 Model, Noise, and Test Data

First, we define our model A. For this example, let A be the following 3×2 matrix

$$A = \begin{pmatrix} 1 & 2 \\ 1 & 3 \\ 1 & 4 \end{pmatrix}. \tag{15}$$

The design points x_{true} and the noise ϵ are modeled through normal distributions with mean μ_x and μ_ϵ respectively and variance σ^2.

$$\begin{aligned} x_{\text{true},p} &\sim \mathcal{N}(\mu_x,\, \sigma^2) \quad p = 1, \dots, n, \\ \epsilon_q &\sim \mathcal{N}(\mu_\epsilon,\, \sigma^2) \quad q = 1, \dots, m \end{aligned} \tag{16}$$

These values are implanted through a function which takes besides of the mean μ and variance σ^2, an argument to specify the number of desired values to draw from the respective distribution. For the drawing the design points x_{true} and the noise ϵ the model A is required, so that Equation (2) is well defined with regard to its dimensions.

For the computation of our predictor for a certain design point $\widehat{x}_{\text{true}}(\lambda)$, we implement Equation (11). The respective loss is then given by $\|Ax - b\|_2^2 + \lambda^2 \|x\|_2^2$ and the misfit by $\|\widehat{x}(\lambda) - x_{\text{true}}\|_2$.

Remark 3.1.

(i) Solving equations is more efficient and numerically stable then computing inverses.

(ii) Vectorized functions, i.e. functions which accepts one or more arrays and apply the funtction on every entry of an array are an option for less code.

The following code can be used to draw the design points x_{true} and the noise ϵ according to the above introduction. Also, the computation of derived values predictor of the design point $\widehat{x}(\lambda)$, the loss $\mathcal{J}(x, b) + \mathcal{R}(x, \lambda)$, and the misfit are implemented.

3.1.1 Code

```python
A = np.array([[1, 2], [1, 3], [1, 4]])
dist = np.random.normal

def draw_x_true(length = 1, mu = 0, sigma = 1, model = np.array([1]).reshape(1, 1), dist = np.
    random.normal):
    shape = model.shape
    X_true = dist(mu, sigma, np.array([length, shape[1]]))
    return X_true

def draw_epsilon(length = 1, mu = 0, sigma = 1, model = np.array([1]).reshape(1, 1), dist = np.
    random.normal):
    shape = model.shape
    Epsilon = dist(mu, sigma, np.array([length, shape[0]]))
    return Epsilon

def compute_x_hat(lambda_, A, b):
  return np.linalg.solve(A.T @ A + lambda_ * np.identity((A.T @ A).shape[0]) , A.T @ b)

def compute_loss(lambda_, A, x, b):
  C = np.dot(A,x)
  return np.linalg.norm((C - b.reshape(C.shape)))**2 + lambda_**2 * np.linalg.norm(x)
vcompute_loss = np.vectorize(compute_loss, excluded=['A','x','b'])

def compute_x_misfit(lambda_, A, x, b):
  return np.linalg.norm(x - compute_x_hat(lambda_ = lambda_, A = A, b = b))
vcompute_x_misfit = np.vectorize(compute_x_misfit, excluded=['A','x','b'])
```

3.2 Generation of Training Data

For an inverse problem, the design points x_{true} are usually not available. Hence, we implement a way to draw observations b without the use of the model A or design points x_{true}. Still, the procedure is directly comparable to the generation of those latter ones and make use of a given distribution. To complete the training data, we need to produce the observations b to the respective design points x_{true}. Now, we make use of our defined functions to get a data set including a design matrix X_{true}, respective noise ϵ, and an observations matrix B.

3.2.1 Code

```python
def draw_B(length = 1, mu = 0, sigma = 1, model = np.array([1]).reshape(1, 1), dist = np.random.
    normal):
    shape = model.shape
    B = dist(mu, sigma, np.array([length, shape[0]]))
    return B

def compute_b(length = 2, mu = 0, sigma = 1, model = np.array([1]).reshape(1, 1)):
    X_true = draw_x_true(length, mu, sigma, model, dist)
    Epsilon = draw_epsilon(length, mu, sigma, model, dist)
    B = model @ X_true.T - Epsilon.T # + 10 * np.ones((3, 1))
    return X_true, Epsilon, B

X_true, epsilon, B = compute_b(length = 1500, mu = 0, sigma = 5, model = A)
```

3.3 Computation of the Optimal Regularization Parameter

After generating all data, we need to compute the optimal regularization parameter λ_{opt} to complete the training set. This is done for each individual design point x_{true} and respective observation b with its noise ϵ. The optimal regularization parameter λ_{opt} varies not only between mentioned tuples, but also depend on the kind of regularization applied. In our example, we choose the Tikhonov Regularization.

3.3.1 Tikhonov Regularization

The altered least square optimization problem is given by Equation (7) and its solution via substitution is given by Equation (11). After defining the Tikhonov regularization, we now briefly discuss advantages. In general, solutions using this regularization, tend to be simpler. Therefore odd results for our design points x_{true} are unlikely, e.g., resulting in absurdly high values. Also, closeness to the original least squares problem and the equivalence to a lagrangian and likelihood estimation making its derivation and application easy to understand.

3.3.2 Finding the Optimal Regularization Parameter

There are many possible options to regularize an ill-posed problem and even more to chose the regularization parameter λ. Prominent methods include the discrepancy principle (DP), the generalized cross-validation (GCV) method, the unbiased predictive risk estimator (UPRE), and the residual periodogram. Some of these methods require also knowledge about the variance σ^2 (or estimation) and additional hyperparameter. Others are costly with regard to its computation. For this example, we chose a simple grid search without any heuristic and only applying a discretization over the regularization parameter λ and finding the minimum over all discrete values of regularization parameter λ with regard to the misfit in Equation (4)

By doing this for each tuple (x_{true}, b with ϵ), we were able to display the distribution of the optimal regularization parameter λ_{opt}

3.3.3 Code

```python
lambda_grid = np.arange(-12, 12, 1)
best_lambda = []

fig, ax = plt.subplots()
for i in np.arange(0, X_true.shape[0]): #B_z.shape[0]
  losses = vcompute_x_misfit(lambda_ = lambda_grid, A = A, x = X_true[i, :], b = B[:, i])
  best_lambda.append(lambda_grid[np.argmin(losses)])
  color = next(ax._get_lines.prop_cycler)['color']
  plt.axvline(x=lambda_grid[np.argmin(losses)], color = color, ymin=0, ymax=0.75)
  plt.plot(lambda_grid, losses, color = color)
plt.xlabel(r"$\lambda$")
plt.ylabel("Misfit")
plt.title("Misfit of the Tikhonov regression")
ax.set_ylim([0, ax.get_ylim()[1]])
best_lambda = np.array(best_lambda).reshape((len(best_lambda), 1))
axtw = ax.twinx()
plt.ylabel(r"Frequency of $\lambda_{opt}$")
plt.hist(x = best_lambda, bins = lambda_grid, fc = (0, 0, 1, 0.3))
name = "normal"
plt.savefig(name + ".svg", format = "svg")
plt.show()
```

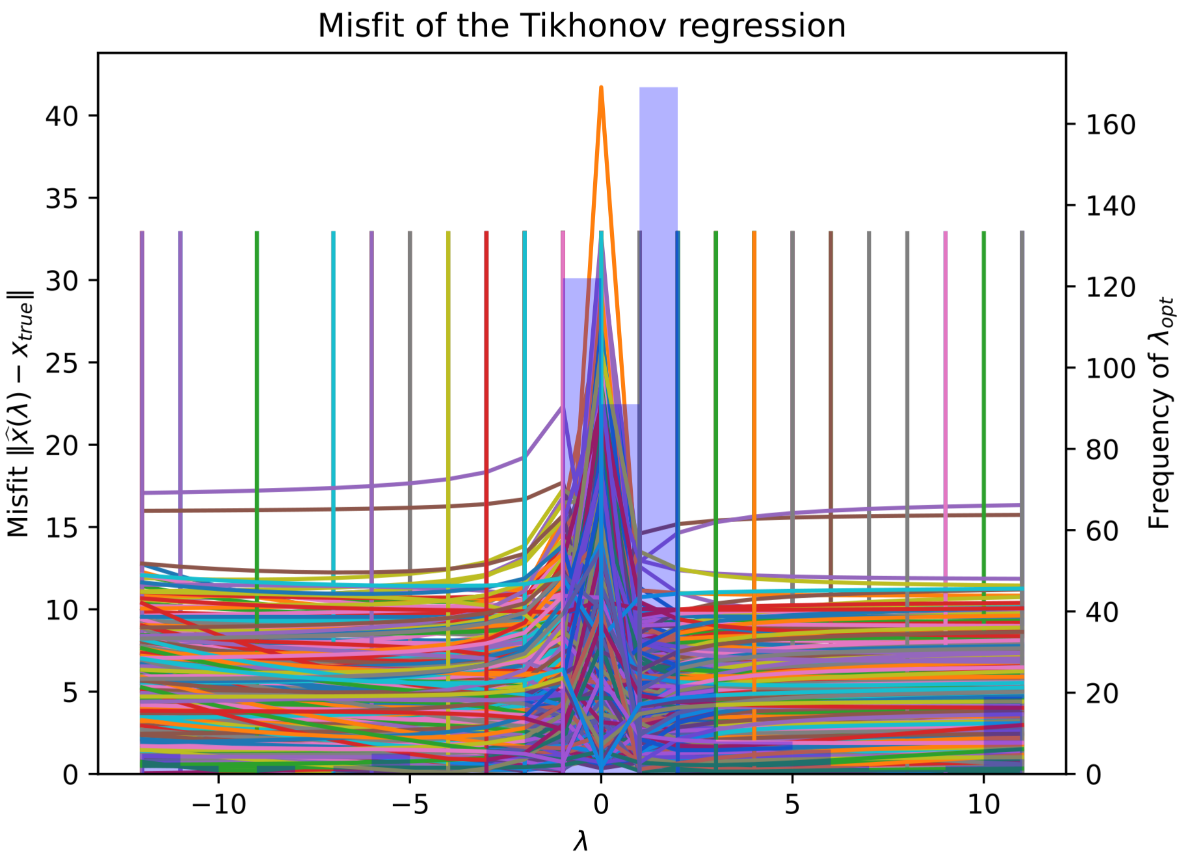

Figure 1: Misfit compared to x_{true} dependent on λ and histogram of chosen λ_{opt}

3.4 Setting up the Neural Network

Neural networks is a term widely used for many different architectures. Whereas most of them are sophisticated and difficult to implement, the one used in this example is a rather simple fully connected feed forward neural network. Such a simple architecture might not be able to solve any difficult task, but for this example, we only need a few parameters. Training less parameters require less data and therefore less time and computational resources. We start by defining a class for our feed forward network, containing a constructor (__init__) and a forward pass function. The constructor builds the architecture, here we have the input layer defined by the model A followed by two hidden layers with four and five neurons respectively, and last, the output layer, which is a single scalar, our regularization parameter λ.

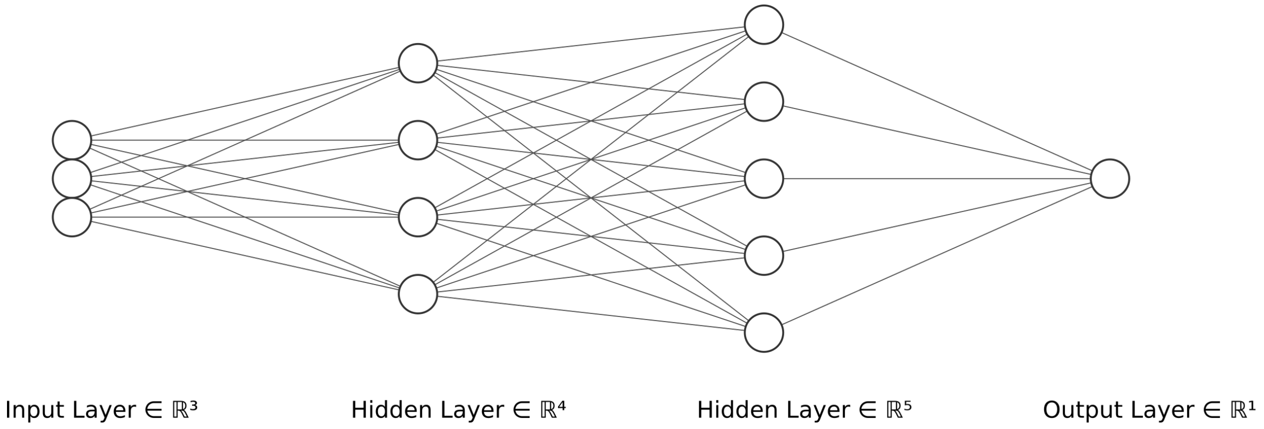

Figure 2: Network architecture

For training of the neural network's parameter θ and testing its predictive capabilities, we also define respective functions. The training loop expects the data $\left\{b^{(j)}, \lambda_{\mathrm{opt}}\right\}_{j=1}^{J}$, model A, a loss function(here MSE), and an optimizer (here ADAM).

$$\mathrm{MSE}(\lambda) = \frac{1}{J}\sum_{i=1}^{J}(\lambda - \lambda_{\mathrm{opt}})^2 \tag{17}$$

The Mean squared error (MSE) can represent the difference between the actual targets λ_{opt} and the targets λ predicted by the model. In this context, it is used to determine the extent to which the model fits the data well. The MSE loss is one of the most widely used loss functions for regression problems, though its widespread use stems more from mathematical convenience than considerations of actual loss in applications (Hastie et al., 2009).

Adaptive Moment Estimation (ADAM) is an optimizer combining gradient descent and the momentum method. In this optimization algorithm, running averages with exponential forgetting of both the gradients and the second moments of the gradients are used (Kingma & Ba, 2017).

Given all the expected inputs, the training loop computes a forward pass for every tuple in the batch, calculates the loss and distributes the errors according to the optimizer during the backward pass. This is done for each batch containing a certain number of training samples.

To assess the performance of the fully connected feed forward neural network, a second loop is implemented as a function. The test loop requires the same inputs as the training loop except the optimizer, because it only asses not change the neural network. For that, the test loop computes a forward pass for every tuple in the batch, calculates the losses and stores them.

An epoch describes one iteration over all tuples in the training set divided in batch size large blocks. The gradient step uses the information of a whole batch. This implies, that the number of gradient steps per epoch is equal to the size of the training set divided through the size of a batch. The change of weights are then evaluated with a scaling of respective errors. This scaling of the learning process is called learning rate. These values are known as hyperparameter and are crucial to shape the training process. However, typically simple networks are fairly resistant against slightly sub-optimal hyperparameter, which results in slower, but still accurate learning. Therefore, hyperparameter are chosen with a rule of thumb and no particular tuning is applied.

Remark 3.2. To make sure that the parameters are not changed during the assessment, the gradients are frozen.

3.4.1 Code

```python
class FeedForward(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(A.shape[0],4),
            nn.ReLU(),
            nn.Linear(4,5),
            nn.ReLU(),
            nn.Linear(5,1),
        )

    def forward(self,x):
        x = self.net(x)
        return x

def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        trainingStep_loss.append(loss.item())

        if batch % 10 == 0:
            loss, current = loss.item(), (batch + 1) * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")


def test_loop(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    validation_loss, correct = 0, 0

    with torch.no_grad():
        for X, y in dataloader:
            # X is here a batch sized tensor
            pred = model(X)
            validation_loss += loss_fn(pred, y).item()
            validationStep_loss.append(loss_fn(pred, y).item())
            testStep_loss.append(validation_loss)

    validation_loss /= num_batches
    print(f"Avg loss: {validation_loss:>8f} \n")

model = FeedForward()
loss_fn = nn.MSELoss()
epochs = 8
batch_size = 64
learning_rate = 1e-3
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

3.5 Train the Neural Network

To make use of our predefined functions for training and testing the pytorch neural network, we transform our observation matrix B into a tensor (with datatype float32 for technical reasons) and repeat the procedures for the optimal regularization parameter λ_{opt} of the respective observations.

After the transformation to make the data processable for the pytorch neural network, we split them into a training set and a test set. Here, the test set contains 20% of the overall data set.

Remark 3.3. The argument random_state is used to make the splitting reproducible.

By referring from using the whole data set to make one gradient step, we need to introduce a data loader, which makes it possible to feed the pytorch neural network batch sized training sets during the training loop. This procedure is called mini batch learning and speeds up the learning process by trading of a little bit of the accuracy of each gradient step.

Now, we let the train an test loop change the network's parameter through iterating over a predetermined number of epochs. At the side, the losses for each epoch will be computed and stored, which is another way to display performance compared to show the performance on only a single batch. To get a number for the performance, we can apply the loss function after completed training to the predicted regularization parameter λ after the forward pass of the observations B together with the known optimal regularization parameter λ_{opt}. Last, we plot the gathered losses to display the training progress and performance on the test set.

3.5.1 Code

```python
# convert into PyTorch tensors
X = torch.tensor(B.T, dtype=torch.float32)
y = torch.tensor(best_lambda, dtype=torch.float32).reshape(-1, 1)

# set up train test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# create DataLoader, then take one batch
train_dataloader = DataLoader(list(zip(X_train, y_train)), shuffle=False, batch_size=batch_size)
test_dataloader = DataLoader(list(zip(X_test, y_test)), shuffle=False, batch_size=batch_size)

trainingEpoch_loss = []
validationEpoch_loss = []
testEpoch_loss = []

for t in range(epochs):
    trainingStep_loss = []
    validationStep_loss = []
    testStep_loss = []
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_fn, optimizer)
    trainingEpoch_loss.append(np.array(trainingStep_loss).mean())
    test_loop(test_dataloader, model, loss_fn)
    validationEpoch_loss.append(np.array(validationStep_loss).mean())
    testEpoch_loss.append(np.array(testStep_loss).mean())
print("Done!")

plt.plot(trainingEpoch_loss, label='Training loss')
plt.plot(validationEpoch_loss,label='Validation loss')
plt.xlabel("Epoch")
plt.ylabel("MSE")
plt.title("Training progress")
plt.legend()
name = "train"
plt.savefig(name + ".svg", format = "svg")
plt.show

y_pred = model.forward(X_test)
loss_fn(y_pred, y_test)
```

4 Online Phase

The online phase is rather short compared to the offline phase. We will discuss the generation of new observations B_{new}, their forward propagation through the pytorch neural network, and the computation of the inverse solution, which was our original goal.

4.1 How to set up the Forward Propagation

As basis for the forward propagation through our trained pytorch neural network, we generate new observations B_{new}. We use the function introduced in Section 3.2 in a similar fashion to the generation of the training data. At this point, we need to transform the new observations B_{new} like the observations B into a tensor.

4.2 Computation of the Estimates for the Regularization Parameter

For the computation of the regularization parameter λ for the new observations B_{new} for the same model A, we only need to forward pass the tensor through the pytorch neural network and store the output.

4.3 Computation of the Inverse Solution

Finally, we compute the inverse solution as well by using a predefined function from Section 3.1, Equation (11). For that, we pass the from the pytorch neural network predicted regularization parameter λ with the respective observation b_{new} and the model A to the beforehand mentioned function.

4.4 Code

```
B_new = draw_B(100, 0, 1, A)
B_new = torch.tensor(B_new, %dtype=torch.float32)

lambda_hat = model.forward(B_new)

B_new = B_new.detach().numpy()
lambda_hat = lambda_hat.detach().numpy()
X_hat = []

for b, lambda_ in zip(B_new, lambda_hat):
  X_hat.append(compute_x_hat(lambda_, A, b))
```

5 Discussion

5.1 Evaluation

To evaluate the performance of the algorithm, we have in general two options. The first option is to implement an established method to solve the problem and compare results on one or more test sets. The second option would be to set up a measure of misfit like the MSE for regression or the accuracy for classification tasks. Since we are working here on a regression problem and already introduced the MSE, we will use the latter option to assess the performance of the algorithm.

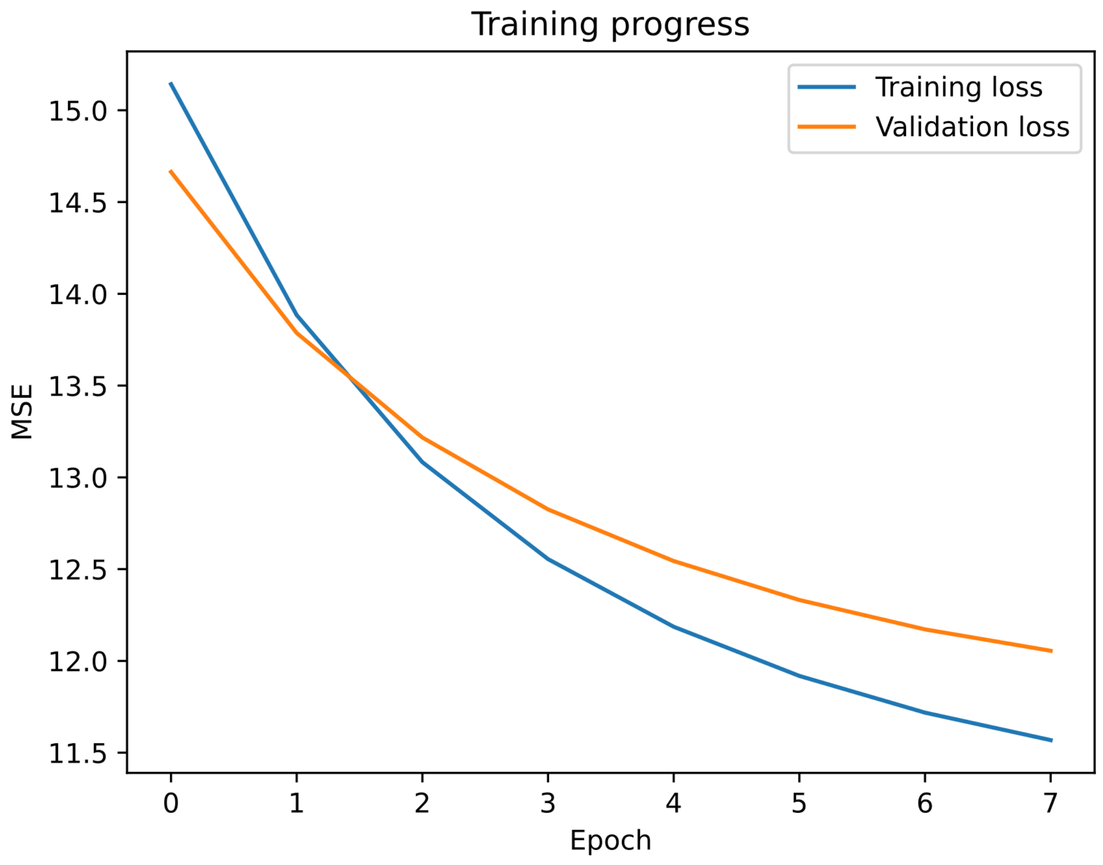

Figure 3: Decline of the MSE over the 8 training epochs

For a particular set of 1500 generated data points, we scored an overall MSE of 12.21.

5.2 Future Work

In our example, the respective noise vector ϵ consists of a scalar multiplied and stacked to the needed amount. This solution makes the code stable against changes of the model and requirements to the noise (automatic up scaling), but the noise on the observation in each element is the same. The stability can also be achieved through slightly more complex implementations of the drawing process, e.g., drawing process of design points x_{true}. Also, with regard to the covariance matrix $diag(\sigma_1, \ldots, \sigma_n)$ where $\sigma_i = \sigma_{i+1} \forall i$ and all off diagonals are equal to 0, which allow element wise drawing of values, we use a rather optimal assumption which certainly not always hold for real world data.

Future work could tackle these simplifications and asses the stability of the algorithm in this matter while iterating through multiple correlated design point element, i.e, some off diagonal elements are not equal to 0.

A Packages

Following packages are required for all inbuilt function we used:

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor

from google.colab import files
device = (
    "cuda"
    if torch.cuda.is_available()
    else "mps"
    if torch.backends.mps.is_available()
    else "cpu"
)
print(f"Using {device} device")
```

References

Afkham, B. M., Chung, J., & Chung, M. (2021). Learning Regularization Parameters of Inverse Problems via Deep Neural Networks [Publisher: arXiv Version Number: 1]. https://doi.org/10.48550/ARXIV.2104.06594
Other 27 pages, 16 figures

Hastie, T., Tibshirani, R., & Friedman, J. (2009). The Elements of Statistical Learning [ISBN: 9780387848570]. *Elements*, *1*, 337387. https://doi.org/10.1007/b94608

Hilt, D. E., Northeastern Forest Experiment Station (Radnor, Pa.), Seegrist, D. W., & United States. (1977). *Ridge, a computer program for calculating ridge regression estimates* /. Dept. of Agriculture, Forest Service, Northeastern Forest Experiment Station, https://doi.org/10.5962/bhl.title.68934

Kingma, D. P., & Ba, J. (2017). Adam: A Method for Stochastic Optimization [arXiv:1412.6980 [cs]]. Retrieved August 24, 2023, from http://arxiv.org/abs/1412.6980
Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015

Neal, B. (2019). On the Bias-Variance Tradeoff: Textbooks Need an Update [arXiv:1912.08286 [cs, stat]]. Retrieved August 23, 2023, from http://arxiv.org/abs/1912.08286
Comment: MSc Thesis