Information und Technik
Nordrhein-Westfalen

PYTPRA
Maschinelles Lernen in der Praxis
PyTorch

Lernen

- Typischerweise beobachten wir Paare von Zahlen, etwa Stundenlohn (wage) und Jahre an Berufserfahrung (experience)
- Daraus wollen wir **lernen**, wie der Zusammenhang zwischen beiden Grössen im Wesentlichen ist
- D.h. uns ist klar, dass der Zusammenhang eher im Mittel als in jedem einzelnen Fall gilt
- In Formeln ausgedrückt gibt es einen datengenerierenden Prozess, ausgedrückt durch die folgende Gleichung. Das **wahre** Modell (w) sei: $y=f(x)+e$
- Ein **Verfahren**, die beiden Modellparameter zu bestimmen, ist die Methode der kleinsten Quadrate, d.h. wir minimieren die Summe der Fehlerquadrate u^2 durch Wahl von f()
- Zum Glück führt Python die Berechnung der **„Kleinsten Quadrate“ (KQ)** für uns aus
- Wir wollen dazu später ein Laborexperiment mit dem Jupyter Notebook **PYTPRA_02_02_PyTorchNonLinReg.ipynb** durchführen,
- Gefolgt von einer Fallstudie zu Autopreisen in **PYTPRA_02_03_LinRegMLP.ipynb**
- Aber zuvor müssen wir uns noch an Tensoren gewöhnen **PYTPRA_02_01_PyTorchTensors.ipynb**

Tensoren

- **Welche Stellen im Code sind unklar?**
- **Bescheiben Sie, wie ich von numpy arrays zu torch tensors komme!**
- **Und wie komme ich zurück?**

```
15.0

 Welcome to PyTorch
tensor(15.)
15.0
torch.Size([3]) torch.Size([3, 1]) torch.Size([3])

1
```

- Die 3 Tensoren c, z, u haben verschiedene Shapes. Das lösen wir!

```python
import numpy as np
import torch #no "as t" for pedagogic reasons to be aware for torch

#fill your first tensor, take numpy array as starting point
#use l_vars as naming convention to signal data type
#note that arrays and tensors are C++ objects
#Python and PyTorch are written in C++
#tensors try to mirror arrays as far as possible

a_c = np.array([0.5,  14.0, 15.0], dtype='float32')#use float cause GPU slow down with 64bit
print(a_c[2])#usual indexing
a_u = np.array([48.4, 60.4, 68.4], dtype='float32')
a_z = np.zeros((1,3), dtype='float32')#such constructors re-appear later

#we want to build data matrix and the arrays shall become columns

#we transpose and create tensor from array, could also use torch.from_numpy(a_c)

t_c = torch.t(torch.tensor(a_c))
print("\n Welcome to PyTorch")
print(t_c[2])#note the diff! hence to zoom in need
print(t_c[2].numpy())#numpy gives back a numpy array

t_u = torch.t(torch.tensor(a_u))
t_z = torch.t(torch.tensor(a_z))
#check shape
print(t_c.shape, t_z.shape, t_u.shape)
t_u.dim()
```

Tensoren

- Die 3 Tensoren c, z, u haben verschiedene Shapes. Das lösen wir!
- Wir unterziehen c und u einer „unsqueeze(p) Behandlung
- Diese ergänzt an Stelle p eine Dimension. Hier an zweiter, weil wir von Null an zählen
- Danach können wir drei 3x1 Matrizen = 3 Spaltenvektoren zu einer 3x3 Matrix t_m zusammenfassen

```
#align shapes, perform this only once!!!
t_cc = t_c.clone().detach().unsqueeze(1)#clone creates copy in different memory location
t_uc = t_u.clone().detach().unsqueeze(1)
#check shape
print(t_cc.shape, t_z.shape, t_uc.shape)

t_m = torch.cat((t_cc,t_uc,t_z),1)

print(t_m)

print(t_m.shape)
```

```
torch.Size([3, 1]) torch.Size([3, 1]) torch.Size([3, 1])
tensor([[ 0.5000, 48.4000,  0.0000],
        [14.0000, 60.4000,  0.0000],
        [15.0000, 68.4000,  0.0000]])
torch.Size([3, 3])
```

Tensoren

- **Welche Stellen im Code sind unklar?**
- **Warum sollte uns der Hauptspeicher interessieren?**

```
#play with index

print("Eine Zelle", t_m[1,1],"\n Erste Spalte",t_m[:,1],"\n Ausschnitt",t_m[:,::2])
print("\n Abfragen", t_m[t_m>50])

#how is it stored in RAM?

print("all elements are stored in a sequence starting at", t_m.data_ptr())
print("if u want to go to beginning of next line move right in RAM x times. x =", t_m.stride()[0])

#enforce that RAM storage contains no gaps
#t_m = t_m.contiguous
#print(t_m)#gives back adress in hexadecimal eg 0x000001F2EBE0DFD0, in decimal 127723
#use https://bin-dez-hex-umrechner.de/
```

```
Eine Zelle tensor(60.4000)
 Erste Spalte tensor([48.4000, 60.4000, 68.4000])
 Ausschnitt tensor([[ 0.5000,  0.0000],
        [14.0000,  0.0000],
        [15.0000,  0.0000]])

 Abfragen tensor([60.4000, 68.4000])
all elements are stored in a sequence starting at 1655947500352
if u want to go to beginning of next line move right in RAM x times. x = 3
```

Tensoren

- **Welche Stellen im Code sind unklar?**
- **Was bewirkt die Anwendung einer Funktion tanh()?**

```
#saving in default working dir of Jupyter, serialisation
torch.save(t_m, "save_t_m.pt")#use pt and not pth! Latter one collides with python path
t_ml = torch.load("save_t_m.pt", map_location = 'cpu')#avoid default access to GPU
print(t_ml)
```

```
tensor([[ 0.5000, 48.4000,  0.0000],
        [14.0000, 60.4000,  0.0000],
        [15.0000, 68.4000,  0.0000]])
```

```
#compute, apply methods of tensor to it, names like in Numpy
torch.tanh(t_m)
```

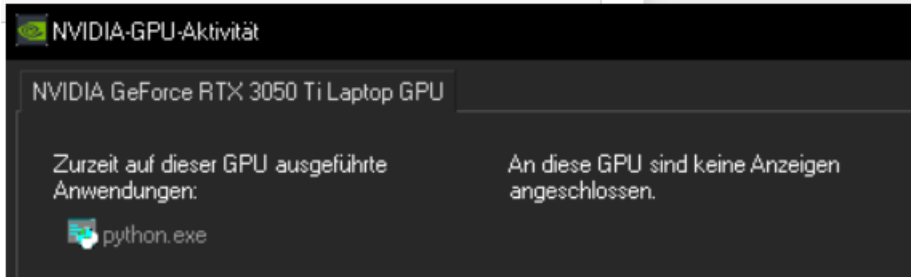
```
tensor([[0.4621, 1.0000, 0.0000],
        [1.0000, 1.0000, 0.0000],
        [1.0000, 1.0000, 0.0000]])
```

Tensoren GPU/CPU

Falls Sie bei sich CUDA* und eine entsprechende GPU haben, so sieht die Ausführung etwa wie folgt aus:

```
print(torch.__version__)
print(torch.cuda.device_count())
if(torch.cuda.device_count()):
    print("GPU available")
```

```
2.1.2
1
GPU available
```



- Wir können zwischen den Tensortypen CPU/GPU hin und her wechseln

```
if(torch.cuda.device_count()):
    #create tensor on GPU
    t_gpu_c = torch.tensor(a_c, device = "cuda")
    print(t_c)
    print(t_gpu_c)
    #cuda:0 counts the GPUs, if only one c=0, else 0,1,2
    print("\n")
    #alternative route copy tensor to tensor_gpu
    t_gpu_u = t_u.to(device = "cuda")
    print(t_u)
    print(t_gpu_u)
    #and the return ticket
    t_uu = t_gpu_u.to(device = "cpu")
    print(t_uu)
```

*CUDA (Compute Unified Device Architecture) wurde von Nvidia entwickelt und ermöglicht Rechnen auf der GPU

Tensoren GPU/CPU

Falls Sie bei sich CUDA* und eine entsprechende GPU haben, so sieht die Ausführung wie folgt aus:

- Ab einer bestimmten Grösse ist die GPU schneller

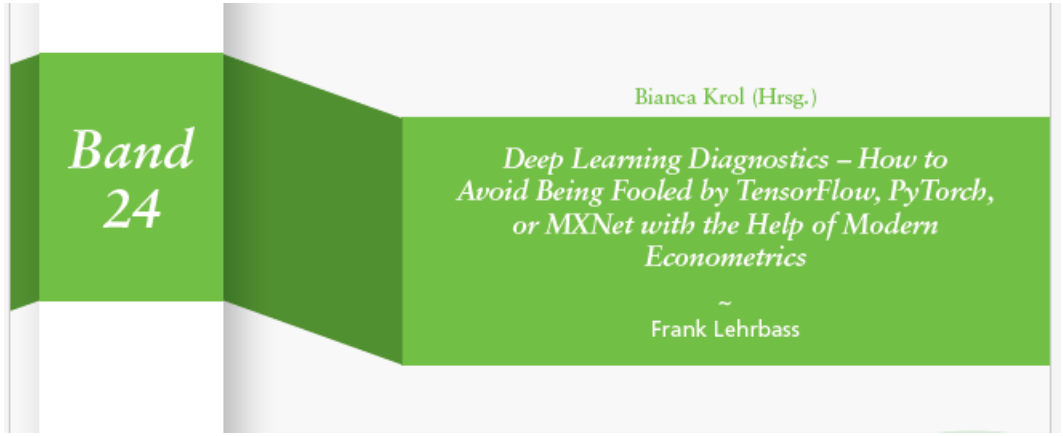
```
torch.manual_seed(2023)

if(torch.cuda.device_count()):
    for n in range(1,11):

        t_r = torch.randint(0, 5, (n*1000000,))
        t_gpu_r = t_r.to(device = "cuda")

        start_time = time.time()
        t_r ** 2
        print("Dimension des quad. Vektors", n*1000000)
        print('---Rechenzeit CPU: %s seconds---' % (time.time() - start_time))

        start_time = time.time()
        t_gpu_r ** 2
        print('---Rechenzeit GPU: %s seconds---' % (time.time() - start_time))
        print("\n\n")
```

```
Dimension des quad. Vektors 6000000
---Rechenzeit CPU: 0.02281808853149414 seconds---
---Rechenzeit GPU: 0.0 seconds---


Dimension des quad. Vektors 7000000
---Rechenzeit CPU: 0.003142118453979492 seconds---
---Rechenzeit GPU: 0.0 seconds---


Dimension des quad. Vektors 8000000
---Rechenzeit CPU: 0.016026735305786133 seconds---
---Rechenzeit GPU: 0.0 seconds---


Dimension des quad. Vektors 9000000
---Rechenzeit CPU: 0.016028165817260742 seconds---
---Rechenzeit GPU: 0.005105733871459961 seconds---


Dimension des quad. Vektors 10000000
---Rechenzeit CPU: 0.01775670051574707 seconds---
---Rechenzeit GPU: 0.008194684982299805 seconds---
```

Experiment

- Wir führen dasselbe Experiment durch wie in

https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3803328



- Es gibt einen Tageszähler x, der **nicht-linear** auf das Gewicht der Hühnchen abgebildet wird
- Der wahre Zusammenhang f(x) ist somit nicht-linear
- Nicht messbare „hühnchen-individuelle“ Merkmale werden durch einen Störterm reflektiert
- Eine lineare Regression würde verzerrte Schätzer haben

The data is generated in order "to mimic … real chicken-growth data" (Riazoshams, Midi, Ghilagaber, 2019, 227), which was recorded of "broiler chicken supply in an area of Marvdasht, Fars province, Iran" (Riazoshams, Midi, Ghilagaber, 2019, 216).

To make this data easily learnable for an MLP with a typical sigmoid[15] activation function I simplify the four-parameter logistic function, which is given by Riazoshams, Midi, and Ghilagaber (2019, 227), to a two-parameter sigmoid function. Specifically, the data are simulated from the following logistic model:

$$y_t = \frac{1}{1+exp(b_0+b_1 x_t)} + u_t \qquad (7)$$

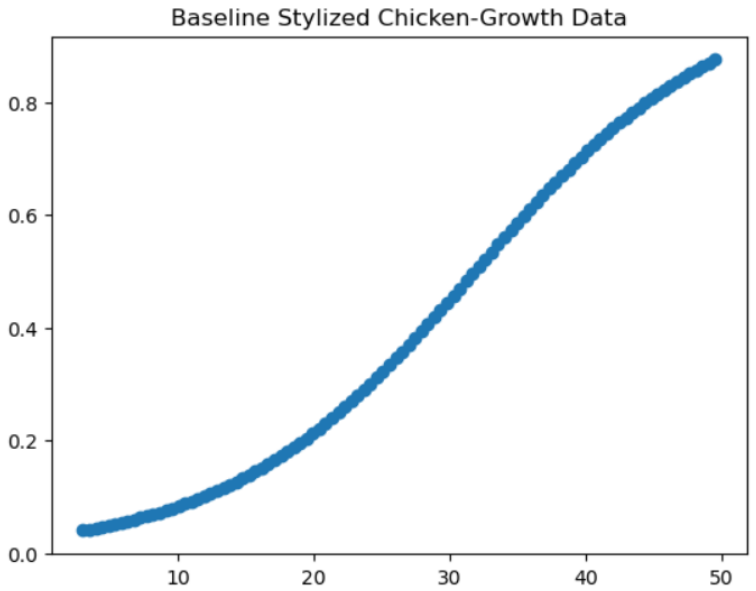
The u_t are error terms following a normal distribution with a variance of s^2 and an expected value of zero. The independent variable x_t ranges from 3 to 50 and counts the days on which the dependent variable chicken weight y_t is measured. The weight measurement is normalized such that the maximum weight is one. The parameters of the logistic function are set as follows:

Parameter	b_0	b_1	s
True Value	3.5	-0.11	0.05

Table 1: Parameters of Logistic Function

Experiment

- **Welche Stellen im Code PYTPRA_02_02_PyTorchNonLinReg sind unklar?**



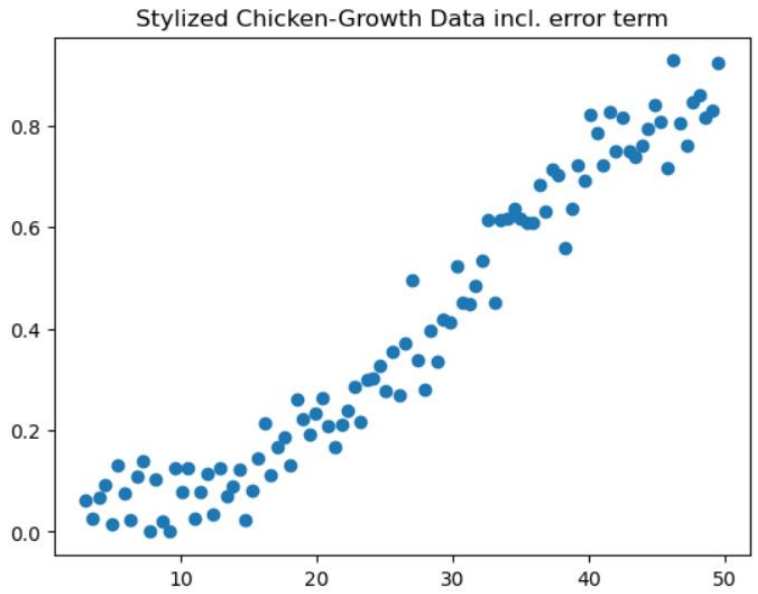

```
import time
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
print(torch.__version__)
```

```
1.12.1
```

```
# data generation
torch.manual_seed(2023) #always first

#set paras as in DL Diag ifes WP
n=100 #size of training sample

#original chicken-growth data
b0=3.50
b1=-0.11
s=0.05

start_obs = 3
end_obs = 50
no_obs = n
t_x = torch.arange(start_obs, end_obs, step = (end_obs-start_obs)/no_obs) #Returns a 1-D tensor
print(t_x.shape)
t_y = 1/(1+torch.exp(b0+b1*t_x))

plt.scatter(t_x.data.numpy(), t_y.data.numpy())
plt.title('Baseline Stylized Chicken-Growth Data')
plt.show()
```

Experiment

- **Welche Stellen im Code sind unklar?**




```
eps_mean = torch.zeros(n)
t_y = torch.add(t_y,torch.normal(eps_mean,s))#When the shapes do not match,
#the shape of mean is used as the shape for the returned output tensor
t_y = torch.maximum(t_y,torch.zeros(n)) #avoid negative weights
t_y = torch.minimum(t_y,torch.ones(n)) #avoid y outside of scope of sigmoid
#plt.scatter(t_x.data.numpy(), t_y.data.numpy())
plt.scatter(t_x, t_y)
plt.title('Stylized Chicken-Growth Data incl. error term')
plt.show()
print(t_x.shape, t_y.shape)
#NOTE that u cannot use this for training! YOU need matrix with n rows and one column
#to get this follow DL Book p. 155
t_x = t_x.clone().detach().unsqueeze(1)#old style torch.tensor(t_x) leads to warning
print(t_x.shape)
t_y = t_y.clone().detach().unsqueeze(1)
```

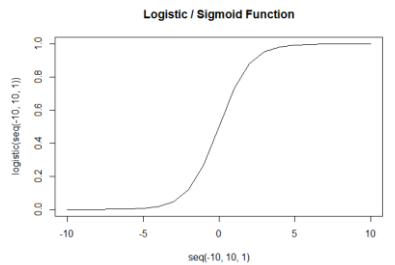
Experiment

- Wir bereiten eine Trennung der Stipo mit 100 Elementen vor. Wir teilen die Daten in 20% (Kreuz-) Validierungs- und 80% Trainingsdaten auf – das nutzen wir jedoch erst in der nächsten Fallstudie
- Wir definieren f(x) als sigmoide Funktion
- Wir definieren unsere Fehlernorm als mittlere Residuenquadrate = Mean Squared Error
- In f() geht eine Linearkombination a+bx ein, also dort wo rechts x steht
- **Welche Stellen im Code sind unklar?**

```
p_val = 0.2 #percentage for validation

n_samples = t_x.shape[0]
n_val = int(p_val * n_samples)

shuffled_indices = torch.randperm(n_samples)

train_indices = shuffled_indices[:-n_val]
val_indices = shuffled_indices[-n_val:]

train_indices, val_indices  # <1>

train_t_x = t_x[train_indices]
train_t_y = t_y[train_indices]
val_t_x = t_x[val_indices]
val_t_y = t_y[val_indices]
```

```
n_input = 1
n_hidden = 1
n_out = 1

#how to construct models out of classes read 6.2.1 in "DL with Pytorch
mlp_chicken_pt = nn.Sequential(nn.Linear(n_input, n_hidden),
                               nn.Sigmoid())#defines the graph
#linear is a+bX, then follows a hidden layer of n_hidden-s sigmoids, l
loss_fn = nn.MSELoss()
optimizer = torch.optim.Adam(mlp_chicken_pt.parameters(), lr=0.02)
losses_train = []
losses_val = []
start_time = time.time()
```

$$f(x) = \frac{1}{1 + e^{-x}}$$



OPTIONAL: OOP Exkurs

- PyTorch hat ein **Submodul**, welches Neuronalen Netzen (nn) gewidmet ist
- Dynamisch können wir im Code eine Instanz der **Klasse** (nn.Seq.) definieren, also ein **Objekt**
- Die **Datenstruktur** dieser Klasse enthält u.a. die Gewichte (Modellparameter) wie wir später sehen werden
- Die **Member Funktionen** dieser Klasse werden wie üblich mit nn.Funktion aufgerufen – hier **Methode** named_parameters()
- Dieses Submodul enthält auch diverse **Fehlerfunktionen**, z B Mean Sqrd Error = **MSE**

```
import torch
import torch.nn as nn
```

```
#how to construct models out of classes read 6.2.1 in "DL with Pytorch", Stevens et al
mlp_chicken_pt = nn.Sequential(nn.Linear(n_input, n_hidden),
                               nn.Sigmoid())#defines the graph
```

```
print("\n Params of Pytorch mlp_chicken_pt")
for name, param in mlp_chicken_pt.named_parameters():
    print(name, param.shape)
    print(name, param.data.numpy())
```

```
 Params of Pytorch mlp_chicken_pt
0.weight torch.Size([1, 1])
0.weight [[0.11273148]]
0.bias torch.Size([1])
0.bias [-3.5769615]
```

```
loss_fn = nn.MSELoss()
```

Vertiefung nach Bedarf durch eigenständige Lektüre Kap. 6 in Stevens et al (DL with PyTorch)

Theorie nicht-lineare Regression

- Wir definieren f(a+bx) als sigmoide Funktion
- Die beiden Modellparameter a und b nennen wir ab jetzt (Netz-) Gewichte, englisch Weights W.
- Jede Vorhersage ist somit eine Funktion von den gegebenen Inputdaten x und den Modellparametern, deshalb wie in (1)
- Abermals vereinfacht unser Modell die Realität
- Deshalb gibt es ein Residuum in (3)
- Zu minimieren: Residuenquadrate = Mean Squared Error

2 Theoretical Background

The goal of estimating an econometric model or training an MLP is to discover the true process which generates our data. In short: The aim is to uncover the Data Generating Process (DGP).

Assumption 1 (Specification)

Therefore, it is assumed, that – if there were no noise in the world – there would be a true relationship as follows:

$$y_t = f(x_t, W) \qquad (1)$$

For the sake of simplicity, I assume the function *f(x,W)* to be a continuous mapping from the independent variable *x* to the dependent variable *y*. The variable *x* might be a vector of regressors or factors.

The parameters of the function are denoted by a parameter or weight matrix *W*. The function might be linear or not. To allow for gradient- and Hessian-based learning the function needs to be twice differentiable in *W*.[10] Note that this implies continuity in *W*.

With a noisy world the mapping in equation (1) can be expected to hold only on average, which I denote via a conditional expectation:

$$E(y_t|x_t) = f(x_t, W) \qquad (2)$$

Let us put the effects of noise into a random variable *u*. The full data generating process can then be expressed as follows:

$$y_t = f(x_t, W) + u_t \qquad (3)$$

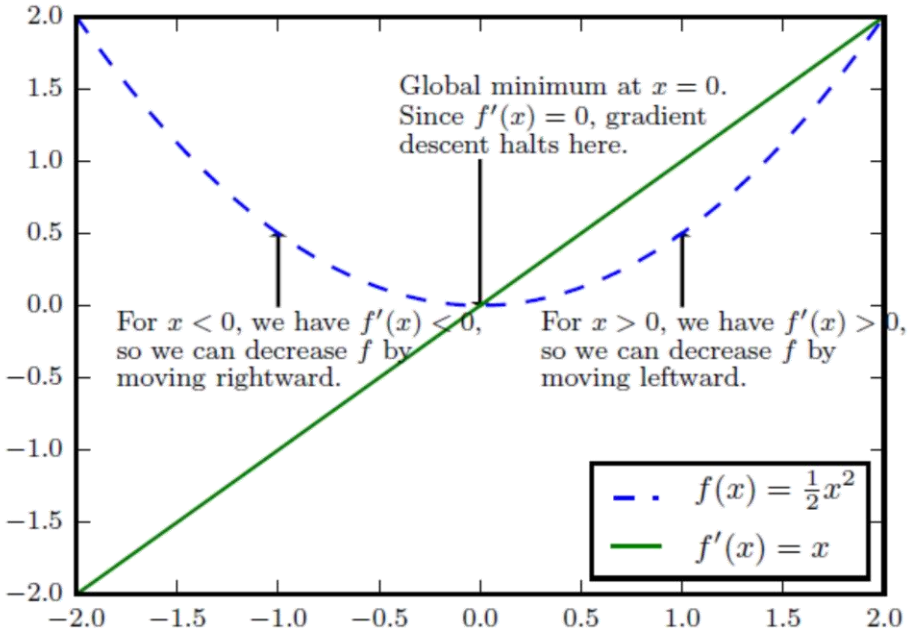
The data in our samples is disturbed by *u*, which makes us err concerning the true relation between *y* and *x*. Therefore, the u_t are called error terms. Equation (3) is already a strong assumption.

Lernverfahren

- Diesmal lösen wir das Problem der Minimierung der Residuenquadrate **numerisch**
- Wir fangen mit einer zufälligen Belegung der Modellparameter a und b an
- Dann speisen wir ein x ein und schauen, wie weit wir vom gegebenen y Wert abweichen
- Wir betrachten den summierten quadrierten Fehler - abermals - als Funktion von (a, b)
- Wir wollen diese Funktion minimieren und nutzen dazu das Vorzeichen der partiellen Ableitungen
- Man kann sich dies wie ein Tal durchschreiten vorstellen - und zwar mit verbundenen Augen wo wir nur Trippelschritte machen – auch um zu überleben;-)

Lernverfahren

- Konkret sog. Gradientenabstieg im Bsp. Y = ½ x² (Goodfellow, 2016, 80)



- Rechts von x=0 ist der Gradient positiv, also ist die Richtung zur Erhöhung des Funktionswertes x+. Da wir minimieren wollen gehen wir in die entgegengesetzte Richtung also x-. Man bedenke, dass wir über a und b minimieren! D.h. das Bild müsste auf der Abzisse a oder b haben.

Lernverfahren

Backpropagation

Man betrachtet den Prognosefehler und verändert die Modellparameter a und b so, dass man besser wird. Dieses Verfahren nennt man Backpropagation = Zurücktragen des Fehlers. Backpropagation ist „a method for calculating gradients on the parameters of a network. In particular, backprop is just an .. implementation of the chain rule from calculus“ (Taddy, 2019, 303).

Die Berechnung des Gradienten wächst mit der Datenmenge. Deshalb nutzt man in der Praxis „estimates of those gradients based upon a **subset of the data**. This is the **Stochastic Gradient Descent (SGD)** algorithm“ (Taddy, 2019, 304).

Den Subset nennt man „**minibatch**“ (Goodfellow, 2016, 148 + 272). Typische Grössen sind 32 – 256.

Dieser Subset wird in jeder Epoche neu ausgewürfelt.

Lernverfahren

Backpropagation mit Autograd

Die Berechnung des Gradienten wird durch eine Eigenschaft der PyTorch Tensoren unterstützt (Stevens, 2020, 125). Diese Tensoren merken sich, welche Operationen auf Ihnen bereits ausgeführt worden sind. Angenommen t_x enthält nur einen Wert x=10. Nun wird t_x quadriert. Der neue Tensor „t_x^2“ enthält nicht nur 100 als Wert, sondern im Attribut „grad“ die ausgeführte Funktion.

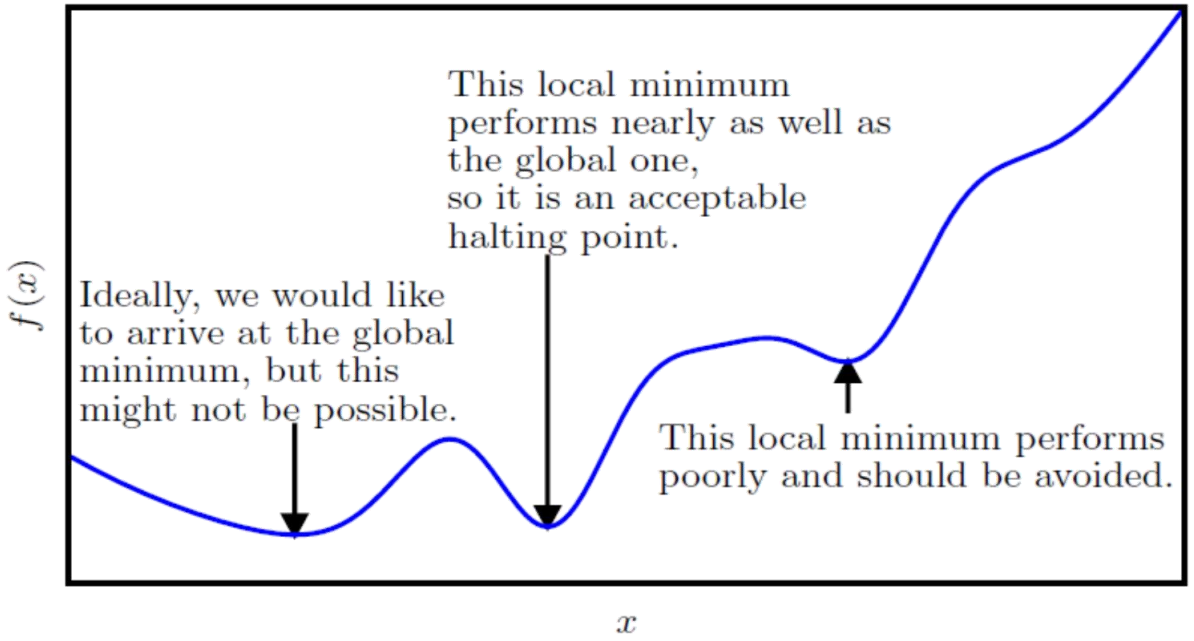
```
#show grad feature
t_x = torch.tensor([10.0], requires_grad = True)
t_x = t_x.pow(2)
print(t_x)
print(t_x.detach().numpy())
print("\nXXX DONE XXX")
```

```
tensor([100.], grad_fn=<PowBackward0>)
[100.]

XXX DONE XXX
```

Kapitel 5 in Stevens (2020) empfiehlt sich zum Nacharbeiten für alle, die tiefer einsteigen möchten.

Lernverfahren

- Beim sog. Gradientenabstieg kann es zu Problemen kommen (Goodfellow, 2016, 81)



- Hierzu gibt es Abhilfen. Für eine vertiefte Behandlung empfiehlt sich ein Master-Studium.

Experiment

- Konkret führen wir das Lernverfahren wie rechts definiert aus
- **Welche Stellen im Code sind unklar?**
- **Wo findet Feedforward statt?**

Verhindere, dass Gradienten kumuliert werden!

Unser Berichtswesen

```
def generic_training_loop(n_epochs, optimizer, model, loss_fn, train_t_u, val_t_u,train_t_c, val_t_c):
    for epoch in range(1, n_epochs + 1):
        train_t_p = model(train_t_u) #all details are in model
        train_loss = loss_fn(train_t_p, train_t_c) #and the loss function
        losses_train.append(train_loss.item())
        with torch.no_grad(): # <1>
            val_t_p = model(val_t_u)
            val_loss = loss_fn(val_t_p, val_t_c)
            losses_val.append(val_loss.item())
            assert val_loss.requires_grad == False

        optimizer.zero_grad()
        train_loss.backward()
        optimizer.step()
        if epoch <= 10 or epoch % 100 == 0:
            print(f"Epoch {epoch}, Training loss {train_loss.item():.4f},"
                  f" Validation loss {val_loss.item():.4f}")
```

Lernen (Gewichtsanpassung) ohne Validierungsdaten!

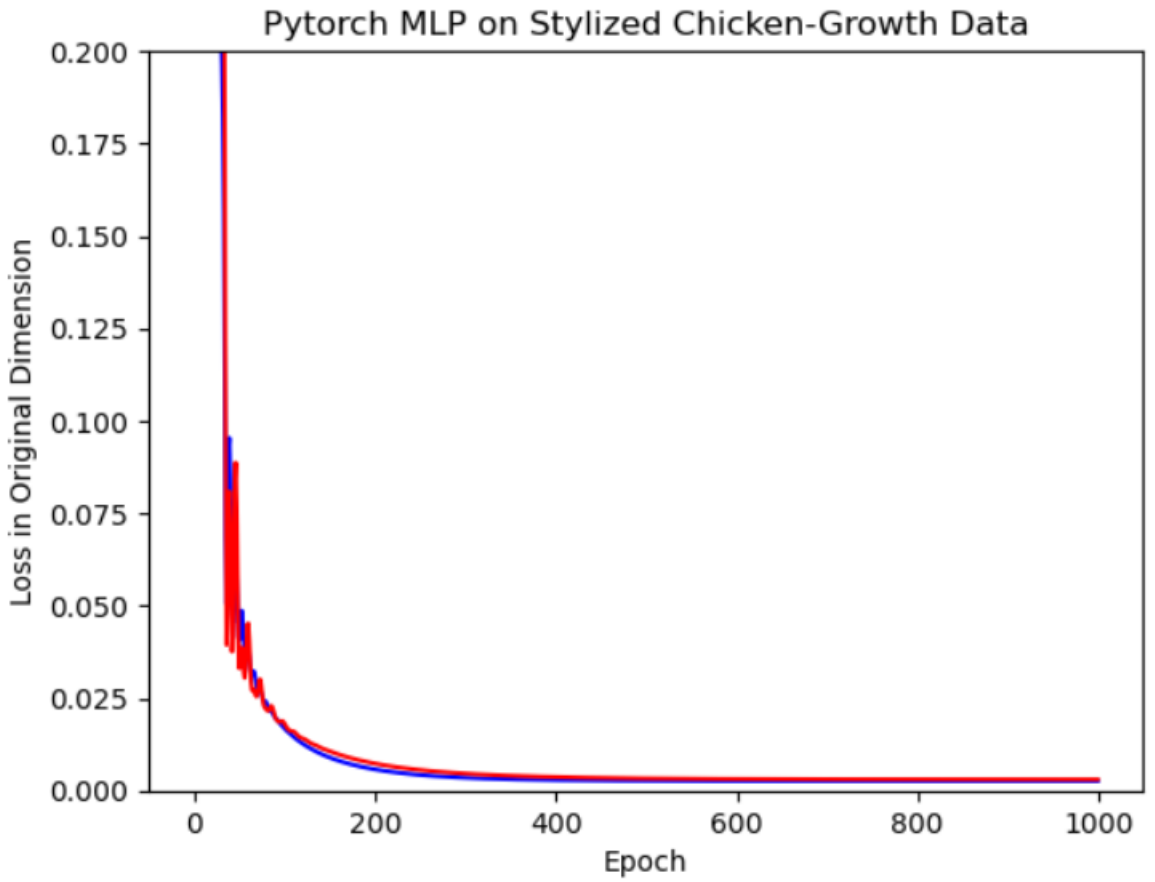
Backprop

Bem.: With torch.no_grad() ist Kontextmanager, assert prüft, ob in diesem Block das Argument auf False steht
Optimizer.step() führt die Gewichtsanapssung aus

Experiment

- Der Fehler entwickelt sich wie gezeigt
- **Welche Stellen im Code sind unklar?**

```
t_ctr = torch.arange(0, n_epoch, step = 1)
t_ctr = t_ctr.clone().detach().unsqueeze(1)
plt.plot(t_ctr, losses_train, 'b') # plotting separately
plt.plot(t_ctr, losses_val, 'r')
plt.ylabel('Loss in Original Dimension')
plt.xlabel('Epoch')
plt.title('Pytorch MLP on Stylized Chicken-Growth Data')
plt.ylim(top = 0.2)
plt.ylim(bottom = 0)
plt.show()
```



Experiment

- Wir können uns die gelernten Modellparameter anschauen

```python
#deep dive into net
print("\n Params of Pytorch mlp_chicken_pt")
for name, param in mlp_chicken_pt.named_parameters():
    print(name, param.shape)
    print(name, param.data.numpy())
print("\nA specific forecast")
print(t_fc[0])#this is the first forecast of net given x[0]
#print(mlp_chicken_pt.bias())
print("\nThe ingredients")
print(train_t_x[0])
print(list(mlp_chicken_pt.parameters())[0])
print(list(mlp_chicken_pt.parameters())[1])
t_by_feet = list(mlp_chicken_pt.parameters())[1] + list(mlp_chicken_pt.parameters())[0] * train_t_x[0]
t_by_feet = t_by_feet.sigmoid()
print("\nAfter cooking we get", t_by_feet)
```

```
 Params of Pytorch mlp_chicken_pt
0.weight torch.Size([1, 1])
0.weight [[0.11273148]]
0.bias torch.Size([1])
0.bias [-3.5769615]
```

- Und einzelne Prognosen nachrechnen

```
A specific forecast
[0.11173666]

The ingredients
tensor([13.3400])
Parameter containing:
tensor([[0.1127]], requires_grad=True)
Parameter containing:
tensor([-3.5770], requires_grad=True)

After cooking we get tensor([[0.1117]], grad_fn=<SigmoidBackward0>)
```

Experiment

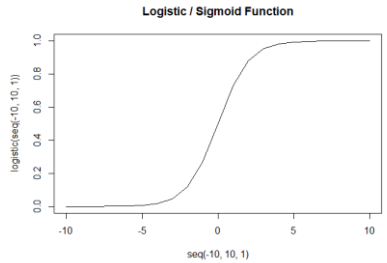
- **Wie beurteilen Sie den Lernerfolg?**

```
t_fc = mlp_chicken_pt(train_t_x).detach().numpy()
t_resi = train_t_y-t_fc
plt.scatter(t_fc, t_resi.data.numpy())
plt.title('Residuals vs Fitted')
plt.show()

t_ctr = torch.arange(0, train_t_x.shape[0], step = 1)
t_ctr = t_ctr.clone().detach().unsqueeze(1)
plt.scatter(t_ctr, t_resi)
plt.title('Residuals vs Index')
plt.show()

#summary stats
print("\n Mean of Resis: ", torch.mean(t_resi).numpy())

print("\n Stdev of Resis vs Stdev of Error: ", torch.std(t_resi).numpy(), s)

print("\n Params of Pytorch mlp_chicken_pt")
for name, param in mlp_chicken_pt.named_parameters():
    print(name, param.shape)
    print(name, param.data.numpy())

print("\n Same order as PyTorch output:", -b0, -b1)
print('\n Training AT HOME needed: %s seconds---' % (time.time() - start_time))
```

```
 Mean of Resis:  -0.0010269269

 Stdev of Resis vs Stdev of Error:  0.051104028 0.05

 Params of Pytorch mlp_chicken_pt
0.weight torch.Size([1, 1])
0.weight [[0.11273148]]
0.bias torch.Size([1])
0.bias [-3.5769615]

 Same order as PyTorch output: -3.5 0.11
```

Erweiterung

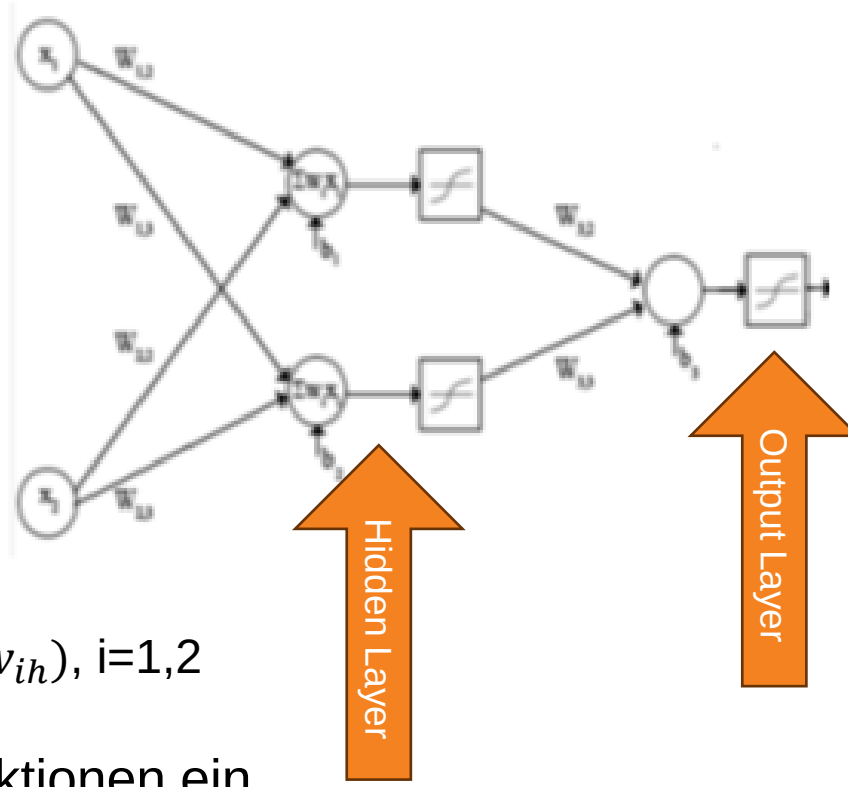
- Die Funktion f(x) kann als Modell **eines** Neurons betrachtet werden. Sobald x einen Schwellwert (Bias) überschreitet, steigt die Aktivität des Neurons von 0 auf 1



- Eine Sprungfunktion wäre als Modell weniger geeignet, da man ableiten können möchte
- Kombiniert man mehrere Neuronen, so gelangt man zum Multi-Layer-Perceptron (MLP)
- Die Kombination dreier Neuronen führt zum einfachst möglichen MLP „Single-Hidden-Layer“ Perceptron wie nachfolgend gezeigt
- Die sigmoide Funktion enthält lineare, konkave und konvexe Abschnitte
- Insofern ist zu vermuten, dass man beliebige y(x) nachbilden kann

Theorie MLP

Multi-Layer-Perceptron – 1 Hidden Layer



$y = \boldsymbol{f}(w_{h1o}$* h1 + w_{h2o}* h2 + $w_{hbo})$ mit $h_i = \boldsymbol{f}(w_{i1}$* x1 + w_{i2}* x2 + $w_{ih})$, i=1,2

- Zwei Inputs x1 und x2 gehen gewichtet in zwei Sigmoidfunktionen ein
- The first layer „is commonly referred to as hidden layer … since its outputs are not observed directly but fed into the output layer“ (Stevens, 2020, 158)
- In jedem „als Quadrat gezeigten“ Neuron wird die Funktion f(.) ausgeführt
- **Wir haben somit eine Hintereinanderschaltung von linearen und nicht-linearen Funktionen**
- Bemerkung: Nur lineare hintereinander bleibt linear!

Theorem & Training

- **Universal Approximation Theorem**

  Jede stetige Funktion y(x) kann durch ein MLP mit einem Hidden Layer approximiert werden

  Jede (auch un-)stetige Funktion y(x) kann durch ein MLP mit zwei Hidden Schichten approximiert werden.

- Das Schätzen der Parameter heisst ab jetzt **Training** des KNN (technisch Training des MLP)
- Im **Training** (->Schätzung) werden die **Residuenquadrate** minimiert, d.h. unsere Zielfunktion ist $(Y-f(X,W))^2$ und wird minimiert
- Je nachdem wie die Gewichte W gewählt werden, erhalten wir eine konkave oder konvexe Funktion. Je nachdem wieviele Hidden Neuronen wir erlauben, erhalten wir recht komplexe Funktionen
- Unser bisheriges Beispiel können wir subsumieren als noch sparsamere Variante des MLP mit 1 Hidden Layer (-> Oberstes Neuron so parametrisieren, dass wir im linearen Bereich sind)

Quellen: Goodfellow et al, 2016, 192, bewiesen in: Hornik, K., Stinchcombe, M., White, H., 1989, “Multilayer Feedforward Networks are Universal Approximators”, Neural Networks, 2, 359-366
Ripley, B. D. (1993): Statistical aspects of neural networks, in: Barndorff-Nielsen, O. E., Jensen, J. L., Kendall, W. S. (Hrsg.): Networks and chaos - Statistical and probabilistic aspects, London: Chapman & Hall Verlag, 1993, S. 40-123
, S. 41 ff und Brause, 1991, 50ff, Russell & Norvig, 2016, S 732.

Theorem & Training

- Das Theorem sagt, dass wir die Funktion im Prinzip lernen können, aber nicht wie und ob wir sie lernen / entdecken
- Die Gewichte werden zufällig initialisiert, dann wird ein Feedforward durchgeführt, der Prognosefehler berechnet und backpropagiert. Die Gewichte werden durch den stoch. Gradientenabstieg (SGD) verbessert.

- **Theorem über SGD**

  Unter schwachen Bedingungen (endliche Varianz der Fehlerfunktion, Beschränktheit n.o. und n.u., etc) kann gezeigt werden, dass die „wahre“ Gewichtsmatrix“ w* mit WS 1 nach unendlich vielen Iterationen i erreicht wird auf dieser Wanderung durch den w-Raum.

  Formal: $P(\lim_{i->inf} w(i) = w^*) = 1$

  Vgl. Brause, R. (1991): Neuronale Netze, Stutttgart: Teubner, S. 64.

- Die gelernte Gewichtsmatrix hat weitere Eigenschaften (w* heisst nachfolgend w Dach):

Theorem & Training

- Aus Lehrbass (2021)
- Assumption 1 hatten wir bereits zuvor y=f()+u

Assumption 2 (Exogeneity)

Due to the linearity of the expectation operator, the following assumption for the distribution of the error term is implied:

$$E(u_t|x_t) = 0 \qquad (4)$$

Assumption 3 (I.I.D. Error Term)

In the sequel, the error terms *u* are assumed to be independently and identically distributed (i.i.d.) with a finite variance > 0.

- Was müssen wir tun, um von Konsistenz ausgehen zu können?
- Kann man auch hier „*H0: Gewicht i = 0*“ testen?

Assumption 4 (Most Parsimonious Architecture)

We need two more assumptions: There are no "redundant inputs" and there are no "irrelevant hidden units" (White, 1992, 105). Now we can state:

Theorem 3 (White)

It can be shown that the weight matrix of the trained MLP $\widehat{W}_n$ is a strongly consistent estimator for *W* (White, 1992, 123).

Again, for an increasing sample size *n* the MLP, represented by its weights $\widehat{W}_n$, converges to the true MLP with probability one. Note that a trained MLP corresponds to an NLS estimator.[13] To conclude I ask which functions by *f(x, W)* can be learned by an MLP.

Theorem 4 (Hornik, Stinchcombe, White)

An MLP can approximate any continuous function arbitrarily well (Hornik, Stinchcombe, White, 1989). It does not need more than a single hidden layer.

Hence, an MLP is a universal approximator, i.e. *f(x, W)* might be any function as long as it is continuous. Note that it might be the case that a huge number of hidden units is required. In terms of layers, one is sufficient. Goodfellow et al. (2016, 192-195) highlight important aspects of this theorem in more detail.

This theorem has a strong consequence for ML: "Any lack of success in applications must arise from inadequate learning, insufficient number of hidden units or the lack of a deterministic relationship between input" *x* and output *y* (White, 1992, 20).

Diagnostik

- Aus Lehrbass (2021)

12 Diagnostic Steps and Common Wisdom

We summarize the steps of the diagnostics and add comments, why these properties of the residuals are of practical value, too.

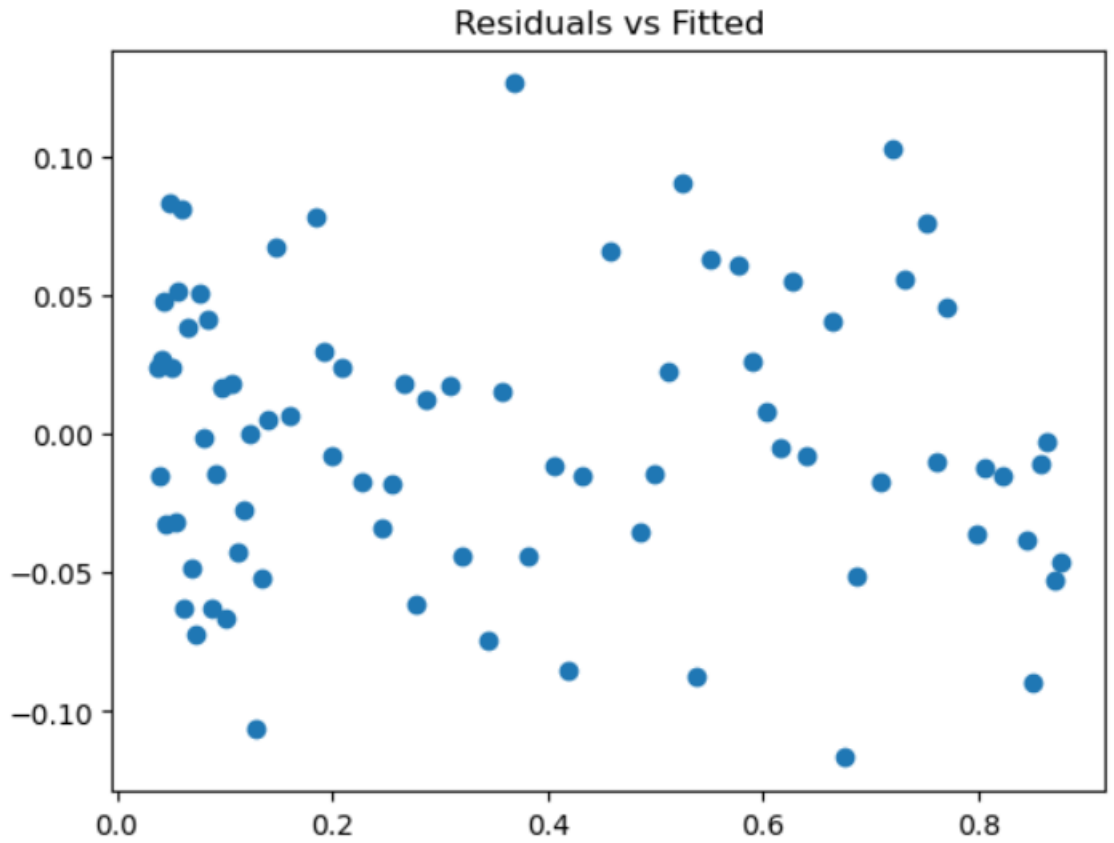
Check	Practical Value
The mean of the residuals is zero	There is no systematic forecasting error
No autocorrelation among residuals	There is no unused structure that I have not yet integrated into the model
No variation in the variance of the residuals	One may assume that each pair *y*, *x* in the sample is equally valid

Table 7: Summary and Practical Value

Insights from this section

Diagnostics encompass the analysis of the residuals, which means "learning from your mistakes". If the diagnostics do not put the assumptions into doubt, one may enjoy the trained MLP as a consistent estimator of the true relationship. This is the main benefit of diagnostics.

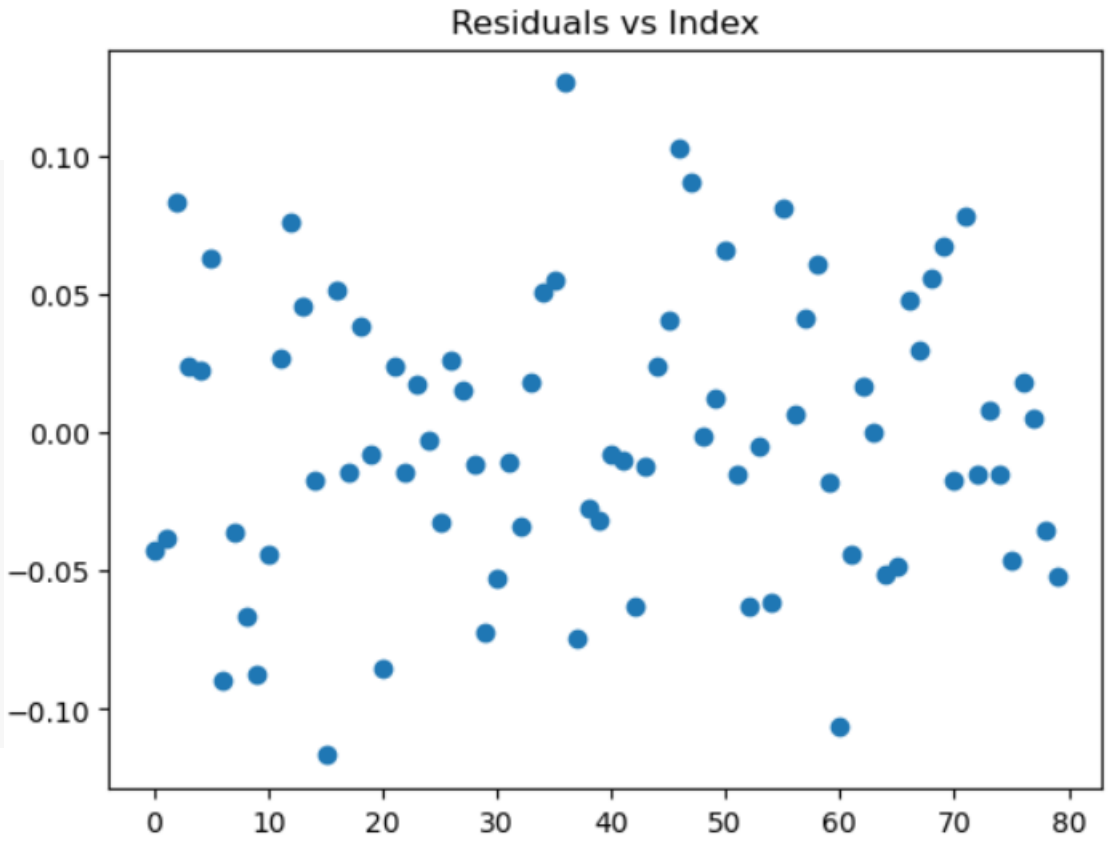
```
Mean of Resis:  -0.0010269269

Stdev of Resis vs Stdev of Error:  0.051104028 0.05
```



Diagnostik

- Coding

```
#diags
#summary stats
print("\n Mean of Resis: ", torch.mean(t_resi).numpy())

print("\n Stdev of Resis vs Stdev of Error: ", torch.std(t_resi).numpy(), s)

t_fc = mlp_chicken_pt(train_t_x).detach().numpy()
t_resi = train_t_y-t_fc
plt.scatter(t_fc, t_resi.data.numpy())
plt.title('Residuals vs Fitted')
plt.show()

t_ctr = torch.arange(0, train_t_x.shape[0], step = 1)
t_ctr = t_ctr.clone().detach().unsqueeze(1)
plt.scatter(t_ctr, t_resi)
plt.title('Residuals vs Index')
plt.show()
```



Experiment

- Aus Lehrbass (2021) **Lernerfolg=Zufall?**

To illustrate this point the sample size is increased and the result from estimation resp. training is shown below. Note that the values for the MLP differ from the entries in the table above because we refer to the MLP trained with RMSProp, which has the highest R^2:

Parameter	Model / Sample Size	b_0	b_1	s
True Value		3.5	-0.11	0.05
Estimated Value	NLR / 100	3.4117	-0.1082	0.0562
Trained Value	MLP / 100	2.0917	-0.0654	0.1023
Estimated Value	NLR / 1,000	3.5394	-0.1111	0.0519
Trained Value	MLP / 1,000	3.5643	-0.1097	0.0532
Estimated Value	NLR / 10,000	3.5135	-0.1104	0.0499
Trained Value	MLP / 10,000	3.5462	-0.1155	0.0548
Estimated Value	NLR / 100,000	3.5065	-0.1102	0.0499
Trained Value	MLP / 100,000	3.5274	-0.1103	0.0499
Estimated Value	NLR / 1,000,000	3.4984	-0.1100	0.0500
Trained Value	MLP / 1,000,000	3.5148	-0.1159	0.0584

Table 5: Result from Estimation Resp. Training

For the NLR, Theorem 2 can be seen in action. The estimates converge to the true *W* for an increasing sample size. Surprisingly, the training results of the

Assumption 4 (Most Parsimonious Architecture)

We need two more assumptions: There are no "redundant inputs" and there are no "irrelevant hidden units" (White, 1992, 105). Now we can state:

Theorem 3 (White)

It can be shown that the weight matrix of the trained MLP $\widehat{W}_n$ is a strongly consistent estimator for *W* (White, 1992, 123).

Again, for an increasing sample size *n* the MLP, represented by its weights $\widehat{W}_n$, converges to the true MLP with probability one. Note that a trained MLP corresponds to an NLS estimator.[13] To conclude I ask which functions by *f(x, W)* can be learned by an MLP.

Theorem 4 (Hornik, Stinchcombe, White)

An MLP can approximate any continuous function arbitrarily well (Hornik, Stinchcombe, White, 1989). It does not need more than a single hidden layer.

Hence, an MLP is a universal approximator, i.e. *f(x, W)* might be any function as long as it is continuous. Note that it might be the case that a huge number of hidden units is required. In terms of layers, one is sufficient. Goodfellow et al. (2016, 192-195) highlight important aspects of this theorem in more detail.

This theorem has a strong consequence for ML: "Any lack of success in applications must arise from inadequate learning, insufficient number of hidden units or the lack of a deterministic relationship between input" *x* and output *y* (White, 1992, 20).

Fallstudie Autopreise

- Aus Lehrbuch von Shmueli et al (2019), Coding **PYTPRA_02_03_LinRegMLP.ipynb**

- **ZIEL: Erklärung der Gebrauchtwagenpreise von Toyota Corollas, möglichst hohes R^2**
- **MITTEL: Lineare Regression & MLP sowie**
- Data: Prices of 1000 used Toyota Corollas, with their specification information

- **Für wen ist dies eine kommerziell relevante Fragestellung?**
- **Welchen Mehrwert / Extra-Service bietet Ihnen ein Fachhändler beim Kauf eines Neuwagens?**
- **Für welche Art Banken ist dies ebenfalls von Interesse?**

Fallstudie Autopreise

Verfügbare Daten

- Price in Euros
- Age in months as of 8/04
- KM (kilometers)
- Fuel Type (diesel, petrol, CNG)
- HP (horsepower)
- Metallic color (1=yes, 0=no)
- Automatic transmission (1=y, 0=no)
- CC (cylinder volume)
- Doors
- Quarterly_Tax (road tax)
- Weight (in kg)

Price	Age	KM	Fuel_Type	HP	Metallic	Automatic	cc	Doors	Quarterly_Tax	Weight
13500	23	46986	Diesel	90	1	0	2000	3	210	1165
13750	23	72937	Diesel	90	1	0	2000	3	210	1165
13950	24	41711	Diesel	90	1	0	2000	3	210	1165
14950	26	48000	Diesel	90	0	0	2000	3	210	1165
13750	30	38500	Diesel	90	0	0	2000	3	210	1170
12950	32	61000	Diesel	90	0	0	2000	3	210	1170
16900	27	94612	Diesel	90	1	0	2000	3	210	1245
18600	30	75889	Diesel	90	1	0	2000	3	210	1245
21500	27	19700	Petrol	192	0	0	1800	3	100	1185
12950	23	71138	Diesel	69	0	0	1900	3	185	1105
20950	25	31461	Petrol	192	0	0	1800	3	100	1185

Fallstudie Autopreise

Datenvorbereitung

```
mySEED = 2023
torch.manual_seed(mySEED) #always first
np.random.seed(mySEED) #make it reproducible

car_df = pd.read_csv('ToyotaCorolla.csv')

# select the suggested variables
selected_var = ['Price', 'Age_08_04', 'KM', 'Fuel_Type', 'HP', 'Automatic', 'Doors', 'Quarterly_Tax',
                'Mfr_Guarantee', 'Guarantee_Period', 'Airco', 'Automatic_airco', 'CD_Player',
                'Powered_Windows', 'Sport_Model', 'Tow_Bar']
car_df = car_df[selected_var]

# convert the categorical data into dummy variables
categorical_var = ['Fuel_Type']
car_df = pd.get_dummies(car_df, columns=['Fuel_Type'], drop_first=True)

# separate out predictors and response variables
X_df = car_df.drop(columns=['Price'])
Y_df = car_df[ ['Price'] ]

# normalize the data, scale each attribute on the input vector X to [0, 1]
scaleInput = MinMaxScaler()
scaleOutput = MinMaxScaler()
X = scaleInput.fit_transform(X_df)
y = scaleOutput.fit_transform(Y_df)

# partition data & establish benchmark model, note that seed is under control
X_train, X_valid, y_train, y_valid = train_test_split(X, y, test_size=0.2, random_state=mySEED)
```

- **Da das MLP im Outputneuron nur den Wertebereich [0, 1] hat, skalieren wir den Output Y**
- **Um nicht in Bereichen der Sigmoidfunktion mit schwachen Gradienten zu sein, skalieren wir X**

Fallstudie Autopreise

Lineare Regression

Neu ist der RESET Test

```
#############################################################################
#   LinReg Statsmodels
#############################################################################

X_train_temp = sm.add_constant(X_train)
car_linreg_sm = sm.OLS(y_train, X_train_temp)#reverse order of args
car_linreg_sm_results = car_linreg_sm.fit(cov_type='HAC',cov_kwds={'maxlags':1})
print(car_linreg_sm_results.summary())

X_fc = X_valid
X_fc = sm.add_constant(X_fc)
y_fc = car_linreg_sm_results.predict(X_fc)#do not overwrite X_valid

print("\nREG: R squared on training data")
print(round(car_linreg_sm_results.rsquared,3))
print("\n")
print("REG: R squared on test data")
print(round(r2_score(y_valid, y_fc),3))

reset = dg.linear_reset(car_linreg_sm_results, power = 4, test_type = 'fitted', use_f = True, )
print("\nRamsey RESET test with H0: Correct specs \n ")
print("F Statistic: ", np.round(reset.fvalue, 4))
print("P Value: ", np.round(reset.pvalue, 4))
#logic: with Rsq close to 100 percent the linear function is in question, not the X
```

Fallstudie Autopreise

Lineare Regression

- **Interpretieren Sie!**

```
REG: R squared on training data
0.894


REG: R squared on test data
0.867

Ramsey RESET test with H0: Correct specs

F Statistic:  94.5828
P Value:  0.0
```

OLS Regression Results

Dep. Variable:	y	R-squared:	0.894
Model:	OLS	Adj. R-squared:	0.892
Method:	Least Squares	F-statistic:	355.4
Date:	Fri, 14 Jul 2023	Prob (F-statistic):	0.00
Time:	14:10:07	Log-Likelihood:	1999.8
No. Observations:	1148	AIC:	-3966.
Df Residuals:	1131	BIC:	-3880.
Df Model:	16		
Covariance Type:	HAC		

	coef	std err	z	P>|z|	[0.025	0.975]
const	0.3158	0.023	13.678	0.000	0.271	0.361
x1	-0.3078	0.009	-33.328	0.000	-0.326	-0.290
x2	-0.1603	0.014	-11.343	0.000	-0.188	-0.133
x3	0.1598	0.024	6.628	0.000	0.113	0.207
x4	0.0190	0.005	3.950	0.000	0.010	0.028
x5	0.0172	0.004	3.951	0.000	0.009	0.026
x6	0.1345	0.035	3.793	0.000	0.065	0.204
x7	0.0071	0.002	2.937	0.003	0.002	0.012
x8	0.0960	0.020	4.886	0.000	0.058	0.135
x9	0.0047	0.003	1.523	0.128	-0.001	0.011
x10	0.1146	0.010	11.189	0.000	0.095	0.135
x11	0.0104	0.004	2.571	0.010	0.002	0.018
x12	0.0165	0.003	5.771	0.000	0.011	0.022
x13	0.0149	0.003	5.358	0.000	0.009	0.020
x14	-0.0112	0.003	-4.346	0.000	-0.016	-0.006
x15	0.0697	0.014	5.115	0.000	0.043	0.096
x16	0.0530	0.017	3.106	0.002	0.020	0.086

Fallstudie Autopreise

MLP

Der RESET Test deutet auf Nicht-Linearitäten hin

Gemäss der bewährten Praktikerregel bestimmen wir die Anzahl der Hidden Neuronen

Die Stipo ist gross genug dafür

Wir setzen den Batch auf 512

Num_workers erstmal auf 0

Welche Stellen im Code sind unklar?

```
#################################################################################
#   Data & Model Prep Pytorch
#################################################################################
#motivated by RESET we use richer model to explore
myHIDDENone = 11 #we have 16 dim input + 1 out = 17, two thirds of 17 as max number of hiddens = 11
myMAX_EPOCHS = 500 #500 so far net is taken from end
myLEARNING_RATE = 0.01
myBATCH_SIZE = 512 #nearly half of data, X_train.shape[0] would be full sample, 32 is min

n_input = X_train.shape[1]
n_out = 1

def func_turn_df_into_t(df_arg):
    t_ret = torch.from_numpy(np.array(df_arg))
    t_ret = t_ret.to(torch.float)
    return t_ret
```

```
t_X_train = func_turn_df_into_t(X_train)
t_y_train = func_turn_df_into_t(y_train)
t_X_valid = func_turn_df_into_t(X_valid)
t_y_valid = func_turn_df_into_t(y_valid)
print(t_X_train.shape,t_y_train.shape,t_X_valid.shape,t_y_valid.shape)

torch_dataset = Data.TensorDataset(t_X_train, t_y_train)
train_loader = Data.DataLoader(
    dataset=torch_dataset,
    batch_size=myBATCH_SIZE,
    shuffle=True, num_workers=0,)#use default
```

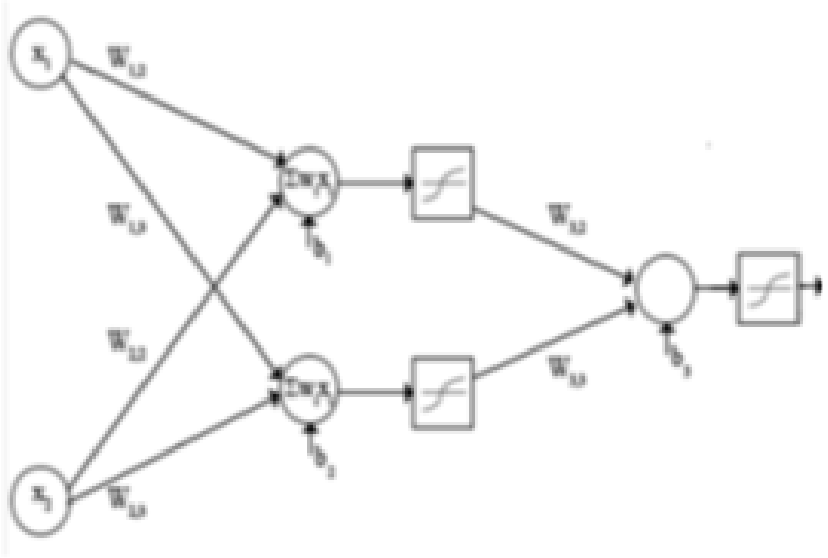
```
torch.Size([1148, 16]) torch.Size([1148, 1]) torch.Size([288, 16]) torch.Size([288, 1])
```

Fallstudie Autopreise

MLP

Welche Stellen im Code sind unklar?

Wo passt die Grafik und wo nicht?



```
#############################################################################

car_mlp_pt = nn.Sequential(nn.Linear(n_input, myHIDDENone),
                      nn.Sigmoid(),#the sublayer has n_hidden outputs (right argument)
                     #this has to be the left argument on the following layer
                      nn.Linear(myHIDDENone, n_out),
                      nn.Sigmoid())#defines the graph
print(car_mlp_pt)
#linear is a+bX, then follows a hidden layer of n_hidden-s sigmoids, lienarly fed into

loss_fn = nn.MSELoss()
optimizer = torch.optim.Adam(car_mlp_pt.parameters(), lr=myLEARNING_RATE)
losses_train = []
losses_val = []
```

Fallstudie Autopreise

MLP – Extended Loop

Welche Stellen im Code sind unklar?

Was ist neu?

```
def generic_training_loop_extended(n_epochs, optimizer, model, loss_fn, train_t_u, val_t_u,train_t_c, val_t_c):
    for epoch in range(1, n_epochs + 1):
        loss_train_avg = []
        for step, (batch_x, batch_y) in enumerate(train_loader):
            b_x = Variable(batch_x)
            b_y = Variable(batch_y)
            train_t_p = model(b_x) #all details are in model
            train_loss = loss_fn(train_t_p, b_y) #and the loss function
            loss_train_avg.append(train_loss.item())
            optimizer.zero_grad()
            train_loss.backward()
            optimizer.step()
        with torch.no_grad():
            val_t_p = model(val_t_u)
            val_loss = loss_fn(val_t_p, val_t_c)
            losses_val.append(val_loss.item())
            assert val_loss.requires_grad == False
        losses_train.append(train_loss.item())
        if epoch <= 10 or epoch % 50 == 0:
            print("Epoch", {epoch}, f"Avg. training loss over all batches: {np.average(loss_train_avg[0:step]):.4f}",
                  f"Validation loss:  {val_loss.item():.4f}",
                  "\n \tLast batch out of", {step}, f"has training loss: {train_loss.item():.4f}")

    return

generic_training_loop_extended(n_epochs = myMAX_EPOCHS, optimizer = optimizer, model = car_mlp_pt,
                      loss_fn = loss_fn , train_t_u = t_X_train, val_t_u = t_X_valid,
                      train_t_c = t_y_train, val_t_c = t_y_valid)
```

Fallstudie Autopreise

MLP

Wie beurteilen Sie das Lernverhalten?

```
Epoch {1} Avg. training loss over all batches: 0.1089 Validation loss:  0.0819
       Last batch out of {2} has training loss: 0.0891
Epoch {2} Avg. training loss over all batches: 0.0765 Validation loss:  0.0552
       Last batch out of {2} has training loss: 0.0563
Epoch {3} Avg. training loss over all batches: 0.0512 Validation loss:  0.0366
       Last batch out of {2} has training loss: 0.0399
Epoch {4} Avg. training loss over all batches: 0.0342 Validation loss:  0.0249
       Last batch out of {2} has training loss: 0.0291
Epoch {5} Avg. training loss over all batches: 0.0247 Validation loss:  0.0187
       Last batch out of {2} has training loss: 0.0166
Epoch {6} Avg. training loss over all batches: 0.0194 Validation loss:  0.0162
       Last batch out of {2} has training loss: 0.0150
Epoch {7} Avg. training loss over all batches: 0.0180 Validation loss:  0.0156
       Last batch out of {2} has training loss: 0.0119
Epoch {8} Avg. training loss over all batches: 0.0165 Validation loss:  0.0157
       Last batch out of {2} has training loss: 0.0222
Epoch {9} Avg. training loss over all batches: 0.0161 Validation loss:  0.0158
       Last batch out of {2} has training loss: 0.0281
Epoch {10} Avg. training loss over all batches: 0.0166 Validation loss:  0.0156
       Last batch out of {2} has training loss: 0.0254
Epoch {50} Avg. training loss over all batches: 0.0057 Validation loss:  0.0054
       Last batch out of {2} has training loss: 0.0045
Epoch {100} Avg. training loss over all batches: 0.0029 Validation loss:  0.0033
       Last batch out of {2} has training loss: 0.0031
Epoch {150} Avg. training loss over all batches: 0.0020 Validation loss:  0.0023
       Last batch out of {2} has training loss: 0.0028
Epoch {200} Avg. training loss over all batches: 0.0016 Validation loss:  0.0018
       Last batch out of {2} has training loss: 0.0025
```

Fallstudie Autopreise

MLP

Wie beurteilen Sie das Lernverhalten?

```
#diags of learning process
print("\n Params of Pytorch Net car_mlp_pt")
for name, param in car_mlp_pt.named_parameters():
    print(name, param.shape)

    #summary stats
t_fc = car_mlp_pt(t_X_train).detach().numpy()
t_resi = t_y_train-t_fc
print("\n Mean of Resis: ", torch.mean(t_resi).numpy())
print("\n Last MSE: ", torch.mean(t_resi*t_resi).numpy())
print("\n Stdev of Resis: ", torch.std(t_resi).numpy())
```

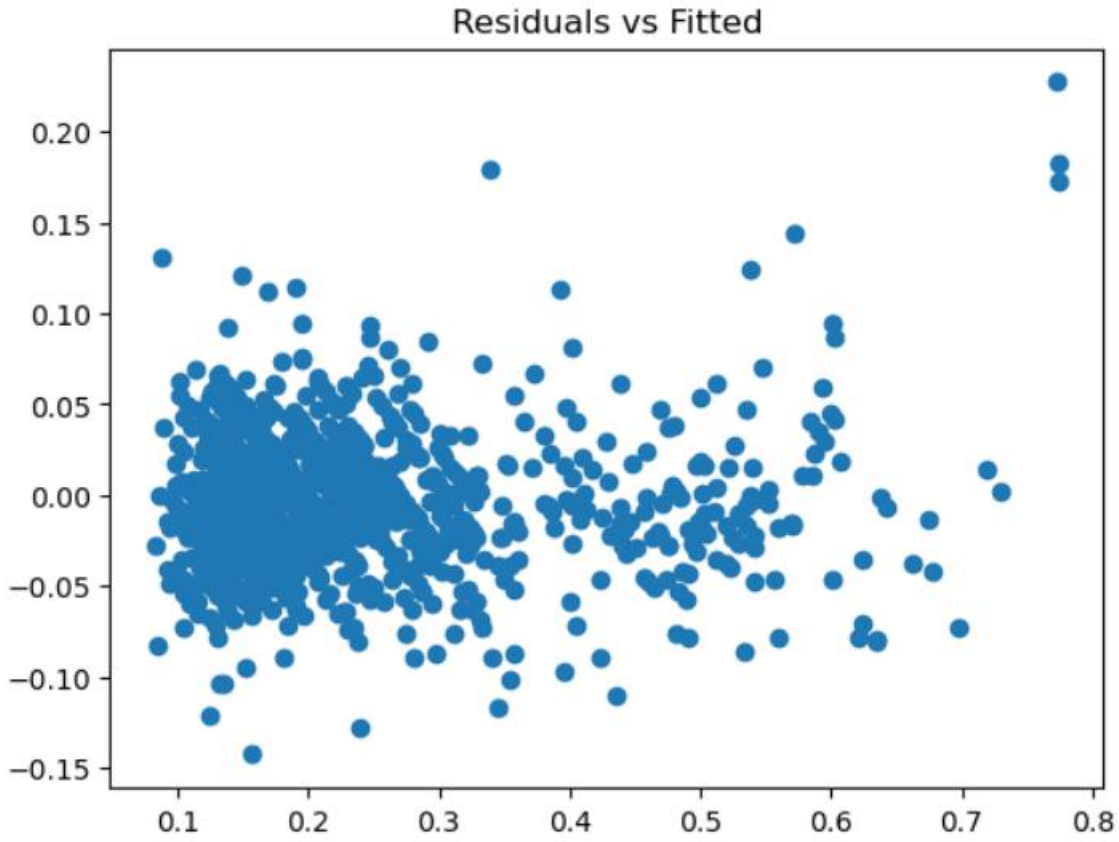
```
 Params of Pytorch Net car_mlp_pt
0.weight torch.Size([11, 16])
0.bias torch.Size([11])
2.weight torch.Size([1, 11])
2.bias torch.Size([1])

 Mean of Resis:  -0.0026585893

 Last MSE:  0.0013630583

 Stdev of Resis:  0.036839824
```

```
Epoch {200} Avg. training loss over all batches: 0.0016 Validation loss:  0.0018
       Last batch out of {2} has training loss: 0.0025
Epoch {250} Avg. training loss over all batches: 0.0015 Validation loss:  0.0017
       Last batch out of {2} has training loss: 0.0022
Epoch {300} Avg. training loss over all batches: 0.0015 Validation loss:  0.0017
       Last batch out of {2} has training loss: 0.0015
Epoch {350} Avg. training loss over all batches: 0.0015 Validation loss:  0.0017
       Last batch out of {2} has training loss: 0.0011
Epoch {400} Avg. training loss over all batches: 0.0013 Validation loss:  0.0017
       Last batch out of {2} has training loss: 0.0022
Epoch {450} Avg. training loss over all batches: 0.0013 Validation loss:  0.0017
       Last batch out of {2} has training loss: 0.0020
Epoch {500} Avg. training loss over all batches: 0.0013 Validation loss:  0.0017
       Last batch out of {2} has training loss: 0.0017
```

Fallstudie Autopreise

MLP

Wie beurteilen Sie die Diagnostik?



Fallstudie Autopreise

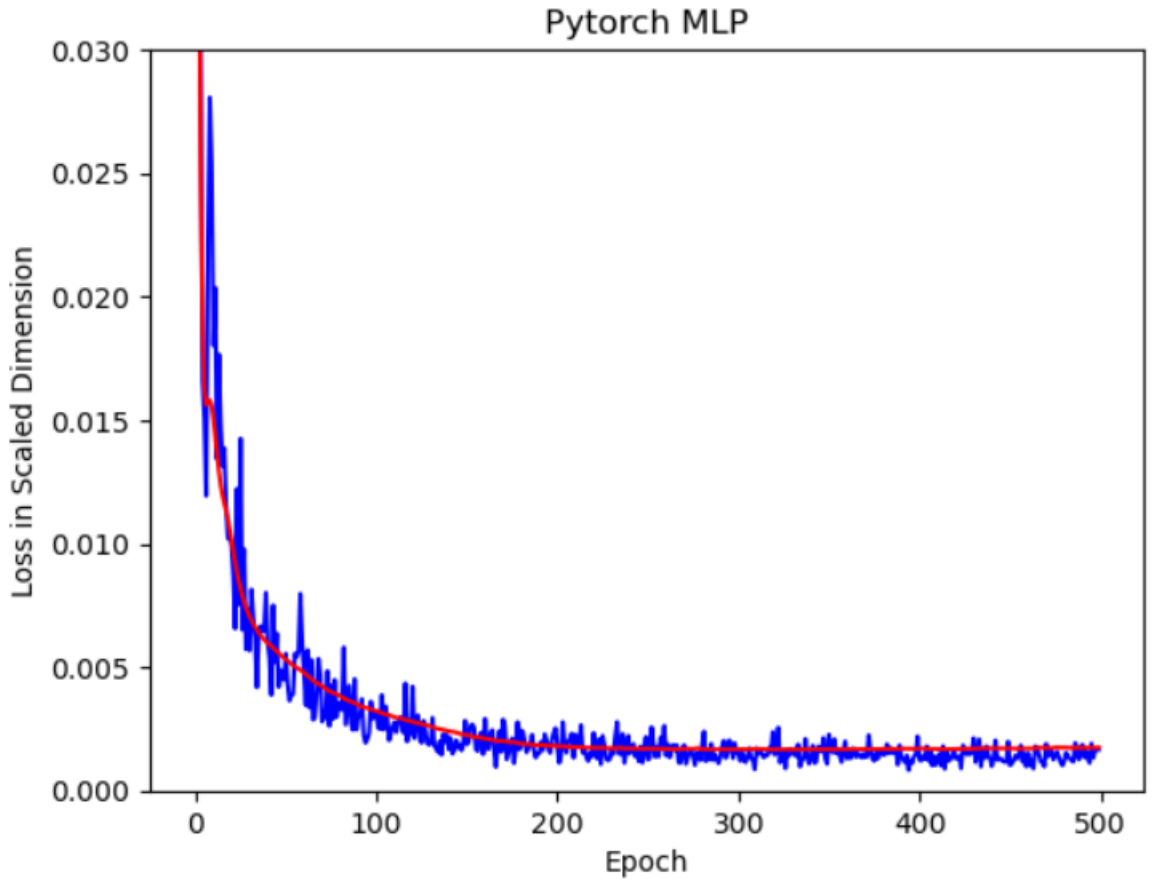
MLP

Wie beurteilen Sie das Lernverhalten?

Hat sich der Einsatz des MLP gelohnt?

```
t_ctr = torch.arange(0, myMAX_EPOCHS, step = 1)
t_ctr = t_ctr.clone().detach().unsqueeze(1)
plt.plot(t_ctr, losses_train, 'b') # plotting separately
plt.plot(t_ctr, losses_val, 'r')
plt.ylabel('Loss in Scaled Dimension')
plt.xlabel('Epoch')
plt.title('Pytorch MLP')
plt.ylim(top = 0.03)
```

```
REG: R squared on training data
0.894
REG: R squared on test data
0.867

MLP: R squared on training data
0.912
MLP: R squared on test data
0.875
```



Fallstudie Autopreise

MLP

Vorher:

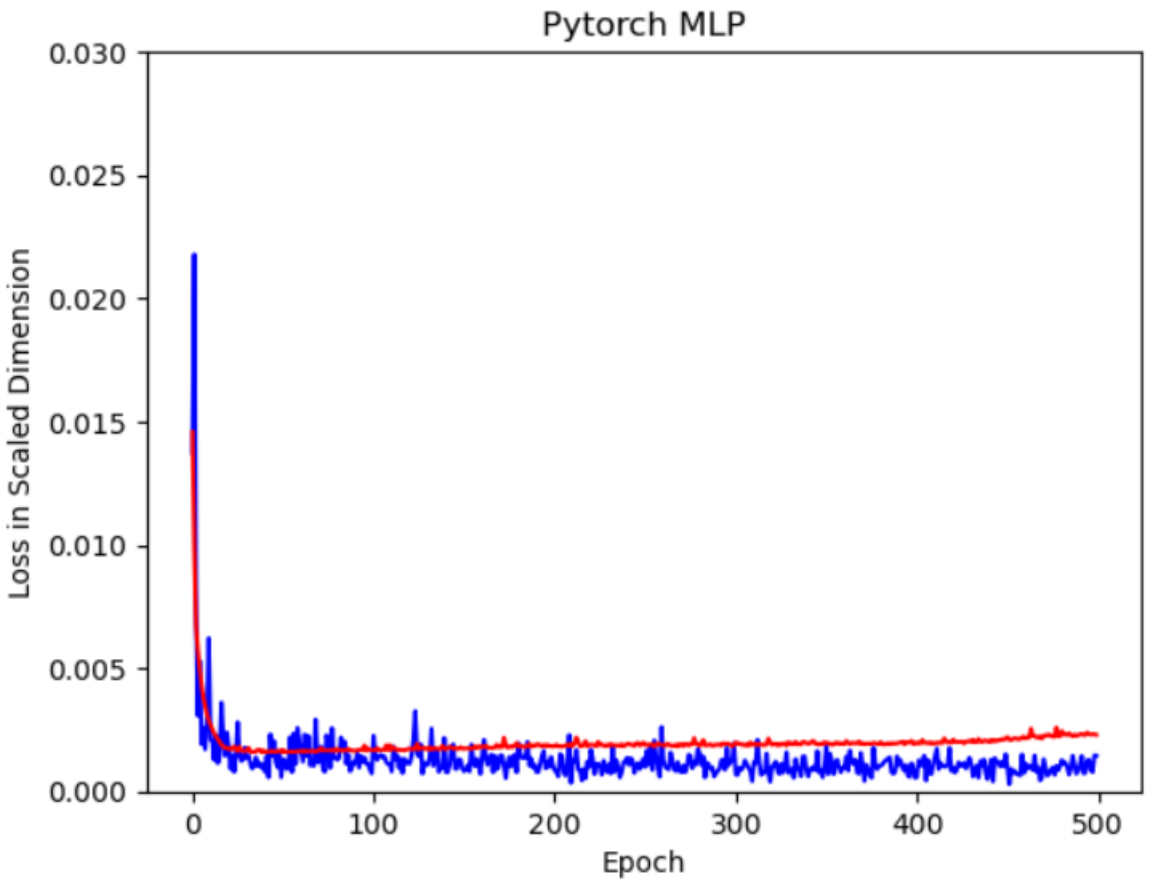
```
REG: R squared on training data
0.894
REG: R squared on test data
0.867

MLP: R squared on training data
0.912
MLP: R squared on test data
0.875
```

Nachher:

```
REG: R squared on training data
0.894
REG: R squared on test data
0.867

MLP: R squared on training data
0.944
MLP: R squared on test data
0.826
```

```
myHIDDENone = 11 #we have 16
myMAX_EPOCHS = 500 #500 so f
myLEARNING_RATE = 0.01
myBATCH_SIZE = 32 #512 max,
```

Wie beurteilen Sie das Lernverhalten mit verkleinertem Batch = 32 (statt 512)?



Hyperparameter Optimierung

- Welche Kombination von Netzarchitektur und Trainingsparametern führt zum Erfolg?

- Bislang Faustregeln und Erfahrung genutzt.

- Was ist mit Random-Search Ansatz, bei dem zufällige Kombinationen von Hyperparametern ausprobiert werden?
- Oder dem Grid-Search Ansatz, bei dem im Rahmen einer festgelegten Intervall- und Schrittgröße jede verfügbare Kombination durchgetestet wird?
- Evolutionäre Optimierung von Hyperparametern (May the winners have children)?

Was spricht gegen obige Ansätze?

Hyperparameter Optimierung

- Scikit unterstützt den Grid Search
- Deshalb trainieren wir mit **Scikit MLP Regressor** erneut das MLP wie gehabt

```
myHIDDENone = 11 #we have
myMAX_EPOCHS = 500 #500 s
myLEARNING_RATE = 0.01
```

```
# Modell initialisieren as in Pytorch
mlp_model = MLPRegressor(max_iter=myMAX_EPOCHS, hidden_layer_sizes=(myHIDDENone), activation='logistic',
                         solver='adam', batch_size = myBATCH_SIZE,
                         shuffle = True, random_state=mySEED,
                         verbose = False)#set true if details matter

print(mlp_model.get_params(mlp_model.get_params))
#if u want full picture of default and other paras used

# Modell trainieren mit default
mlp_model.fit(X_train, y_train.ravel())
y_fc_scikit = mlp_model.predict(X_train)
print("\n\nREG: R squared on train data SCIKIT")
print(round(r2_score(y_train, y_fc_scikit),3))#Pytorch achieved more on training
```

```
{'activation': 'logistic', 'alpha': 0.0001, 'batch_size': 512, 'beta_1': 0.9, 'beta_2': 0.999, 'early_stopping': False, 'epsilo
n': 1e-08, 'hidden_layer_sizes': 11, 'learning_rate': 'constant', 'learning_rate_init': 0.001, 'max_fun': 15000, 'max_iter': 50
0, 'momentum': 0.9, 'n_iter_no_change': 10, 'nesterovs_momentum': True, 'power_t': 0.5, 'random_state': 2023, 'shuffle': True,
'solver': 'adam', 'tol': 0.0001, 'validation_fraction': 0.1, 'verbose': False, 'warm_start': False}


REG: R squared on train data SCIKIT
0.459
```

Was fällt bei der Modellgüte auf im Vgl zu PyTorch MLP?

Hyperparameter Optimierung

- Wir verbessern mit Hilfe von grid search

Welche beiden Hyperparas variieren wir?

Welche könnten wir noch variieren?

```
# Modell initialisieren mit Hyperparameter-Tuning,
##inkludiere Paras, die bei PyTorch genutzt wurden, weil hier scikit
#variiere nur über learning rate and architecture
mlp_model2 = GridSearchCV(mlp_model,
    {
    "hidden_layer_sizes": [(11), (12), (13), (8,6)],
    #"activation":["logistic"], #["identity", "logistic", "tanh", "relu"],
    #"solver":["adam"],#["lbfgs", "sgd", "adam"]
    #"max_iter": [myMAX_EPOCHS],
    "learning_rate_init": [myLEARNING_RATE,(myLEARNING_RATE*10),(myLEARNING_RATE/10)],
    #"random_state": [mySEED],#control this already above
    },
    cv=2, #the default is 5-fold cross validation
    #parameter search uses the score function of the estimator to evaluate a parameter setting.
    #with MLP Regressor it is R2
    verbose = 0 #2 = time and R2 reported
)
mlp_model2.fit(X_train, y_train.ravel()) #this executes the grid search
```

```
# Ergebnisse des Hyperparameter-Tuning
#print(pd.DataFrame(mlp_model2.cv_results_))#print if details wanted

# Hyperparameter nach dem Tuning mit Kreuzvalidierung anzeigen
print("Best Score:", mlp_model2.best_score_)
print("Best Params:", mlp_model2.best_params_)
```

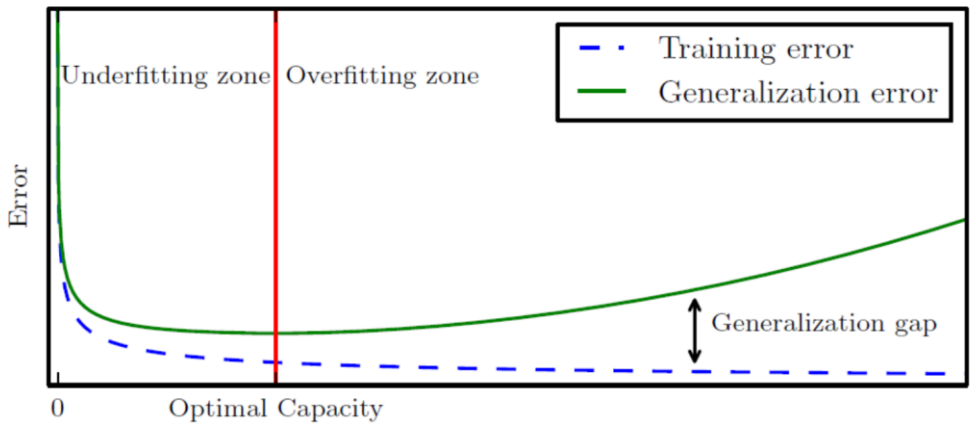
```
Best Score: 0.8507096049415694
Best Params: {'hidden_layer_sizes': (8, 6), 'learning_rate_init': 0.1}
```

Was wurde geändert, um das R^2 zu erhöhen?

Welche Gefahr lauert hinter HPO, wenn man das Lernverhalten nicht beobachtet?

Tipps fürs Training

- Für das Trainieren neuronaler Netze beobachtet man den MSE auf den Trainings- und sogenannten Validierungs- oder Generalisierungsdaten.



- Sobald die grüne Kurve (Goodfellow, 2016, 127) ansteigt vermutet man ein Auswendiglernen von alten Störtermrealisationen **= OVERFITTING**
- Die blau gestrichelte Kurve zeigt, wie der Fehler durch Modellvereinfachung durch die schrittweise verbesserte Bestimmung der Gewichte gemäß Back Propagation reduziert wird. Da sich in den Trainingsdaten nur ein durch den Störterm verrauschter Zusammenhang y(x) befindet, ist eine zu gute Anpassung an die Trainingsdaten gefährlich.
- History doesn’t repeat itself, but it rhymes.

Overfitting & Underfitting

Heutzutage werden MLP mit sehr vielen Schichten trainiert. Die Anzahl der Gewichte geht bis in die Milliarden.
Der universelle Approximator MLP ist stets der Gefahr ausgesetzt, auch Rauschen mitzulernen. Dies nennt man Overfitting.

„When you have big data and many inputs, it is easy to overfit the training data so that your model is being driven by noise that will not be replicated in future observations. That adds errors to your predictions, and it is possible that the overfit model becomes worse than no model at all“ (Taddy, 2019, 70).

In Bildern (Goodfellow, 2016, 110):

Overfitting & Underfitting

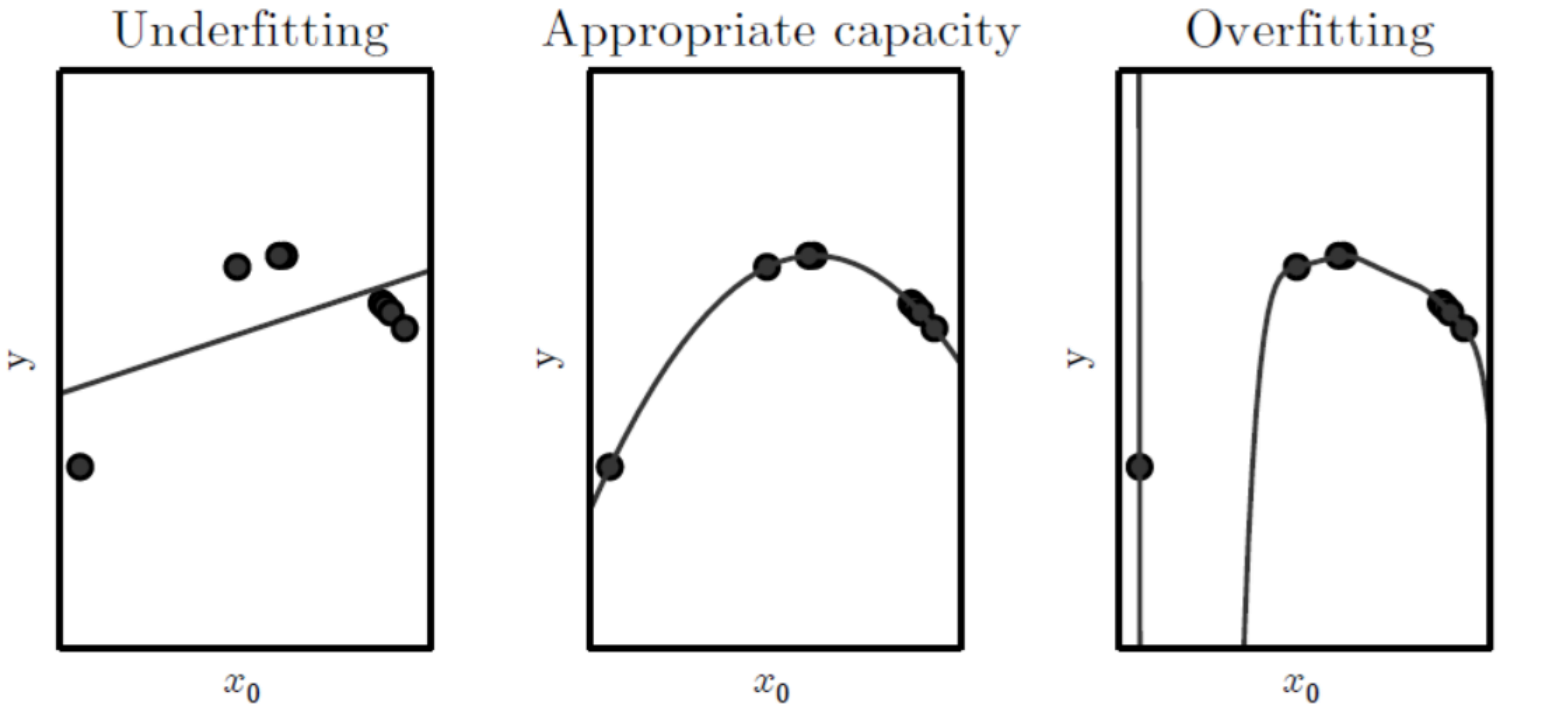


Figure 5.2

(Goodfellow 2016)

Overfitting & Underfitting

- “Three situations, where the model family being trained either (1) excluded the true data generating process—corresponding to underfitting and inducing **bias**, or (2) matched the true data generating process, or (3) included the generating process but also many other possible generating processes—the overfitting regime where **variance** rather than bias dominates the estimation error. The goal of **regularization** is to take a model from the third regime into the second regime.” (Goodfellow, 2016, 222)
- “… the overfitting regime where **variance** rather than **bias** dominates the estimation error” (Goodfellow, 2016, 222)

Overfitting & Underfitting

Underfitting	Overfitting
But similar forecasting quality when applied to new data (e.g. validation, test set)	Fails when applied to new data because (past) random error realizations have been memorized
Less costly to set up / maintain model	High costs, violation of Ockhams´s rule
Incomplete Fitting Quality within training data	Fifference in Fitting Quality between data sets

Wenn Sie wählen müssten, was nehmen Sie?

Interpretation

- In der linearen Regression hatte x1 = Alter des Kfz in Monaten einen klar negativen Einfluss von -0.3
- Und beim MLP?

```
#inspect model
ig = IntegratedGradients(car_mlp_pt)#Specifically, integrated gradients
#are defined as the path intergral of the gradients along the straightline path
#see https://arxiv.org/pdf/1703.01365.pdf
ig_input_tensor = t_X_train
ig_input_tensor.requires_grad_()
attr, delta = ig.attribute(ig_input_tensor,target=0, return_convergence_delta=True)
#target class No 0 because we only have one output dimension
attr = attr.detach().numpy()
np.round(attr,2)
importances = np.mean(attr, axis=0)
feature_names = list(X_df.columns)
for i in range(len(feature_names)):
        print(feature_names[i], ": ", '%.3f'%(importances[i]))


print("\n Params of Pytorch MLP")
for name, param in car_mlp_pt.named_parameters():
    print(name, param.shape)
    print(name, param.data)
```

```
Age_08_04 :  -0.248
KM :  -0.058
HP :  0.057
Automatic :  0.001
Doors :  0.014
Quarterly_Tax :  0.043
Mfr_Guarantee :  0.003
Guarantee_Period :  0.003
Airco :  0.002
Automatic_airco :  0.005
CD_Player :  0.003
Powered_Windows :  0.014
Sport_Model :  0.005
Tow_Bar :  -0.001
Fuel_Type_Diesel :  0.006
Fuel_Type_Petrol :  0.043
```

```
                                                                    03,
        -0.2440, -0.0790, -0.4374])
2.weight torch.Size([1, 11])
2.weight tensor([[ 0.8811,  0.5608, -0.5201, -0.3065, -0.4215, -0.6313, -0.4141, -0.1072,
         -0.2080, -0.3976, -0.5777]])
2.bias torch.Size([1])
2.bias tensor([0.3214])
```

Was enthält diese Tapete?

Interpretation

Ein alternatives Verfahren zur Ermittlung der Bedeutung der Regressoren ist die **Permutation Feature Importance** wie folgt:

Eine zufällige Permutation (der Auspärgungen) eines Regressors wird vorgenommen und somit der unterliegende Realzusammenhang beseitigt. Der Verlust des R^2 auf dem Validierungsdatensatz im Vergleich zur Benchmark drückt aus, wie groß der Einfluss der Variable für die Vorhersagegüte des Modells ist. Der Vorteil liegt darin, dass die Architektur des Modells und die trainierten Gewichte genau gleich bleiben.

Wichtig zu beachten: Beim Training wird der Regressor nicht permutiert! Nur bei der Prediction.

Einfach nur Weglassen? Lasse ich den Regressor weg und trainiere ein neues Modell ohne das Feature, vergleiche ich Äpfel mit Birnen.

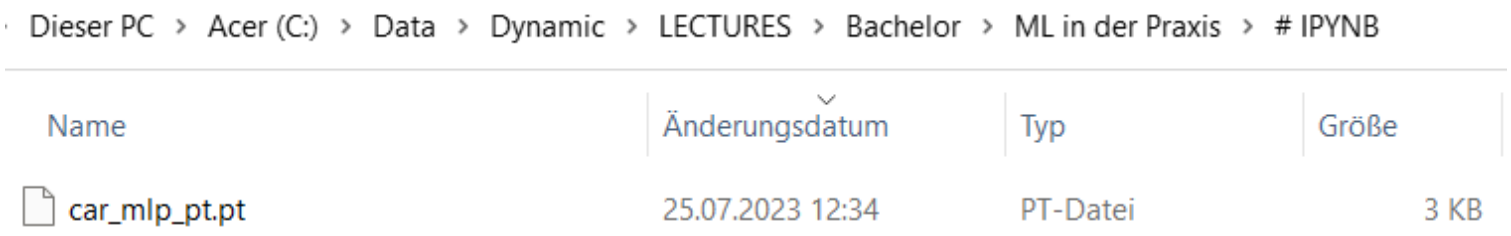
Quelle: https://scikit-learn.org/stable/modules/permutation_importance.html#id2, Zugriff 26 Jan 2024

Was enthält diese Tapete?

Deployment

- Bislang war das Ziel “Erklärung”
- Praktisch sind aber auch Vorhersagen gewünscht
- Wir würden dann noch mehr Wert auf Kreuzvalidierung und kommerzielle Aspekte legen
- HIER skizzieren wir nur die Schritte hin zu einem praktischen Einsatz
- Konkret möchten wir die Modellgewichte auf der Festplatte sichern, um Prognosen zu erzeugen

```
#for deployment we save the weights
#saving in default working dir of Jupyter, serialisation
torch.save(car_mlp_pt.state_dict(), "car_mlp_pt.pt")#use pt and not pth! Latter one collides with python path
#then we can load and maybe write a main-style program for batch exe
loaded_model = torch.load("car_mlp_pt.pt", map_location = 'cpu')#avoid default access to GPU
print("\nXXX DONE XXX")
```



Diese Datei enthält die Modellparameter

Deployment

- Notebook **PYTPRA_02_04_MLPdeploy.ipynb**
- Nun möchten wir die Modellgewichte von der Festplatte laden, um etwa später in einem batch-tauglichen Programm von der Kommandozeile aus Prognosen zu erzeugen
- Wir ändern dabei den Programmierstil etwas
- **Was ist unklar?**

In [1]:
```
#torch
import torch
import os

global_var = 2.0
```

In [2]:
```
def set_target_directory():
    targetDirectory = "C:\Data\Dynamic\LECTURES\Bachelor\ML in der Praxis\# IPYNB"
    os.chdir(targetDirectory)
    return True #returning an explicit true helps you in checking successful execution
#otherwise None is returned, i.e. we make the function fruitful
```

In [3]:
```
def get_model_weight():
    #then we can load and maybe write a main-style program for batch exe
    loaded_model = torch.load("car_mlp_pt.pt", map_location = 'cpu')#avoid default access to GPU
    ret_val = loaded_model.get('0.weight')[0,1].detach().numpy()
    return ret_val
```

Deployment

- Notebook **PYTPRA_02_04_MLPdeploy.ipynb**
- Nun möchten wir die Modellgewichte von der Festplatte laden, um etwa später in einem batch-tauglichen Programm von der Kommandozeile aus Prognosen zu erzeugen
- Wir ändern dabei den Programmierstil etwas
- **Was ist unklar?**

In [4]:
```
def show_model():
    #then we can load and maybe write a main-style program for batch exe
    loaded_model = torch.load("car_mlp_pt.pt", map_location = 'cpu')#avoid default access to GPU
    #diags of learning process
    print("\nParams of Pytorch Net car_mlp_pt")
    print(loaded_model)
    #if u want to adress specific elements might make a list of it - but not useful
    #loaded_model_list = list(loaded_model)
    #print("\nVia List:", loaded_model_list[3])
    print("\nWhat is available:", dir(loaded_model))
    print("\nDirect Access to 2nd weight:", loaded_model.get('0.weight')[0,1].detach().numpy())
    return
```

In [5]:
```
def make_forecast(gv_arg):
    user_input = float(input('\nGib zuletzt gefahrene KM ein! '))

    if user_input <= 0:
        print('Invalider Input! Nur pos Zahlen!!')
    else:
        print("Wertverlust in EUR", gv_arg * user_input)
    return (max (0, gv_arg * user_input))
```

Deployment

- Notebook **PYTPRA_02_04_MLPdeploy.ipynb**
- Wir skizzieren die Prognose nur, um die Technik zu zeigen
- Eigentlich müssten wir einen vollen Feedforward machen mit dem Userinput

```
def make_forecast(gv_arg):
    user_input = float(input('\nGib zuletzt gefahrene KM ein! '))

    if user_input <= 0:
        print('Invalider Input! Nur pos Zahlen!!')
    else:
        print("Wertverlust in EUR", gv_arg * user_input)
    return (max (0, gv_arg * user_input))
```

```
def main():
    print("\nMAIN STARTS")
    set_target_directory()
    global_var = get_model_weight()
    make_forecast(global_var)
    print("\nMAIN DONE")
```

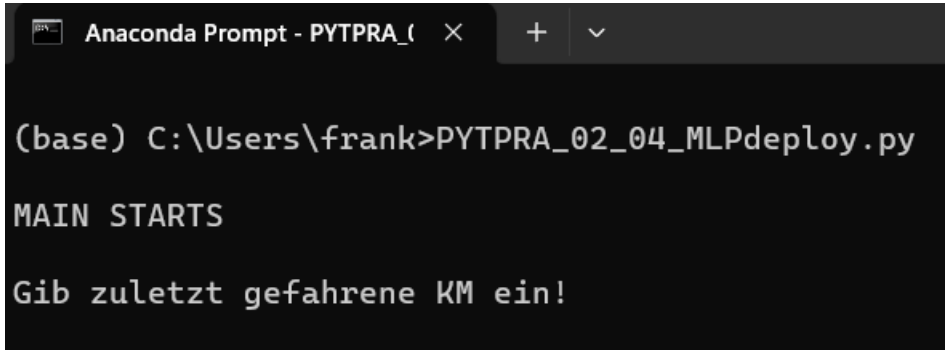
```
main() #this is the main program
```

```
MAIN STARTS

Gib zuletzt gefahrene KM ein! 100
Wertverlust in EUR -48.228615522384644

MAIN DONE
```

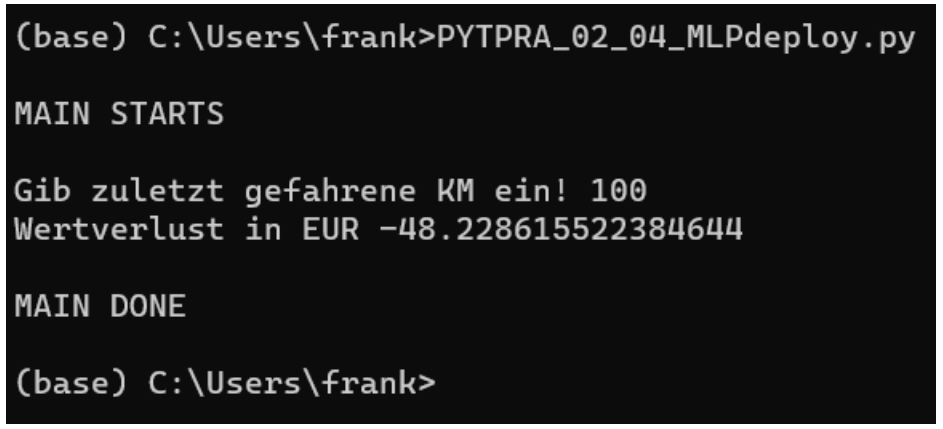
- Wir können dieses Notebook downloaden als py

Deployment

- Das py Skript muss im Arbeitsverzeichnis von Juypter liegen!
- IdR C:\Users ….
- Ausführen von **PYTPRA_02_04_MLPdeploy.py** via Prompt ergibt



- Nach Eingabe von 100 km
- Man kann auch direct hinter dem Prompt Argumente übergeben
- Damit haben wir eine Version, die wir in die batch Verarbeitung aufnehmen können
- Wir stoppen hier aus Zeitgründen



Fehlerquellen

Ergänzend zu ersten Trainingstipps noch ein paar Hinweise auf mögliche Fehlerquellen. Wir beginnen mit **statistischen Fehlerquellen** und reissen danach technische Problemlagen an.

Notebook **PYTPRA_02_05_SpuriousRegMLP**

Zeitreihen sind zeitlich indizierte Beobachtungen, z B zwei Aktienkurse, Tagestemperaturen …

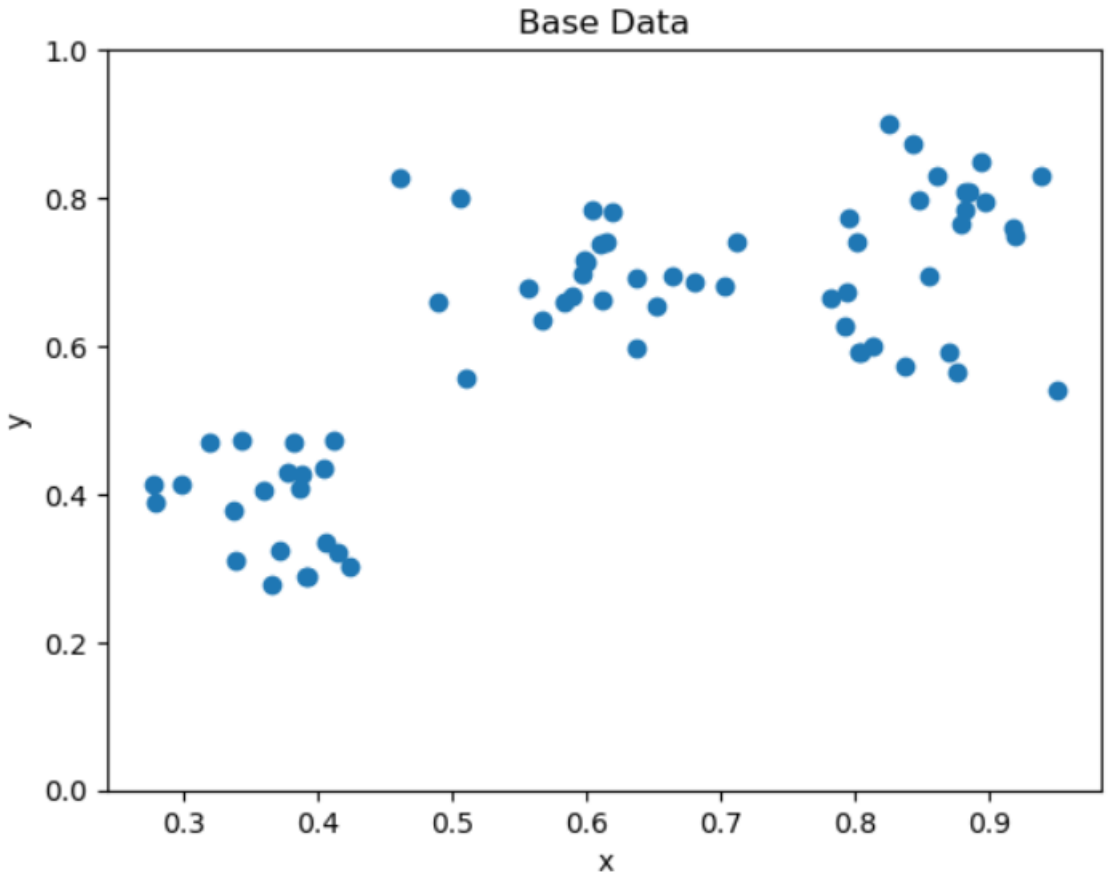
Diesmal betrachten wir zwei Zeitreihen y und x (blau), bei denen wir einen Zusammenhang vermuten:

Die Abzisse ist der Zeitindex!

Was vermuten Sie ist y(x)?

Fehlerquellen

Diesmal betrachten wir zwei Zeitreihen y und x (blau), bei denen wir einen Zusammenhang vermuten:



Die Abzisse ist x!

Was vermuten Sie ist y(x)?

Fehlerquellen

Diesmal betrachten wir zwei Zeitreihen y und x (blau), bei denen wir einen Zusammenhang vermuten:

Interpretieren Sie!

OLS Regression Results

Dep. Variable:	y	R-squared:	0.563
Model:	OLS	Adj. R-squared:	0.556
Method:	Least Squares	F-statistic:	87.46
Date:	Wed, 02 Aug 2023	Prob (F-statistic):	7.85e-14
Time:	11:04:05	Log-Likelihood:	52.850
No. Observations:	70	AIC:	-101.7
Df Residuals:	68	BIC:	-97.20
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	0.2231	0.044	5.089	0.000	0.136	0.311
x1	0.6228	0.067	9.352	0.000	0.490	0.756

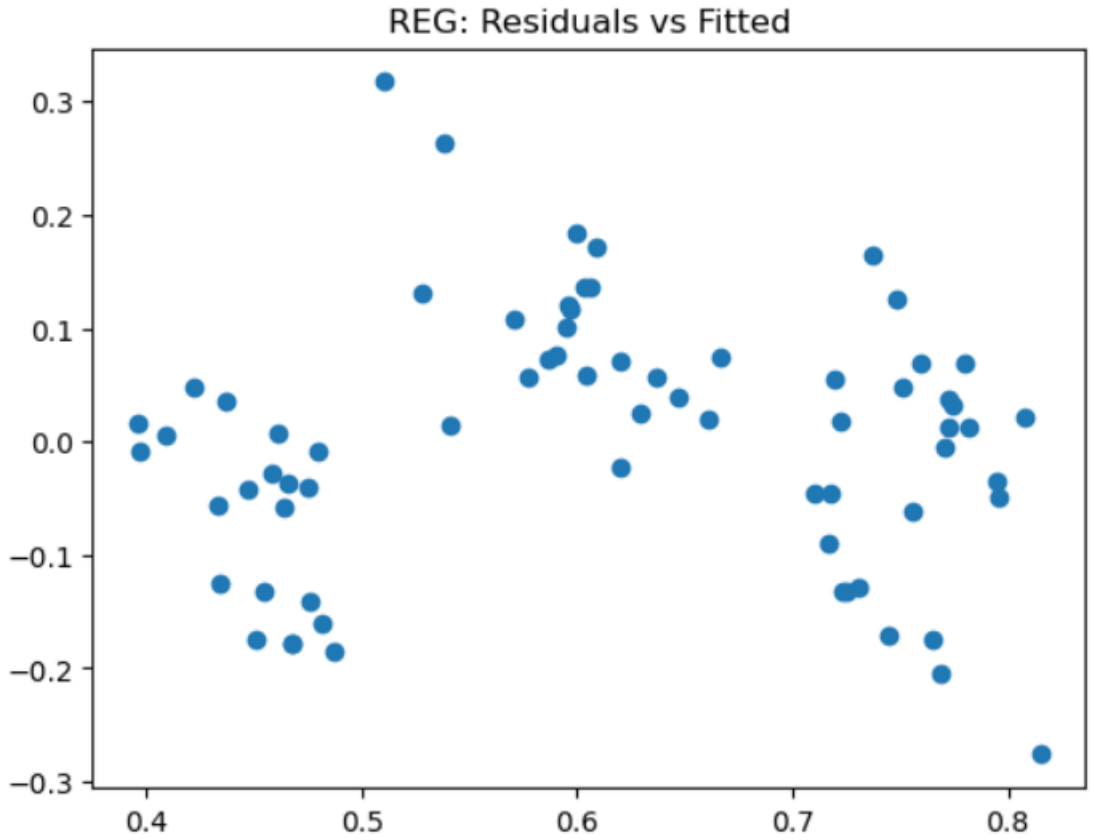
Omnibus:	0.232	Durbin-Watson:	0.300
Prob(Omnibus):	0.891	Jarque-Bera (JB):	0.012
Skew:	0.013	Prob(JB):	0.994
Kurtosis:	3.059	Cond. No.	6.77

Fehlerquellen

Da wir zwei Zeitreihen y und x (betrachten, müssen wir zeitliche (serielle) Korrelationen bei den Residuen befürchten

Deshalb ACF ergänzend zum üblichen Plot

Interpretieren Sie!

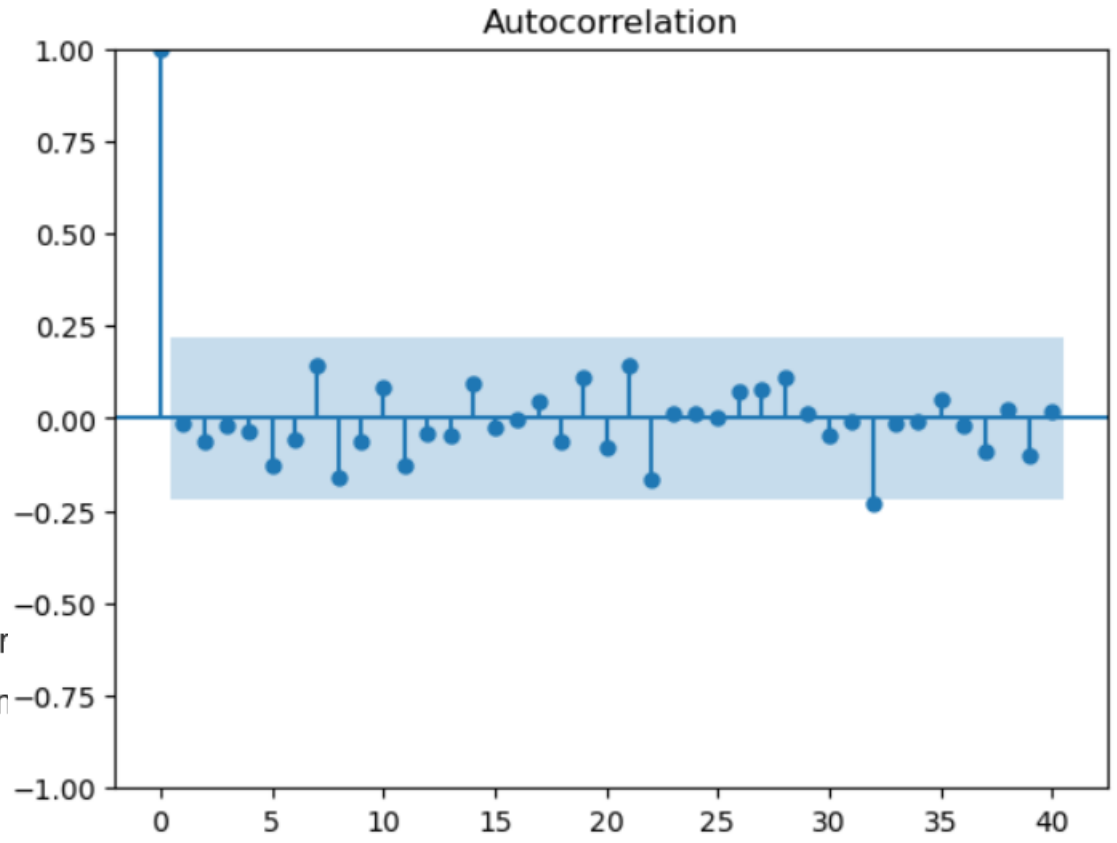


Fehlerquellen

Da wir zwei Zeitreihen y und x (betrachten, müssen wir zeitliche (serielle) Korrelationen bei den Residuen befürchten

Deshalb ACF ergänzend zum üblichen Plot

Interpretieren Sie! Welche (ai) scheint gefährdet?



Wenn für die n Störterme e_i die folgenden Bedingungen gelten:

- (a1) $E[e_i]$=0, i=1,…,n (Im Mittel heben sich Stör ignorierte Faktoren heben
- (a2) e und x sind unabhängig (Exogenität von x)
- (a3) $Var[e_i]$ variiert, d.h. Heteroskedastizität liegt vor
- (a4) Autokorrelation im Störterm geht nach „wenigen“ Lags auf Null
- (a5) Im Falle einer multiplen Regression: Keine lineare Abhängigkeit in x

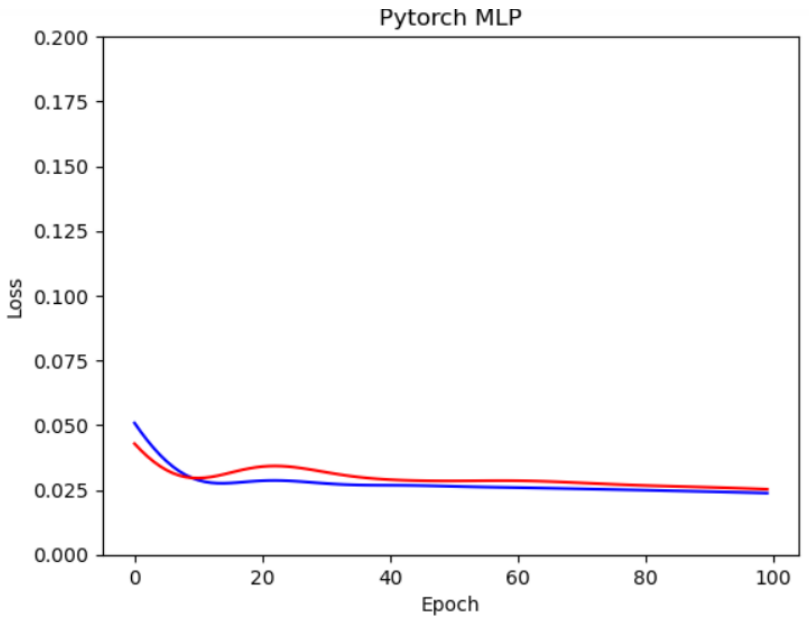
Fehlerquellen

Wir trainieren ein MLP

Interpretieren Sie!

```
n_input = 1
n_hidden = 2
n_out = 1

mlp_pt = nn.Sequential(nn.Linear(n_input, n_hidden),
                       nn.Sigmoid(),#the sublayer has n_hidd
                      #this has to be the left argument on t
                       nn.Linear(n_hidden, n_out),
                       nn.Sigmoid())#defines the graph
#mlp_pt = nn.Sequential(nn.Linear(n_input, n_hidden),
 #                       nn.Sigmoid())#defines the graph
#linear is a+bX, then follows a hidden layer of n_hidden-s
loss_fn = nn.MSELoss()
optimizer = torch.optim.Adam(mlp_pt.parameters(), lr=0.02)
losses_train = []
losses_val = []
start_time = time.time()
```

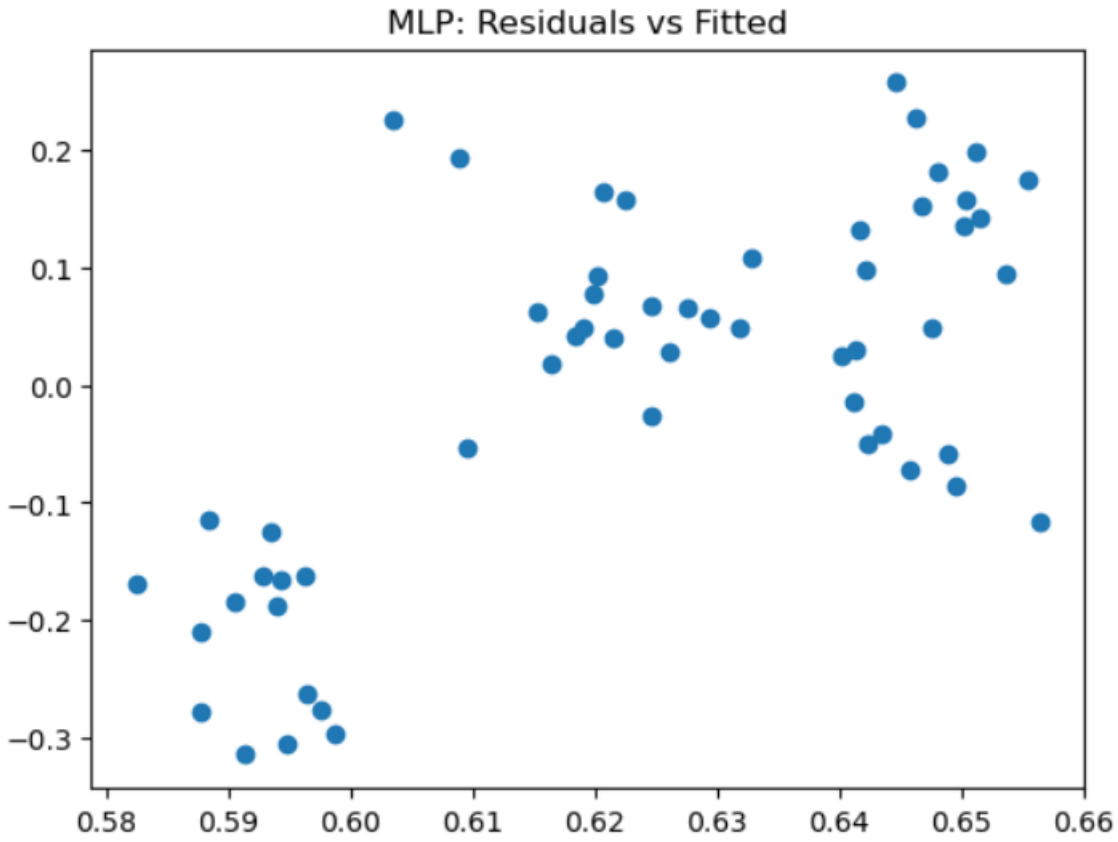
```
Epoch 1, Training loss 0.0508, Validation loss 0.0429
Epoch 2, Training loss 0.0474, Validation loss 0.0403
Epoch 3, Training loss 0.0441, Validation loss 0.0380
Epoch 4, Training loss 0.0412, Validation loss 0.0359
Epoch 5, Training loss 0.0385, Validation loss 0.0341
Epoch 6, Training loss 0.0361, Validation loss 0.0327
Epoch 7, Training loss 0.0340, Validation loss 0.0315
Epoch 8, Training loss 0.0322, Validation loss 0.0306
Epoch 9, Training loss 0.0308, Validation loss 0.0300
Epoch 10, Training loss 0.0296, Validation loss 0.0296
Epoch 100, Training loss 0.0238, Validation loss 0.0253
```



Fehlerquellen

Wir diagnostizieren das MLP

Interpretieren Sie!



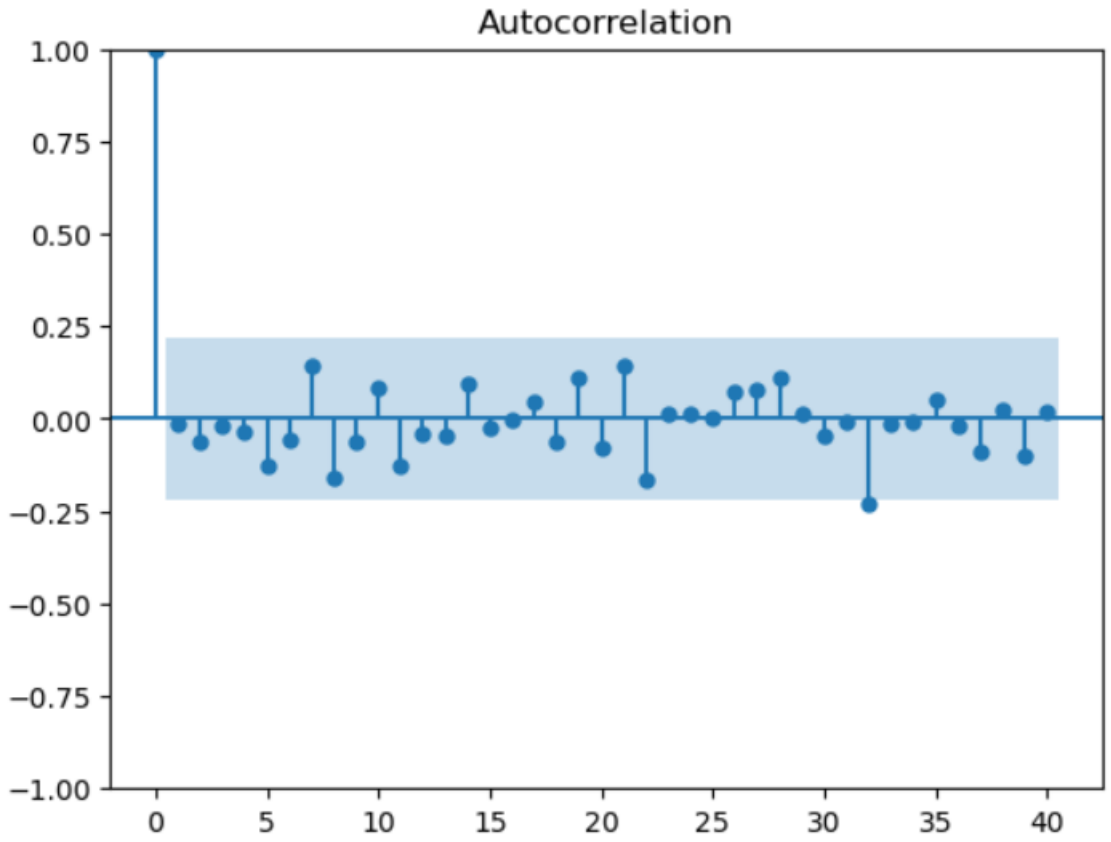
Fehlerquellen

Wir diagnostizieren das MLP

Interpretieren Sie!

Wir vergleichen die Modelle:

```
REG R2 is 0.563
MLP R2 is 0.549
```



Fehlerquellen

Wir gucken hinter die Kulisse

Wieviel hat y mit x zu tun?

Dies ist ein Fall wo „Correlation does not mean causation“ gilt

Take away: Need for domain knowledge and a theory of causation!

```
# set up experimental data
torch.manual_seed(2024)
n=70 #size of training sample

#random walks
s=0.05
eps_mean = torch.zeros(n)
t_a = torch.normal(eps_mean,s)
t_x = torch.cumsum(t_a,dim = 0) + 0.4 * torch.ones(n)
t_b = torch.normal(eps_mean,s)
t_y = torch.cumsum(t_b,dim = 0) + 0.3 * torch.ones(n)

t_ctr = torch.arange(0, n, step = 1)
t_ctr = t_ctr.clone().detach().unsqueeze(1)
plt.plot(t_ctr, t_x, 'b') # plotting separately
plt.plot(t_ctr, t_y, 'r')
plt.ylabel('Time series x and y')
plt.xlabel('Sample')
plt.title('Base Data')
plt.ylim(top = 1)
plt.ylim(bottom = 0)
plt.show()
```