educX

Autor: Andreas Schau - AS

Leitfaden für nachvollziehbare Schritte

1. Kurze Darstellung des Problembereichs / Aufriss des Themas

1.1 Inhaltlich

Kann mittels eines neuronalen Netzes zuverlässig eine Erkennung von Ziffern von 0 – 9 durchgeführt werden, auch wenn diese von schlechten Aufnahmen mit einigen Verzerrungseffekten stammen?
Dies soll mithilfe des „The Street View House Numbers (SVHN) Datasets“ analysiert werden. Darin sind 73257 Ziffern für das Training und 26032 Ziffern zum Testen enthalten.

1.2 Begründung desThemas

Darstellung der Relevanz des Themas?

Eine Ziffernerkennung dieser Form könnte besonders im automatisierten Fahren eine Rolle spielen. In einem von Menschen-Hand unterstütztem Fahrzeug könnte es einen genauer vor der gewünschten Haustür absetzen und bei komplett autonomen Fahrzeugen könnte es die Zielgenauigkeit von Start und Stopp auch verbessern. Obwohl hier wohl eher eine Anwendung in Lagerhäusern und Waren-Transport Robotern die möglichst perfekt zum gewünschten Regal fahren vorstellbar wäre.

Darstellung eines persönlichen Erkenntnisinteresses.

Durch den Siegeszug des Online-Shoppings gibt es immer mehr Nachfrage nach Systemen, die fehlerfrei und kostengünstig eine riesen Auswahl an Artikeln, die in geringer Charge vorhanden sind, an die entsprechenden Interessenten liefern können.
Früher gab es 2-3 Handy-Hüllen im Einzelhandel, heute kann man für tausende von Mobil-Telefon-Modellen und dutzenden verschiedenen Farben genau die Hülle finden, die am besten zu einem passt.
Damit diese Vielfalt bezahlbar bleibt benötigt dieser Industriezweig Systeme, die möglichst viele der monotonen Menschen-ungeeigneten Aufgaben übernimmt.
Hier können automatisierte Fahrsysteme, die ihre Ziel-Kennziffer zuverlässig erkennen eine große Hilfe sein.

2. Nachvollziehbare Schritte

2.1 Der Stand der Forschung / Auswertung der vorhandenen Literatur / Tutorials ...

Der SVHN Datensatz beschreibt sich selbst als MNIST ähnlich.
Im MNIST Datensatz geht es um die Erkennung von handgeschriebenen Ziffern. Im SVHN Datensatz um die Erkennung von Ziffern von Hausnummern. Es wurden also schon ähnliche Neuronale Netze zu einem ähnlichen Zweck untersucht und darunter gibt es einige die eine Genauigkeit von über 99% erreichen.

Der SVHN Datensatz ist im Vergleich zum MNIST Datensatz jedoch nicht so sauber. So gibt es auf vielen der Bilder des SVHN Datensatzes noch rechts oder links die anderen Ziffern, die bei einer mehr-ziffrigen Hausnummer auftreten und diese sind mitunter sehr nah an der zu erkennenden Ziffer wodurch das Netzwerk manchmal fälschlicherweise versucht diese am Rand stehende Ziffer zu erkennen.
Der MNIST Datensatz hat außerdem den Vorteil, dass die Zahlen immer von der „Oben drauf“-Perspektive aufgenommen wurden, wohingegen die SVHN Ziffern häufig auch leicht gedreht, schräg von der Seite oder mit schlechtem Kontrast aufgenommen wurden.

Erklärung der Vorgehensweise im Code

```
    Zelle Ausführen | Oben laufen | Zelle debuggen | Gehe zu [2]
26  # %%
27  # Import required libraries
28
29  import os
30  from collections import Counter
31  from pprint import pprint
32
33  import matplotlib.pyplot as plt
34  import numpy as np
35  import torch
36  import torch.nn as nn
37  import torch.optim as optim
38  import torchvision
39  from scipy.io import loadmat
40  from torch.utils.data import Dataset
41  from torchvision.utils import make_grid
42  from tqdm import tqdm
43
```

Zunächst werden alle nötigen Bibliotheken importiert und wenn nötig mit einem alias versehen. In diesem Script sind es:

- os – Zum setzten der Umgebungsvariable
- collections Counter – Um das Dataset zusammen zu fassen
- pprint – Pretty print um die Daten schöner darzustellen
- matplotlib – Wird für das Plotten von Beispielen und den Kurven verwendet
- numpy – Für die Datenaufarbeitung damit diese mit Torch kompatibel sind
- torch – Die verwendete Neuronales Netz Bibliothek (Version 2.2.0 mit ROCm Support)
- scipy.io – Zum laden der Matlab *.mat Dateien mit den Datensätzen
- tqdm – Zum darstellen eines Fortschritts während des Trainings

```
    Zelle Ausführen | Oben laufen | Zelle debuggen
44  # %%
45  # Configure run
46
47  # Set 780M GPU "cuda instruction set" environment variable
48  HSA_OVERRIDE = "HSA_OVERRIDE_GFX_VERSION"
49  GFX_VERSION = "11.0.0"
50  os.environ[HSA_OVERRIDE] = GFX_VERSION
51  print('\n'.join([
52      f'Set environment variable for current python environment:',
53      f'{HSA_OVERRIDE}={GFX_VERSION}']))
54
55
56  def set_compute_device():
57      return torch.device("cuda" if torch.cuda.is_available() else "cpu")
58
59
60  device = set_compute_device()
61  print(f'Use computing device: {device}')
62
63
64  def configure_run():
65      return {
66          "epoch_count": 50,
67          # since there are classifications of 10 a multiple of that should
68          # hopefully always have roughly equal amounts of each classification
69          # in a batch size
70          "batch_size": 10
71      }
72
73
74  config = configure_run()
75
```

Damit auf meinem Laptop die GPU mit zum berechnen verwendet werden kann folgen im Konfigurations-Schritt einige System-Spezifische Befehle:

- Zeilen 47-53 erstellen Variablen, diese werden dann verwendet um eine Systemvariable für die Ausführung des Scripts zu setzen und es wird ausgegeben, dass diese gesetzt wurde.
  - Diese Variable sagt der Torch Bibliothek, dass meine 780M GPU den Befehlssatz mit der Version 11.0.0 versteht. Ohne diese Variable würde Torch beim Training auf meinem System mit einem Fehler abstürzen.
- Zeilen 56-61 prüfen dann, ob CUDA Kerne zur Verfügung stehen und geben aus, ob diese oder die CPU zur Berechnung verwendet werden.
- Zeilen 64-74 definieren dann ein Dict, welches die Epochen-Anzahl und Batch-Größe für den Lauf enthält.
  - Die Batch-Größe wurde hier bewusst auf genau 10 gesetzt, da es genau 10 unterschiedliche Ziffern/Klassifikationen gibt und in Kombination mit dem folgend beschriebenen Benutzerdefinierten DataLoader erreicht werden sollte, dass pro Forward-Propagation jeweils einmal alle Ziffern für die Loss-Berechnung zusammengefasst wurden. So kann pro Batch-

Backward-Propagation für alle Ziffern gleichzeitig die Gewichte angepasst werden. Dieser Ansatz hat zu einer sehr Muster-Beispielhaften Loss- und Accuracy-Kurve geführt, da so das Dataset maximal balanciert ist und nicht erst ganz viele 1en und dann in späteren Batches vorrangig 2en oder 0en trainiert werden.

```
76
    Zelle Ausführen | Oben laufen | Zelle debuggen
77  # %%
78  # Data preparations
79
80  def load_data():
81      data_train = loadmat('train_32x32.mat')
82      data_test = loadmat('test_32x32.mat')
83
84      return (data_train, data_test)
85
86
87  def shapeshift_input_data(X: np.ndarray) → np.ndarray:
88      # convert shape:
89      # (32, 32, 3, 26032)
90      # to:
91      # (26032, 32, 32, 3)
92      # ⇒ (image_number, Y=row, X=column, RGB)
93      return np.moveaxis(X, -1, 0)
94
95
96  def fix_labels(y: np.ndarray) → np.ndarray:
97      """
98      The SVHN MNIST-like dataset describes the target labels as follows:
99      10 classes, 1 for each digit.
100     Digit '1' has label 1, '9' has label 9 and '0' has label 10.
101     Leaving the 10 in will result in a possible confusion
102     when the dataset is validated later.
103
104     TLDR: Thus the 10 is renamed to 0.
105     """
106     return np.array([0 if yi == 10
107                      else yi
108                      for yi in y])
109
```

In diesem Abschnitt werden einige Funktionen für die folgende Datenaufbereitung definiert:

- load_data() – Lädt via scipy.io's loadmat Funktion die Datensets aus den *.mat Matlab Dateien.
- shapeshift_input_data() – Bekommt die Input-Daten und verändert Ihre Dimensionen so, dass in der ersten Dimension das jeweilige Sample indiziert werden kann.
- fix_labels() – Das Datenset verwendet für die Ziffern mit einer Null das Label mit dem Integer Wert 10.
    - Dies liegt sehr wahrscheinlich daran, dass Matlab eine „One-Starting-Indices"- Sprache ist, bei der Arrays nicht, wie in Python bei 0, sondern bei 1 starten.

- Um Verwirrungen für Python Programmierer zu vermeiden werden in dieser Funktion die Labels mit einer 10 zu Labels mit einer 0 umgeschrieben.

```
Zelle Ausführen | Oben laufen | Zelle debuggen
# %%
# Data preparation

def prepare_data():
    (train, test) = load_data()

    shifted_train_X = shapeshift_input_data(train['X'])
    shifted_test_X = shapeshift_input_data(test['X'])

    fixed_train_y = fix_labels(train['y'].flat)
    fixed_test_y = fix_labels(test['y'].flat)

    print('Digits and their amount in the training dataset:')
    pprint(dict(sorted(Counter(fixed_train_y).items())))

    print('Digits and their amount in the testing dataset:')
    pprint(dict(sorted(Counter(fixed_test_y).items())))

    return ((shifted_train_X, fixed_train_y),
            (shifted_test_X, fixed_test_y))


((train_X, train_y),
 (test_X, test_y)) = prepare_data()

```

Im nächsten Schritt werden diese Funktionen nun wie in einer Pipeline hintereinander ausgeführt:

- Daten laden.
- Eingangsdaten umformen.
- Labels korrigieren.
- Für Trainings- und Test-Datensatz die Anzahl der jeweiligen Ziffern anzeigen. (Im Datensatz kommt die 1 und 2 deutlich häufiger vor als die 9.)
- Den Datensatz Zurückgeben und auf die global verfügbaren Variablen „train_X“ und „train_y“ sowie „test_X“ und „test_y“ setzen.

```
Zelle Ausführen | Oben laufen | Zelle debuggen
# %%
# "Advanced" Custom DataLoader - data preparation

def transformer():
    # alias for readability
    tf = torchvision.transforms

    transform = tf.Compose([
        tf.ToTensor()
    ])
    return transform


transform = transformer()


def create_label_index_dict(labels):
    label_index_dict = {
        digit: [index_in_labels
                for index_in_labels, label in enumerate(labels)
                if label == digit]
        for digit in range(10)
    }
    return label_index_dict

```

Es folgen weitere Funktionen, die für das Benutzerdefinierte Dataset benötigt werden:

- transformer() – Definiert eine Transformer Pipeline für die Eingangsdaten. → Hier müssen nur die Bildwerte von 0-255 auf eine Skala zwischen 0.0 und 1.0, sowie in das Tensor-Format gebracht werden.
- create_label_index_dict() – Bekommt eine Liste aller Labels eines Datensatzes und baut daraus ein Dictionary bei dem die unterschiedlichen Ziffern von 0-9 die Schlüssel sind und die Werte eine Liste aus Indicies an denen diese jeweilige Ziffer im Datensatz vorkommt.

```
162
163 class DigitsDataset(Dataset):
164     def __init__(self, digit_imgs_X, target_labels_y, transform=None):
165         self.digit_imgs_X = digit_imgs_X
166         self.target_labels_y = target_labels_y
167         self.transform = transform
168
169         self.label_index_dict = create_label_index_dict(self.target_labels_y)
170         # holds "pointer" to each labels index advancement
171         self.label_index_itaration = [0] * 10
172
173     def __len__(self):
174         max_count_of_single_digit = max(
175             len(label_indicies)
176             for label_indicies in self.label_index_dict.values())
177         return max_count_of_single_digit * 10
178
179     def __getitem__(self, index):
180         digit = index % 10
181
182         # get current digits index iteration
183         index_of_digit_iteration = self.label_index_itaration[digit]
184
185         # advance iterator
186         self.label_index_itaration[digit] += 1
187         # if iterator advanced out of digit indices arrays bounds reset it to zer
188         if self.label_index_itaration[digit] >= len(self.label_index_dict[digit]):
189             self.label_index_itaration[digit] = 0
190
191         index_of_digit_in_data = self.label_index_dict[digit][index_of_digit_iter
192
193         sample = self.digit_imgs_X[index_of_digit_in_data]
194         target = self.target_labels_y[index_of_digit_in_data]
195
196         if self.transform:
197             sample = self.transform(sample)
198
199         return sample, target
200
```

In diesem Abschnitt wird nun das Benutzerdefinierte Dataset erstellt:

- __init__() – Hier werden die Parameter auf Member der Klasse kopiert. Und die create_label_index_dict() Funktion aufgerufen um das oben beschriebene label_index_dict zu erstellen und ebenfalls als Member der Klasse zu speichern.
- Zeile 171 erstellt noch eine List aus „Pointern“ zum jeweiligen Index im label_index_dict zur jeweiligen Zahl und initialisiert diese Liste mit 10 Nullen.
- __len__() – Gibt die Länge des Datensets zurück. Hier steckt folgende Logik dahinter:
  - Für jede Ziffer gibt es unterschiedlich viele Eingangs-Bilder/Label. (Für die 1 ca. 13000 und für die 9 nur ca. 4000)
  - Aus dem label_index_dict wird die Ziffer mit den meisten Bildern gesucht. (Dürfte die 1 sein)
  - Damit in einem Epoch alle Bilder dieser Ziffer mindestens einmal mit trainiert wurden wird diese

Anzahl dann mit 10, also der Anzahl aller unterschiedlichen Ziffern multipliziert.
    - Die Ziffern, für die weniger Beispiele vorhanden sind, werden am Ende wieder von Vorne angefangen und in dieser Schleife wiederholt.
    - In der __get_item__() Funktion wird dann klar warum.
- __get_item__() – Diese Funktion gibt dem DataLoader jeweils ein Datum aus dem Dataset zurück, woraus dieser dann bei der Batchsize von 10 pro Batch-Loading also 10 Samples abfragt.
    - Damit in jedem Batch jede Ziffer genau ein Mal vorkommt wird der abgefragte Index modulo 10 gerechnet und dann in Zeile 183 der aktuelle „Pointer“ aus für diese Ziffer aus der „Pointer-Liste“ label_index_iteration geholt.
    - Dann wird für diese Ziffer der „Pointer“ um eins erhöht und sozusagen weiter in der Liste nach hinten verschoben.
    - Überläuft er dabei die Anzahl an Indices für diese Ziffer, so wird er wieder auf den Anfang der Liste, also auf Null, zurück gesetzt.
    - Zeile 191 – Dann kann für diese Ziffer mit diesem Index nun der Index den sie im Datensatz hat geholt werden.
    - Sozusagen, für die nächste 5 des Datensatzes siehe nach welchen Index die nächste 5 im Ziffern zu Indices Mapping hat und dann nimm den Index aus diesem Mapping um für diese 5 den Index im Datensatz zu erhalten.
    - Zeile 193-194 – Mit diesem Index kann man nun das Bild und das Label für diese nächste Ziffer auslesen.
    - Falls eine Transform-Funktion angegeben wurde, wird das Bild noch entsprechend Transformiert. (Hier wird es nur in einen Tensor umgewandelt, wobei es auch gleich zwischen 0.0 und 1.0 skaliert wird.)
    - Dann wird das gegebenenfalls transformierte Bild und sein Label zurück gegeben.

Durch diesen Aufwand können bei einer Batch-Größe von 10 nun in jedem Batch alle Ziffern von 0-9 genau ein Mal vorkommen. Und wenn nach 4000 Samples, bzw. 400 Batches, dann dem Datensatz die Neunen „ausgehen“ werden für die folgenden Batches einfach wieder die vom Anfang des Datensatzes genommen. Dadurch ist der Datensatz auf eine gewisse Weise augmentiert, da auf diese Art und Weise bis zum Ende des Epochs immer ganz neu-kombinierte Batches entstehen. Außerdem haben diese noch eine perfekte Balance, da alle Batches jeweils alle Ziffern enthalten und so das Neuronale Netzt nicht erst alle Nullen, dann alle Einsen und so weiter lernt, sondern immer alle Ziffern gleichzeitig „verstehen“ muss.

```
dataset_train = DigitsDataset(train_X,
                              train_y,
                              transform=transformer())
dataset_test = DigitsDataset(test_X,
                             test_y,
                             transform=transformer())


print(f'Length of train dataset: {dataset_train.__len__()}')
print(f'Length of test dataset: {dataset_test.__len__()}')


_tud = torch.utils.data
loader_train = _tud.DataLoader(dataset_train,
                               batch_size=config["batch_size"],
                               shuffle=False)
loader_test = _tud.DataLoader(dataset_test,
                              batch_size=config["batch_size"],
                              shuffle=False)


```

Nun folgt die Instanziierung dieses Datasets für die Trainings- und Test-Daten. Es werden die Längen der augmentierten Datensätze ausgegeben. Und ein Dataloader mit der konfigurierten Batch-Größe von 10 erstellt. Wie schon gesagt, wird in der Trainings- und Validierungs-Schleife der jeweilige DataLoader verwendet um ein Batch an Ziffern aus dem Datensatz zu laden.

```
#
Zelle Ausführen | Oben laufen | Zelle debuggen | Gehe zu [6]
# %% Show a batch of numbers

def show_batch(data_loader):
    for images, _ in data_loader:
        _, ax = plt.subplots(figsize=(12, 12))
        ax.set_xticks([])
        ax.set_yticks([])
        ax.imshow(
            make_grid(images[:config["batch_size"]], nrow=5).permute(1, 2, 0))
        break


show_batch(loader_train)

```

Die show_batch() Funktion stellt dann Beispielhaft das erste Batch an Ziffern-Bildern in einem Diagramm dar.

Beispiel Ausgabe:

```
Zelle Ausführen | Oben laufen | Zelle debuggen | Gehe zu [7]
# %% Model building


class CNNModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=(3, 3), stride=1, padding=1)
        self.act1 = nn.ReLU()
        self.drop1 = nn.Dropout(0.3)

        self.conv2 = nn.Conv2d(32, 32, kernel_size=(3, 3), stride=1, padding=1)
        self.act2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=(2, 2))

        self.flat = nn.Flatten()

        self.fc3 = nn.Linear(8192, 512)
        self.act3 = nn.ReLU()
        self.drop3 = nn.Dropout(0.2)

        self.fc4 = nn.Linear(512, 128)
        self.act4 = nn.ReLU()
        self.drop4 = nn.Dropout(0.1)

        self.fc5 = nn.Linear(128, 32)
        self.act5 = nn.ReLU()
        self.drop5 = nn.Dropout(0.1)

        self.fc6 = nn.Linear(32, 10)
        self.act6 = nn.LogSoftmax()

```

Im nächsten Schritt wird dann mit der Torch-Bibliothek ein Neuronales Netz definiert. Dieses besteht aus zwei Convolution-Schichten, drei Linearen Schichten und einer Klassifizierungs Schicht mit Logaritmischer-

Softmax Aktivierungs-Funktion, welche das Model besonders stark für seinen Loss „bestraft“.

```
269
270     def forward(self, x):
271         # input 3x32x32, output 32x32x32
272         x = self.act1(self.conv1(x))
273         x = self.drop1(x)
274         # input 32x32x32, output 32x32x32
275         x = self.act2(self.conv2(x))
276         # input 32x32x32, output 32x16x16
277         x = self.pool2(x)
278         # input 32x16x16, output 8192
279         x = self.flat(x)
280         # input 8192, output 512
281         x = self.act3(self.fc3(x))
282         x = self.drop3(x)
283         # input 512, output 128
284         x = self.act4(self.fc4(x))
285         x = self.drop4(x)
286         # input 128, output 32
287         x = self.act5(self.fc5(x))
288         x = self.drop5(x)
289         # input 32, output 10
290         x = self.act6(self.fc6(x))
291         return x
292
```

In diesem Modell wird dann auch noch die forward-Propagation Pipeline erstellt, welche im Endeffekt nur die vorher definierten Schichten miteinander verbindet, beziehungsweise deren Datenfluss.

```
294 # create model instance and move it to the GPU
295 model = CNNModel().to(device)
296 print(model)
297
298 # define loss function
299 loss_fn = nn.CrossEntropyLoss()
300 # define optimizer
301 # uncomment one of the two options
302 optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
303 # optimizer = optim.Adam(model.parameters(), lr=0.001)
304
```

Dann wird das Model instanziiert und eine Zusammenfassung ausgegeben. Sowie noch die zu verwendende Loss-Funktion definiert und ein Gewicht-Optimierungs-Algorithmus gesetzt. Hier der SGD, da dieser beim Testen zuverlässigeren Lernerfolg brachte.

```
Zelle Ausführen | Oben laufen | Zelle debuggen | Gehe zu [8]
306  # %%
307  # Out of model accuracy
308
309  def evaluate(model, validation_loader):
310      correct = 0
311      total = 0
312
313      # Set the model to evaluation mode
314      model.eval()
315
316      # Turn off gradients for evaluation
317      with torch.no_grad():
318
319          for images, labels in validation_loader:
320              # Move to device
321              images, labels = images.to(device), labels.to(device)
322
323              # Forward pass
324              output = model(images)
325
326              # Get the index of the max log-probability
327              _, predicted_test = torch.max(output.data,  1)
328
329              total += labels.size(0)
330              correct += (predicted_test == labels).sum().item()
331
332      accuracy = 100 * correct / total
333      return accuracy
334
335
```

Damit während des Lernvorganges erkenntlich ist, wie gut das Model nach der jeweiligen Epoche den Test-Datensatz bewältigen würde, wird hier eine Out-Of-Training-Validierungs-Funktion erstellt. Diese schaltet das Lernverhalten des Netzwerks kurz aus und berechnet für den Test-Datensatz, wie viele Label das Netz aktuell korrekt erkennt.

```
337  # Model training
338
339  train_losses = []  # to store training losses
340  train_accuracy = []  # to store training accuracies
341  test_accuracy = []  # to store test accuracies
342
343  for epoch in range(config["epoch_count"]):
344      epoch_loss = 0.0
345      correct_train = 0
346      total_train = 0
347
348      # Training loop
349      model.train()
350
351      for index, (images, labels) in tqdm(enumerate(loader_train)):
352
353          # Move images and labels to the GPU
354          images, labels = images.to(device), labels.to(device)
355
356          # Forward pass
357          outputs = model(images)
358          loss = loss_fn(outputs, labels)
359
360          # Backward pass and optimize
361          optimizer.zero_grad()
362          loss.backward()
363          optimizer.step()
364
365          epoch_loss += loss.item()
366
367          # Calculate training accuracy
368          _, predicted_train = torch.max(outputs.data, 1)
369          total_train += labels.size(0)
370          correct_train += (predicted_train == labels).sum().item()
371
372      # Calculate average training loss and accuracy for the epoch
373      avg_epoch_loss = epoch_loss / len(loader_train)
374      train_losses.append(avg_epoch_loss)
375
376      # In training accuracy
377      acc_train_perc = 100 * correct_train / total_train
378      train_accuracy.append(acc_train_perc)
379
380      # Out of training accuracy
381      acc_test_perc = evaluate(model, loader_test)
382      test_accuracy.append(acc_test_perc)
383
384      print('\n'.join([
385          '',  # add empty line
386          f'Epoch {epoch +1}:',
387          f'  Average Training Loss: {avg_epoch_loss:>6.2f}',
388          f'  Training Accuracy: {acc_train_perc:>6.2f} %',
389          f'  Testing  Accuracy: {acc_test_perc:>6.2f} %',
390          f''  # add empty line
391      ]))
```

- Es folgt die Trainings-Schleife, welche ein paar leere Listen für die Loss- und Accuracy-Kurven verwendet werden.
- Dann folgt eine Schleife, die die Anzahl der konfigurierten Epochen durchläuft.
- In dieser wird das Model auf trainieren gesetzt.
- Dann folgt eine Schleife über alle Batches, die aus dem loader_train DataLoader geholt werden.
- Das Batch wird an das verwendete Device geschickt. (CPU oder GPU)
- Die Predictions und der Loss wird berechnet.
- Der Optimizer wird resettet und eine Backward-Propagation, sowie ein Optimierungs-Schritt durchgeführt.
- Der Loss für dieses Batch wird in die Loss-Liste aufgenommen.
- Es wird für das Batch berechnet, wie viele Vorhersagen korrekt waren und diese werden ebenfalls in ihre jeweilige Liste aufgenommen.
- Dann folgen nach der Schleife für alle Batches Berechnungen für die Performance der aktuellen Epoche in welcher auch die Out-Of-Training-Accuracy berechnet wird, welche ebenfalls ihrer Liste hinzugefügt wird.
- Zum Schluss werden noch ein paar Performance-Daten der abgeschlossenen Epoche ausgegeben.

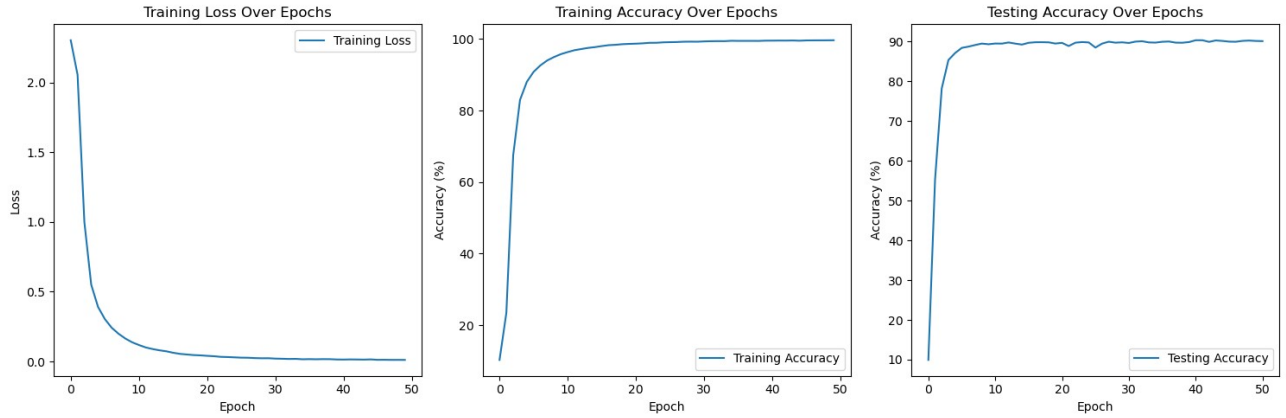
```
Zelle Ausführen | Oben laufen | Zelle debuggen | Gehe zu [10]
# %%
# Model testing (as separate step)


correct_test = 0
total_test = 0

model.eval()

with torch.no_grad():
    for images, labels in loader_test:
        # Move images and labels to the GPU
        images, labels = images.to(device), labels.to(device)

        # Forward pass
        outputs = model(images)

        # Get the index of the max log-probability
        _, predicted_test = torch.max(outputs.data, 1)

        total_test += labels.size(0)
        correct_test += (predicted_test == labels).sum().item()

# Calculate test accuracy
acc_test_perc = 100 * correct_test / total_test
test_accuracy.append(acc_test_perc)

print(f'Test Accuracy: {acc_test_perc:.2f} %')

```

In diesem Abschnitt kann man nach dem Training noch einmal einzeln das trainierte Neuronale Netz mit den gesamten Test-Datensatz durchlaufen lassen um zu prüfen, wie gut es Ziffern aus nicht-trainierten Bildern erkennt.

```
421  # %%
422  # Visualize training history
423
424  plt.figure(figsize=(15, 5))
425
426  plt.subplot(1, 3, 1)
427  plt.plot(train_losses, label='Training Loss')
428  plt.xlabel('Epoch')
429  plt.ylabel('Loss')
430  plt.title('Training Loss Over Epochs')
431  plt.legend()
432
433  plt.subplot(1, 3, 2)
434  plt.plot(train_accuracy, label='Training Accuracy')
435  plt.xlabel('Epoch')
436  plt.ylabel('Accuracy (%)')
437  plt.title('Training Accuracy Over Epochs')
438  plt.legend()
439
440  plt.subplot(1, 3, 3)
441  plt.plot(test_accuracy, label='Testing Accuracy')
442  plt.xlabel('Epoch')
443  plt.ylabel('Accuracy (%)')
444  plt.title('Testing Accuracy Over Epochs')
445  plt.legend()
446
447  plt.tight_layout()
448  plt.show()
```

Es folgt die Ausgabe der Loss-, Training-Accuracy- und Test-Accuracy-Kurven.

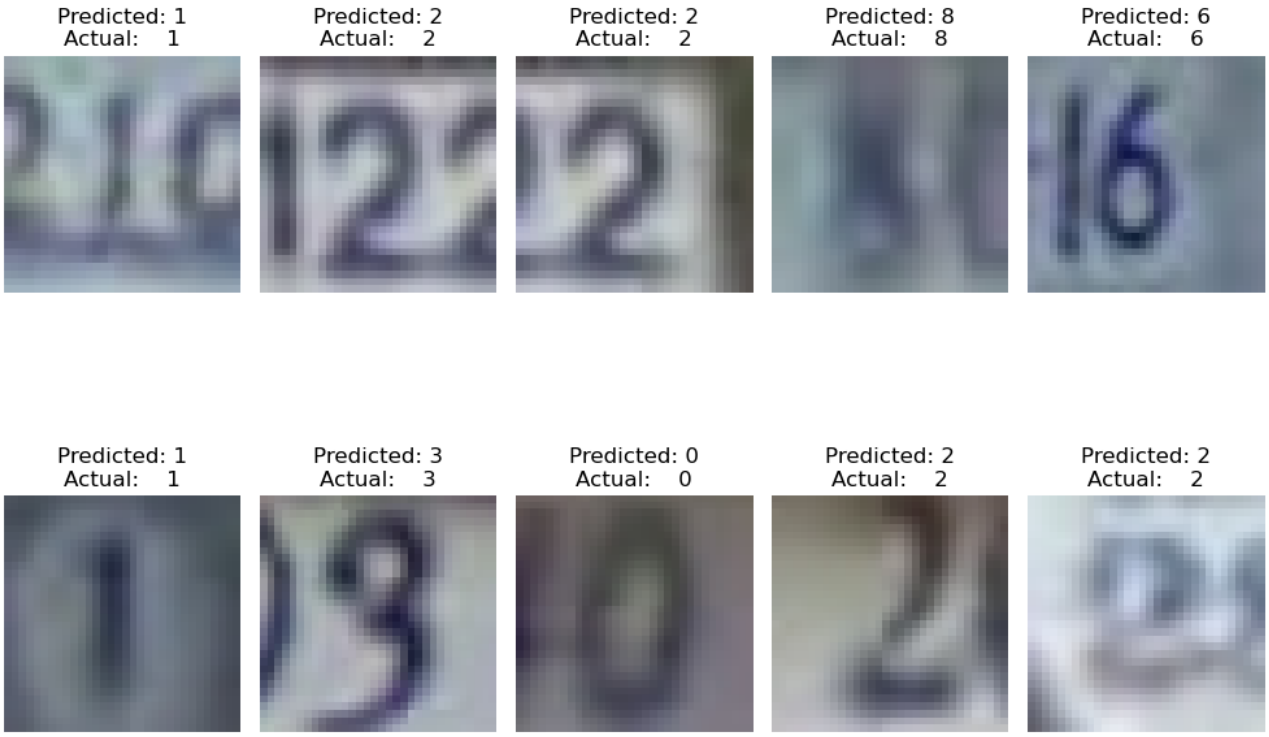
Beispiel Ausgabe:



```
450  # %%
451  # Visualize prediction
452
453  # Create a DataLoader for random sampling from the test dataset
454  random_testloader = torch.utils.data.DataLoader(
455      dataset_test, batch_size=config["batch_size"], shuffle=True
456  )
457
458  # Visualize predictions for batch-size random images
459  model.eval()
460  with torch.no_grad():
461      images, labels = next(iter(random_testloader))
462      images, labels = images.to(device), labels.to(device)
463
464      outputs = model(images)
465      _, predicted = torch.max(outputs, 1)
466
467      plt.figure(figsize=(config["batch_size"], 8))
468      for i in range(config["batch_size"]):
469          plt.subplot(2, 5, i + 1)
470          plt.imshow(images[i].cpu().permute(1, 2, 0).numpy())
471          plt.title('\n'.join([
472              f'Predicted: {predicted[i].item()}',
473              f'Actual:    {labels[i].item()}'
474          ]))
475          plt.axis('off')
476
477      plt.tight_layout()
478      plt.show()
```

Und ganz zum Schluss werden aus dem Test-Datensatz ein paar Bilder zufällig ausgewählt und das Neuronale Netz befragt, welche Ziffern in diesen Bildern zu sehen sein sollten. Aus dem Ergebnis wird dann ein Diagramm mit der Batch-Größe an Ziffern erstellt und welche Ziffern erkannt wurden, sowie welche es laut Label wirklich waren.

Beispiel Ausgabe:



2.6 Ergebnisse

Mit dem erstellten und trainierten Neuronalen Netzt kann man nun mit einer Genauigkeit von bis zu 90% Ziffern in Bildern erkennen, selbst wenn diese noch andere Ziffern enthalten oder von schlechter Qualität sind.
Dieses Netzwerk könnte damit als Ausgangspunkt weiterer Experimente dienen, bei denen automatisiert Ziffern via Kamera erkannt werden müssen.

2.7 Ausblick

Das hier erstellte Neuronale Netz ist noch von relativ geringer Größe und Komplexität und erreicht dennoch schon 90% Genauigkeit. Bei weiteren Untersuchungen könnte man mit Größeren und tieferen Netzen noch höhere Genauigkeiten erreichen.
Je nach Anwengunsfall könnte man so zum Beispiel autonom Steuernde Roboter mit diesem Netz ausstatten und diese zuverlässiger durch ein Lager navigieren lassen oder es könnte auch in der Postleitzahl-Erkennung von automatischen Brief-Sortierungs-Anlagen Verwendung finden.