Machine Learning 2

Projektarbeit

Timon Spichtinger

27. Juni 2025

1 Einleitung

Ziel dieser Projektarbeit ist die Entwicklung eines Deep-Learning-Klassifikators für handgeschriebene Zeichen aus dem EMNIST-ByClass-Datensatz. Klassifiziert werden 36 Klassen bestehend aus Buchstaben (A–M, a–m) und Ziffern (0–9). Zunächst wird ein Transfer-Learning-Ansatz mit einem bestehenden CNN umgesetzt und optimiert. Anschließend wird ein modularer Klassifikator entwickelt, der zusätzlich zwischen Großbuchstaben, Kleinbuchstaben und Zahlen unterscheidet.

Inhaltsverzeichnis

2 Datensatz

Mit der `get_emnist_test_train()` Funktion werden 6000 Bilder von jedem der Symbole extrahiert. Bei Bedarf werden weitere Bilder augmentiert. Dabei wurde Rauschen eingefügt und kleine Rechtecke entfernt, um Overfitting zu vermeiden.

3 Klassifikator

Vorbereitung

Zuerst muss das Device gewählt werden. CUDA ansonsten CPU(nicht zu empfehlen) Danach werden jeweil Test- und Traindatensatz in Tensordataset-Objekte gepackt. (Bild als X und Label als y)

Hyperparameter-Optimierung mit Optuna:

Mit einer Optuna-Studie werden die besten Hyperparameter für das Modell gefunden. Das Ziel ist es die Validierungsgenauigkeit zu maximieren. Ich habe Batch-Größe, Lernrate, Momentum, Step Size und Gamma für den Learning Rate Scheduler untersucht. Da ich mit dem Scheduleransatz schon gut erfolg hatte, hab ich ihn beibehalten. Am ende konnten sehr gute Parameter Gefunden werden und ein Accuracy von 87

Modellarchitektur

Ein modifiziertes ResNet18 mit `CrossEntropyLoss`, SGD + Momentum und `StepLR` als Scheduler. Als Modell verwende ich ein ResNet18. Das ist ein relativ tiefes neuronales Netz mit 18 Schichten, das ich für die 36 Klassen aus dem EMNIST angepasst habe. ResNet ist praktisch, weil es durch die Residual-Verbindungen auch bei vielen Epochen stabil bleibt und nicht so schnell das Vanishing Gradient Problem kriegt.

Als Loss-Funktion nehme ich CrossEntropyLoss – Standard bei Klassifikationsaufgaben. Optimiert wird mit SGD (Stochastic Gradient Descent) und Momentum, damit das Training etwas schneller und stabiler läuft. Zusätzlich hab ich wieder den StepLR-Scheduler verwendet – er verringert die Lernrate nach einer gewissen Zeit schrittweise, was bei mir in früheren Projekten schon gute Ergebnisse gebracht hat.

Training

Das Training läuft in einem klassischen Loop ab. Vor jeder Epoche wird das Modell in den Trainingsmodus versetzt, dann werden die Daten batchweise durch das Netzgeschickt. Ablauf pro Batch: Gradienten werden zurückgesetzt Vorwärtsdurchlauf: Modell macht Vorhersage Loss wird berechnet Rückwärtsdurchlauf: Gradienten werden berechnet Optimizer aktualisiert die Gewichte

Dazu hab ich auch Early Stopping eingebaut – wenn sich die Validierungsgenauigkeit über mehrere Epochen nicht verbessert, wird das Training automatisch gestoppt, um Overfitting zu vermeiden. Nach jeder Epoche wird das Modell auf dem Testset evaluiert (Loss Accuracy).

Evaluation

Genauigkeit, Konfusionsmatrix, AUC-Wert zur Bewertung. Schwierige Fälle: z.B. “1” vs “l”.

4 Modularer Klassifikator

Architektur

- **TM1**: ResNet18 für Zeichenklassifikation (36 Klassen)
- **TM2**: Eigenes CNN für Typklassifikation (Zahl, Großbuchstabe, Kleinbuchstabe)

Kombination

Die Outputs beider Module werden multipliziert. Damit wird der Zeichenoutput durch den Typ-Output gewichtet.

TM2 - Codeauszug

```
class TypeClassifier(nn.Module):
    def __init__(self, dropout_rate=0.3):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(1, 32, 3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(2),

            nn.Conv2d(32, 64, 3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(64 * 7 * 7, 128),
            nn.ReLU(),
            nn.Dropout(dropout_rate),
            nn.Linear(128, 3)  # 3 Typen
        )
```

ModularClassifier - Codeauszug

```
class ModularClassifier(nn.Module):
    def __init__(self, tm1, tm2, class_type_map):
        super().__init__()
        self.tm1 = tm1
        self.tm2 = tm2
        self.register_buffer('class_type_map', torch.tensor(
   class_type_map, dtype=torch.long))

        # Trainierbarer "Kombinationskopf"
```

```
        self.combination_head = nn.Sequential(
            nn.Linear(36, 64),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(64, 36)  # zur ck auf die Klassenzahl
        )

    def forward(self, x):
        out_cls = self.tm1(x)
        out_type = self.tm2(x)

        probs_cls = F.softmax(out_cls, dim=1)
        probs_type = F.softmax(out_type, dim=1)

        class_type_weights = probs_type[:, self.class_type_map]
        combined = probs_cls * class_type_weights

        out = self.combination_head(combined)
        return out
```

Training und Optimierung

Gemeinsames Training von TM1 und TM2, Tuning via Optuna: `batch_size`, `lr`, `momentum`, `step_size`, `gamma`.

Ergebnisse

Das Ergebnis war sehr schlecht bei 10

5 Fazit

Ich bin mit meinem Klassifikator aus 1.2 Sehr zufrieden, da er nichtn nur bei 87Leider konnte ich die letzte aufgabe nicht richtig beenden.

6 Quellen

- https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html
- https://pytorch.org/docs/stable/generated/torch.nn.Module.html
- https://optuna.org/
- https://discuss.pytorch.org/
- https://github.com/pytorch/vision/blob/main/torchvision/models/resnet.py
- https://www.youtube.com/watch?v=EMXfZB8FVUA&list=PLqnslRFeH2UrcDBWF5mfPGpqQDSta6V
- https://chatgpt.com/