Model został wytrenowany na przygotowanych danych i przetestowany pod kątem skuteczności klasyfikacji 36 klas znaków.

Zdecydowaliśmy się na użycie CNN, a wybrana architektura zawiera:

* 3 warstwy konwolucyjne z aktywację ReLU i poolingiem
* Warstwę dropout
* 2 w pełni połączone warstwy (fc1 i fc2)
* Funkcję aktywacji softmax (CrossEntropyLoss)

Model ASLClassifier składa się z funkcji:

* Conv2d – do ekstracji obrazów (krawędzie -> kształty -> gesty)
* MaxPool2d – zmniejszenie rozmiaru obrazka celem zmniejszenie liczby operacji
* Dropout – wyłączenie losowo neuronów (zapobiega przeuczeniu modelu)
* Linear – klasyfikator
* import torch
* import torch.nn as nn
* import torch.nn.functional as F
* class ASLClassifier(nn.Module):
* def \_\_init\_\_(self, num\_classes=36):
* super(ASLClassifier, self).\_\_init\_\_()
* # Konwolucja
* self.conv1 = nn.Conv2d(3, 32, kernel\_size=3, padding=1) # Pierwszy etap - znajduje linie krawędzie
* self.conv2 = nn.Conv2d(32, 64, kernel\_size=3, padding=1) # Drugi etap - znajduje kształy np. palce, kontury dłoni
* self.pool = nn.MaxPool2d(2, 2) # Zmniejsza rozdzielczość, rozmiar obrazu na pół
* self.conv3 = nn.Conv2d(64, 128, kernel\_size=3, padding=1) # Trzeci etap - znajduje całe gesty np. litera A, B, C
* self.dropout = nn.Dropout(0.3) # Zmniejsza liczbę neuronów o 30%
* self.fc1 = nn.Linear(128 \* 28 \* 28, 512)  # zakładając wejście 224x224, po spłaszczeniu łączy w 512 neuronów
* self.fc2 = nn.Linear(512, num\_classes) # Po jednym na klasę
* def forward(self, x):
* x = self.pool(F.relu(self.conv1(x)))  # 112x112
* x = self.pool(F.relu(self.conv2(x)))  # 56x56
* x = self.pool(F.relu(self.conv3(x)))  # 28x28
* x = x.view(x.size(0), -1)
* '''
* To jest flatten – spłaszcza dane przed podaniem do warstwy liniowej (fc1).
* Potrzebne, bo nn.Linear() nie umie działać na obrazach 3D (czyli [batch\_size, channels, H, W]).
* '''
* x = self.dropout(F.relu(self.fc1(x)))
* x = self.fc2(x)
* return x

Trenowanie modelu:

* CrossEntropyLoss – funkcja strat dla klasyfikacji wieloklasowej
* Adam – optymalizator implementujący algorytm **Adama**
* Wyznaczenie dokładności po każdej epoce
* import torch
* import torch.nn as nn
* import torch.optim as optim
* import time
* def train\_model(model, train\_loader, val\_loader, epochs=10, lr=0.001, device="cpu"):
* start\_time = time.time()
* model.to(device) # Przenieisenie modelu na cpu/gpu
* criterion = nn.CrossEntropyLoss() # Funkcja kosztu
* optimizer = optim.Adam(model.parameters(), lr=lr) # optymalizator
* '''
* Adam - szybki i adaptacyjny optymalizator
* '''
* for epoch in range(epochs):
* model.train() # ustawia model w tryb treningowy
* running\_loss = 0.0 # Do śledzenia błędu
* correct = 0 # Do śledznia dokładności
* total = 0   # ---------||------------
* for inputs, labels in train\_loader:
* inputs, labels = inputs.to(device), labels.to(device)
* optimizer.zero\_grad() # Zerowanie gradientu, przed nowym batchem trzeba zawsze wyzerować
* outputs = model(inputs) # Przepuszacznie danych przez sieć
* loss = criterion(outputs, labels) # obliczenie straty
* loss.backward() # Obliczanie gradientów
* optimizer.step() # Aktualizacja wag
* running\_loss += loss.item() # loss to tensor a item wyciaga wartość, sumuje strate z kazdego batcha
* \_, predicted = outputs.max(1) # zwraca przewidywanie
* total += labels.size(0) # labels.size(0) to batch size
* correct += predicted.eq(labels).sum().item() # sprawdza czy dobrze trafiło, sum liczy ile trafień
* train\_acc = 100. \* correct / total
* print(f"Epoch {epoch+1} | Loss: {running\_loss:.4f} | Accuracy: {train\_acc:.2f}%")
* evaluate(model, val\_loader, device)
* total\_time = time.time() - start\_time
* print(f"\nTraining complete, elapsed time: {total\_time:.2f} seconds")
* def evaluate(model, data\_loader, device="cpu"):
* model.eval() # tryb testowy
* correct = 0
* total = 0
* with torch.no\_grad(): # wyłącza śledzenie gradientów
* for inputs, labels in data\_loader: # to samo co w treningu
* inputs, labels = inputs.to(device), labels.to(device)
* outputs = model(inputs)
* \_, predicted = outputs.max(1)
* total += labels.size(0)
* correct += predicted.eq(labels).sum().item()
* acc = 100. \* correct / total
* print(f"Validation Accuracy: {acc:.2f}%")

main.py:

* Ładuje dane
* Inicjalizuje model
* Uruchamia trening na CPU lub GPU (w zależności co jest dostępne, ale jeśli są oba to wybiera gpu, tu cuda)

A screenshot of a computer

AI-generated content may be incorrect.