SI100B Project Face Detection and emotion classification

Lecture 4

Objectives

- Implement the forward propagation by pytorch.
- Try to train the model.

1 Back propagation

1.1 Network output

We have three labels in our emotion network: happy, sad and neutral.
In a three-class classification network, the output layer typically has three neurons (or units) corresponding to the three classes. This meaning the network produces a 3-dimensional output vector, such as $[o_1, o_2, o_3]$. Each component corresponds to the predicted probability or score for one of the three classes.
The labels are usually represented as integers (e.g., 0, 1, 2) or as one-hot encoded vectors. One-hot representation: Class 0 is $[1, 0, 0]$, class 1 is $[0, 1, 0]$, and class 2 is $[0, 0, 1]$.

Figure 1: Output

We can use a formula to represent our network:

$$[o_1, o_2, o_3] = f(Image)$$

1.2 Loss Function

During training, a loss function like categorical cross-entropy is used to compare the predicted probabilities with the true labels.

Following is a common loss function called cross-entropy:

$$L_{CE} = -\sum_{i=1}^{C} y_i log(\hat{y}_i) \tag{1}$$

The happy case in 1: $L = -(1 \times log(0.7) + 0 \times log(0.2) + 0 \times log(0.1)) = -log(0.7)$
The neutral case in 1: $L = -(0 \times log(0.2) + 1 \times log(0.6) + 0 \times log(0.2)) = -log(0.6)$
The sad case in 1: $L = -(0 \times log(0.1) + 0 \times log(0.1) + 1 \times log(0.8)) = -log(0.8)$

The optimization aim is minimize $\sum_{all} \mathcal{L}_{\mathcal{CE}}$.

1.3 Predictions

At inference time, the network predicts the class with the highest probability:

$$PredictedClass = \arg\max([o_1, o_2, o_3])$$

After the previous lesson, you may have noticed that the output scores are quite abstract and not directly probabilities. Typically, we apply a function such as softmax to convert the output vector into probability values, ensuring that the sum across all classes equals 1. This way, we can directly interpret the result as the probability of each class. The Softmax function (also known as the normalized exponential function) is typically used as the final activation function in a neural network designed for multi-class classification.
For a given input vector o, the Softmax output for the i-th element, denoted $\sigma(o)_i$, is calculated as:

$$\sigma(o)_i = \frac{e^{o_i}}{\sum_{j=1}^{K} e^{o_j}}$$

2 PyTorch Basics

2.1 The Core: Tensors

All data manipulation in PyTorch is done using **Tensors**, which are multi-dimensional arrays optimized for GPU acceleration.

- **Concept:** Multi-dimensional array (like NumPy, but with GPU support).
- **API:** `torch.Tensor`, `torch.randn()`, `torch.zeros()`

2.2 The Neural Network Module: `torch.nn`

The `torch.nn` module contains the classes and functions for defining the neural network architecture. All layers inherit from `nn.Module`.

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

2.3 Key CNN Components (Layers)

2.3.1 Convolutional Layer

Performs feature extraction by sliding a kernel (filter) across the input.

- **API:** `nn.Conv2d(in_channels, out_channels, kernel_size)`
- **Purpose:** Extracts spatial hierarchies of features (edges, textures, etc.).

2.3.2 Activation Function: Leaky ReLU

Introduces non-linearity to the network, enabling it to learn complex mappings. Leaky ReLU prevents "dying neurons."

- **API:** `nn.LeakyReLU(negative_slope=0.01)`
- **Formula:**
$$f(x) = \max(0, x) + \alpha \cdot \min(0, x)$$
where α is the `negative_slope`.

2.3.3 Pooling Layer (Max Pooling)

Reduces the spatial dimensions (height and width) of the feature map, reducing computation and increasing invariance to input shifts.

- **API:** `nn.MaxPool2d(kernel_size)`
- **Purpose:** Downsampling the feature map.

2.3.4 Flattening

Converts the multi-dimensional feature map output from the convolutional layers into a one-dimensional vector, preparing it for the Fully Connected layers.

- **API:** `torch.flatten(x, start_dim=1)`
- **Note:** `start_dim=1` preserves the batch dimension (dimension 0).

2.3.5 Fully Connected Layer (Linear Layer)

A standard layer where every input is connected to every output. Performs the final classification based on the extracted features.

- **API:** `nn.Linear(in_features, out_features)`
- **Purpose:** Mapping the features to the final output class scores.

3 Training Components

1. **Loss Function (Criterion):** Measures the error (`nn.CrossEntropyLoss`).
2. **Optimizer:** Updates the model weights based on the loss gradient (`torch.optim.Adam`).
3. **Data Loader:** Manages batching and shuffling of the dataset.

4 Try to train a model

A convolutional neural network can be viewed as a function f whose output is the classification result and input is the image.
The training process aims to optimize the parameters of the function f.

$$\arg\min_{f} \sum_{all} Loss(f(Image), y)$$

There are two main types of parameters in a neural network: hyperparameters and model parameters. We will not discuss the detail in this class. We only need to know how certain hyperparameters will affect the model, such as the convolutional kernel, stride, and so on

The various modules in the neural network are connected in series, where the output of one module becomes the input of the next.
The given network is build as following, can you calculate the dimensions in every step?

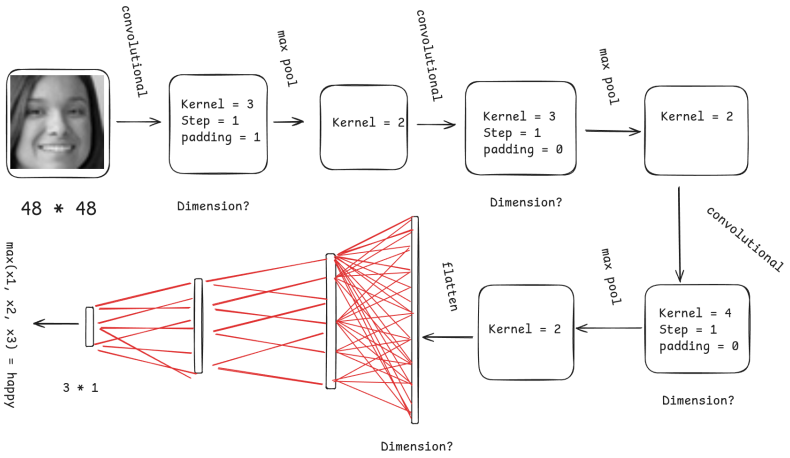

Figure 2: SI100FaceNet

After passing through a module of the CNN, the dimensions of the image will change. In the context of neural network modules, hyperparameters play a crucial role in determining the changes in input and output dimensions Hyperparameters are configurations that help control the learning process of a machine learning model. They are not adjusted during training. Instead, they need to be specified prior to training, and their values often require experimentation to optimize model performance.

The following code construct the network by PyTorch. Implement following to realize the network showing in Figure 2. There are several requirements:

1. The forward function should return a 3×1 array, where each element represents the score for the corresponding class.
2. There are a total of three convolutional layers, and the number of channels after each convolutional layer should be 64, 128, and 256, respectively.
3. After each convolutional layer and fully connected layer, there should be a Leaky ReLU activation function.

```python
class emotionNet(nn.Module):
  def __init__(self, printtoggle):
      super().__init__()

      self.print = printtoggle

      ### write your codes here ###
      ##############################
      # step1:
      # Define the functions you need: convolution, pooling, activation, and fully connected functions.



  def forward(self, x):
      #Step 2
      # Using the functions your defined for forward propagate
      # First block
      # convolution -> maxpool -> relu

      # Second block
      # convolution -> maxpool -> relu

      # Third block
      # convolution -> maxpool -> relu

      # Flatten for linear layers

      # fully connect layer

      return x
```

Ok! Let's execute the train_emotion_classifier.py.
When training starts, a window will first pop up, displaying a few randomly sampled training images to give you an intuitive sense of the data being used. Once you close this window, the actual training process begins in the background, and the terminal will continuously scroll messages showing the training progress.
In these messages, "**Epoch**" indicates how many times the model has gone through the entire dataset—we've set it to train for 10 epochs in total. "**Batch**" refers to the number of images fed into the model at once; the whole dataset is divided into 535 such batches.
When training finishes, you'll find a file named *face_expression.pth* saved locally. This file contains all the learned weight parameters of the model. Recall our earlier discussion about forward propagation: during inference, an input image passes through multiple layers of the network, and at each layer, the data is transformed using weight matrices—ultimately producing a classification result. The goal of training is precisely to iteratively adjust these weight matrices using many labeled examples, so that the outputs of forward propagation become increasingly accurate. In other words, the weights stored in *face_expression.pth* are the "optimized" parameters that enable the model to correctly recognize facial expressions.

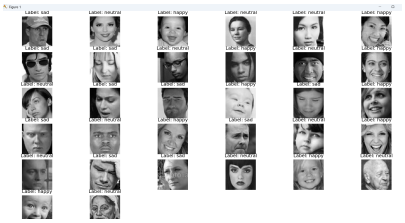

Figure 3: windows

```
Training Device:  cuda:0
Data shapes (train/test):
torch.Size([32, 3, 48, 48])

Data value range:
(tensor(-1.), tensor(1.))
Epoch: 1/10, Batch: 0, 1/535
Epoch: 1/10, Batch: 1, 2/535
Epoch: 1/10, Batch: 2, 3/535
Epoch: 1/10, Batch: 3, 4/535
Epoch: 1/10, Batch: 4, 5/535
```

Figure 4: log

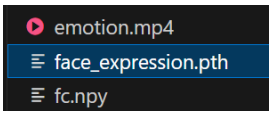

Figure 5: product

Notes

If you use Mac with M series SoC. Modify the begin of *train_emotion_classifier.py* with following code

```python
#FROM
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
#TO
device = torch.device('mps' if torch.cuda.is_available() else 'cpu')
```

Notes

If your computer has an NVIDIA GPU but training is still using the CPU, please reinstall the PyTorch library with CUDA support.

5 Bonus

- Change our model to a 7-class classification model.
- Use netron to show your model, there are lots of parameters and blocks. Write comments for each block, explaining what what the parameters mean.