SI100B Project Face Detection and emotion classification

Lecture 5

Objectives

- Learn how to use a pretrained network.
- Master the evaluation metrics of a classification model.

One of the critical reasons for the widespread use of neural networks is the ability to separate the inference process from the training process
Separating the inference process from the training process in neural networks has several advantages:

1. Efficiency: Inference often requires less computational resources than training. By decoupling the two processes, we can optimize the inference phase for speed, enabling quicker predictions. This is particularly beneficial in real-time applications, such as autonomous driving or online recommendations.
2. Scalability: During training, models can be complex and resource-intensive. However, for inference, we can use simplified versions of the model, such as pruning or quantization, which reduce the model size and enhance performance, making it easier to deploy in various environments, including those with limited compute capabilities.
3. Flexibility: Separating the processes allows for distinct workflows. Training can occur in a more controlled environment, potentially involving larger datasets and more complex setups, while inference can happen in diverse locations (e.g., cloud, edge devices) without needing the full training context or setup.
4. Model Iteration: Developers can iterate on training without affecting the inference environment. This means updates or changes to the model can be made and tested independently, leading to faster development cycles and easier deployment of new models.
5. Error Isolation: By decoupling training and inference, it's easier to identify issues with model performance. Any discrepancies between training and inference can be more systematically analyzed, helping to improve model robustness and generalization.
6. Production Stability: In production systems, models can continue running based on previously trained versions while new models are being trained. This reduces downtime and ensures that there is always a stable model available for inference.

The inference process utilizes the already trained function f to classify new images.

$$\hat{y} = f(InputImage)$$

where $\hat{y}$ is the prediction, get the label according to last lesson can calculate evaluation metrics.
There is one folder named validation in image directory. Images in this folder in used to validate the model you trained in last class. It's import to distinguishing between the test set and the training set. In this class, the task is use validation dataset to evaluate your model.

1 Evaluation Metrics

For a multi-class classification model (like a three-class model), the metrics—accuracy, recall, and F-score (specifically F1-score)—are calculated slightly differently than in a binary classification case. Here's how you can compute them:

1.1 Confusion Matrix

	Predicted A	Predicted B	Predicted C
True A	TP_A	AB	AC
True B	BA	TP_B	BC
True C	CA	CB	TP_C

1.2 Accurary

The overall accuracy for a multi-class model is calculated as:

$$Accurary_i = \frac{TP_i}{TP_i + FP_i} \tag{1}$$

$$Accurary = \frac{NumberofCorrentPredictions}{TotalNumberofPredictions} = \frac{\sum_{i=1}^{C} TP_i}{N} \tag{2}$$

where C is the number of class, TP_i is the number of true positive of class i. N is the total number of instance arcoss all classes.
eg. $acc_A = \frac{TP_A}{TP_A+BA+CA}$

1.3 Recall

Recall for each class in a multi-class classification scenario can be computed as follows:

$$Recall_i = \frac{TP_i}{TP_i + FN_i}$$

FN_i is the false negatives for class i (the instances of class i that were incorrectly predicted as other classes). The weighted recall is defined as:

$$Recall_{weight} = \sum_{i}^{C} \frac{w_i Recall_i}{\sum_i^C w_i}$$

eg. $recall_A = \frac{TP_A}{TP_A+AB+AC}$

Try to predict use the model that trained last lesson

```python
import torch
import my_net

batch_size=32

model, lossfun, optimizer = my_net.classify.makeEmotionNet(False)
PATH = './face_expression.pth'
model.load_state_dict(torch.load(PATH, weights_only=True))
test_loader, classes = my_net.utility.loadTest("./images/", batch_size)

X,y = next(iter(test_loader))

model.eval()
## Test in one batch
with torch.no_grad():
```

```
    yHat = model(X)

    ##Step 1 Obtain predicted labels
    #new_labels =

    ##Show first 32 predicted labels
    my_net.utility.imshow_with_labels(X[:batch_size], new_labels[:batch_size], classes)

    #Step 2
    ##Calculate the accuracy for each category prediction, as well as the overall accuracy
    #Print them to the screen.
    ## "happy:xx.xx%, neutral:xx.xx%, sad:xx.xx%, total:xx.xx%"

    #Step 3
    ##Calculate the recall for each category prediction, as well as the overall accuracy
    #Print them to the screen.
    ## "happy:xx.xx%, neutral:xx.xx%, sad:xx.xx%, total:xx.xx%"

## Get the accuracy and recall in full dataset
##Step 4
for X,y in test_loader:
    pass
```

Put this file to the directory you run in last class and run it

2 Bonus

Adjust the moduls and hyperparameters according to the document and retrain the model, calculate the model's accuracy and recall.