Image Classification Using Deep Neural Networks

Modular PyTorch Pipeline for Visual Recognition/Classification

Enes Aykuter

Bowdoin College

Motivation

- How can we create a algorithm to distinguish between different types of playing cards?

- Can neural networks be helpful for this problem??

- Can we build one pipeline to classify both cards and emotions???

Project Overview

- Built a model to distinguish 53 types of playing cards
(13 ranks x 4 suits + joker)

- Used a pre-trained ImageNet model and customized it with PyTorch library

- Modularized* the workflow to adapt to different datasets

*reusable code for training, preprocessing, etc for different datasets.

PyTorch, ImageNet & timm Libraries

ImageNet:

- Large-scale dataset with 14+ million labeled images

- Pretrained model widely used in model benchmarks

PyTorch:

- Developed by Meta (Facebook)

- Widely used in both research and production

timm library:

- Provides Hundreds of pretrained models, including EfficientNet, ViT, ConvNeXt, etc.

- Simple API: timm.create_model(...)

- Used in this project to load EfficientNet, pretrained on ImageNet

Sample Use of PyTorch

PyTorch stores data as ***tensors.*** Tensors are basically multi-dimensional matrices optimized for matrix operations often used in neural network trainings.

```
#splitting the data into batches for easier/faster processing
dataloader = DataLoader(dataset, batch_size=32, shuffle=True)

#checking the first batch of pictures (32 pics, 3 color, and 128x128 dimension)
for images, labels in dataloader:
    break

print(images.shape, labels.shape)
print(labels) #labels in this batch
```

```
torch.Size([32, 3, 128, 128]) torch.Size([32])
tensor([ 9,  5, 10, 52, 24,  6, 39, 37, 50,  6, 52, 16, 13, 47,  9, 11, 12, 40,
        17, 28, 17, 29, 33,  4, 32, 39, 41, 42, 36, 31, 49, 16])
```

Dataset

I used ImageFolder for easy access to data and to split it into training and validation folders. I also re-scaled the images into 128x128 pixels for faster processing.

```python
train_folder = "/kaggle/input/cards-image-datasetclassification/train/"
valid_folder = "/kaggle/input/cards-image-datasetclassification/valid/"
test_folder = "/kaggle/input/cards-image-datasetclassification/test/"

#using ImageFolder here for modularity, but CardDataSet is the default
train_dataset = ImageFolder(train_folder, transform=transform)
val_dataset = ImageFolder(valid_folder, transform=transform)
test_dataset = ImageFolder(test_folder, transform=transform)

#shuffling is not needed for validation and test datasets
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
test_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
```

Finally, I loaded the scaled images into DataLoader* after for easier access.

*I tried to build a custom CardDataSet but found out later that DataLoader does the same thing more efficiently and more modular.

Model Training

- Used the pre-trained EfficientNet as backbone

- Got rid of the last layer

- Mapped 1280 nodes into 53 classes for my cards (fine tuning!)

Here, we are determining how our model is going to behave and how many classes it will be able to specify.

53 classes set as default because there are 53 card types to distinguish between

```
class CardClassifier(nn.Module):
  def __init__(self, num_classes=53):
    super(CardClassifier, self).__init__()

    self.base_model = timm.create_model("efficientnet_b0", pretrained=True)

    #cutting the last later of the network, so that we can specify output class number
    # the * takes the items out of the list and passes them as is
    self.features = nn.Sequential(*list(self.base_model.children())[:-1])

    #hidden layer feature numbers (efficientnet_b0 default)
    enet_out_size = 1280

    #mapping the 1280 features into only 53 for our card dataset
    self.classifier = nn.Sequential(nn.Flatten(), nn.Linear(enet_out_size, num_classes))

  def forward(self, x):
    x = self.features(x)
    output = self.classifier(x)

    return output
```

Testing if it works:

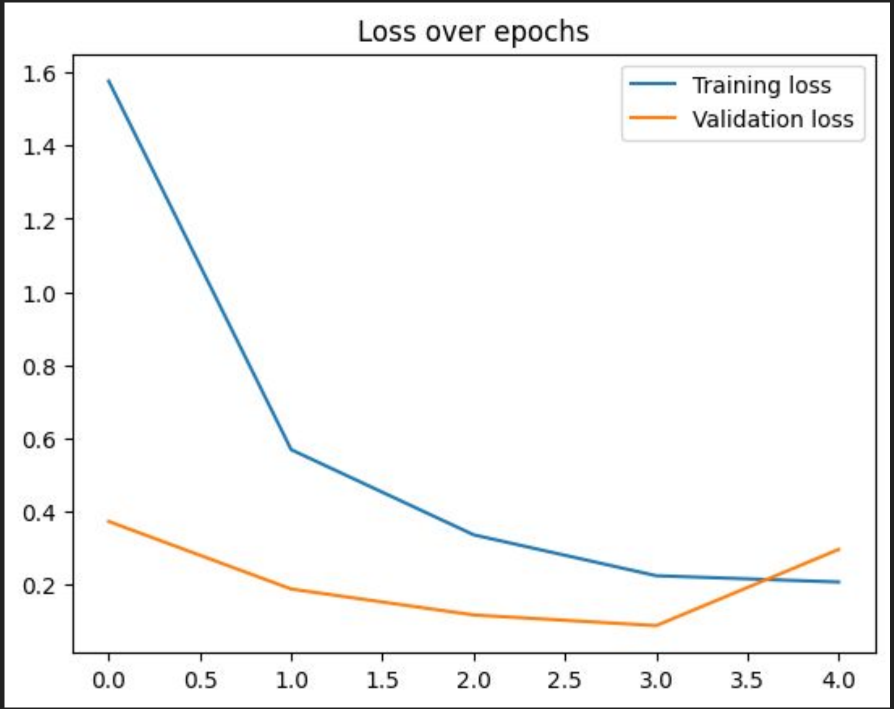
```
num_classes = len(target_to_class)
model = CardClassifier(num_classes=num_classes)

example_out = model(images)
example_out.shape #32 photos, 53 classes
```

```
torch.Size([32, 53])
```

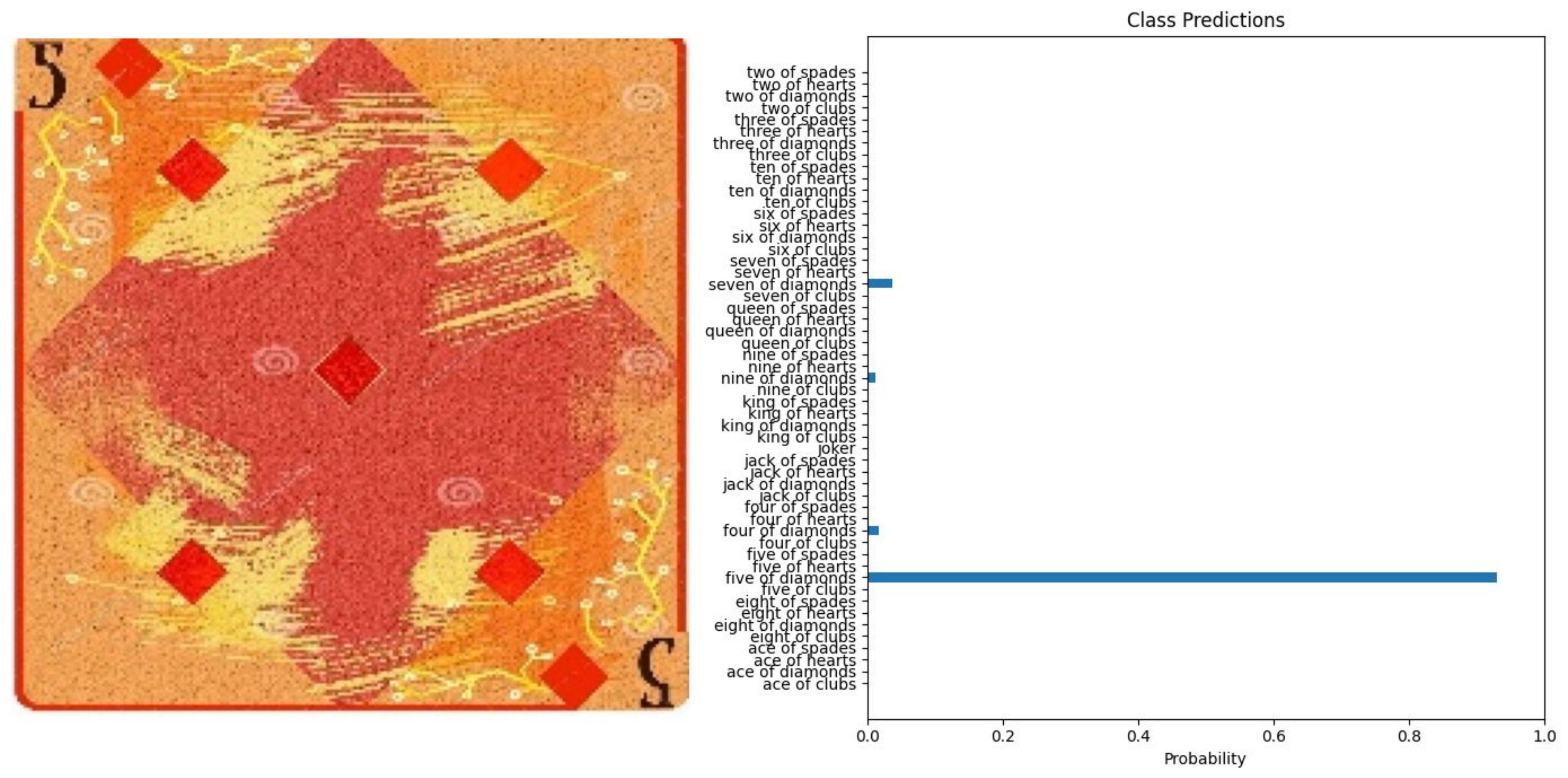
Instantiated the model and tested on a batch of 32 photos. It works!

Training Loop

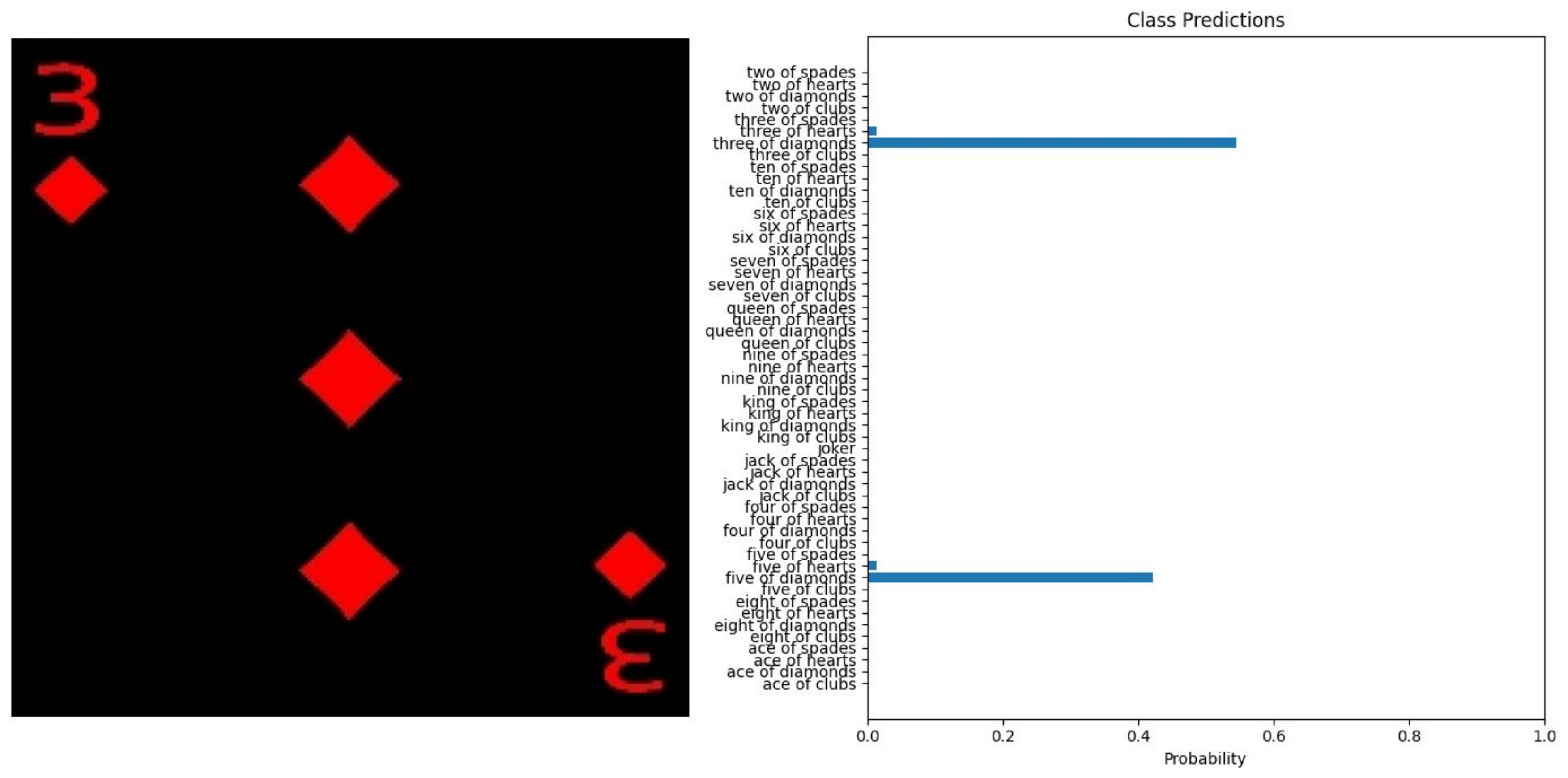
```python
num_epochs = 5
training_losses, val_losses = [], []

#this is for accelerating the training with nvidia's cuda architecture
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model.to(device)

for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0

    #looping over the batches of data
    for images, labels in tqdm(train_loader, desc="Training loop"):
        # Move inputs and labels to the device as well
        images, labels = images.to(device), labels.to(device)

        #clearing out the optimizer
        optimizer.zero_grad()

        outputs = model(images)

        # the loss between the guess and the labels
        loss = criterion(outputs, labels)

        #back propogation
        loss.backward()

        #updating the weights
        optimizer.step()

        #calculates the loss on this run weighted by the batch size
        running_loss += loss.item() * labels.size(0)

    # after running on each batch, calculates the average loss on each epoch
    training_loss = running_loss / len(train_loader.dataset)
    training_losses.append(training_loss)
```



Sample Classification



Sample Classification



Can we reuse this code for happy/sad images?

Yes! The workflow is pretty modular and by just changing the dataset paths and output feature numbers, we can classify happy and sad images.

Changes I had to make

Change the transformations:

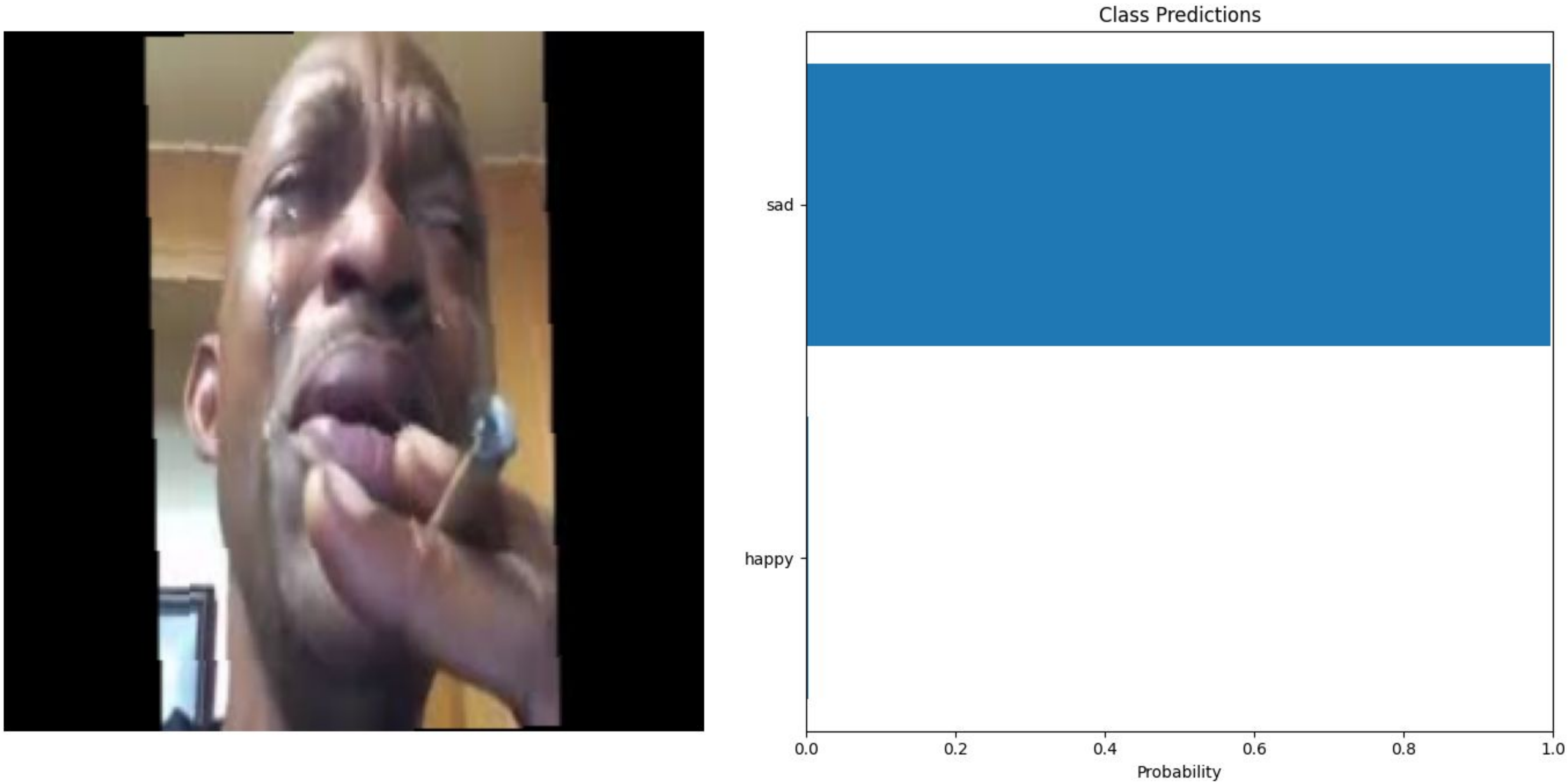
- 128x128 images weren't enough for general pattern recognition. The model was overfitting and the validation loss was constantly high.

- Changed the dimensions to 256x256 and added rotation and flipped the images for more diversity in training.

-Output classes:

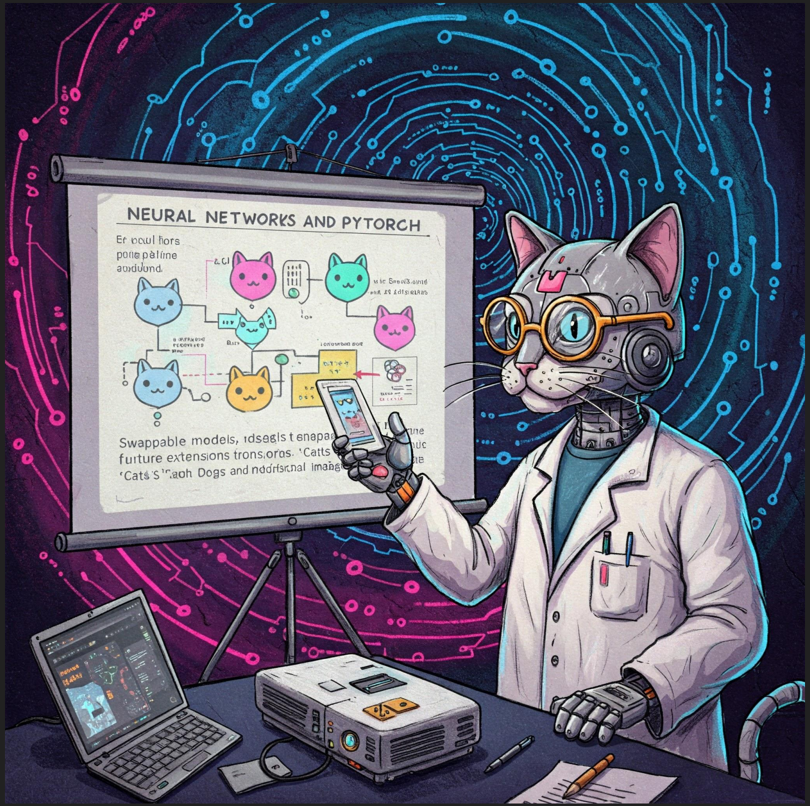
Playing Cards (53) → Emotion Faces (2)

```
[5] #like a script to resize data
    transform = transforms.Compose([
        transforms.Resize((256, 256)),
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(10),
        transforms.ToTensor()
        ])

    #here the script is passed into our data and transformed
    dataset = ImageFolder(root=path, transform=transform)
```


Class Predictions
sad
happy
0.0
0.2
0.4
0.6
0.8
1.0
Probability

Why modularity matters?

- Same codebase → different datasets
- Swappable model, transform, dataset paths
- Makes future extensions (e.g., cats vs. dogs, medical images) trivial



Sources/Inspirations

https://www.kaggle.com/code/robikscube/train-your-first-pytorch-model-card-classifier

https://www.youtube.com/watch?v=tHL5STNJKag&t=1156s

https://www.kaggle.com/datasets/aravindanr22052001/emotiondetection-happy-or-sad

https://docs.pytorch.org/tutorials/

https://www.youtube.com/feed/history

https://www.youtube.com/watch?v=IC0_FRiX-sw

Thank you for listening!

Link to project: https://colab.research.google.com/drive/1ZZ-W16wohrbh22iTSDZ0AcWNC1Dfjd3g?usp=sharing
Happy/sad version: https://colab.research.google.com/drive/18vEKoUCN8sRw2i1nVOcLAkvEnm2kbSRE#scrollTo=gAPCakpfs4Zx