Computer Vision Assignment 2

Authors (Group 2):

- Aditya Shourya (i6353515)
- Ashkan Karimi Saberi (i6365010)

code at : https://github.com/adishourya/vit_fer2013

Structure of the Report

- The focus of the report would not be on attaining the best test score but would be on experimenting with different architectures.
- And looking through how the forward pass looks in each case.

Convolutional Neural Network

- we will first develop a simple convolutional neural network
- And reason the effectiveness of the netwrok on a simple task as FER 2013[https://www.kaggle.com/datasets/msambare/fer2013]

A simple hand made vision transformer (shallow)

- we will try to replicate the paper "AN IMAGE IS WORTH 16X16 WORDS: TRANSFORMERS FOR IMAGE RECOGNITION AT SCALE" -- adosovitskiy et.al [https://arxiv.org/pdf/2010.11929v2] and show through one full forward pass.
- We will try to reason the rate of learning by giving it same number of epochs as our CNN

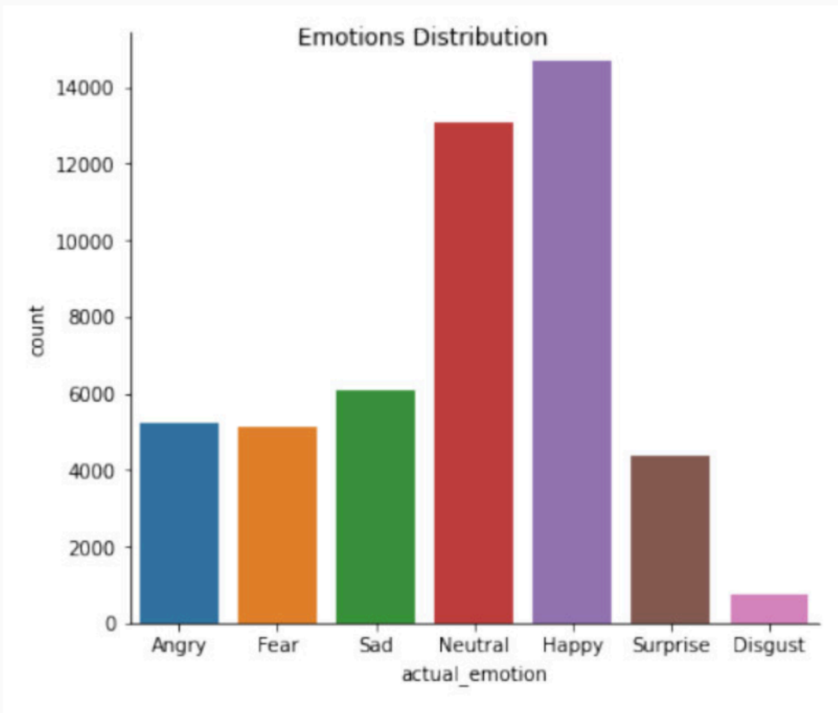
Pretrained Vision Transformer

- we will use vit_b_16 [https://pytorch.org/vision/main/models/generated/torchvision.models.vit_b_16.html#vit-b-16] and unfreeze the last few layers to perform the training.

- And compare the rate of learning with shallow transformer by giving it the same number of epochs.

Dataset

> The dataset comprises **48x48** pixel grayscale images of faces. These faces have been automatically aligned to ensure that each face is roughly centered and occupies a similar amount of space in every image.
>
> The objective is to classify each facial expression into one of seven emotional categories: 0 for Angry, 1 for Disgust, 2 for Fear, 3 for Happy, 4 for Sad, 5 for Surprise, and 6 for Neutral. The training set includes 28,709 examples, while the public test set contains 3,589 examples.
>
> -- [https://www.kaggle.com/datasets/msambare/fer2013]



Class Diftribtion in FER2013, and the challenges asscociated with the dataset are :

> The FER2013 dataset has several inherent issues that make it challenging for deep learning architectures to achieve optimal results. Key problems include imbalanced data, intra-class variation, and occlusion. Specifically, the database exhibits significant imbalance in the training data, with classes having vastly different numbers of samples. For example, the 'happy' emotion has over 13,000 samples, whereas 'disgust' has only about 600 samples, as shown in the figure above.
>
> Intra-class variation refers to the differences within the same class. Reducing intra-class variation while increasing inter-class variation is crucial for effective classification. Variations, uncontrolled lighting conditions, and occlusions are common issues that face recognition systems encounter in real-world applications . These challenges often lead to a drop in accuracy compared to performance in controlled experimental settings. Occlusion occurs when an object blocks part of a person's face, such as a hand, hair, cap, or sunglasses. Although occlusion complicates face recognition, it can also provide useful information, as people often use their hands while communicating through gestures.
>
> -- https://www.oaepublish.com/articles/ir.2021.16

CNN

```python
# define a small convolutional network
# see beautiful mnist in tinygrad .
import torch.nn as nn
import torch.nn.functional as F

# shape after operations n,n →(with padding p and stride s) (n + 2p - f + 1 )/s + 1

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.pool = nn.MaxPool2d(2, 2) # this is not a learnable operaation just performs downsampling
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 7) # we have 7 classes
```

```python

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        logits = self.fc3(x)
        return logits


net = Net()
```

In 99

```python
loss.item()
```

Executed at 2024.05.19 16:04:36 in 18ms

Out 99

```
1.3972270488739014
```

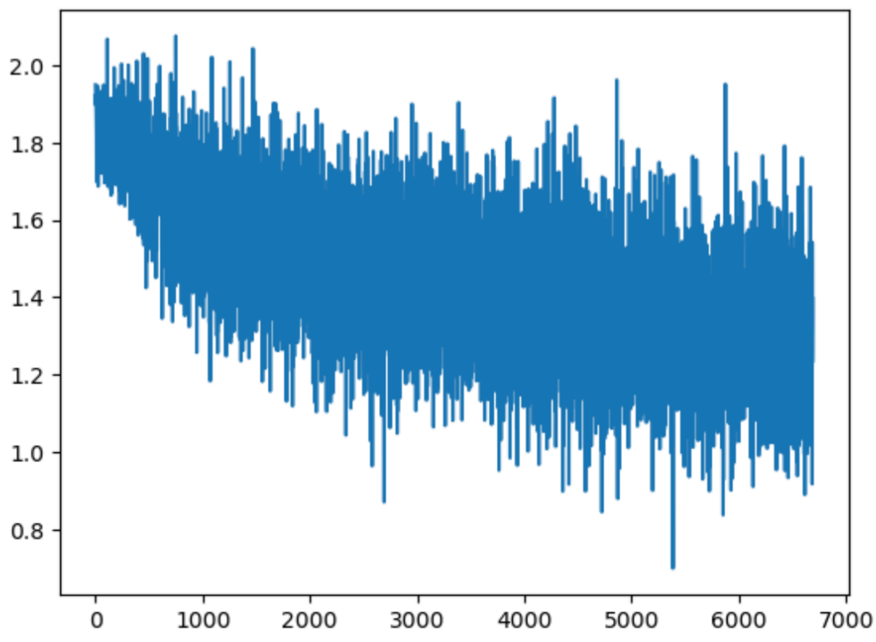
In 100

```python
plt.plot(track_loss)
```

Executed at 2024.05.19 16:04:37 in 121ms

Out 100

```
[<matplotlib.lines.Line2D at 0x15b72bf40>]
```



Shallow Transformer Replicating paper

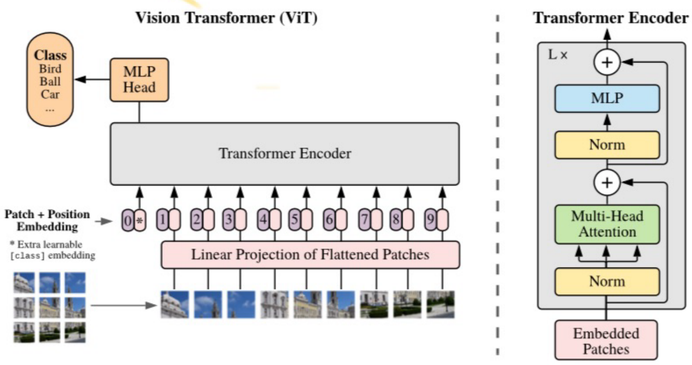


Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification token" to the sequence. The illustration of the Transformer encoder was inspired by Vaswani et al. (2017).

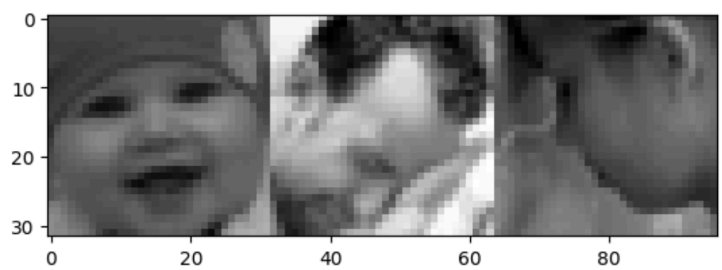
$$N = \frac{HW}{P^2} = \frac{32 \times 32}{16 \times 16} = 4$$

P_1	P_2
P_3	P_4

$P \in \mathbb{R}^{16,16,C=1}$

3 METHOD

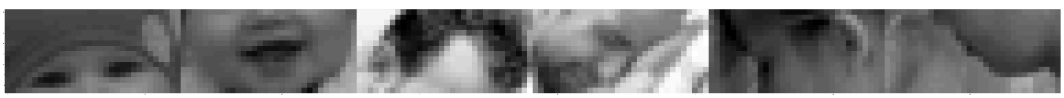
- Images to 16 * 16 patches



In 105

```
# now what we want is to turn them into patches
# patches = einops.rearrange(xb,"b 1 (p1 h) (p2 w) → b (p1 p2) (h w)",p1=2,p2=2)
patches = einops.rearrange(xb,"b 1 (p1 h) (p2 w) → h (b p1 p2 w)",p1=2,p2=2) # increase p1,p2 to
 get 16 x 16 patches
# here p1 and p2 must be 14 x 14
plt.figure(figsize=(150,80))
plt.imshow(patches, cmap="gray")
plt.show()
```

Executed at 2024.05.19 11:53:46 in 90ms



```python
# check appendix for unit test of this class
class SingleHead(nn.Module):
    """
    Implements a single head of attention (unmasked)
    """

    def __init__(self,n_embed=32,head_size=8):
        super().__init__()
        # single head
        self.head_size = torch.tensor(head_size)
        self.n_embed = torch.tensor(n_embed)
        self.Q = nn.Parameter( torch.randn(self.n_embed,head_size) * (1/torch.tensor(2.82)))
        self.K = nn.Parameter( torch.randn(self.n_embed,head_size) * (1/torch.tensor(2.82)))
        self.V = nn.Parameter( torch.randn(self.n_embed,head_size) * (1/torch.tensor(2.82)))

    def forward(self,x):
        query = x @ self.Q
        key =   x @ self.K
        value=  x @ self.V

        # hand implementation
        # scale => sqrt head size
        scale = 1 / torch.sqrt(self.head_size)

        # we will not use any masking here as its an image
        # and no dropout consideration in this implementation
        comm = query @ key.transpose(-2,-1)
        comm = comm* scale
        soft_comm = torch.softmax(comm, dim=2)
        att = soft_comm @ value

        return att
```

```python
class Multihead(nn.Module):
    def __init__(self,n_embed,n_heads):
        super().__init__()

        self.n_embed = n_embed
        self.n_heads = n_heads
```

```python
        self.head_size = self.n_embed // self.n_heads

        self.multiheads = nn.ModuleList(
            [SingleHead(self.n_embed,self.head_size)
             for _ in range(self.n_heads)]
        )

    def forward(self,x):
        return torch.cat([head(x) for head in self.multiheads],dim=2)
```

```python
# only multihead -> skip connection -> layernorm
# Batch norm : couples examples in and normalizes it .. (also has a regularization effect) but we need to keep a running mean to track new mean and sigma
# layernorm : normalizes the features of each example (does not couple examples across the batch) more popular in transformers

class TranformerBlock(nn.Module):
    def __init__(self, n_embed,n_head):
        super().__init__()
        self.multi_head = Multihead(n_embed,n_head)
        # i am not going to implement my own layer norm it wont be efficient and will be janky at best
        self.norm = nn.LayerNorm(n_embed) # we want to normalize feeatures (each patch gets normalized)

    def forward(self,x):
        # pass through multihead
        attention = self.multi_head(x)
        # skip connection and non linarity
        attention = torch.relu( x + attention)
        # layer norm
        attention = self.norm(attention)
        return attention # B , n_patch , n_embed
        ...
```

```python
# most of the comments are pasted verbatim from the paper
class SmallVIT(nn.Module):

```

```
    def __init__(self):
        super().__init__()
        # patches
        # embedding
        self.vocab_size = torch.tensor(256) # 0 to 255 pixels
        # each patch will only get one n_embed representation
        self.n_embed = 32 # we will project each patch 16*16 to a 32 dimensional representation
        # so the lookup table would be of the shape
        # unlike in nlp where we embed token to a vector like below we would project matrix of patch size to a vector
        # self.C = nn.Embedding(self.vocab_size, self.n_embed)

        self.C = nn.Parameter(torch.randn(self.vocab_size, self.n_embed) * 1/torch.sqrt(self.vocab_size) )

        # positional embedding
        # the paper says Position embeddings are added to the patch embeddings to retain positional information. We use standard learnable 1D position embeddings, since we have not observed significant performance gains from using more advanced 2D-aware position embeddings
        self.pe = nn.Parameter(torch.randn(1,4,self.n_embed)) # each pach and representation should get positional embedding

        # we use the standard approach of adding an extra learnable “classification token” to the sequence
        self.classification_token = nn.Parameter(torch.randn(1, 1, self.n_embed))
        # we will keep the step above optional .. i dont understand why we should use it yet.

        # transformer block
        self.n_heads = 4 # we will use 4 heads for now
        self.transformer_block = TranformerBlock(self.n_embed,self.n_heads)

        # MLP Head for final logit calculation
        # n_patch * n_embed → fer["emotion"].nuinque() : 7
        self.mlp_head = nn.Parameter(torch.randn(4*32, 7) * torch.sqrt(torch.tensor(4*32)))




```

```python
        ...

    def forward(self,X):
        batch_size = X.shape[0]
        patches = X.view(-1,4,256) # B , p_num , 16*16
        # B , p_num , 256 @ 256 , 32
        emb = patches @ self.C # B , p_num , n_embed
        emb = emb + self.pe # kind of acts like a bias towards each patches.

        # 2 transformers
        tf = self.transformer_block(emb)
        tf = self.transformer_block(tf)

        # flatten it : across patches
        tf = tf.view(batch_size,-1)

        # logits
        logits = tf @ self.mlp_head



        # broadcasting steps in the above command
        # B,p_n , p*p , n_embed
        #1,4,32


        return logits


```

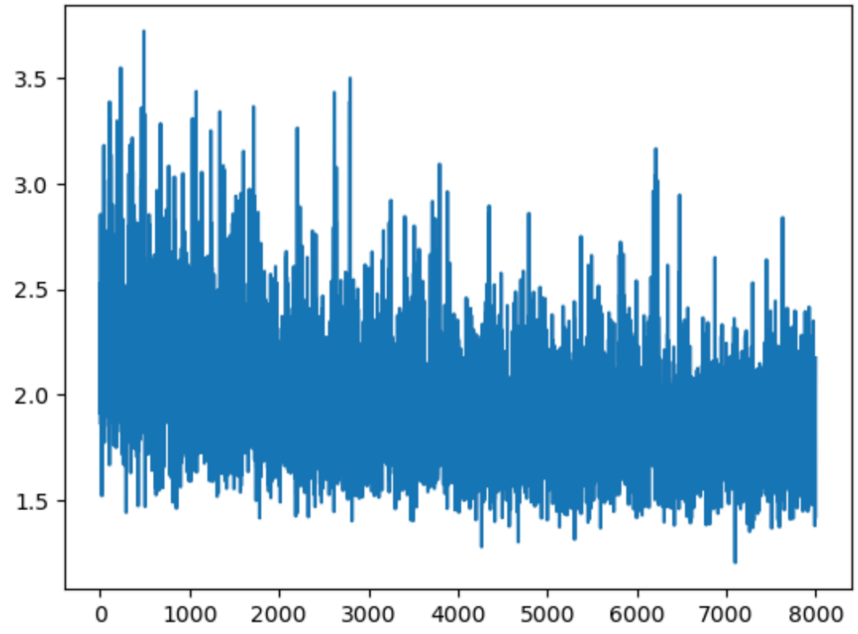
In 101

```
plt.plot(track_loss[-8000:])
```

Executed at 2024.05.19 21:54:47 in 129ms

Out 101

```
[<matplotlib.lines.Line2D at 0x1436d3250>]
```



In 95

```
np.median(
    track_loss[:-10]
)
```

Executed at 2024.05.19 21:53:49 in 20ms

Out 95

```
1.9874691367149353
```

Pretrained Vision Transformer (perform transfer learning)

```python

# output test loss statistics
print('Test Loss: {:.6f}'.format(test_loss))

```

```
100%|██████████| 113/113 [36:49<00:00, 19.55s/it]

Test Loss: 1.314458
```

```
## Results for each class
Disgust : 1%   (1/55)
Fear  : 29%    (155/528)
Happy : 78%    (688/879)
Sad : 35%    (211/594)
Surprise  : 62%    (260/416)
Neutral : 54%    (344/626)

Test Accuracy of Dataset:    50%   (1799/3589)
```

Remarks

Appendix

unit test for self attention with pytorch's implementation

```
def single_head(query, key,value):
    head_size = torch.tensor(query.shape[-1])
    # hand implementation
    # scale => sqrt head size
    scale = 1 / torch.sqrt(head_size)

    # we will not use any masking here as its an image
    # and no dropout consideration in this implementation
    comm = query @ key.transpose(-2,-1)
    comm = comm* scale
    soft_comm = torch.softmax(comm, dim=2)
    att = soft_comm @ value
    print(att.shape)
    return att


g=torch.Generator().manual_seed(123)
```

```python
query, key, value = torch.randn(2, 3, 8 , generator = g), torch.randn(2, 3, 8, generator = g), torch.randn(2, 3, 8 , generator = g)

# our implementation
sh  = single_head(query,key,value)

# pytorch implementation
py_sa = nn.functional.scaled_dot_product_attention(query, key, value)

# > torch.allclose(py_sa , sh) prints True

```