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

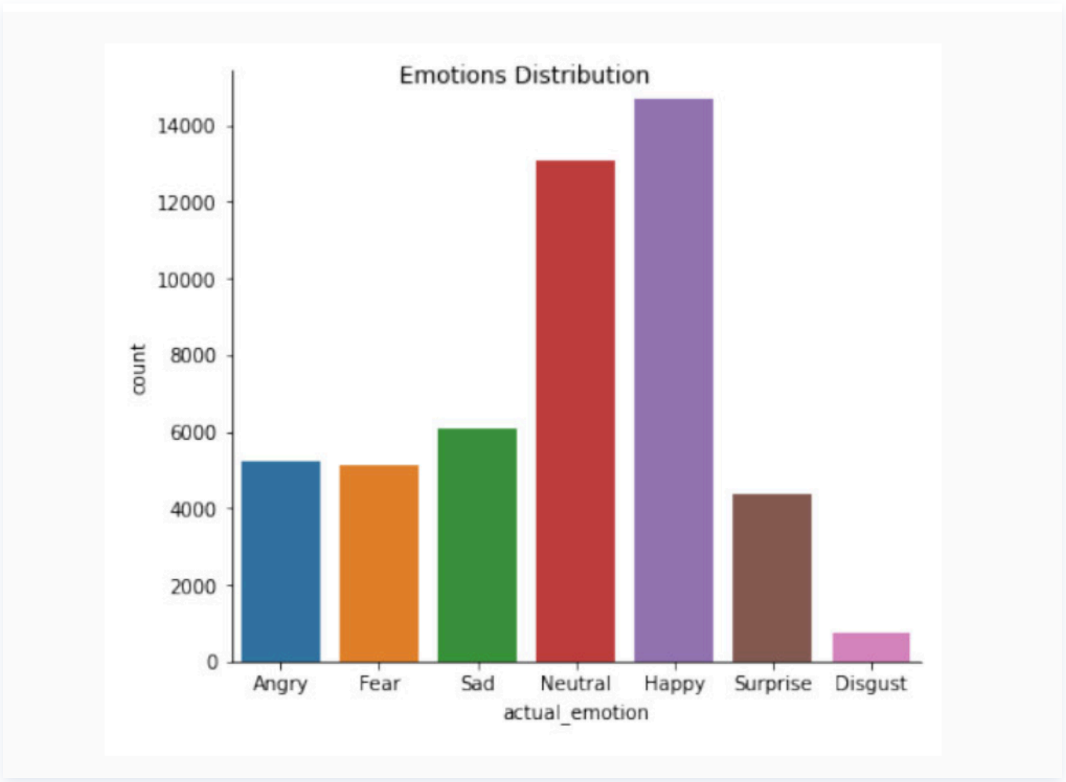
Pretrained Vision Transformer

- 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

- All the experiments and the code are from the notebook : https://github.com/adishourya/VIT_FER2013/blob/main/convolutional.ipynb

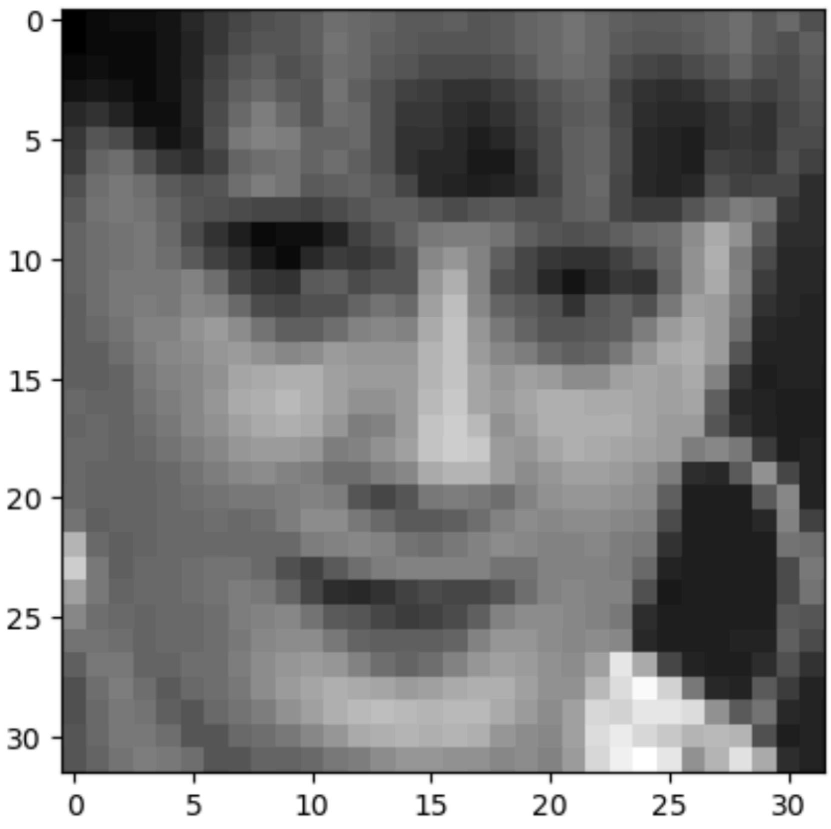
```
# define a small convolutional network
# see beautiful mnist in tinygrad .
import torch.nn as nn
import torch.nn.functional as F

# 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) # input channel 1 , 6 filter banks each of kernels size (5,5)
        self.pool = nn.MaxPool2d(2, 2) # this is not a learnable operaation just performs downsampling
        self.conv2 = nn.Conv2d(6, 16, 5) # 16 kernels
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 7) # we have 7 classes

    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()
```

forward pass in our learnt convolutional layers

```
# lets visualize the first layer on a sample image
x = self.pool(F.relu(self.conv1(x)))

# 1,6 input channel = 1 (greyscale)
# output dimension = 6 (filterbanks)
# kernel size = (5,5) kernels
net.conv1 , net.conv1.weight.shape

# > out: (Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1)), torch.Size([6, 1, 5, 5]))
```

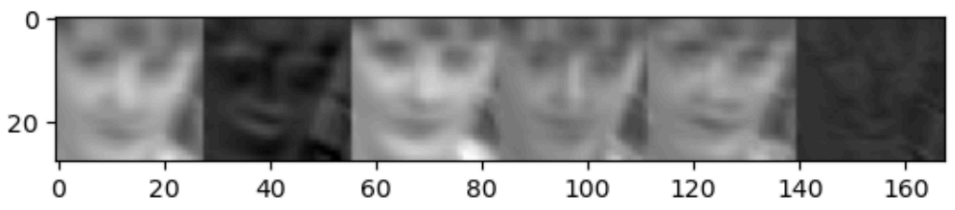
In 52
```python
import einops
sample_img = img[0]
plt.imshow(
    einops.rearrange(sample_img,"1 h w → h w"),
    cmap="grey")
```
Executed at 2024.05.24 22:29:07 in 247ms

Out 52
```
<matplotlib.image.AxesImage at 0x15796c460>
```



```python
import torch.nn.functional as F
with torch.no_grad():
    # first layer convolution
    out1 =F.conv2d(sample_img,net.conv1.weight,
                   bias=None, stride=1, padding=0)
    print(out1.shape) # input channel = 1 filter banks = 6


plt.imshow(
    einops.rearrange(out1,"out_c h w → h (out_c w)"),
        cmap="grey"
)
# visualizing the output of convolution from the first learnt layer.
# notice how some kernels are vastly different
```

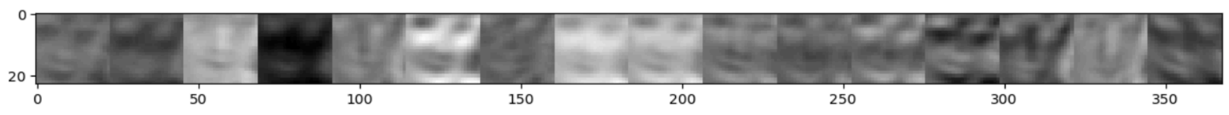
```
torch.Size([6, 28, 28])

Out 63      <matplotlib.image.AxesImage at 0x157e77550>
```



- we will not show the pooling operation here (check notebook) as it is not a learnable parameter

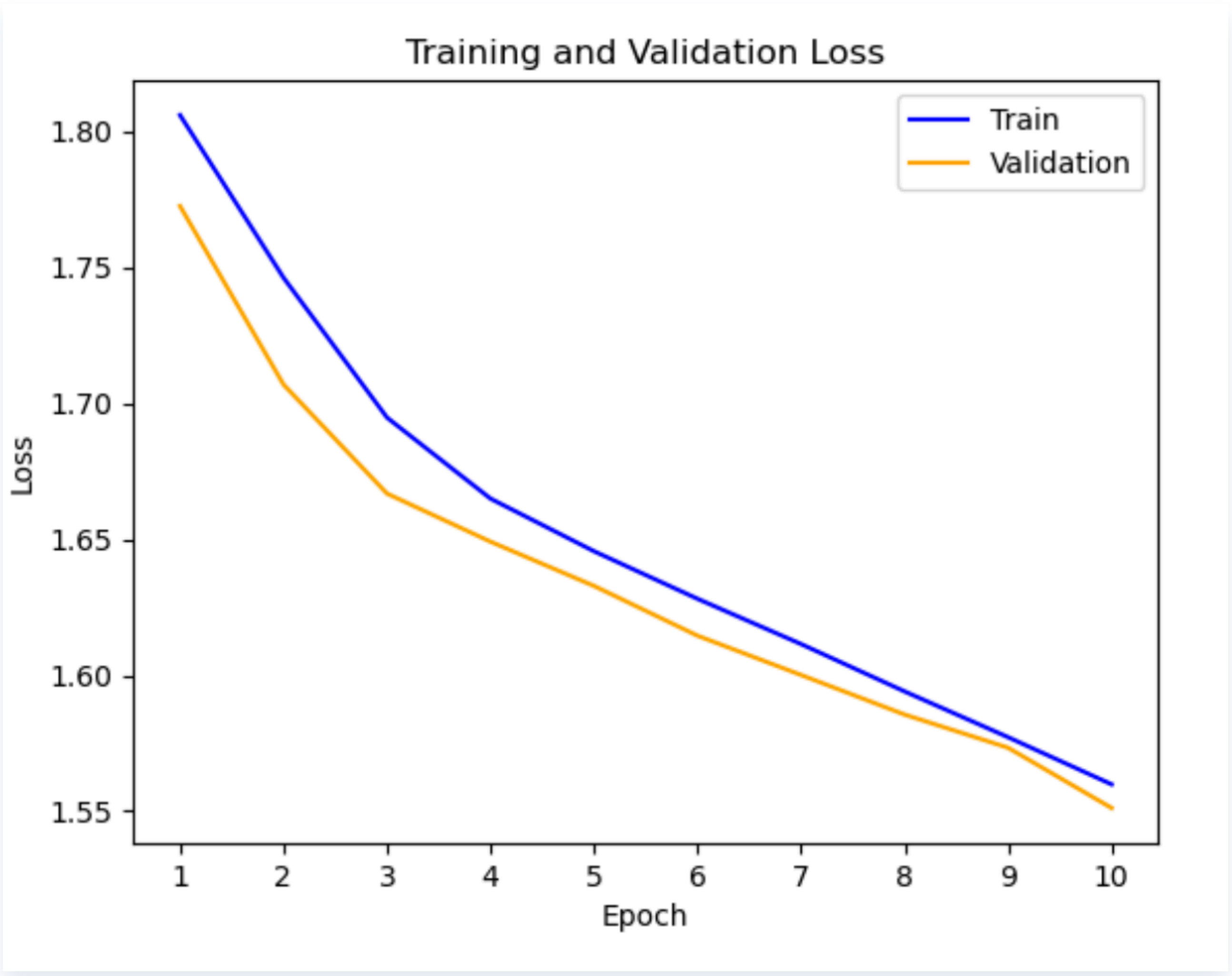
```
with torch.no_grad():
    # first layer convolution
    out2 =F.conv2d(out1_p,net.conv2.weight,
                   bias=None, stride=1, padding=0)
    print(out2.shape) # input channel = 1 filter banks = 6

plt.figure(figsize=(15,8))
plt.imshow(
    einops.rearrange(out2,"out_c h w → h (out_c w)"),
    cmap="grey"
)
```

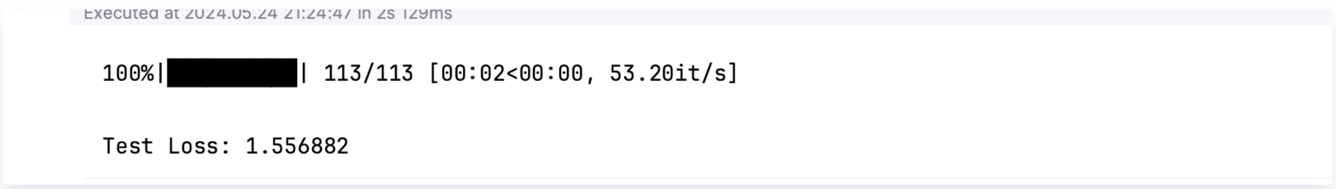
```
torch.Size([16, 23, 23])

<matplotlib.image.AxesImage at 0x157fcbdc0>
```



- and then we maxpool it and then flatten it to pass to a feed forward network to arrive at the logits for classification

- we only train for 10 epochs as attaining the best test accuracy is not our main goal of this report




Executed at 2024.05.24 21:24:47 in 2s 129ms

```
100%|██████████| 113/113 [00:02<00:00, 53.20it/s]

Test Loss: 1.556882
Test Accuracy of Classes (Generalization scores of our Convolutional Model)

Angry : 10%    (54/491)
Disgust : 0%   (0/55)
Fear  : 7%   (41/528)
Happy : 80%    (710/879)
Sad : 32%    (191/594)
Surprise  : 47%    (197/416)
Neutral : 33%    (208/626)

Test Accuracy of Dataset:    39%   (1401/3589)

```

Shallow Transformer

- we will try to replicate a shallow vit from the paper adosovitskiy et.al [https://arxiv.org/pdf/2010.11929v2]

Published as a conference paper at ICLR 2021

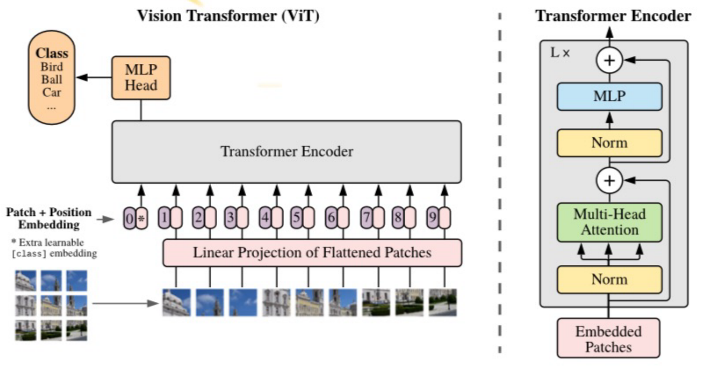


Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification 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

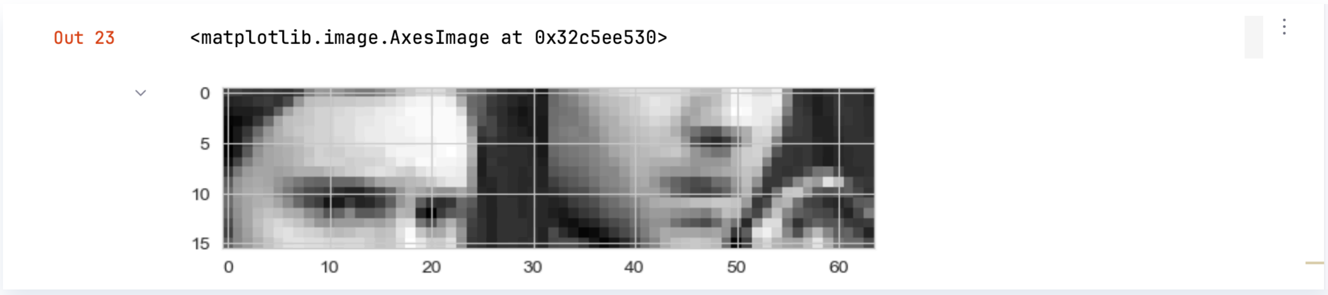
- All the experiments and the code are from : https://github.com/adishourya/VIT_FER2013/blob/main/fer_vit.ipynb
- we will first start with splitting the image as patches

generating patches

> To do so, we split an image into patches and provide the sequence of linear embeddings of these patches as an input to a Transformer. Image patches are treated the same way as tokens
>
> -- Introduction adosovitskiy et.al

> Naive application of self-attention to images would require that each pixel attends to every other pixel.
>
> -- Related Work adosovitskiy et.al

```python
## from class FerDataset(utils.data.Dataset): (check Notebook fer_vit)
  def get_patches(self, arr):
        # best place to use einops
        # squeeze out the channel dimension.. the loader will add it ..
        arr = einops.rearrange(arr,"1 h w → h w")
        patches = einops.rearrange(
            arr, "(p1 h) (p2 w) →(p1 p2) h w",p1=2 , p2=2
        ) # lay it on as the batch for the image
        # we need to crop twice for a 48 * 48 image to get a 16 * 16 patch
        return patches

```



```
Out 23    <matplotlib.image.AxesImage at 0x32c5ee530>
```

Embed Patches

- We first flatten all the patches and embed them with a lookup table

```
# initalize a lookup table
 self.C = nn.Parameter(torch.randn(self.vocab_size, self.n_embed) *
1/torch.sqrt(self.vocab_size) )

# flatten the patches
patches = einops.rearrange(X,"b p h w → b p (h w)")

# embed them : the block matrix multiplication looks like :
# B , p_num , 256 @ 256 , 32
emb = patches @ self.C # B , p_num , n_embed
# so each patch (a matrix) gets and n_embed dimensional representation (row
vector)
```

class token

- we dont actually use the class token in our implemented model. check screenshot of the issue : `` ` Is the extra class embedding important to predict the results, why not simply use feature maps to predict? ``
- But the pretrained models implement them anyway. So we do it like :

```
# its like adding an extra patch . but this patch is learnable
learnable_patch = torch.randn(1,n_embed)
# add same learnable patch to all the images in the batch
learnable_patch = einops.repeat(learnable_patch,"1 n_embed → 3 1 n_embed")
print(learnable_patch.shape)
# prints torch.Size([3, 1, 128]) # batch_size , 1 token , 128 embed
dimension

# you actually need all dims except one to be different to pack or concat
it
# in the paper they add it at the first location. (dont really see how it
matters if append it at the end instead)
```

```python
xb ,ps = einops.pack([learnable_patch,xb],"b * n_embed") # we want to append it on top of patch
xb.shape
# prints torch.Size([3, 5, 128]) # batch , tokens , n_embed

# 4 patches from the images , 1 patch from class token
```

positional embedding

```python
# 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

# init a patch embedding
self.pe = nn.Parameter(torch.randn(1,4,self.n_embed)) # each pach and representation should get positional embedding

# add patch embedding after class token
emb = emb + self.pe # kind of acts like a bias towards each patches.
```

pass through transformer blocks

- We will first start with making self attention blocks. and then concatenate them to get multihead attention (vasawani et.al 2017) .
- Note multihead attention blocks returns back the same size of representation `n_embed` .
- unit test for our implementation of self head attention is in the appendix

```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
```

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

        self.n_embed = n_embed
        self.n_heads = n_heads
        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)]
        )

```

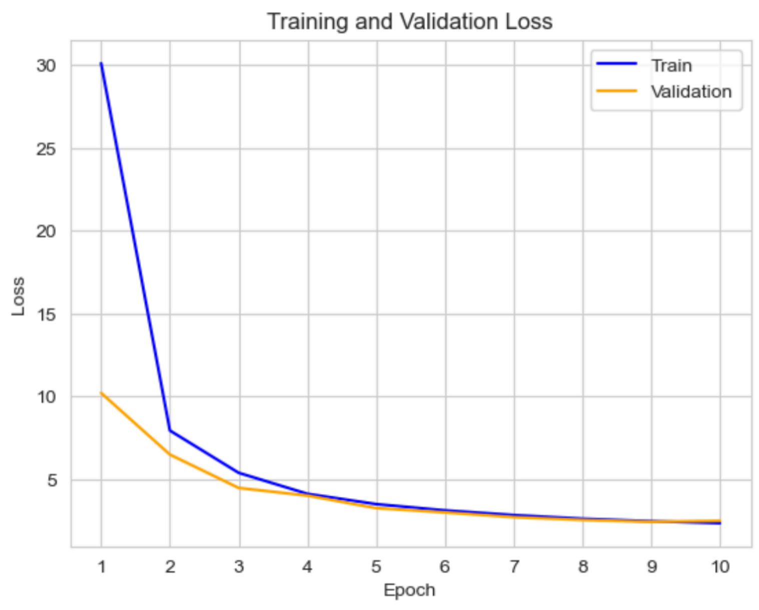
```python
    def forward(self,x):
        return torch.cat([head(x) for head in self.multiheads],dim=2)
```

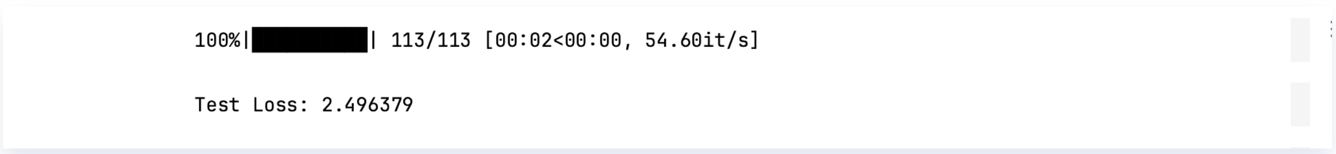
- And then a transformer block with it

```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
        ...

```

- So our architecure looks like this :

```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)))




        ...

    def forward(self,X):
        batch_size = X.shape[0]
        # flatten the patches
        patches = einops.rearrange(X,"b p h w → b p (h w)")
        # 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
```





```
100%|██████████| 113/113 [00:02<00:00, 54.60it/s]

Test Loss: 2.496379
```

```
Test Accuracy of Classes

Angry : 12%     (61/491)
Disgust : 0%    (0/55)
Fear   : 31%    (168/528)
Happy : 62%     (551/879)
Sad : 11%    (68/594)
Surprise  : 13%    (56/416)
Neutral : 10%    (64/626)

Test Accuracy of Dataset:    26%   (968/3589)
```

- Note how the accuracy of a shallow transformer (significantly higher number of parameters compared to CNN) produces worse result. which is to be expected.

Pretrained Vision Transformer (perform transfer learning)

- The code and experiments for this section can be found at : https://github.com/adishourya/VIT_FER2013/blob/main/vit_pretrained.ipynb
- we will be transfer learning from the model : 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.

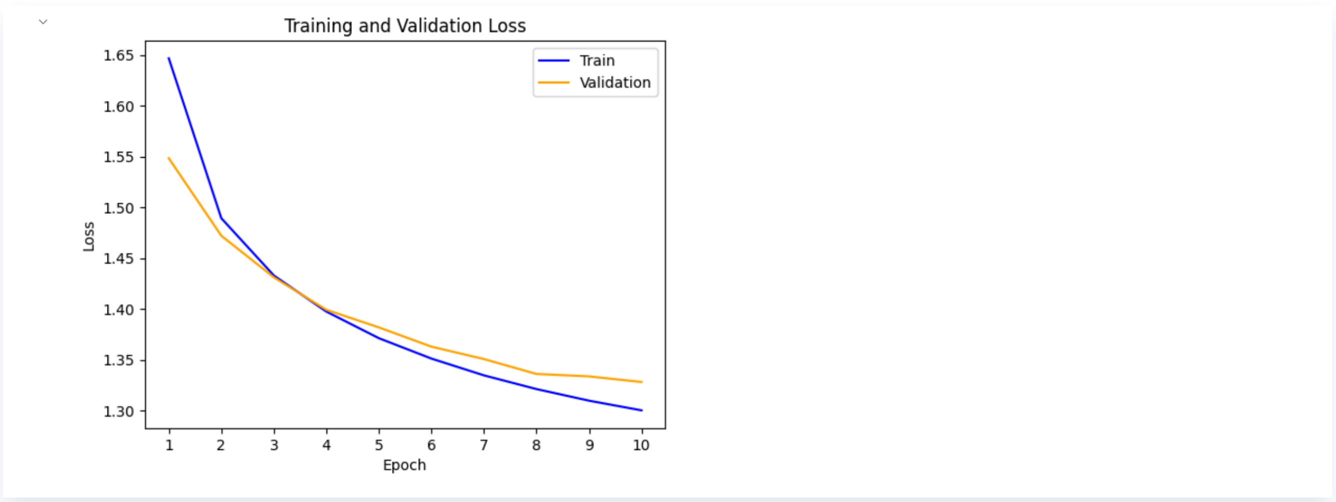
```
vision_transformer =
models.vit_b_16(weights=models.ViT_B_16_Weights.DEFAULT)
```

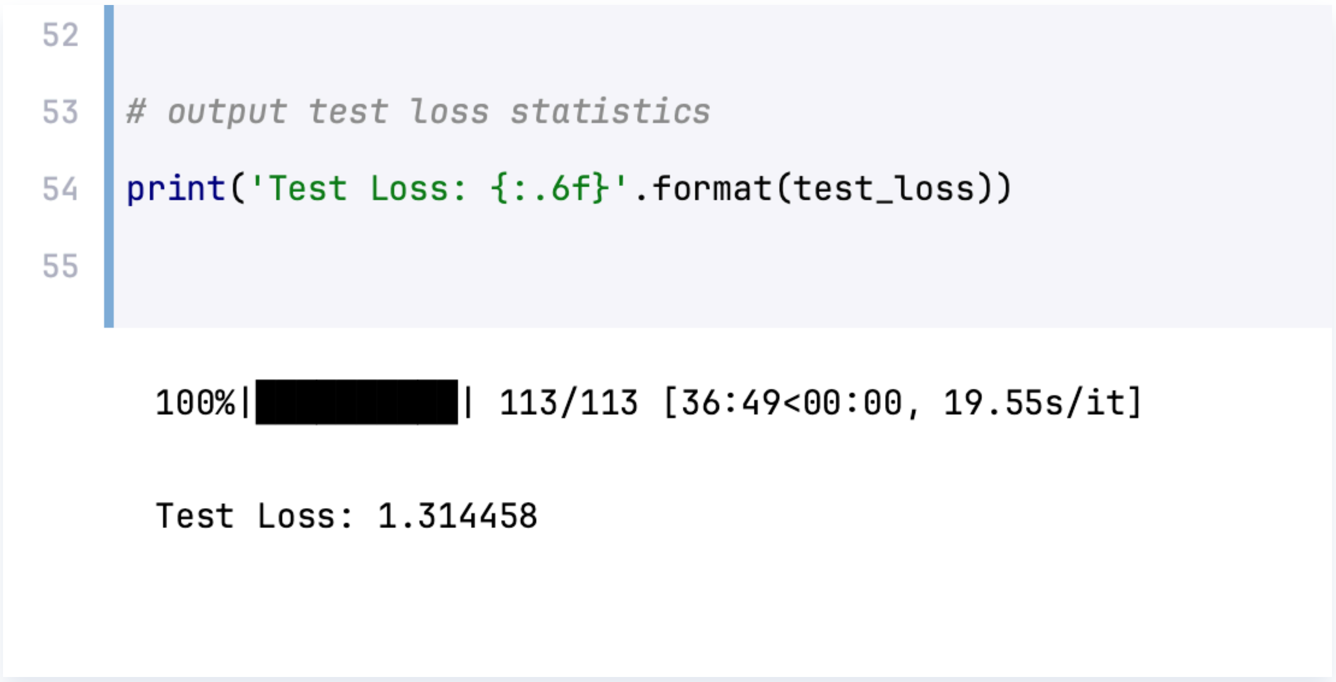
```
vision_transformer.heads

#Sequential(
#  (head): Linear(in_features=768, out_features=1000, bias=True)
#)
```

Fine Tune

```python
# fine-tune with dataset

# change the number of output classes
vision_transformer.heads = nn.Linear(in_features=256*3, out_features=7,
bias=True)

# freeze the parameters except the last linear layer
#
# freeze weights
for p in vision_transformer.parameters():
    p.requires_grad = False

# unfreeze weights of classification head to train
for p in vision_transformer.heads.parameters():
    p.requires_grad = True
```

- and then train





```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
```

![image-20240524201609270](/Users/adi/Library/Application Support/typora-user-images/image-20240524201609270.png

```
## 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)
```

- Note how its considerably better than shallow transformer that we made
- It also performs better than our small convolutional network (Note that the CNN has extremely lower number of parameters compared to vit_b16)

Remarks

- Discuss nuber of parameters

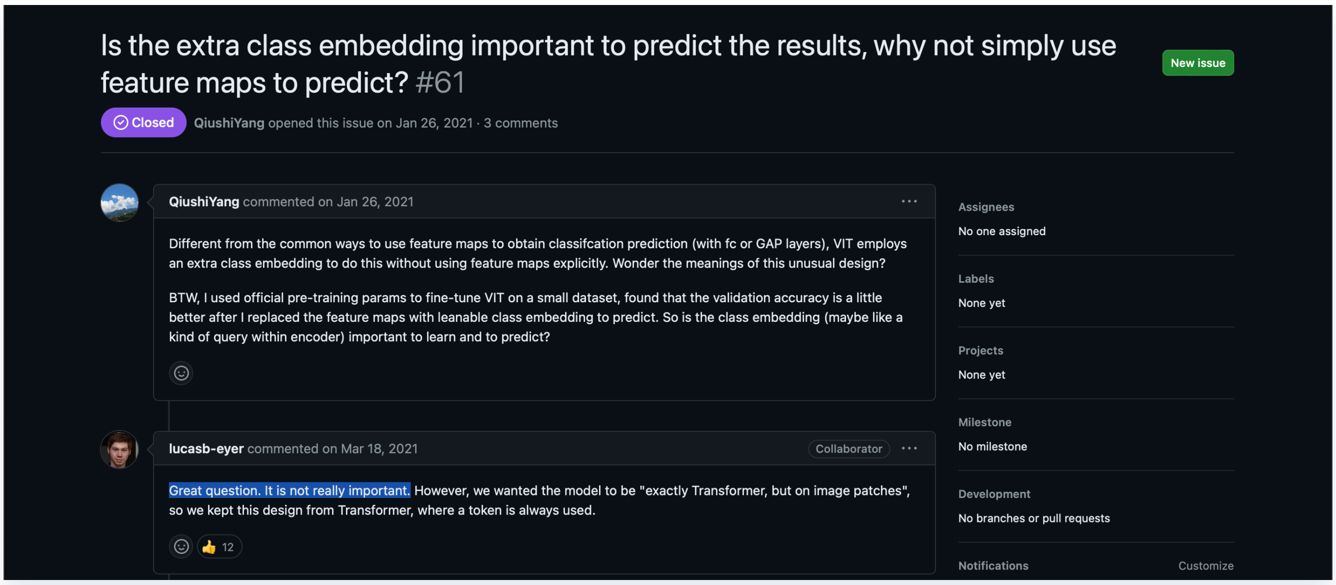
- Discuss difficulty of task at hand

Appendix

unit test for self attention with pytorch's implementation

```python
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)
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

```

importance of class tokens ?



Is the extra class embedding important to predict the results, why not simply use feature maps to predict? #61

Closed QiushiYang opened this issue on Jan 26, 2021 · 3 comments

QiushiYang commented on Jan 26, 2021

Different from the common ways to use feature maps to obtain classifcation prediction (with fc or GAP layers), VIT employs an extra class embedding to do this without using feature maps explicitly. Wonder the meanings of this unusual design?

BTW, I used official pre-training params to fine-tune VIT on a small dataset, found that the validation accuracy is a little better after I replaced the feature maps with leanable class embedding to predict. So is the class embedding (maybe like a kind of query within encoder) important to learn and to predict?

lucasb-eyer commented on Mar 18, 2021 — Collaborator

Great question. It is not really important. However, we wanted the model to be "exactly Transformer, but on image patches", so we kept this design from Transformer, where a token is always used.

👍 12