pytorch paper replicating

0: getting setup ## need torch 1.12+ and torchvision 0.13+

```
import torch... import torchvision ... from torch import nn ... from torchvision import transforms ... from torchinfo import summary
from going_modular.going_modular import data_setup, engine
from helper_functions import download_data, set_seeds, plot_loss_curves
## try and exception blocks if uninstalled
device = "cuda" if torch.cuda.is_available() else "cpu"
```

1: get data

```
image_path = download_data(source = " github raw link", destination ="pizza_steak_sushi")
train_dir = image_path / "train" ... test_dir = image_path / "test"
```

2: create datasets and dataloaders

```
img_size = 224 ... batch_size = 32
manual_transforms = transforms.Compose([transforms.Resize((img_size, img_size)), transforms.ToTensor(),])
train_dataloader, test_dataloader, class_names = data_setup.create_dataloaders(train_dir=train_dir,
        test_dir=test_dir, transforms = manual_transforms, batch_size = batch_size)
```

4: equation 1

```
height = 224 ... width = 224 ... color_channels = 3 ... patch_size = 16
number_of_patches = int((height * width) / patch_size **2)
class PatchEmbedding(nn.Module):
    def __init__(self, in_channels:int=3, patch_size:int=16, embedding_dim: int = 768):
        super().__init__()
        self.patcher = nn.Conv2d(in_channels=in_channels, out_channels=embedding_dim, kernel_size = patch_size,
                                 stride = patch_size, padding=0)
        self.flatten = nn.Flatten(start_dim=2, end_dim=3)
    def forward(self,x):
        image_resolution = x.shape[-1] ... assert image_resolution % patch_size == 0, f"invalid size"
        x = self.flatten(self.patcher(x))
        return x.permute(0,2,1)
```

5: equation 2, multi-head attention

```
class MultiheadSelfAttentionBlock (nn.Module):
    def __init__(self, embedding_dim: int = 768, num_heads: int=12, attn_dropout: float: 0):
        super().__init__()
        self.layer_norm = nn.LayerNorm (normalized_shape = embedding_dim)
        self.multihead_attn = nn.MultiheadAttention (embed_dim = embedding_dim, num_heads = num_heads,
                                                     dropout: attn_dropout, batch_first = True)
    def forward (self, x):
        x = self.layer_norm(x)
        attn_output, _ = self.multihead_attn (query=x, key=x, value=x, need_weights = False)
        return attn_output
class MLPBlock (nn.Module):
    def __init__(self, embedding_dim: int: 768, mlp_size: int = 3072, dropout: float = 0.1);
        super().__init__()
        self.layer_norm = nn.LayerNorm (normalised_shape = embedding_dim)
        self.mlp = nn.Sequential (nn.Linear (in_features = embedding_dim, out_features = mlp_size),
                                  nn.GELU(), nn.Dropout (p= dropout), nn.Linear (in_features = mlp_size, out_features=
                                  embedding_dim), nn.Dropout (p: dropout))
    def forward (self, x):
        x = self.mlp( self.layer_norm (x))    ... return x
```

7: Create transformer encoder

```
class TransformerEncoderBlock (nn.Module):
    def __init__(self, embedding_dim: int =768, num_heads: int=12, mlp_size: int= 3072,
                 mlp_dropout: float = 0.1, attn_dropout: float = 0):
        super().__init__()
                                                                                          attn_dropout =
        self.msa_block = MultiheadSelfAttentionBlock (embedding_dim, num_heads = num_heads,  attn_dropout)
        self.mlp_block = MLPBlock (embedding_dim = embedding_dim, mlp_size = mlp_size, dropout = mlp_dropout)
    def forward (self, x):
        x = self.msa_block (x) + x      ...     x = self.mlp_block (x) + x
        return x
```

Can also use   torch.nn.TransformerEncoderLayer()

Pytorch paper replicating continued...

```
class ViT(nn.Module):
    def __init__(self, img_size:int=224, in_channels:int=3, patch_size:int=16, num_transformer_layers:int=12,
            embedding_dim:int=768, mlp_size:int=3072, num_heads:int=12, attn_dropout:float=0,
            mlp_dropout:float=0.1, num_classes:int=1000):
        super().__init__()
        assert img_size % patch_size == 0
        self.num_patches = (img_size * img_size) // patch_size**2
        self.class_embedding = nn.Parameter(data=torch.rand(1,1, embedding_dim), requires_grad=True)
        self.position_embedding = nn.Parameter(data=torch.rand(1, self.num_patches+1, embedding_dim), requires_grad=
                                                                                                       True)
        self.embedding_dropout = nn.Dropout(p=embedding_dropout)
        self.patch_embedding = PatchEmbedding(in_channels=in_channels, patch_size=patch_size, embedding_dim=
                                                                                             embedding_dim)
        self.transformer_encoder = nn.Sequential(*[TransformerEncoderBlock(embedding_dim=embedding_dim, num_heads,
                                                                                                 mlp_size,
                                                                                                 mlp_dropout)
                                        for _ in range(num_transformer_layers)])
        self.classifier = nn.Sequential(nn.LayerNorm(normalized_shape=embedding_dim), nn.Linear(in_features=embedding_dim,
                                                                                                out_"  = num_classes))
    def forward(self, x):
        batch_size = x.shape[0]
        class_token = self.class_embedding.expand(batch_size, -1, -1)
        X = self.patch_embedding(x) ... x = torch.cat((class_token, x), dim=1) ... x = self.position_embedding + X
        x = self.embedding_dropout(x) ... x = self.transformer_encoder(x) ... x = self.classifier(x[:, 0])
        return x
vit = ViT(num_classes=len(class_names))
```

9: Setting up and training code for vit model

```
from going_modular.going_modular import engine
optimizer = torch.optim.Adam(params=vit.parameters(), lr=3e-3, betas=(0.9, 0.999), weight_decay=0.3)
loss_fn = torch.nn.CrossEntropyLoss()
set_seeds()
results = engine.train(model=vit, train_dataloader=train_dataloader, test_dataloader=test_dataloader,
                       optimizer=optimizer, loss_fn=loss_fn, epochs=10, device=device)
```