IT353-LAB_ASSIGNMENT2

Akshara

211AI012

Dataset:

I have used an image dataset for the binary classification task. The dataset is of size 1821 with dandelion and daisy images. It was split into train(1275), test(182) and validation sets(364). The size of image in original dataset is 512 x 512.

Link: https://www.kaggle.com/datasets/alsaniipe/flowers-dataset/data

Google colab notebook:

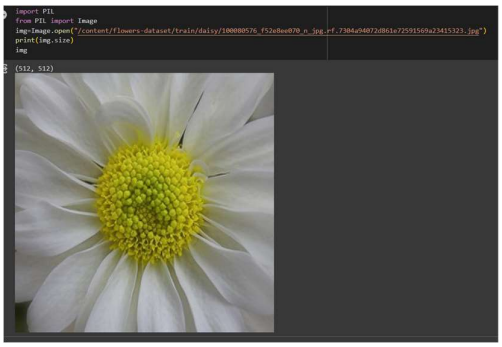
code can be found here –

https://colab.research.google.com/drive/1ItTHSp930fyVvN-i9AcQisofJnzyC5mS?usp=sharing

Tasks:

A. Download a binary classification dataset and implement your own feed forward network with 2 architectures a) Start with X units in a layer and narrow down the network till 2 unit output is reached b) Start with X units in a layer and increase till 2X units are in a layer, then narrow down the network till 2 unit output is reached (6 Marks)

1 Preparing dataset (preprocessing)



The initial image size is 512x512 it is visualized above using pillow.

First, the dataset is loaded into train_loader, test_loader and valid_loader for computer vision tasks. for this I defined three sets of transformations for training (train_transform), testing (test_transform), and validation (valid_transform). These transformations are applied to the input images to augment and preprocess them before feeding them into a neural network.

RandomHorizontalFlip(): Randomly flips the image horizontally. This helps to introduce variety in the training data and improve the model's generalization.

ColorJitter(): Randomly adjusts the brightness, contrast, saturation, and hue of the image. Again, this helps in making the model more robust to different lighting conditions.

Resize(255): Resizes the image to a square of size 255x255 pixels.(512 x 512 -→ 255 x 255)

CenterCrop(224): Crops the center region of the image to a size of 224x224 pixels. This is a common size for many pre-trained deep learning models.

(224 is an arbitrarily chosen value. This will be the final image size and it is generally chosen to be some multiple of 2^x where larger the x the better this makes it easier for maxpooling to be applied)

ToTensor(): Converts the image to a PyTorch tensor. Neural networks in PyTorch generally work with tensors as input.

After this we use datasets.Imageloader in pytorch to get train_data , test_data and valid_data in required format.

Each instance of training data is a tensor of dimensions (3 x 244 x 244 —image tensor, label (0 or1)). The first dimension consists of image tensor and second consists of label. 0 for dandelion and 1 for daisy. Below is an image of random 600th instance of train_data

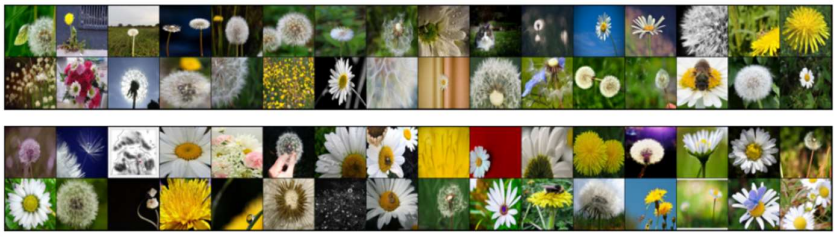
```
print(len(train_data[0][0][0]))
train_data[600]
#3d tensor representation of an image
```

```
224
(tensor([[[0.7490, 0.7373, 0.7255,  ..., 0.5882, 0.6000, 0.6000],
          [0.7529, 0.7451, 0.7255,  ..., 0.5843, 0.5922, 0.5804],
          [0.7608, 0.7569, 0.7333,  ..., 0.5725, 0.5765, 0.5647],
          ...,
          [0.6157, 0.5412, 0.4902,  ..., 0.5373, 0.4980, 0.4549],
          [0.6000, 0.5608, 0.5529,  ..., 0.5725, 0.5333, 0.4863],
          [0.6039, 0.6000, 0.6314,  ..., 0.6275, 0.5725, 0.5255]],

         [[0.7176, 0.7059, 0.6863,  ..., 0.5961, 0.6078, 0.6078],
          [0.7216, 0.7176, 0.6941,  ..., 0.5922, 0.6000, 0.5882],
          [0.7294, 0.7255, 0.6980,  ..., 0.5804, 0.5843, 0.5725],
          ...,
          [0.5608, 0.4824, 0.4275,  ..., 0.5412, 0.5059, 0.4902],
          [0.5412, 0.5020, 0.4902,  ..., 0.5725, 0.5412, 0.5216],
          [0.5529, 0.5412, 0.5647,  ..., 0.6275, 0.5804, 0.5529]],

         [[0.6588, 0.6471, 0.6196,  ..., 0.5490, 0.5608, 0.5569],
          [0.6667, 0.6549, 0.6314,  ..., 0.5412, 0.5529, 0.5373],
          [0.6784, 0.6667, 0.6392,  ..., 0.5373, 0.5412, 0.5216],
          ...,
          [0.4235, 0.3451, 0.2941,  ..., 0.5216, 0.4784, 0.4353],
          [0.4039, 0.3725, 0.3608,  ..., 0.5725, 0.5333, 0.4863],
          [0.4431, 0.4275, 0.4471,  ..., 0.6157, 0.5686, 0.5255]]]),
 1)
```

After this we choose batch size as 32 and load train, test, and valid data into train, test and valid, loaders respectively. I have visualized training images batchwise to show how the dataset looks.

```
[ ] train_loader = torch.utils.data.DataLoader(train_data, batch_size=32,shuffle=True)
    test_loader = torch.utils.data.DataLoader(test_data, batch_size=32)
    valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=32)
```

```
import matplotlib.pyplot as plt
from torchvision.utils import make_grid
def show_batch(dl, invert=True):
    for images, labels in dl:
        fig, ax = plt.subplots(figsize=(12, 6))
        ax.set_xticks([])
        ax.set_yticks([])
        ax.imshow(make_grid(images, nrow=16).permute(1, 2, 0))

def show_sample(image, label, invert=True):
    print('Label: ' + train_data.classes[label] + "(" + str(label) + ")")
    plt.imshow(image.permute(1, 2, 0))
show_batch(train_loader, invert=True)
```

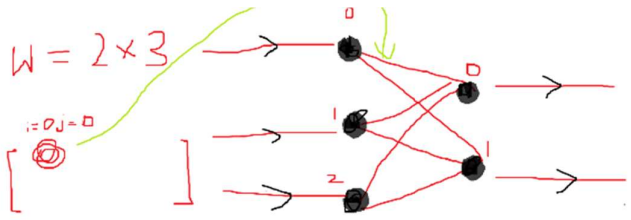


2 Implementing custom linear layer (without using torch.nn.linear)

```python
import torch
import torch.nn as nn
import torch.nn.init as init
import math

class MyLinearLayer(nn.Module):
    """ Custom Linear layer but mimics a standard linear layer """
    def __init__(self, input_size, output_size):
        super().__init__()
        self.input_size, self.output_size = input_size, output_size
        weights = torch.Tensor(output_size, input_size) # creates a matrix of size output x input where wij represent weights between ith input neuron to jth output neuron
        self.weights = nn.Parameter(weights)  # nn.Parameter is a Tensor that's a module parameter.
        bias = torch.Tensor(output_size) # bias is added while sending to the output neuron
        self.bias = nn.Parameter(bias) # bias is also declared as model parameter

        # initialize weights and biases
        nn.init.kaiming_uniform_(self.weights, a=math.sqrt(5)) # weight initialising

        bound = 1 / math.sqrt(self.input_size)
        nn.init.uniform_(self.bias, -bound, bound)  # bias initialisng

    def forward(self, x):
        # Linear operation: y = x * W^T + b
        w_times_x = torch.mm(x, self.weights.t())  # Matrix multiplication of input x and transposed weights W
        output = torch.add(w_times_x, self.bias)  # Element-wise addition of w_times_x and bias b
        return output
```

This is the custom implementation of linear layer.

- First I created Constructor method for the custom linear layer, taking input_size and output_size as arguments.
- Next I created weight and bias tensors, and wrapping them as nn.Parameter to make them trainable parameters of the model. In pytorch back-propogation is implied (no separate function for back-propogation while defining layers) so defining weights and bias as model parameters tells it to update them while back propagating.
- Weights is initialised as a matrix of size output_nodes_size x input_nodes_size where w(i , j) represents weights from input node i to output node j



- Biases is of output size as it is added only while computing output layer.
- Initializing the weights using the Kaiming uniform initialization, which is a common initialization for deep neural networks. (it used when we are using ReLu as activation in our neural network)
- Initializing the biases using uniform initialization within a specific bound. Any value from the bound (-1/root(input_size),+ 1/root(input_size)) is chosen at random and initialised
- Implementing the forward pass of the custom linear layer
  w_times_x = torch.mm(x, self.weights.t())
  Performing the matrix multiplication of the input x and the transposed weights W.

output = torch.add(w_times_x, self.bias)

Performing element-wise addition of the product of input and weights with the bias.

- Return the output matrix

3. **Implementing Feed forward neural network architectures:**

    1. **layer size increasing to 2x and then reducing to 2**

```
class NN_1(nn.Module):
    def __init__(self):
        super(NN_1, self).__init__()

        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.batchnorm1 = nn.BatchNorm2d(32)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.dropout1 = nn.Dropout2d(0.2)

        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.batchnorm2 = nn.BatchNorm2d(64)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.dropout2 = nn.Dropout2d(0.2)

        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.batchnorm3 = nn.BatchNorm2d(128)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.dropout3 = nn.Dropout2d(0.2)

        self.flatten = nn.Flatten()

        self.fc1 = MyLinearLayer(128 * 28 * 28, 32)
        self.batchnorm_fc1 = nn.BatchNorm1d(32)
        self.dropout_fc = nn.Dropout(0.2)

        self.fc2 = MyLinearLayer(32 , 64)
        self.batchnorm_fc2 = nn.BatchNorm1d(64)

        self.fc3 = MyLinearLayer(64 , 128)
        self.batchnorm_fc3 = nn.BatchNorm1d(128)

        self.fc4 = MyLinearLayer(128, 2)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.batchnorm1(x)
        x = self.pool1(x)
        x = self.dropout1(x)

        x = F.relu(self.conv2(x))
        x = self.batchnorm2(x)
        x = self.pool2(x)
        x = self.dropout2(x)

        x = F.relu(self.conv3(x))
        x = self.batchnorm3(x)
        x = self.pool3(x)
        x = self.dropout3(x)

        x = self.flatten(x)

        x = F.relu(self.fc1(x))
        x = self.batchnorm_fc1(x)
        x = self.dropout_fc(x)

        x = F.relu(self.fc2(x))
        x = self.batchnorm_fc2(x)
        x = self.dropout_fc(x)

        x = F.relu(self.fc3(x))
        x = self.batchnorm_fc3(x)
        x = self.dropout_fc(x)

        x = self.fc4(x)

        return F.softmax(x, dim=1)
```

This is the first neural network architecture here three convolution layers with batch normalisation and pooling were used and after that the tensor is flattened to 1d and fully connected feed forward layer is applied. Four linear layers were applied using my custom linear function. The size of layers are 32, 64,128, 2. (layer size increasing to almost 3x then decreasing to 2)

1. Input Layer:
   - The network takes input images with a size of 224x224 pixels and three color channels (RGB).

2. Convolutional Layers:
   - Three convolutional layers (`conv1`, `conv2`, `conv3`) are used with ReLU activation functions.
   - The first convolutional layer has 3 input channels (RGB) and 32 output channels, followed by batch normalization, max-pooling, and dropout.

- The second convolutional layer has 32 input channels and 64 output channels, again followed by batch normalization, max-pooling, and dropout.
- The third convolutional layer has 64 input channels and 128 output channels, followed by batch normalization, max-pooling, and dropout.

3. Flatten Layer: - The output from the convolutional layers is flattened using `nn.Flatten()` to prepare it for the fully connected layers.

4. Fully Connected Layers:
- Three fully connected layers (`fc1`, `fc2`, `fc3`) are employed with ReLU activation functions.
- `fc1` takes the flattened input with a size of 128 * 28 * 28 (output channels from the last convolutional layer), and has 32 output neurons.
- `fc2` takes the output of `fc1` with 32 input neurons and has 64 output neurons.
- `fc3` takes the output of `fc2` with 64 input neurons and has 128 output neurons.
- All fully connected layers are followed by batch normalization and dropout for regularization.

5. Output Layer:
- The final fully connected layer (`fc4`) has 128 input neurons and 2 output neurons, corresponding to the binary classification task.
- The output is processed through a softmax activation function (`F.softmax(x, dim=1)`) to obtain probabilities for the two classes (binary classification).

6. Dropout and Batch Normalization:
- Dropout layers (`dropout1`, `dropout2`, `dropout3`, `dropout_fc`) are applied after each max-pooling layer and fully connected layer for regularization, reducing overfitting.
- Batch normalization is applied after each convolutional layer and fully connected layer to normalize activations and accelerate training.

Note: The custom linear layer (`MyLinearLayer`) is employed for the fully connected layers, allowing for customization of weight initialization.

2. **layer size gradually reducing to 2**

This is the first neural network architecture here three convolution layers with batch normalisation and pooling were used and after that the tensor is flattened to 1d and fully connected feed forward layer is applied. Four linear layers were applied using my custom linear function. The size of layers are 128, 64,32, 2. (gradually decreasing layer size)

```python
class NN_2(nn.Module):
    def __init__(self):
        super(NN_2, self).__init__()

        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.batchnorm1 = nn.BatchNorm2d(32)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.dropout1 = nn.Dropout2d(0.2)

        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.batchnorm2 = nn.BatchNorm2d(64)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.dropout2 = nn.Dropout2d(0.2)

        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.batchnorm3 = nn.BatchNorm2d(128)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.dropout3 = nn.Dropout2d(0.2)

        self.flatten = nn.Flatten()

        self.fc1 = MyLinearLayer(128 * 28 * 28, 128)
        self.batchnorm_fc1 = nn.BatchNorm1d(128)
        self.dropout_fc = nn.Dropout(0.2)

        self.fc2 = MyLinearLayer(128, 64)
        self.batchnorm_fc2 = nn.BatchNorm1d(64)

        self.fc3 = MyLinearLayer(64, 32)
        self.batchnorm_fc3 = nn.BatchNorm1d(32)

        self.fc4 = MyLinearLayer(32, 2)



    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.batchnorm1(x)
        x = self.pool1(x)
        x = self.dropout1(x)

        x = F.relu(self.conv2(x))
        x = self.batchnorm2(x)
        x = self.pool2(x)
        x = self.dropout2(x)

        x = F.relu(self.conv3(x))
        x = self.batchnorm3(x)
        x = self.pool3(x)
        x = self.dropout3(x)

        x = self.flatten(x)

        x = F.relu(self.fc1(x))
        x = self.batchnorm_fc1(x)
        x = self.dropout_fc(x)

        x = F.relu(self.fc2(x))
        x = self.batchnorm_fc2(x)
        x = self.dropout_fc(x)

        x = F.relu(self.fc3(x))
        x = self.batchnorm_fc3(x)
        x = self.dropout_fc(x)

        x = self.fc4(x)

        return F.softmax(x, dim=1)
```

1. Input Layer:
   - The network takes input images with a size of 224x224 pixels and three color channels (RGB).

2. Convolutional Layers:
   - Three convolutional layers (`conv1`, `conv2`, `conv3`) are used with ReLU activation functions.
   - The first convolutional layer has 3 input channels (RGB) and 32 output channels, followed by batch normalization, max-pooling, and dropout.
   - The second convolutional layer has 32 input channels and 64 output channels, again followed by batch normalization, max-pooling, and dropout.
   - The third convolutional layer has 64 input channels and 128 output channels, followed by batch normalization, max-pooling, and dropout.

3. Flatten Layer:
   - The output from the convolutional layers is flattened using `nn.Flatten()` to prepare it for the fully connected layers.

4. Fully Connected Layers:

- Three fully connected layers (`fc1`, `fc2`, `fc3`) are employed with ReLU activation functions.
- `fc1` takes the flattened input with a size of 128 * 28 * 28 (output channels from the last convolutional layer), and has 128 output neurons.
- `fc2` takes the output of `fc1` with 128 input neurons and has 64 output neurons.
- `fc3` takes the output of `fc2` with 64 input neurons and has 32 output neurons.
- All fully connected layers are followed by batch normalization and dropout for regularization.

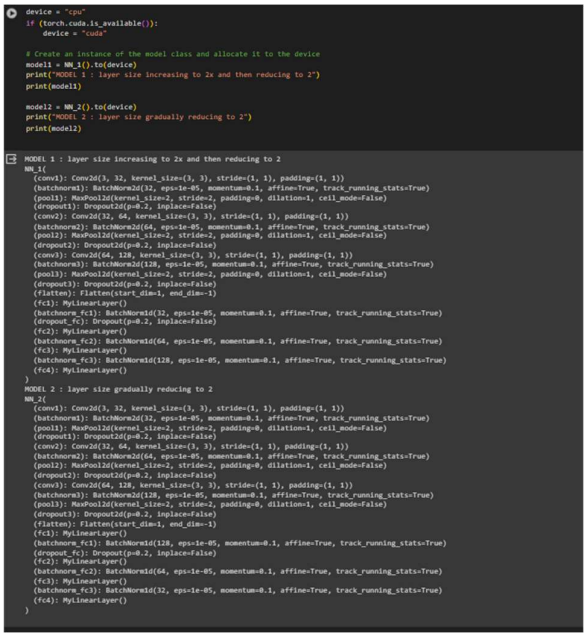
5. Output Layer:
- The final fully connected layer (`fc4`) has 32 input neurons and 2 output neurons, corresponding to the binary classification task.
- The output is processed through a softmax activation function (`F.softmax(x, dim=1)`) to obtain probabilities for the two classes (binary classification).

6. Dropout and Batch Normalization:
- Dropout layers (`dropout1`, `dropout2`, `dropout3`, `dropout_fc`) are applied after each max-pooling layer and fully connected layer for regularization, reducing overfitting.
- Batch normalization is applied after each convolutional layer and fully connected layer to normalize activations and accelerate training.

- The custom linear layer (`MyLinearLayer`) is employed for the fully connected layers, allowing for customization of weight initialization.

Results:

This is the resulting models after compiling NN_1 and NN_2 (zoom in for better view)



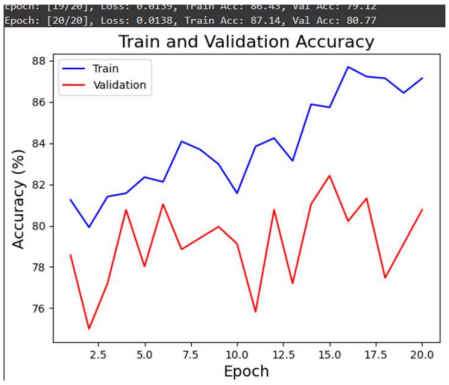
Next we train the model simulataneously on train set and validation set. We use crossentropyloss as loss criteria to calculate loss and stochastic gradient descent(SGD) as optimizer in backpropogation of loss. Code can be found in the colab link.

B. Compute Accuracy, Precision, Recall and F1 score and visualize the same for Training set, Validation set and Test set (4 Marks)

After building the models we train them on cuda. while training we get both training loss and validation loss and adjust our weights accordingly. Learning rate(lr) is chosen to be 0.01 and I trained it for 20 epochs. The results are as follows :

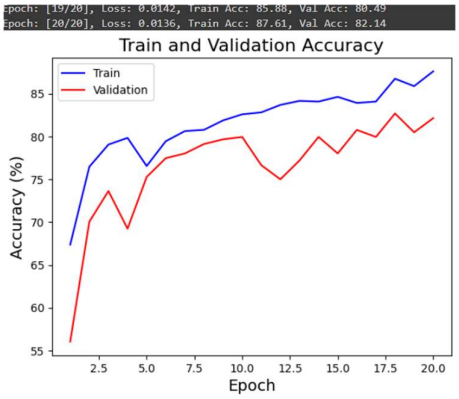
Model1 (layers increasing to 2x then reducing to 2)

```
optimizer = optim.SGD(model1.parameters(), lr=0.01, momentum=0.9)
train(model1, optimizer, train_loader, valid_loader, n_epoch=20, device=device)
plot()

Epoch: [1/20], Loss: 0.0156, Train Acc: 81.25, Val Acc: 78.57
Epoch: [2/20], Loss: 0.0158, Train Acc: 79.92, Val Acc: 75.00
Epoch: [3/20], Loss: 0.0155, Train Acc: 81.41, Val Acc: 77.20
Epoch: [4/20], Loss: 0.0154, Train Acc: 81.57, Val Acc: 80.77
Epoch: [5/20], Loss: 0.0151, Train Acc: 82.35, Val Acc: 78.02
Epoch: [6/20], Loss: 0.0152, Train Acc: 82.12, Val Acc: 81.04
Epoch: [7/20], Loss: 0.0148, Train Acc: 84.08, Val Acc: 78.85
Epoch: [8/20], Loss: 0.0147, Train Acc: 83.69, Val Acc: 79.40
Epoch: [9/20], Loss: 0.0151, Train Acc: 82.98, Val Acc: 79.95
Epoch: [10/20], Loss: 0.0154, Train Acc: 81.57, Val Acc: 79.12
Epoch: [11/20], Loss: 0.0147, Train Acc: 83.84, Val Acc: 75.82
Epoch: [12/20], Loss: 0.0146, Train Acc: 84.24, Val Acc: 80.77
Epoch: [13/20], Loss: 0.0150, Train Acc: 83.14, Val Acc: 77.20
Epoch: [14/20], Loss: 0.0140, Train Acc: 85.88, Val Acc: 81.04
Epoch: [15/20], Loss: 0.0141, Train Acc: 85.73, Val Acc: 82.42
Epoch: [16/20], Loss: 0.0137, Train Acc: 87.69, Val Acc: 80.22
Epoch: [17/20], Loss: 0.0137, Train Acc: 87.22, Val Acc: 81.32
Epoch: [18/20], Loss: 0.0136, Train Acc: 87.14, Val Acc: 77.47
Epoch: [19/20], Loss: 0.0139, Train Acc: 86.43, Val Acc: 79.12
Epoch: [20/20], Loss: 0.0138, Train Acc: 87.14, Val Acc: 80.77
```

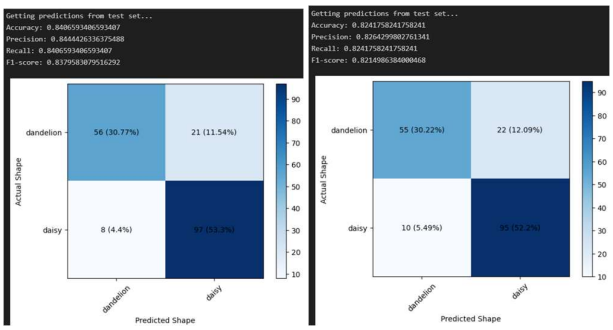


Model2 (layers gradually reducing to 2)

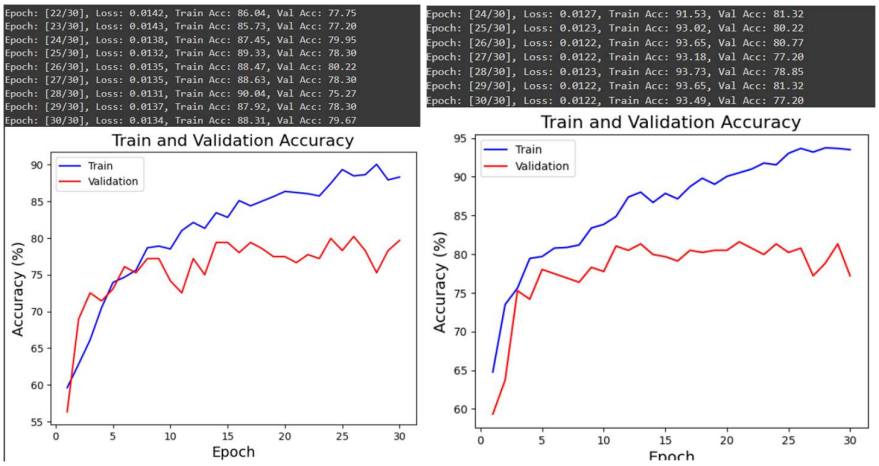
```
optimizer = optim.SGD(model2.parameters(), lr=0.01, momentum=0.9)
train(model2, optimizer, train_loader, valid_loader, n_epoch=20, device=device)
plot()

Epoch: [1/20], Loss: 0.0195, Train Acc: 67.37, Val Acc: 56.04
Epoch: [2/20], Loss: 0.0171, Train Acc: 76.47, Val Acc: 70.05
Epoch: [3/20], Loss: 0.0164, Train Acc: 79.06, Val Acc: 73.63
Epoch: [4/20], Loss: 0.0159, Train Acc: 79.84, Val Acc: 69.23
Epoch: [5/20], Loss: 0.0166, Train Acc: 76.55, Val Acc: 75.27
Epoch: [6/20], Loss: 0.0160, Train Acc: 79.45, Val Acc: 77.47
Epoch: [7/20], Loss: 0.0156, Train Acc: 80.63, Val Acc: 78.02
Epoch: [8/20], Loss: 0.0156, Train Acc: 80.78, Val Acc: 79.12
Epoch: [9/20], Loss: 0.0153, Train Acc: 81.88, Val Acc: 79.67
Epoch: [10/20], Loss: 0.0150, Train Acc: 82.59, Val Acc: 79.95
Epoch: [11/20], Loss: 0.0151, Train Acc: 82.82, Val Acc: 76.65
Epoch: [12/20], Loss: 0.0149, Train Acc: 83.69, Val Acc: 75.00
Epoch: [13/20], Loss: 0.0147, Train Acc: 84.16, Val Acc: 77.20
Epoch: [14/20], Loss: 0.0147, Train Acc: 84.08, Val Acc: 79.95
Epoch: [15/20], Loss: 0.0146, Train Acc: 84.63, Val Acc: 78.02
Epoch: [16/20], Loss: 0.0145, Train Acc: 83.92, Val Acc: 80.77
Epoch: [17/20], Loss: 0.0145, Train Acc: 84.08, Val Acc: 79.95
Epoch: [18/20], Loss: 0.0139, Train Acc: 86.75, Val Acc: 82.69
Epoch: [19/20], Loss: 0.0142, Train Acc: 85.88, Val Acc: 80.49
```



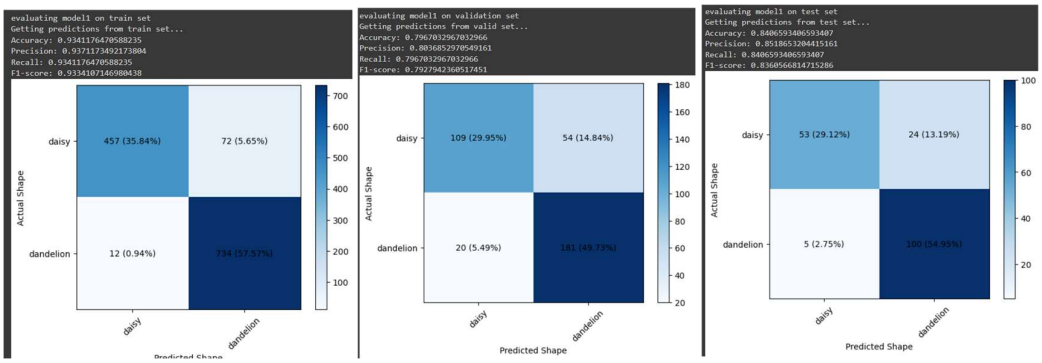
Accuracy on test set (model1 and model2 respectively)(20 epochs)



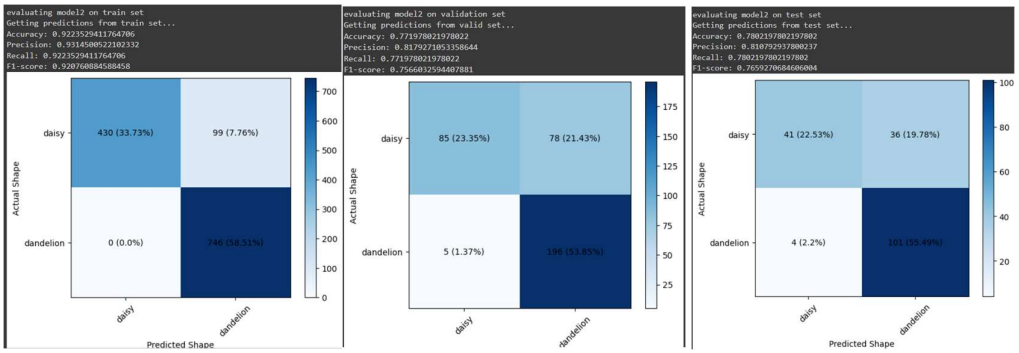
After training for 30 epochs: (model1 and model2 respectively)



Accuracy precision recall f1 score for model1 on train test and validation sets(30 epochs):



Accuracy precision recall f1 score for model2 on train test and validation sets:

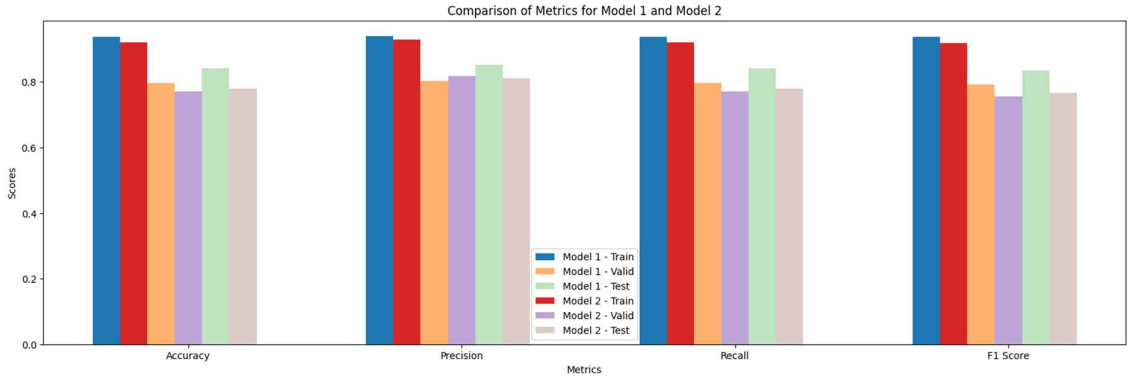


Detailed analysis and conclusion of training and testing results:

- When we observe the training and validation accuracy vs epochs graph of both model1 and model2 we observe that model1 has a very irregular graph and model2 has a relatively smooteher graph. This means model2 converges smoothly while model1 has

trouble in converging. I reduced the learning rate to 0.001 ad epochs to 30 and trained it again to see how the graph looks. The second graph(for 30 epochs) still shows that the model1 has some irregularities compared to model2. So this might be an architectural issue and not training issue.

- Also we notice that training for 20 epochs is sufficient for both the models and after that that the train and validation graphs diverge as epochs increase beyond 20. Overfitting happens beyond 20 epochs.
- Smaller learning rate (0.001) works better for model1 and model2 can converge even with learning rate 0.01
- For 20 epochs model2 gave a better accuracy(82%) than model1(80%). When we evaluate on test sets we got the accuracy of 84% for model1 and 82% for model2 so both are comparitively similar.
- For 20 epochs when we train the two models and evaluate the accuracy on train test and validation sets we see an interesting pattern. Model1 gives an accuracy of 93% on train set, 79% on validation set and 84% on test set. Model2 gives an accuracy of 92% on train set, 77% on validation set and 78% on test set.
- Here we see an issue with model2. The poor performance of model2 is because it predicted many daisies to be dandelions(as shown in coonfusion matrix). This behaviour maybe due to some issue in the architecture. I changed epochs and learning rates to fix this but this issue prevails. It predicts around 36% of daisies to be dandelions.This also results in lower F1 score for model2 in test set. I assume that this issue is with the architecture.
- Model1 had no such issue and it performs decently on test set. But model2 cannot be trained to 30 epochs and should only be trained to 20 epochs where its accuracy is 84%. to avoid overfitting on train data.



In conclusion for my dataset I would prefer using model1 over model2 because model1 has a considerably decent performance in all aspects(accuracy,recall,f1score,precision) compared to model2. Epochs should be set to 20 and lr to 0.001.