Q1)

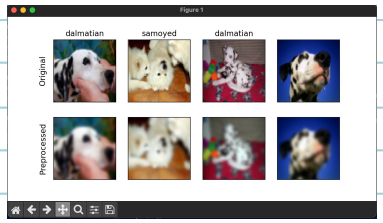
a. After running data.py, these were my results:

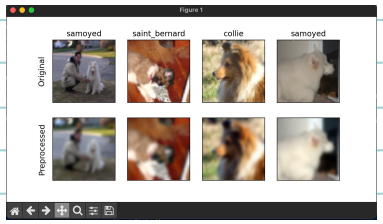
image_mean: [122.72, 115.85, 102.57]

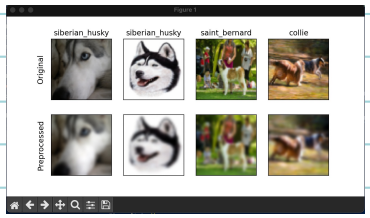
image_std: [63.05, 61.73, 63.96]

b. We extract these from the training set because if we were to use the other partitions, we would be introducing bias.

c.




The pre-processed images are noticeably more blurry / less-focused, which is due to them being down-sized

Q2)

a.

| Layer | Weights | Biases | Total |
|---|---|---|---|
| 1 | $5 \times 5 \times 3 \times 16$ | 16 | 1,216 |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| Grand Total | | | |

b.

(b) [10 pts] Speculate on these architecture choices by completing the table below (use your intuition Fill in the current choice and one possible alternative. In the three rightmost columns, compare th alternative against the current choice by filling in either ↑ or ↓. For each comparison, provide a sho justification that identifies the main differences.

| | Current choice | Alternative | How might the alternative affect... | | |
|---|---|---|---|---|---|
| | | | Training Speed | Approximation Error | Estimation Error |
| Initialization | random normal | zero | ↓ | ↑ | |
| Activation | ReLU | ELU | | | |
| Depth | | | | | |
| Regularization | | | | | |

- For initialization, zero breaks symmetry, all the neurons start the same way, so they will all train the same way, and so the network won't train well, and that increases approx and est error.
- ELU is more expensive per step, which makes it slower, but if there is a negative domain, it can learn better, which reduces approx error. The improved gradient flow can also reduce the overfitting risk (est. error)
- With more layers, that means greater capacity, so lower approx error. But the training speed decreases because there are more parameters, and there is a higher risk of overfitting
- With dropout, it disables neurons stochastically, which means slower convergence, and a higher approx error. But, it does help reduce the risk of overfitting.

c.

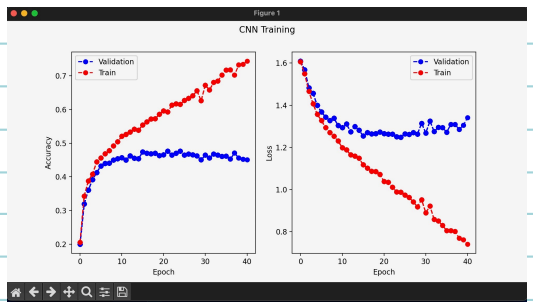
```python
class CNN(nn.Module):
    def __init__(self):
        super().__init__()

        # TODO: define each layer
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16,kernel_size=5, stride=2, padding=2)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=64,kernel_size=5, stride=2, padding=2)
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=32,kernel_size=5, stride=2, padding=2)
        self.fc1 = nn.Linear(512, 64)
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, 5)
        #

        self.init_weights()

    def init_weights(self):
        for conv in [self.conv1, self.conv2, self.conv3]:
            C_in = conv.weight.size(1)
            nn.init.normal_(conv.weight, 0.0, 1 / sqrt(5*5*C_in))
            nn.init.constant_(conv.bias, 0.0)
```

```python
        # TODO: initialize the parameters for [self.fc1, self.fc2, self.fc3]
        layer_input_dims = [512, 64, 32]
        fc_layers = [self.fc1, self.fc2, self.fc3]

        for fc, in_dim in zip(fc_layers, layer_input_dims):
            nn.init.normal_(fc.weight, mean=0.0, std=1.0 / sqrt(in_dim))
            nn.init.constant_(fc.bias, 0.0)

        #

    def forward(self, x):
        N, C, H, W = x.shape

        # TODO: forward pass
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        x = x.view(x.size(0), -1)

        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        z = self.fc3(x)
        #

        return z
```

d.

```python
# TODO: define loss function, and optimizer
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=config('cnn.learning_rate'))
#
```

e.

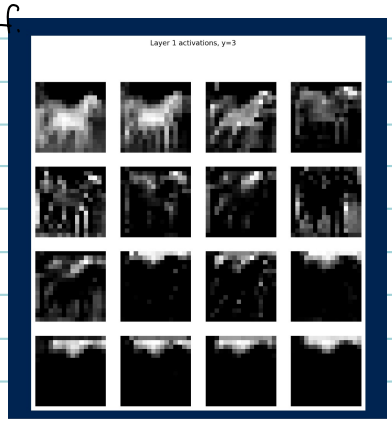


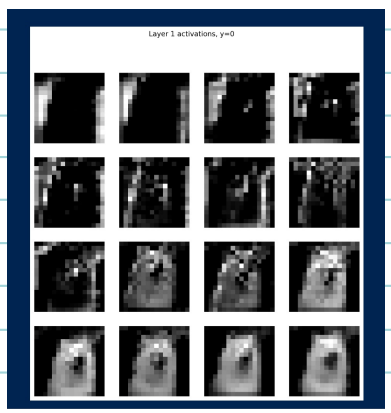

i. One source of noise may be that after every iteration, we use a different mini-batch from the training set, which can cause the training loss to fluctuate due to the various examples/gradients per batch. Another source of noise can come from the learning rate updater because of the noise from gradient estimation for each batch leads to fluctuation in the training and validation metrics.

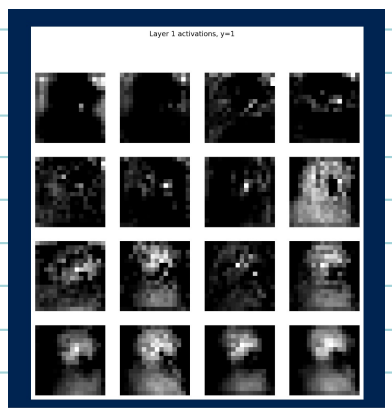
ii. I believe the training loss would continue to decrease, or stay at very low values. As for validation loss, it would decrease for a short while, but then it would start to overfit the data, causing validation loss to increase.

iii. I believe when epoch = 24 because it had the lowest validation loss, with the highest validation accuracy.

f.




i. Certain activation maps highlight edges or outlines, which shows strong contrast. So for class-specific differences, the filters light more around the samoyed because of it's fur, but for the Great Dane and poodle, it may not highlight the same areas. But they all share the same filters, so if they were similar you would get similar outputs.

ii. I would say edges and contours, directional/orientational features, color gradients, as maybe texture patterns.

# Q3: Challenge write-up

- Regularization: I tried to implement weight decay and dropout, but my validation accuracy dropped below 50%, so I decided to leave it out.
- Feature selection: I decided to use the same 5 classes from Q2, and I kept the raw images resized to 32x32
- Model Architecture: I used the same architecture from Q2 (3 conv layers, the fully connected filters, and for initialization I did random normal). I decided to focus more on hyperparameter tuning, and extended training.
- Hyperparameter: I decided to use a smaller LR (1e-5), I kept the batch size the same, I started with 15 epochs then incremented by 5 all the way to 40 to see how the validation accuracy was affected, for the optimizer, I used adam with no weight decay, and finally I decided to use cross entropy loss.
- Model evaluation: Epoch 39 yielded the best results, with a validation accuracy of 57.14%.
- Utilization of Additional data: No extra data was used (5 classes), and I did not incorporate augmentation.
- GPU/Pretrained models: I did not use any pretrained models, and I used the original GPU