Computer Vision - EE 00046746
HW1

Tomer Raz Adi Hatav

July 2024

Part 1: Classic Classifier

1 - The CIFAR10 Dataset

The CIFAR10 dataset contains 60,000 32x32-pixel RGB images split evenly between the following classes: plane, car, bird, cat, deer, dog, frog, horse, ship, truck.

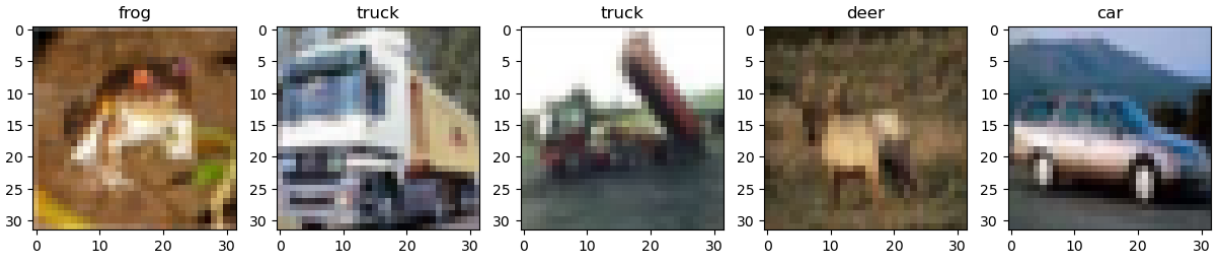

Figure 1: A sample of five images from the CIFAR10 dataset.

2, 3 & 4 - KNN Classifiers

For $K = 10$ a KNN classifier "trained" on 10,000 flattened images yielded an accuracy of 29% over 1,000 test images.

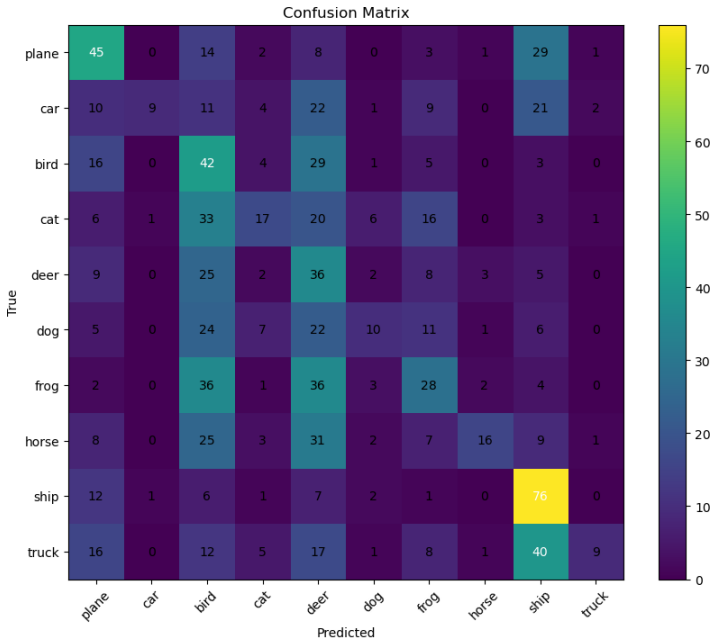

Figure 2: Results from the $K = 10$ KNN classifier evaluated on 1,000 test images, as a confusion matrix.

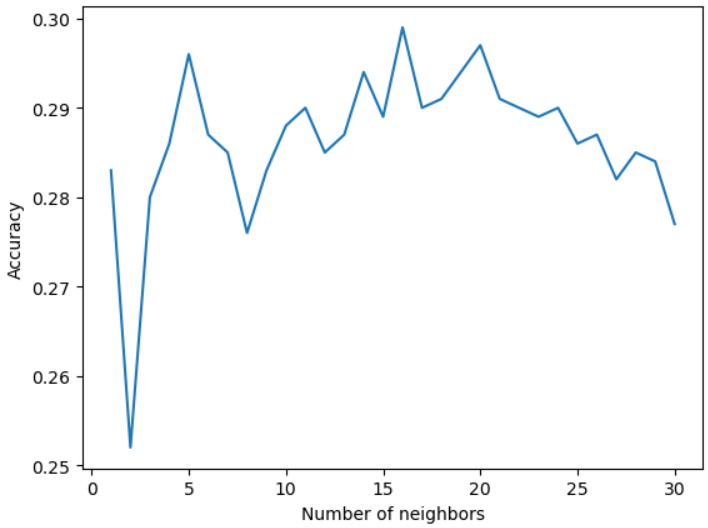

Figure 3: Accuracy of KNN models as a function of K, the number of neighbors. Although the variation in accuracy is relatively small (compared to 29%), the trade-off between expressiveness and over-fitting is evident: for low values of K the model relatively over-fits, leading to worse generalization on the test set than slightly higher Ks. In this case outlying train samples have too great a say when classifying images close to them. For high values of K, the model under-fits, losing expressiveness. This leads to the existence of an optimal K for accuracy, around 16.

Part 2: Design and Build CNN Classifier

1 SvhnCNN Benchmark

Training of the model was conducted with the hyperparameters specified in the exercise: epochs (20), batch size (128), learning rate ($= 10^{-4}$), optimizer (Adam), loss function (cross entropy loss). Additionally, the pytorch seed was set randomly to 123456 for reproducibility.[1] The resulting model achieves an accuracy of 78.43% on the test set.

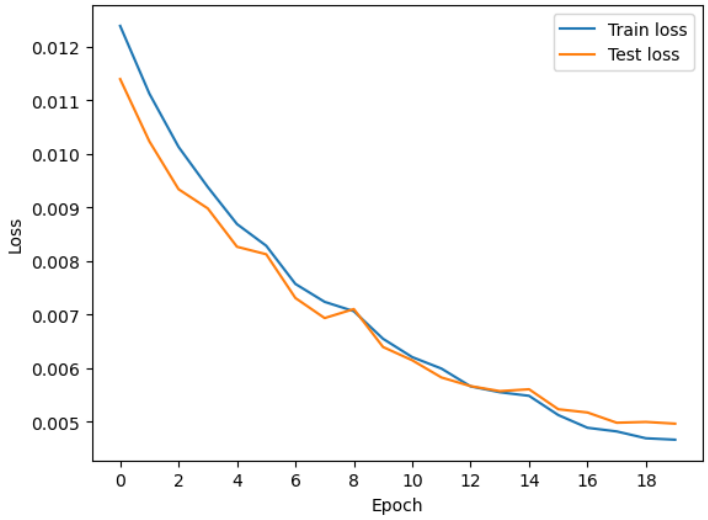

Figure 4: SvhnCNN losses during training.

The top five most confusing pairs (see fig. 6):

1. Cat - mistaken to be dog $149/1000 = 14.9\%$ of the time.
2. Dog - mistaken to be cat $132/1000 = 13.2\%$ of the time.

[1]The first time we ran all of our code, we did not set the pytorch seed. Our model performed similarly to how it performs given this seed, and the SvhnCNN performed as it does here (give or take 1% accuracy). After returning for our final check of the code, we found our model was harder to train and consistently yielded a train accuracy around 9%. This necessitated a search for random seed that yields results similar to the ones we had first viewed, and the first random seed that obeyed this was taken. The SvhnCNN model was rerun under this seed for reproducibility and consistency. No optimization was done to ensure our model outperforms the Svhn model given a set seed.

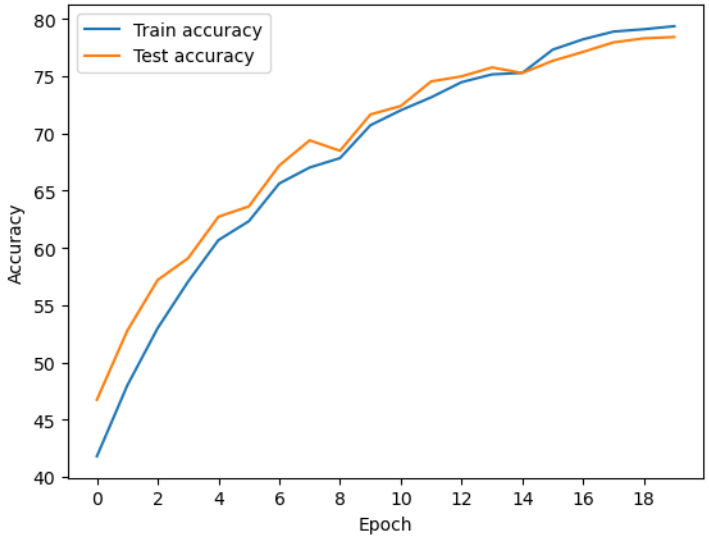

Figure 5: SvhnCNN accuracies during training.

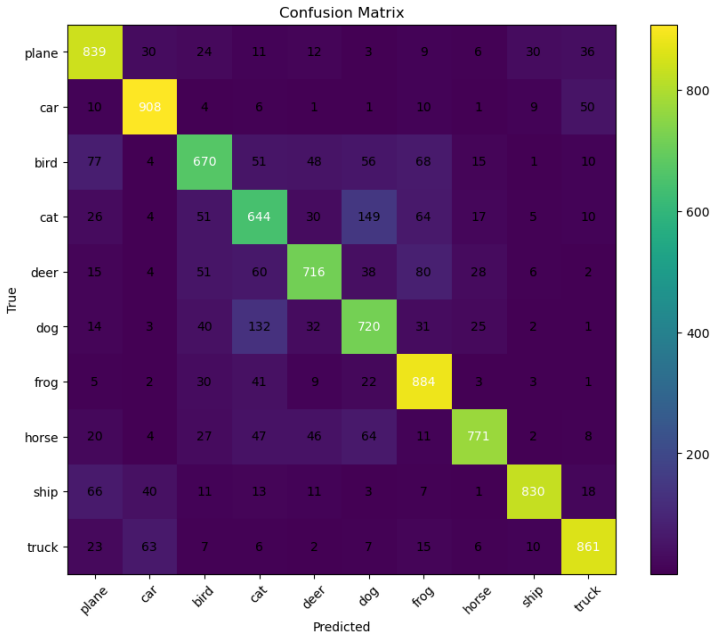

Figure 6: SvhnCNN confusion matrix (test set).

3. Deer - mistaken to be frog $80/1000 = 8\%$ of the time.
4. Bird - mistaken to be plane (how ironic!) $77/1000 = 7.7\%$ of the time.
5. Bird - mistaken to be frog $68/1000 = 6.8\%$ of the time.

2 & 3 - Our Model

Our model (OurCNN1 class) has 9,561,866 parameters as opposed to the SvhnCNN class which has only 4,482,058 parameters. The input dimension is 32x32x3 (32x32 pixels and 3 color channels) and the output dimension is 10 (for the 10 classes of the CIFAR10 dataset). We chose to build off of the SvhnCNN class, keeping the kernel= 3, stride= 1, padding= 1 scheme and adding a convolution block which doubles the final number of channels from 128 to 256. This implies the output vector (flattened final channels, concatenated) is twice as long (ours has $2 \cdot 8192$ entries). The model has 26 layers in total. There are 4 convolution blocks in total with 2 convolution layers per block for a total of 8 convolution layers. The fully connected net for classification of the feature vector was left with 2 linear layers. Maxpooling, batchnorm, dropout, and activation (LeakyReLU) layers make up another 16 layers. We Changed the activation from ReLU to LeakyReLU with the default slope parameter of 10^{-2}, to possibly facilitate better propagation of gradients (no attempt was made to quantify the change).

After training for 20 epochs using the same hyperparameters as those with which the original SvhnCNN model was trained, our model reached a test set accuracy of 84.77%. We conclude that increasing model size can amplify performance.

An attempt was made to improve the model using a train-validation split of the original train set. This was abandoned due to poor performance for our model (it was later understood that the poor performance stemmed from unlucky random seeds during optimization). The train-validation split did, likewise, not improve the SvhnCNN model. This was a test to see if the direction was worth pursuing for optimizing our model and we concluded that there was no need.

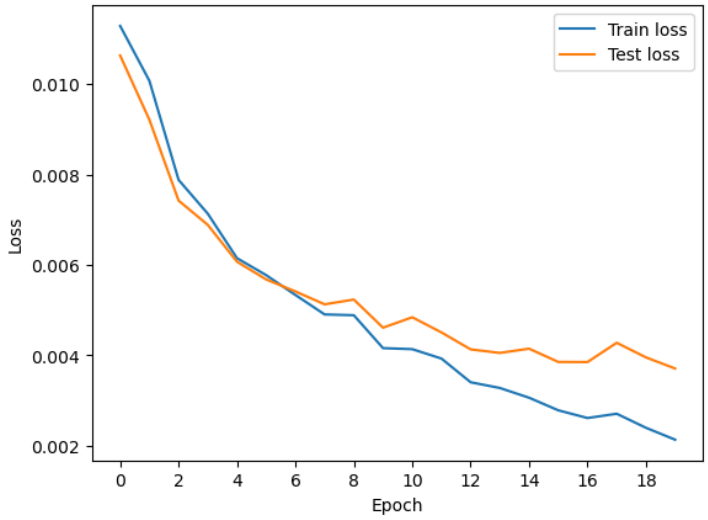

Figure 7: Train set and test set losses as a function of epoch for our CNN (OurCNN1 class).

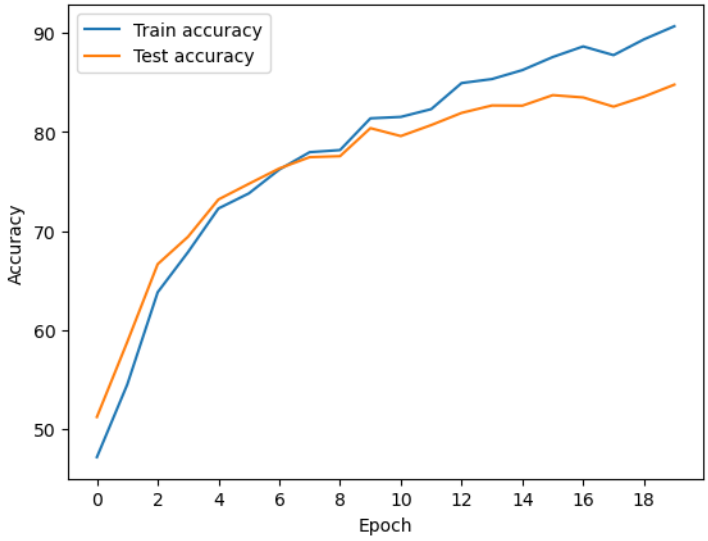

Figure 8: Train set and test set accuracies as a function of epoch for our CNN (OurCNN1 class).

'Cat' and 'dog' are the most frequently confused classes. Interestingly, the confusion between the two is roughly symmetric (149 pictures of cats were classified as pictures of dogs and 132 pictures of dogs were classified as pictures of cats).

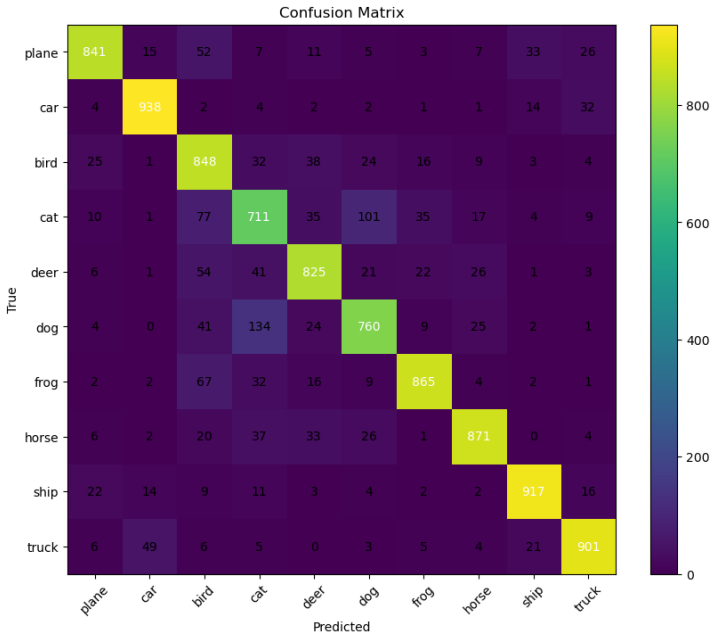

Figure 9: Confusion matrix describing our model's performance on the test set. Off-diagonal numbers are generally smaller than those found in fig. 6.

Part 3: Foundation Models (CLIP)

Note: test images were not loaded in the first two subsections (except for the clustering of the test images, in which only the test images appear).

1 Zero-shot Clustering

The CLIP embeddings (dimension-reduced using T-SNE) adequately cluster images of dogs and cats into two groups. This is evidence showing that CLIP embeddings capture semantics.

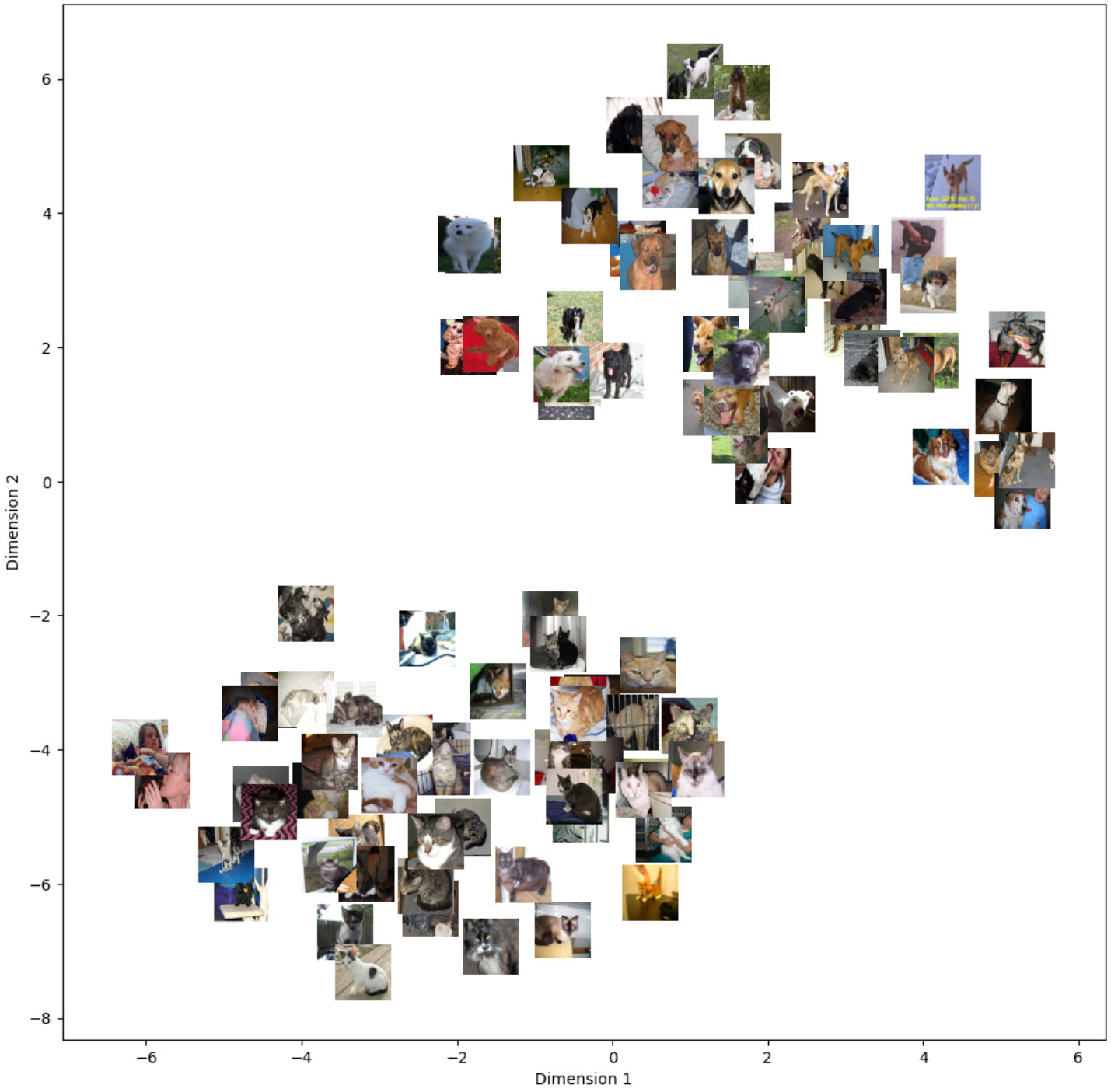

Figure 10: Images of cats and dogs plotted at their respective T-SNE-dimension-reduced CLIP embeddings. Semantically similar pictures are grouped together: the two clusters are clearly 'dogs' and 'cats'. These images may have been seen by CLIP during training (they are not from the test set).

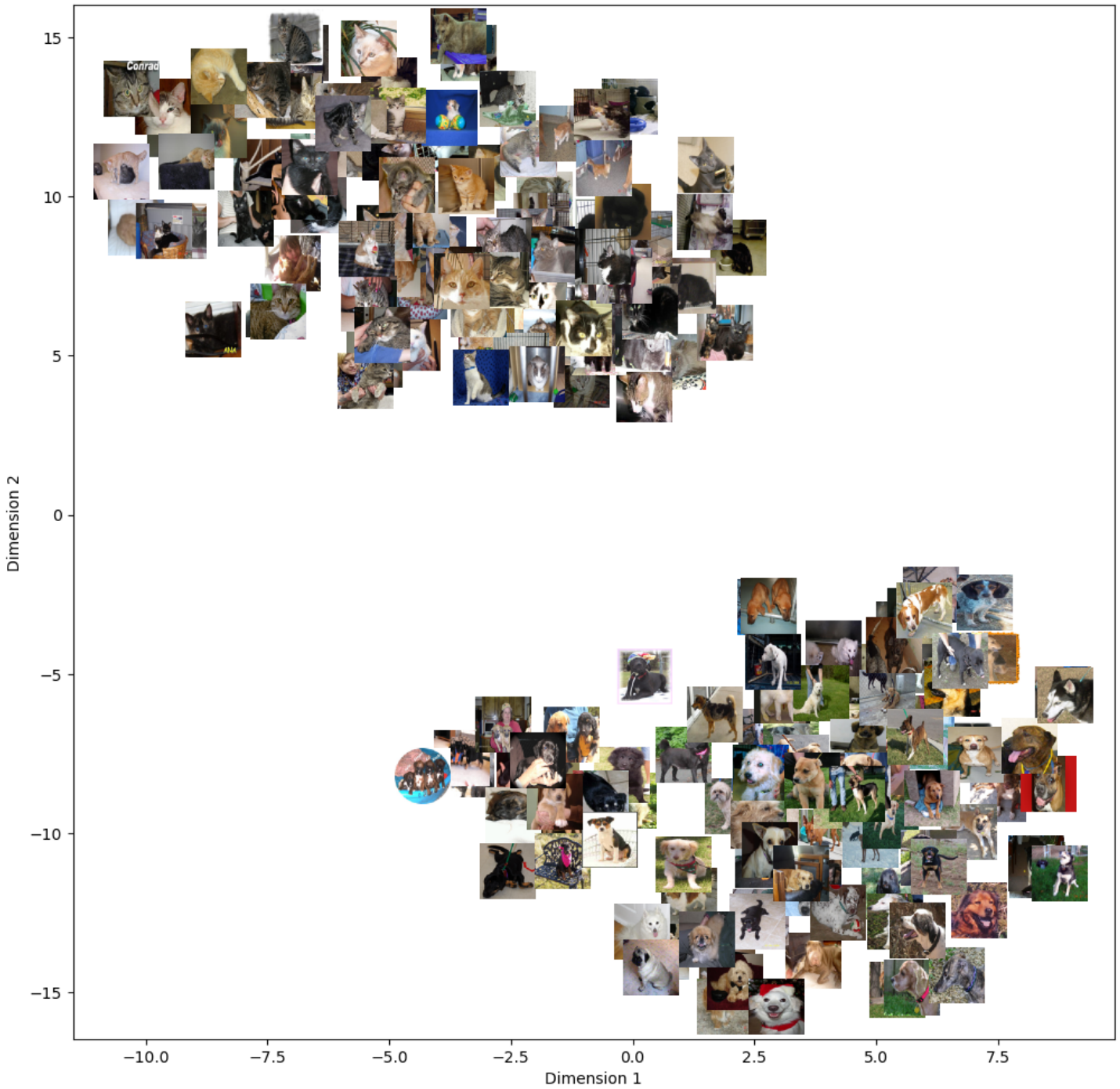

Figure 11: Same as above (fig. 10), but for test images. Embedding fidelity holds even for test images (we interpreted 'test' as unseen by CLIP during training but this may be wrong).

2 - Zero-shot Information Retrieval

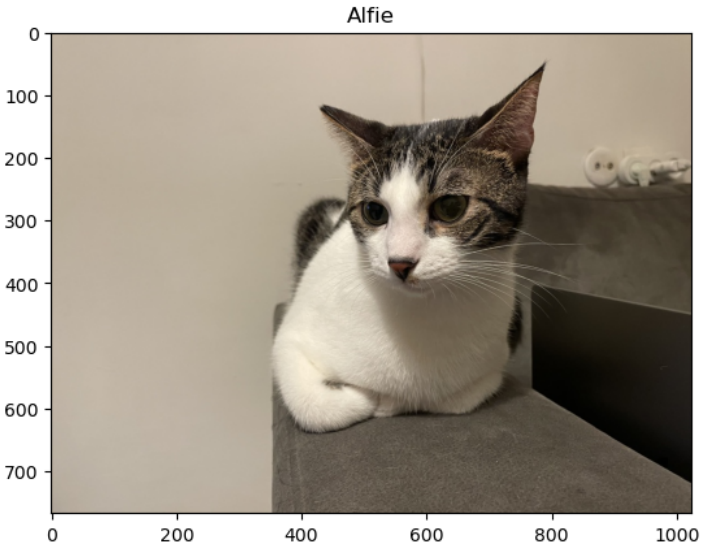

Figure 12: Alfie (cat).

The 5 images most similar (cosine-similarity) to Alfie are shown in fig. 13.

Figure 13: Alfie's buddies (top 5 cosine-similarity out of a joint set of cat and dog images).

This result makes sense: CLIP was trained to maximize the similarity between image and respective text embeddings. Since semantically similar images will have similar descriptions, their text embeddings will also be similar and hence the images will also have similar embeddings.

Here is a schematic explaining the logic behind: Semantically similar images ⇒ similar CLIP embeddings

$$\begin{array}{ccccccccc} & & \text{text1} & \longrightarrow & \text{text1\_embedding} & \overset{\text{CLIP training}}{\longleftrightarrow} & \text{image1\_embedding} & \longleftarrow & \text{image1} \\ \text{implies similar description} & \Downarrow & \updownarrow & & \Longleftarrow & & \Longleftarrow & & \updownarrow \text{ Semantically similar} \\ & & \text{text2} & \longrightarrow & \text{text\_embedding2} & \underset{\text{CLIP training}}{\longleftrightarrow} & \text{image2\_embedding} & \longleftarrow & \text{image2} \end{array}$$

$$\Longrightarrow \text{ Similar text endodings } \Longrightarrow \text{ Similar text embeddings } \Longrightarrow \text{ Similar image embeddings}$$

Perhaps the other direction can be explained like this too: Similar image embeddings ⇒ semantically similar images. Of course this is heuristic logic. In reality the embedding space will undoubtedly have adversarial examples.

3 - Zero-shot Image Classification

Pipeline: embed images, embed class names (text). For each embedded image, predict class by finding the embedded class with highest cosine similarity. The class names in our case: ['dog, 'cat']. This method works without exception on the test data supplied (see fig. 14).

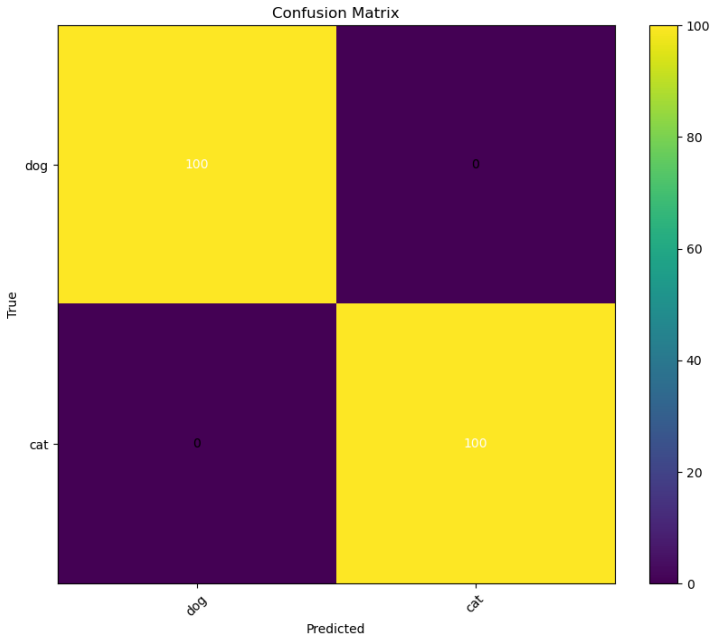

Figure 14: The zero-shot method described above does not fail on the test data.

- What are the strengths of the approach (using clip embeddings for zero-shot)?

  The strengths of the approach using clip embeddings for zero-shot are that CLIP embeddings leverage a powerful model that has been pretrained on a vast amount of image-text pairs. This allows it to create rich, meaningful representations of images and text without the need for extensive task-specific training.

- What are the weaknesses? Not task specific? room for improvement?

  One weaknesses of the approach is that CLIP provides a strong general representation, it might not perform as well on very specialized tasks without fine-tuning. The embeddings are designed to be general-purpose, which can be a limitation when very specific nuances are required like in our task that was distinguish between cats and dogs.

- If the folder contained also images of (not cats and not dogs) how would we classify into cats, dogs, other?

  Instead of just using "cat" and "dog" as labels, enhance the prompts by including descriptive contexts. For example, using prompts like "This is a picture of a pet cat" or "This is a picture of a pet dog" could provide additional contextual information that might help the model differentiate between pets and other objects. After changing the prompt, we will define a threshold that if a picture not pass this threshold for this two prompts it will be classify as other.

4 - Zero-shot Instance Counting

Teaching CLIP to count! Not really, zero-shot again. The riddle here can be phrased: what prompt form is the best for describing to CLIP an image with N cats, so it can classify the image properly? We tried 16 different schemes, from most naive to those that are recommended by Radford and Kim et al. (Learning Transferable Visual Models From Natural Language Supervision, i.e. the CLIP paper). Here is a list of our attempts and the percentage of accuracy they received on a group of 11 images that we tagged ourselves.

1. 'number_word'+'cats": 0.7272727272727273
2. 'written_number': 0.36363636363636365
3. 'number': 0.09090909090909091
4. 'a photo of number_word cats': 0.5454545454545454
5. 'a photo with number_word cats': 0.18181818181818182
6. 'a photo of number_word cats, a type of pet': 0.5454545454545454
7. 'a photo with number_word cats, a type of pet': 0.18181818181818182
8. 'a photo of number_word cats, a pet/pets': 0.5454545454545454

9. 'a photo with number_word cats, a pet/pets': 0.18181818181818182
10. 'number_word'+'kittens": 0.8181818181818182
11. 'a photo of number_word kittens': 0.8181818181818182
12. 'a photo with number_word kittens': 0.45454545454545453
13. 'a photo of number_word kittens, a type of pet': 0.8181818181818182
14. 'a photo with number_word kittens, a type of pet': 0.45454545454545453
15. 'a photo of number_word kittens, a pet/pets': 0.6363636363636364
16. 'a photo with number_word kittens, a pet/pets': 0.36363636363636365

The tips in the CLIP paper seem to be best performing. Based on these results, we decided to write our cat counting function with the string scheme: 'a photo of number_word kittens'. It is the most compact string scheme out of the best performing ones (does not include the 'a type of pet' suffix). 'number_of_cats' is the name of our function.

Part 4: Dry Questions

4.1 & 4.2

Pros and Cons of Bag of Words Algorithm

Pros:

1. **Simplicity**: The Bag of Words algorithm is straightforward to implement and understand. It converts images into a collection of visual words, making it easy to process.
2. **Efficiency**: Bag of Words can be computationally efficient, as it can describe with a small amount of information significant characteristics of the flag, such as colors, structure, shapes, etc.

Cons:

1. **Loss of Spatial Information**: Bag of Words disregards the spatial arrangement of features within an image. For flags, the arrangement of colors and patterns is crucial for correct identification, and ignoring this can lead to incorrect classifications.
2. **Fixed Vocabulary**: The effectiveness of Bag of Words depends on the predefined set of visual words. If the vocabulary is not comprehensive enough, important features might be missed, leading to reduced accuracy.
3. **Feature Overlap**: Different flags might share similar colors and patterns, leading to overlapping visual words that can confuse the classifier.

4.2. Danny is interested to design a dogs classifier. He has dataset of RGB images of 3 types of dogs in dimensions of 64X64 (i.e. the dimension of each image is 3X64X64). In the following table, the network architecture is defined in the leftmost column. You need to fill in the two additional columns: In the middle column fill the output dimensions and in the right column the number of learnd parameters (weights). The answer should be written as multiples (for example: 128 × 128 × 3).No need to consider bias.

Conv7-N: A convolution layer with N neurons, each one has dimensions of 7 × 7 × D where D is the volume depth activation in the previous layer. Padding = 2 and Stride = 1.

POOL2: 2 × 2 Max-Pooling activation with Stride=2. If the input to the layer is an odd number, round it down.

FC-N: Fully Connected layer with N neurons.

Layer	Output dimensions	Number of parameters (weights)
$INPUT$	$64 \times 64 \times 3$	0
$CONV7-16$	$62 \times 62 \times 16$	$16 \times (7 \times 7) \times 3$
$POOL2$	$31 \times 31 \times 16$	0
$CONV7-32$	$29 \times 29 \times 32$	$32 \times (7 \times 7) \times 16$
$POOL2$	$14 \times 14 \times 32$	0
$FC-3$	3	$14 \times 14 \times 32 \times 3$

4.3

3. You have built a neural network model for a classification problem and observed that it is overfitting.

a. For each of the following factors, determine if it could be contributing to the overfitting, and explain how. If it is a contributing factor, suggest a method to mitigate its impact.

1. **Learning Rate (Lr)**
   - **Contribution to Overfitting**: A very low learning rate can lead to overfitting because the model might learn the noise in the training data as it converges very slowly or to stuck in local minimum of the train set.
   - **Mitigation Method**: Use a moderate learning rate. Use techniques such as learning rate decay or adaptive learning rate optimizers like Adam.
2. **Batch Size**
   - **Contribution to Overfitting**: Very small batch sizes can result in noisy gradient estimates, which can lead to overfitting.
   - **Mitigation Method**: Increase the batch size. However, balance is key as very large batch sizes can also cause the model to miss finer details.
3. **Number of Layers**
   - **Contribution to Overfitting**: Having too many layers can make the model too complex, causing it to fit the training data very closely.
   - **Mitigation Method**: Reduce the number of layers or use techniques like regularization (L2 or L1), dropout, or early stopping.
4. **Training Set Size**
   - **Contribution to Overfitting**: A small training set size can lead to overfitting because the model has fewer examples to generalize from.
   - **Mitigation Method**: Increase the training set size by collecting more data or using data augmentation techniques.
5. **Test Set Size**
   - **Contribution to Overfitting**: The test set size doesn't directly contribute to overfitting, but a small test set size can give unreliable estimates of the model's performance.
   - **Mitigation Method**: Ensure a sufficiently large test set to reliably evaluate the model's generalization performance.

b. After conducting multiple trials with your model, you decided to add dropout to it. Describe the role of dropout during training and testing, and explain how it helps in reducing overfitting.

- **During Training**: Dropout forces the network not to count just on few bold neurons, by randomly dropping neurones in train time, it forces the network to learn more connections. That helps in making the model more robust and prevents it from relying too heavily on any individual node.
- **During Testing**: Dropout is not applied. Instead, the weights are scaled down by the dropout rate to account for the missing units during training.

A CNN Training Logs

```
torch.cuda.empty_cache()
torch.manual_seed(123456) # set the seed for reproducibility (this is the seed chosen when training our model, see below)
model = SvhnCNN()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
criterion = nn.CrossEntropyLoss()

train_losses, test_losses, train_accs, test_accs = train(model, trainloader, testloader, 20, optimizer, criterion)
```

✓ 19m 26.1s

```
Device: cuda:0
Epoch: 1, Train loss (over epoch): 0.014075463802814483, Train loss: 0.012391215410232545, Test loss: 0.01139765965938568, Train accuracy: 41.798, Test accuracy: 46.74
Epoch: 2, Train loss (over epoch): 0.011904482007026672, Train loss: 0.011126129839420318, Test loss: 0.010234904992580413, Train accuracy: 47.986, Test accuracy: 52.77
Epoch: 3, Train loss (over epoch): 0.01089143236875534, Train loss: 0.010130898456573487, Test loss: 0.009340470385551452, Train accuracy: 52.994, Test accuracy: 57.2
Epoch: 4, Train loss (over epoch): 0.010050793535709382, Train loss: 0.009381375069618225, Test loss: 0.008981908077001572, Train accuracy: 57.058, Test accuracy: 59.1
Epoch: 5, Train loss (over epoch): 0.009351691701412201, Train loss: 0.008689767236709594, Test loss: 0.008265769058465957, Train accuracy: 60.686, Test accuracy: 62.73
Epoch: 6, Train loss (over epoch): 0.008617190185785294, Train loss: 0.008284746825695037, Test loss: 0.00812586226463318, Train accuracy: 62.336, Test accuracy: 63.63
Epoch: 7, Train loss (over epoch): 0.008091165947914123, Train loss: 0.007571462563276291, Test loss: 0.007308255487680436, Train accuracy: 65.63, Test accuracy: 67.2
Epoch: 8, Train loss (over epoch): 0.007614875855445862, Train loss: 0.007236585049629212, Test loss: 0.006933406955003738, Train accuracy: 67.026, Test accuracy: 69.4
Epoch: 9, Train loss (over epoch): 0.007331978977918625, Train loss: 0.007061827282905579, Test loss: 0.007101905387639995, Train accuracy: 67.848, Test accuracy: 68.49
Epoch: 10, Train loss (over epoch): 0.006988525257110596, Train loss: 0.006546813433170318 5, Test loss: 0.006393248981237411 4, Train accuracy: 70.72, Test accuracy: 71.66
Epoch: 11, Train loss (over epoch): 0.006730930464267731, Train loss: 0.006204070411920548, Test loss: 0.006145097839832306, Train accuracy: 72.028, Test accuracy: 72.4
Epoch: 12, Train loss (over epoch): 0.006430001771450043, Train loss: 0.005989896121025085 5, Test loss: 0.005823911547660827 5, Train accuracy: 73.16, Test accuracy: 74.55
Epoch: 13, Train loss (over epoch): 0.006212407227754593, Train loss: 0.005657234326004982, Test loss: 0.005662323206663131, Train accuracy: 74.484, Test accuracy: 74.99
Epoch: 14, Train loss (over epoch): 0.006004824472665786 5, Train loss: 0.005547919863462448, Test loss: 0.005568601045012474, Train accuracy: 75.162, Test accuracy: 75.77
Epoch: 15, Train loss (over epoch): 0.005780090025067329, Train loss: 0.005481039623022079, Test loss: 0.005603640750050545, Train accuracy: 75.294, Test accuracy: 75.28
Epoch: 16, Train loss (over epoch): 0.005610546870231628, Train loss: 0.005121956030726433, Test loss: 0.005230328327417374, Train accuracy: 77.326, Test accuracy: 76.35
Epoch: 17, Train loss (over epoch): 0.00542282614827156, Train loss: 0.00488418031513691, Test loss: 0.005169665256142616, Train accuracy: 78.228, Test accuracy: 77.13
Epoch: 18, Train loss (over epoch): 0.005282086310596465, Train loss: 0.004816000996232033, Test loss: 0.004978297945857048, Train accuracy: 78.902, Test accuracy: 77.96
Epoch: 19, Train loss (over epoch): 0.00508401025891304, Train loss: 0.004688096504807472, Test loss: 0.004993175688385963 6, Train accuracy: 79.108, Test accuracy: 78.31
Epoch: 20, Train loss (over epoch): 0.004993053911924362, Train loss: 0.004661495271921158, Test loss: 0.00496123925447464, Train accuracy: 79.374, Test accuracy: 78.43
```

Figure 15: Training log for the SvhnCNN class.

```
trainset, trainloader, testset, testloader = load_CIFAR10(batch_size=128, trainset_path='trainset.pt')
model, random_seed, train_losses, test_losses, train_accs, test_accs = train_with_random_seed_optimization(OurCNN1, trainloader, testloader, 20, optimizer, criterion)
✓ 23m 29.5s

Device: cuda:0
Random seed: 123456
Epoch: 1, Train loss (over epoch): 0.013226683311462402, Train loss: 0.011284005970954894, Test loss: 0.010629777443408966, Train accuracy: 47.166, Test accuracy: 51.22
Epoch: 2, Train loss (over epoch): 0.010499546395540238, Train loss: 0.010071092171669006, Test loss: 0.009210294222831726, Train accuracy: 54.502, Test accuracy: 58.8
Epoch: 3, Train loss (over epoch): 0.008841911625862122, Train loss: 0.00788011852145195, Test loss: 0.0074227617084980015, Train accuracy: 63.81, Test accuracy: 66.65
Epoch: 4, Train loss (over epoch): 0.007721492632627487, Train loss: 0.007132147799730301, Test loss: 0.006885045909881592, Train accuracy: 67.872, Test accuracy: 69.42
Epoch: 5, Train loss (over epoch): 0.00681030002951622, Train loss: 0.006148515658378601, Test loss: 0.006070208978652954, Train accuracy: 72.27, Test accuracy: 73.18
Epoch: 6, Train loss (over epoch): 0.006205133535861969, Train loss: 0.005773079621195793, Test loss: 0.005679015272855759, Train accuracy: 73.794, Test accuracy: 74.76
Epoch: 7, Train loss (over epoch): 0.005715915142893791, Train loss: 0.005335211083292961, Test loss: 0.005412734723091126, Train accuracy: 76.194, Test accuracy: 76.31
Epoch: 8, Train loss (over epoch): 0.0052840104067325595, Train loss: 0.004904554026126861, Test loss: 0.005128110286593437, Train accuracy: 77.954, Test accuracy: 77.45
Epoch: 9, Train loss (over epoch): 0.00493411022245884, Train loss: 0.004887150691747665, Test loss: 0.005233823677897453, Train accuracy: 78.166, Test accuracy: 77.55
Epoch: 10, Train loss (over epoch): 0.004639758349657058, Train loss: 0.004159690090417862, Test loss: 0.004611888164281845, Train accuracy: 81.37, Test accuracy: 80.38
Epoch: 11, Train loss (over epoch): 0.004340579578876495, Train loss: 0.004137059115767479, Test loss: 0.004842168000340462, Train accuracy: 81.516, Test accuracy: 79.58
Epoch: 12, Train loss (over epoch): 0.004065601735711097, Train loss: 0.003929072245955467, Test loss: 0.004507243403792382, Train accuracy: 82.298, Test accuracy: 80.69
Epoch: 13, Train loss (over epoch): 0.0038225915825366974, Train loss: 0.003404914931058884, Test loss: 0.0041318033576011656, Train accuracy: 84.932, Test accuracy: 81.92
Epoch: 14, Train loss (over epoch): 0.00365757053822279, Train loss: 0.0032784652742743493, Test loss: 0.004053719803690911, Train accuracy: 85.336, Test accuracy: 82.67
Epoch: 15, Train loss (over epoch): 0.0034272693538665772, Train loss: 0.0030634182155132293, Test loss: 0.004147730660438538, Train accuracy: 86.24, Test accuracy: 82.65
Epoch: 16, Train loss (over epoch): 0.0032270016396045683, Train loss: 0.0027848427802324295, Test loss: 0.00385415278673172, Train accuracy: 87.56, Test accuracy: 83.71
Epoch: 17, Train loss (over epoch): 0.003041237519085407, Train loss: 0.002614236972928047, Test loss: 0.0038516644388437273, Train accuracy: 88.63, Test accuracy: 83.48
Epoch: 18, Train loss (over epoch): 0.0028824976697564126, Train loss: 0.002708662356734276, Test loss: 0.004277992257475853, Train accuracy: 87.752, Test accuracy: 82.55
Epoch: 19, Train loss (over epoch): 0.0027533616781234742, Train loss: 0.002399026725888252, Test loss: 0.003955448362231255, Train accuracy: 89.346, Test accuracy: 83.56
Epoch: 20, Train loss (over epoch): 0.0025444411953985691, Train loss: 0.0021350882449746133, Test loss: 0.003708437475562095, Train accuracy: 90.668, Test accuracy: 84.77
```

Figure 16: Training log for our CNN class (OurCNN1 class).

B OurCNN1 Model Architecture

1. `nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, padding=1)`
   % Conv Layer block 1
2. `nn.BatchNorm2d(16)`
3. `nn.LeakyReLU(inplace=True)`
4. `nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, padding=1)`
5. `nn.LeakyReLU(inplace=True)`
   % Conv Layer block 2
6. `nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)` *% What are the dims after this layer? % How many weights?*
7. `nn.BatchNorm2d(64)`
8. `nn.LeakyReLU(inplace=True)`
9. `nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1)`
10. `nn.LeakyReLU(inplace=True)`
11. `nn.MaxPool2d(kernel_size=2, stride=2)`
    % Conv Layer block 3
12. `nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)`
13. `nn.BatchNorm2d(128)`
14. `nn.LeakyReLU(inplace=True)`
15. `nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, padding=1)`
16. `nn.LeakyReLU(inplace=True)`
17. `nn.MaxPool2d(kernel_size=2, stride=2)`
18. `nn.Dropout2d(p=0.05)` *% Why is this here?*
    % Modified Conv Layer block 4
19. `nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1)`

20. `nn.LeakyReLU(inplace=True)`
21. `nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, padding=1)`
22. `nn.LeakyReLU(inplace=True)`

For the classifying head that receives the flattened, concatenated features of the CNN and outputs class scores:

1. `nn.Dropout(p=0.1)`
2. `nn.Linear(2*8192, 512)`
3. `nn.LeakyReLU(inplace=True)`
4. `nn.Linear(512, 10)`