Mahmut Alp Erkent
2019742009

CMPE 597 Sp. Tp. Deep Learning
Spring 2021 Project II

Answers

1. **(15 pts) Implement data augmentation techniques, such as random flip and random crop. Do not forget to standardize or normalize your dataset before training.**

   I looked online for frequently used augmentation methods on CIFAR10 data [4] and implemented random horizontal flipping and random cropping on the training data. I first used random cropping by adding 4 pixels to the borders of the image and cropping it for the same output size of 32 x 32 pixels. I did not want to increase padding because these images were already scaled and reviewed by the people who composed the dataset, so if I increased padding, I could lose crucial features of images towards the edges. Then I added random horizontal flipping with probability of 0.5. Therefore, at each epoch, the network will use slightly different images, which should reduce overfitting. Here are some sample images with respective labels:

dog bird cat truck horse bird bird ship

Here is the performance difference with or without data augmentation:

augmentation+
Training time: ~16 min | Best performance reached at epoch: 31
Training accuracy: 86.60%
Validation accuracy: 82.41%
Test accuracy: 81.97%

---

augmentation-
Training time: ~6 min | Best performance reached at epoch: 16
Training accuracy: 95.22%
Validation accuracy: 77.34%
Test accuracy: 76.07%

Here is the code for transformations:

```
means = [trainset[:, :, :, 0].mean()/255,
         trainset[:, :, :, 1].mean()/255,
         trainset[:, :, :, 2].mean()/255]

stds = [trainset[:, :, :, 0].std()/255,
        trainset[:, :, :, 1].std()/255,
        trainset[:, :, :, 2].std()/255]

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(means, stds)])
```

```
transform_train = transforms.Compose([
    transforms.ToPILImage(),
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(means, stds)])
```

2. **(30 pts) Design a 3-layer CNN architecture for object classification task. A 3-layer CNN may not give you the state-of-the-art performance on CIFAR10 data. You will explore ways to obtain as high performance as possible with the 3-layer CNN.**

   Here is the final network architecture:

```
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 32, 3, 1, 1)
        self.bn1 = nn.BatchNorm2d(32)
        self.pool = nn.AvgPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 64, 3, 1, 1)
        self.bn2 = nn.BatchNorm2d(64)
        self.conv3 = nn.Conv2d(64, 128, 3, 1, 1)
        self.bn3 = nn.BatchNorm2d(128)
        self.fc1 = nn.Linear(128*4*4, 512)
        self.bn4 = nn.BatchNorm1d(512)
        self.fc2 = nn.Linear(512, 128)
        self.bn5 = nn.BatchNorm1d(128)
        self.fc3 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.bn1(self.conv1(x))))
        # (N, 3, 32, 32)  -> (N, 32, 16, 16)
        x = self.pool(F.relu(self.bn2(self.conv2(x))))
        # (N, 32, 16, 16) -> (N, 64, 8, 8)
        x = self.pool(F.relu(self.bn3(self.conv3(x))))
        # (N, 64, 8, 8)   -> (N, 128, 4, 4)
        x = torch.flatten(x, 1)
        # (N, 128, 4, 4)  -> (N, 2048)
        x = F.relu(self.bn4(self.fc1(x)))
        # (N, 2048)       -> (N, 512)
        x = F.relu(self.bn5(self.fc2(x)))
        # (N, 512)        -> (N, 128)
        x = self.fc3(x)
        # (N, 128)        -> (N, 10)
        return x
```

- **Discuss the reasoning behind your choices on the kernel size, activation functions, number of feature maps, number of fully connected layers, and the size of the fully connected layers.**

  - I tried a few configurations but decided to use kernel size as 3 for all layers with 1 padding on each border, so that the image size stays the same after the convolution operation.

kernels: 7x7 | 5x5 | 3x3
Training time: ~8 min | Best performance reached at epoch: 18
Training accuracy: 84.25%
Validation accuracy: 80.59%
Test accuracy: 80.03%

---

kernels: 5x5 | 3x3 | 3x3
Training time: ~6 min | Best performance reached at epoch: 17
Training accuracy: 84.87%
Validation accuracy: 81.75%
Test accuracy: 81.05%

---

kernels: 3x3 | 3x3 | 3x3
Training time: ~10.5 min | Best performance reached at epoch: 28
Training accuracy: 89.60%
Validation accuracy: 83.37%
Test accuracy: 83.00%

- I used ReLU activation function in all layers because it helps with computations and it does not suffer vanishing/exploding gradients problem like tanh or sigmoid activation functions. I also tried leaky ReLU but could not see a discernable difference.

- I increased number of channels from 3 to 32 in the first convolutional layer, to 64 in the second and to 128 in the third. I did this so that the network could try to learn many different features of the images.

- After applying average pooling to every convolutional layer and flattening the tensor, I had 2048 units in the first fully connected layer. I used three FC layers to decrease the number of features from 2048 to 512, 128 and finally 10. I assumed three FC layers would be enough to classify the features learned from three convolutional layers.

- **Investigate techniques to improve the performance, such as adding dropout, batch normalization, residual blocks. Discuss your observations on the effect of these techniques on the performance.**

    - Adding batch normalization to every layer except the last improved training time and all training, validation, and test accuracies. Therefore, I batch normalization in my model.

      batchnorm+
      Training time: ~15 min | Best performance reached at epoch: 34
      Training accuracy: 84.85%
      Validation accuracy: 81.07%
      Test accuracy: 80.43%

      ---

      batchnorm-
      Training time: ~18 min | Best performance reached at epoch: 41
      Training accuracy: 79.49%
      Validation accuracy: 79.72%
      Test accuracy: 79.10%

    - Adding dropout with 0.1 probability improved training time, but worsened all training, validation, and test accuracies. Since the network did not overfit without it, I thought it was unnecessary to use dropout in the model.

dropout = 0.10
Training time: ~15 min | Best performance reached at epoch: 34
Training accuracy: 84.85%
Validation accuracy: 81.07%
Test accuracy: 80.43%

---

dropout-
Training time: ~18 min | Best performance reached at epoch: 36
Training accuracy: 87.52%
Validation accuracy: 82.40%
Test accuracy: 82.10%

- I did not try residual blocks, because I used pooling at each convolutional layer and since we already had a shallow network, I believed that pooled information coming from the initial layer would not really improve the performance that much.

- **(15 pts) Choose one optimizer and train your network. Plot the value of the loss function with respect to number of epochs. Determine the batch size. Report training and test classification accuracy. Discuss your early stopping procedure.**

  I tried three different batch sizes, here is the results of the comparison:

  batch_size = 16
  Training time: ~13 min | Best performance reached at epoch: 31
  Training accuracy: 86.60%
  Validation accuracy: 82.41%
  Test accuracy: 81.97%

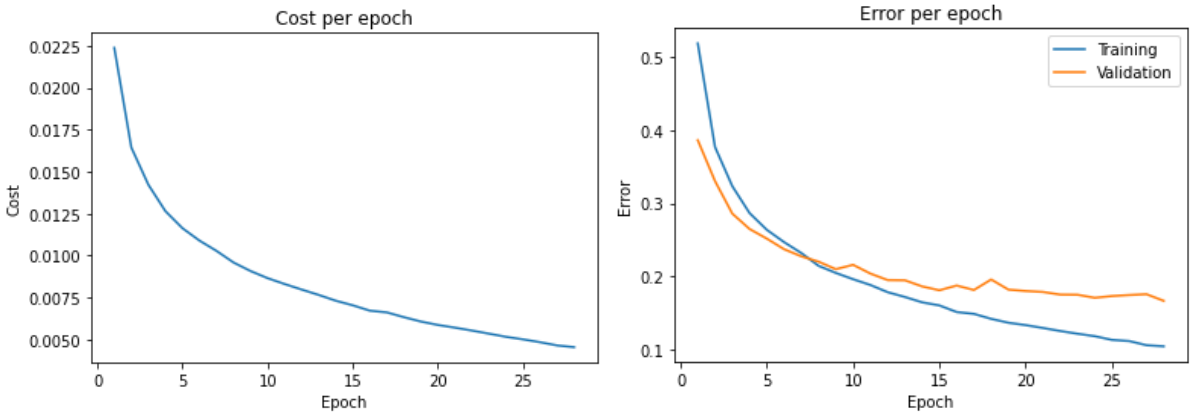
  ---

  batch_size = 32
  Training time: ~9 min | Best performance reached at epoch: 18
  Training accuracy: 84.88%
  Validation accuracy: 81.64%
  Test accuracy: 80.08%

  ---

  batch_size = 64
  Training time: ~10.5 min | Best performance reached at epoch: 28
  Training accuracy: 89.60%
  Validation accuracy: 83.37%
  Test accuracy: 83.00%

  I ran the network with Adam as the optimizer, learning rate of 0.001 and 64 batch size:



I searched for a suitable early stopping procedure, and used a similar protocol to that of [3]. I saved the model parameters if the validation accuracy was better than previous epochs, and waited a maximum of 5 epochs for an improvement on the validation accuracy to continue training. If there were no improvements, I stopped the training and saved the best performance (5 previous epochs).

- **(15 pts) After determining the final hyperparameters, try different optimizers and compare their performance and convergence properties.**

  batch_size = 64 | max_epochs = 100 | learning_rate = 0.001

  Using these hyperparameters, here is the comparison of different optimizers:

  optimizer: Adam
  Training time: ~10.5 min | Best performance reached at epoch: 28
  Training accuracy: 89.60%
  Validation accuracy: 83.37%
  Test accuracy: 83.00%

  ---

  optimizer: SGD with momentum = 0.9
  Training time: ~12 min | Best performance reached at epoch: 33
  Training accuracy: 85.12%
  Validation accuracy: 80.91%
  Test accuracy: 80.03%

  ---

  optimizer: Adagrad
  Training time: ~15 min | Best performance reached at epoch: 43
  Training accuracy: 75.20%
  Validation accuracy: 72.62%
  Test accuracy: 71.92%

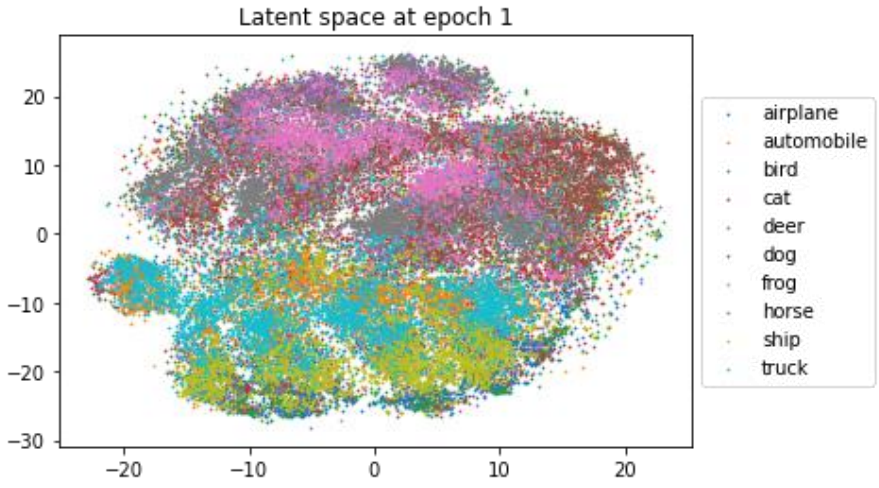
  ---

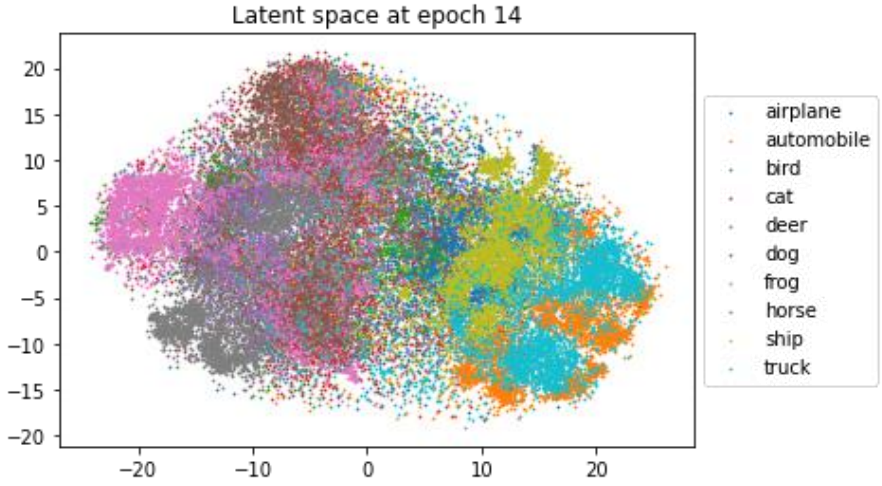
  optimizer RMSprop
  Training time: ~10 min | Best performance reached at epoch: 25
  Training accuracy: 88.24%
  Validation accuracy: 82.58%
  Test accuracy: 82.06%

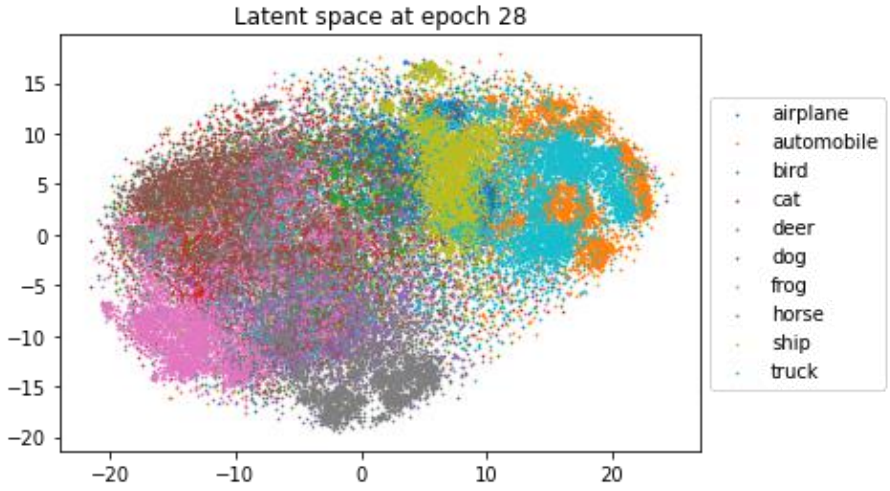
  As a result, I kept the Adam optimizer as its performance was slightly better than RMSprop.

- **(20 pts) Visualize the latent space of the model for the training samples (output of the feature extraction layers) in the beginning of the training, in the middle of the training, and in the end of the training. To visualize, map the latent representations into 2-D space using t-SNE. Use different colors to illustrate the categories. Compare the plots. Can you observe 10 groupings in the end of the training?**

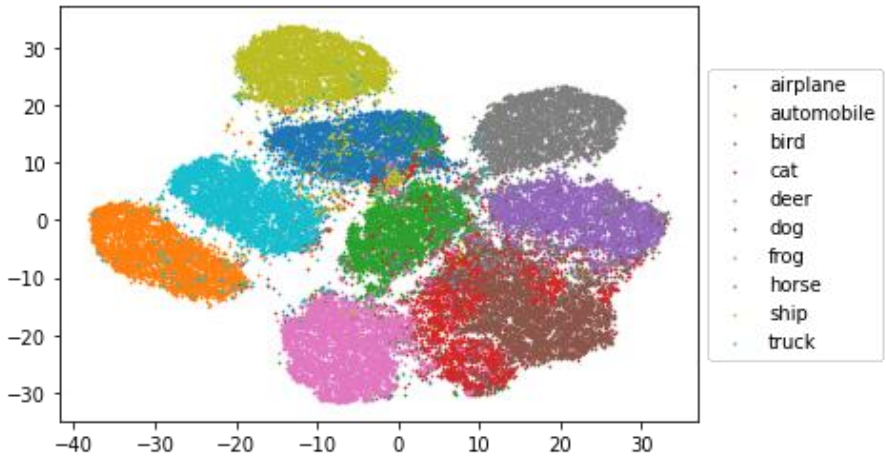
  Since I had to decrease the dimension of 40000 x 2048 matrix that is the output of convolutional layers to 40000 x 2 to train the t-SNE model, it took a very long time with the classic sklearn module. Therefore, I found another module on GitHub, tsne-cuda [5], which uses GPU to train t-SNE much faster. Using this module, I obtained these plots:







Although we can see slight improvements in clusters, it seems that the latent space itself is not enough to fully represent the learned features of the network, because ~90% classification accuracy does not really reflect on these plots. In contrast, when we look at the t-SNE plot for the final FC layer, we can easily see the differentiation of the categories:



We can see that cat and dog categories are somehow intermixed, this is probably the main source of the inaccuracy of the model. Other categories seem to be well separated.

- **(5 pts) Provide a README file where I can find the steps to train, and evaluate your model.**

  See README.txt.

References

1. Amidi A & Amidi S. (2021, May 13). *A detailed example of data loaders with PyTorch.* Stanford Blog. https://stanford.edu/~shervine/blog/pytorch-how-to-generate-data-parallel
2. *Training a classifier.* (2021, May 13). PyTorch Tutorials. https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html
3. Sunde, B M. (2021, May 13). *Early stopping for PyTorch*. GitHub. https://github.com/Bjarten/early-stopping-pytorch
4. Karani, D. (2021, May 13). *How Data Augmentation Improves your CNN performance? — An Experiment in PyTorch and Torchvision.* Medium. https://medium.com/swlh/how-data-augmentation-improves-your-cnn-performance-an-experiment-in-pytorch-and-torchvision-e5fb36d038fb
5. *CannyLab/tsne-cuda.* (2021, May 13). GitHub. https://github.com/CannyLab/tsne-cuda