Assignment 2

Student: MATTIA COLBERTALDO; Student's email: colbem@usi.ch

January 5, 2024

1 Image classification using CNNs [90/100]

1.1 Data (20 pts)

1. I have loaded the CIFAR-10 dataset using torchvision.datasets.CIFAR10. After loading, I visualized one image per class (Figure 3) and displayed histograms showing the distribution of images in both the training and test sets. From Figure 1 and Figure 2 it is clear that there is a perfectly even distribution of the images among the ten classes, in both train and test datasets: we have 5000 images for each class in the train set, and 1000 for each class in the test set. So the train set is five times larger than the test set.

2. Each element of the dataset is a tuple containing a torch.Tensor representing the image and an integer representing the class label. The type of each element is a tuple (torch.Tensor, int). To convert it to a suitable type, we can use the transforms.ToTensor() transformation during dataset creation. The dimension of the image as a torch.Tensor object is (3, 32, 32), representing three color channels and a 32x32 pixel resolution. The meaning of each dimension of the images is (channels, height, width).

3. To ensure a good practice in deep learning, I have normalized the dataset images to have a mean of 0 and a standard deviation of 1. This normalization is performed using the transforms.Normalize function, considering the previously applied transformation. You can see the normalized images in Figure 4. I did it by calculating the mean and standard deviation of the dataset, and then applying `transforms.Normalize(mean,std)`.

4. I split the original training set into a new training set (80%) and a validation set (20%) using torch.utils.data.random_split. This division allows for hyperparameters tuning using the training set and validation set during the model training process.

1.2 Model (10 pts)

The ConvNet that I have implemented comprises four convolutional layers, each followed by a ReLU activation function to introduce non-linearity. Max pooling layers are strategically placed to downsample the spatial dimensions, reducing the computational load and promoting translation invariance.

The initial two convolutional layers (`conv1` and `conv2`) have 3 input channels (for RGB images) and gradually increase the number of output channels to 256 and 512, respectively. Both of these layers use a 3x3 kernel with no padding and a stride of 1.

The subsequent convolutional layers (`conv3` and `conv4`) further process the features with increased output channels (512 to 256 and 256 to 128, respectively). These layers utilize larger 5x5 kernels to capture more complex patterns.

Pooling layers (`pool`) follow each pair of convolutional layers, employing a 2x2 window with a stride of 2 to perform spatial downsampling.

The final layer of the ConvNet is a fully connected layer (`fc1`) that takes the flattened output from the preceding layers and produces an output vector of size 10, corresponding to the number of classes in the CIFAR-10 dataset.

The ConvNet is trained using the cross-entropy loss function and stochastic gradient descent (SGD) with momentum as the optimization algorithm, with learning rate = 0.003 and momentum = 0.9.

1.3 Training (60 pts)

1. In the training pipeline, I have implemented a loop that iterates through the specified number of epochs (`n_epochs = 4`). Within each epoch, there is a nested loop that iterates through the batches of the training dataset. During the training loop, the model performs a forward pass, computes the training loss, performs backpropagation, and updates the model parameters using the optimizer. Additionally, I have included code to print the current training loss and accuracy every 300 steps. This periodic reporting helps in monitoring the training progress, while I record only once per epoch.

   For the validation phase, the model is switched to evaluation mode (with the command `net.eval()`). The validation set is then used to evaluate the model's performance without updating the parameters. The validation loop prints and records the current validation loss and accuracy once per epoch. This information is valuable for hyperparameter tuning and assessing model generalization.

2. With a batch size of 32 and 4 epochs, with the given seed (42) my model reaches a Test Accuracy of 75.36%.

3. After training, the parameters of the trained model are saved through `torch.save(net.state_dict(), 'MATTIA_COLBERTALDO_1.pt')`, that actually saves the model state dictionary.

4. Following the training process, the evolution of both training and validation losses is visualized in plot 5; the same goes for the accuracies in plot 6. The x-axis, in both cases, represents the number of epochs, since I recorded those results once per epoch.

5. The new Convolutional Neural Network model reaches 83% test accuracy (seed = 42) with the subsequent configuration. I'd like to point out that my previous code reached 85% in 100 epochs, but my data were normalized with Normalize((0.5,0.5,0.5),(0.5,0.5,0.5)). Only after I modified

my code by calculating the mean and std dev of the dataset, the accuracy shifted to 83% in 60 epochs. One additional note about the seed is that even if I manually set it to 42 both for both torch and its cuda library, sometimes the process is not deterministic and lead to different (even if similar) results.

(a) **Convolutional Layers**:

- First Convolutional Layer: 3 input channels, 256 output channels, kernel size of 3, stride of 1 and padding of 0.
- Second Convolutional Layer: 256 input channels, 512 output channels, kernel size of 3, stride of 1 and padding of 0.
- Third Convolutional Layer: 512 input channels, 256 output channels, kernel size of 5, stride of 1 and padding of 0.
- Fourth Convolutional Layer: 256 input channels, 128 output channels, kernel size of 5, stride of 1 and padding of 0.

The chosen architecture is relatively deep with multiple convolutional layers, aiming to capture complex features in the dataset.

(b) **Activation Functions**: ReLU is used after each convolutional layer to introduce non-linearity to the model. I didn't find useful for my model accuracy to move to GeLU activation functions, so I kept the ReLU ones.

(c) **Batch Normalization**: Batch Normalization is applied after the first and fourth convolutional layers. It helps stabilize and accelerate the training process by normalizing the inputs to the layer, , which can lead to faster convergence.

(d) **Pooling Layers**: MaxPooling is applied after the first and third convolutional layers with a kernel size of 2 and a stride of 2, to perform spatial downsampling.

(e) **Fully Connected (Linear) Layers**:

- First Fully Connected Layer: 1152 input features, 512 output features.
- Second Fully Connected Layer: 512 input features, 10 output features (corresponding to the number of classes).

(f) **Dropout**: Dropout with a probability of 0.5 is applied after the first fully connected layer to reduce overfitting during training. Dropout is employed after the first fully connected layer to prevent overfitting, especially given the increased model complexity. A dropout rate of 0.5 was chosen to moderately regularize the network..

(g) **Data Augmentation**: During training, data augmentation is applied to the input images. This includes random horizontal flips, random rotations up to 15 degrees, and color jittering for brightness, contrast, saturation, and hue. Data augmentation is utilized during training to increase the diversity of the training dataset, enhancing the model's ability to generalize to unseen data.

(h) **Data Normalization**: Input images are normalized with a mean of (0,0,0) and a standard deviation of (1,1,1), since it is a good practice in Deep Learning. Input normalization is applied to ensure that the input data has a consistent scale, aiding convergence during training.

(i) **Loss Function and Optimizer**:

- CrossEntropyLoss is used as the loss function, suitable for multi-class classification tasks.
- Adam optimizer is employed with a learning rate of 0.0005 and weight decay of 1e-4.

(j) **Learning Rate Scheduler**: A ReduceLROnPlateau learning rate scheduler is used, monitoring the validation loss. The learning rate is reduced by a factor of 0.5 if no improvement is observed for 5 consecutive epochs. A learning rate scheduler is incorporated to adjust the learning rate dynamically based on the validation loss, potentially overcoming learning rate challenges and improving convergence.

6. As seen before, I saved the new model with

```
torch.save(model.state_dict(), 'MATTIA_COLBERTALDO_2.pt')
```

1.4 Bonus question* (5 pts)

I averaged my fastest model (the first one) on 10 different seeds, and obtained a mean Accuracy of 75.91%, and Standard Deviation of 0.34%.

I generally evaluate the performance of different models by comparing their mean accuracy and standard deviation across multiple runs with various seeds. This allows me to gauge the consistency and reliability of each model. A higher mean accuracy indicates better overall performance, while a lower standard deviation suggests more stability in the model's behavior across different runs.

Regarding model robustness, I assess it by examining the standard deviation. A lower standard deviation implies that the model is less sensitive to changes in random initialization (different seeds), indicating a more robust and reliable performance. This ensures that the model's accuracy is consistent across various instances, reinforcing my confidence in its generalization capabilities. In my case the standard deviation is 0.34%, pretty low, so I can firmly state that my model is a robust one.

Questions [5 points]

ResNets are a type of deep neural network architecture designed to address the vanishing gradient problem during training. They introduce residual blocks, consisting of skip connections that allow the gradient to bypass one or more layers. This facilitates the training of very deep networks, mitigating the degradation issue where adding more layers leads to decreased accuracy.

In a residual block, the input of a layer is added to the output, creating a residual mapping.

ResNets offer advantages over traditional Convolutional Neural Networks by enabling the training of extremely deep networks. This design not only improves accuracy but also simplifies the training process.

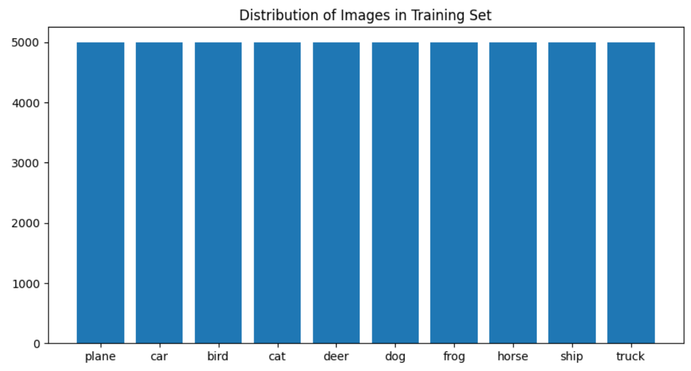

Figure 1: Distribution of images in train set.

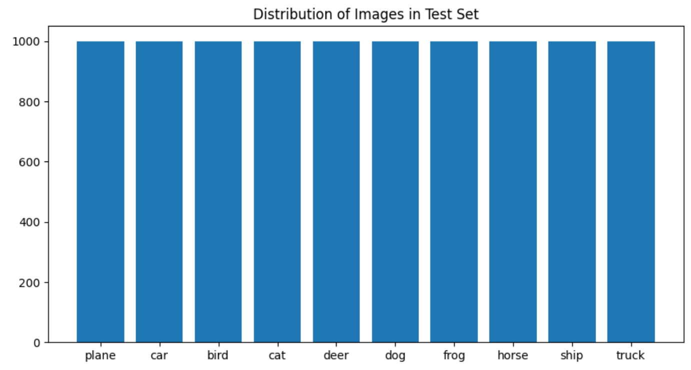

Figure 2: Distribution of images in test set.

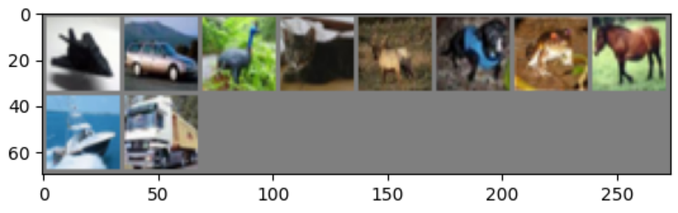

Figure 3: One image per class.

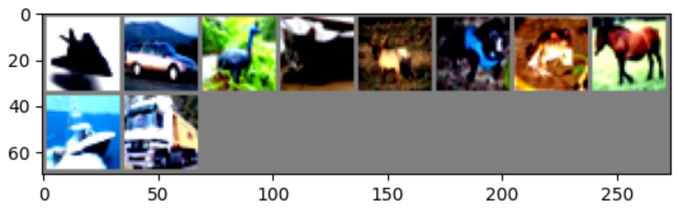

Figure 4: One normalized image per class.

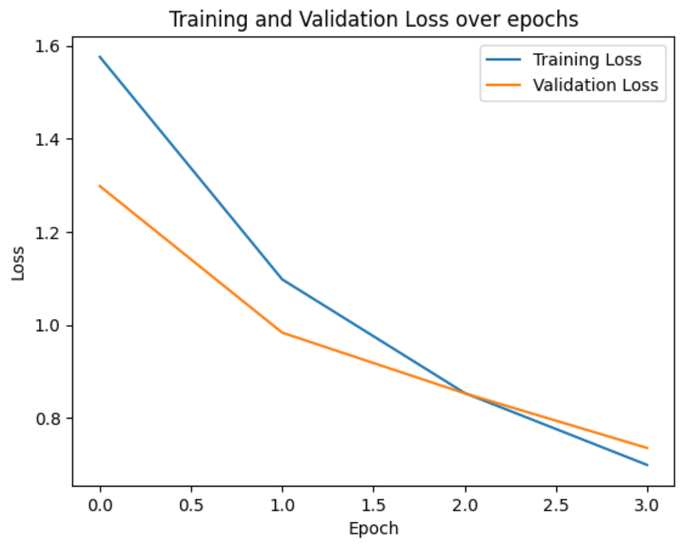

Figure 5: Training and Validation loss over epochs.

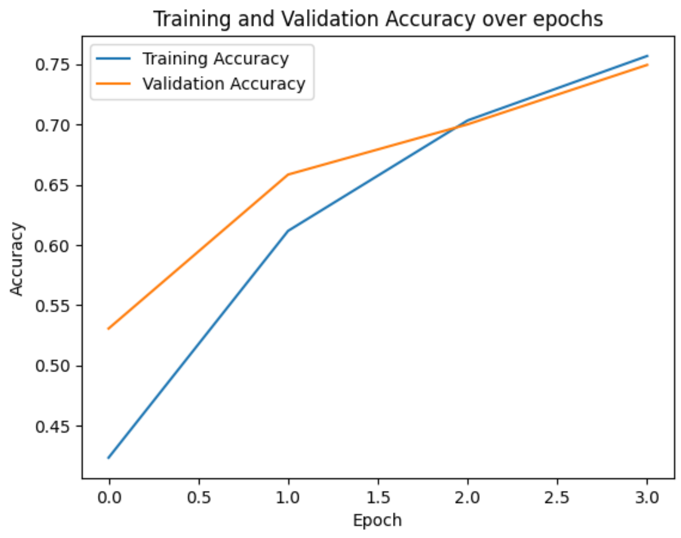

Figure 6: Training and Validation accuracy over epochs.