Neural Network CW

Ngoc Bach Nguyen

Task 1) Read dataset and create dataloaders.

I use the standard code provided by PyTorch documentation on the CIFAR-10 tutorial to read the dataset and create the dataloaders. I also applied data augmentation, like taught in this course's labs, specifically:

- For the training dataset, a series of transformations are applied: **RandomHorizontalFlip** randomly flips images horizontally, **RandomCrop** randomly crops images with a specified padding, **ToTensor** converts the input image from a PIL format to a PyTorch tensor and **Normalize** normalizes the image using the provided mean and standard deviation values.
- For the testing dataset, the same **ToTensor** and **Normalize** transformations are applied, but data augmentation techniques like **RandomHorizontalFlip** and **RandomCrop** are not used as they are only relevant during training.

The data augmentation should aid in reducing overfitting as the images' medley and variety will increase due to the transformations applied.

Task 2) Create the model.

My model's architecture consists of the following components:

- Block Module: The Block module comprises a series of k convolutional layers, each followed by ReLU activation and dropout. MaxPooling is used after the ReLU activation and dropout for each convolutional layer. MaxPooling is a dimensionality reduction technique that reduces the spatial dimensions of the feature maps while preserving the most important information. This is achieved by taking the maximum value within a sliding window (in this case, a 2x2 window with a stride of 2) across the feature map. Adaptive average pooling and a linear layer are used to compute the weights for each convolutional layer's output, with output size 1, as AdaptiveAvgPool2d(1) averages the channels and each one is turned into a single value, as HXW becomes 1x1. The outputs of these convolutional layers are then combined using the computed weights, followed by a batch normalization layer. The Block module is designed to learn different features through multiple convolutional layers and combine them adaptively. This design decision allows the model to focus on the most important features learned by each convolutional layer, enabling it to capture a diverse set of features from the input images. The dropout randomly switches off neurons to help prevent overfitting, and a value of 0.1 is chosen based on studies cited in the sources.

- Classifier Module: The Classifier module is a fully connected feed-forward neural network with three hidden layers (an MLP with 4 layers in total), followed by ReLU activations. An adaptive average pooling (AdaptiveAvgPool2d(1)) layer is applied before the input layer to reduce the spatial dimensions of the input. This module is responsible for mapping the high-level features extracted by the Block modules to the final class probabilities. The design choice of having multiple hidden layers helps in learning complex patterns in the extracted features and mapping them to the correct classes.

- Model Module: The Model module stacks a series of, N, Block modules, where each block takes the output of the previous block as input. The final block's output is passed to the Classifier module to obtain the class probabilities. This module effectively assembles the entire model, allowing the features to be extracted and combined across multiple levels of abstraction. The decision to stack Block modules sequentially helps the model to learn hierarchical features from the input images, enhancing its representational capacity.

Overall, the model leverages the adaptive combination of multiple convolutional layers in each block and effectively assembles the blocks to extract hierarchical features from the input images. The classifier module, with its fully connected layers, enables the mapping of high-level features to the final class probabilities.

Task 3) Create the loss and optimizer.

I defined the loss function as the cross-entropy loss function, since it's appropriate for the classification task, as it measures the difference between the predicted class probabilities and the ground-truth labels.

I chose the Adam optimizer since it is a popular choice for training deep neural networks, as it adapts the learning rate for each parameter during training, allowing for faster convergence and improved performance. Also, it works better for my chosen hyperparameters compared to SGD, during the fine-tuning process. But the learning rate adaptation for the Adam Optimizer is not effective enough by itself when I was fine-tuning the model, so I had to also implement a learning rate scheduler.

The learning rate scheduler used a cosine annealing schedule, including a warmup phase, this is by using an inbuilt function from the Hugging Face Transformers library. The warmup phase gradually increases the learning rate from a small value to the specified initial learning rate, which allows the model to start exploring the parameter space more gently before taking larger steps. The cosine annealing schedule then gradually decreases the learning rate, helping the model converge to a good solution. This approach ensures a balance between exploration and exploitation during optimization and often results in better generalization performance. This is inspired by the implementation of the learning rate scheduler provided in the sources.

Task 4) Training script

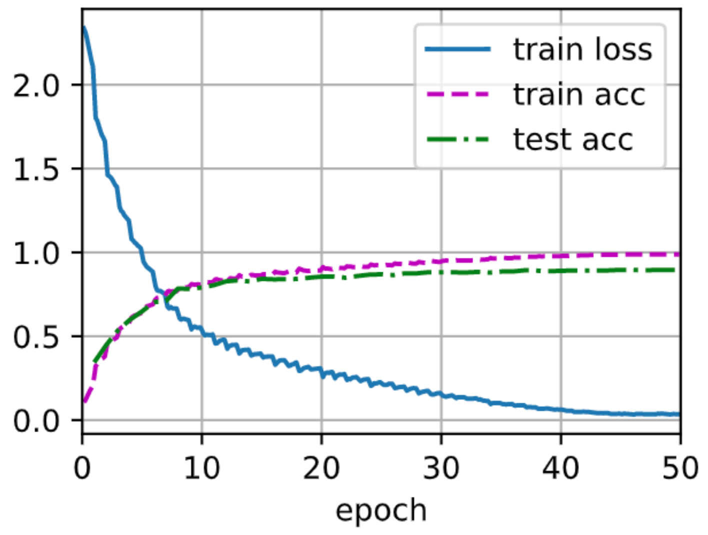
I used the training script provided in the lab 9 solution file and I modified it so that it has a scheduler to update the learning rate after the training is optimized at each iteration. The trainf function trains a given neural network model (**net**) using a specified loss function (**loss**), optimizer (**optimizer**) and device (**device**). The input data is provided through **train_iter** and **test_iter**, and the training process lasts for **num_epochs** epochs. During training, the function measures and accumulates the training loss and accuracy, and it updates a plot every 50 iterations to visualize the progress. After each epoch, the function evaluates the model on the test dataset and updates the plot with the test accuracy. Finally, the function prints the final training loss, training accuracy, test accuracy, and the number of examples processed per second.

<u>Hyperparameters:</u>
batch_size = 128
stats = ((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) for all data augmentation techniques
tt.RandomCrop(32, padding=4) standard parameters for random crop
nn.AdaptiveAvgPool2d(1) reducing input tensor's spatial dimensionality, HxW to 1x1; hence output = 1
nn.MaxPool2d(2, 2) 2x2 window, stride is 2 by default.
nn.Dropout(p=0.1) value assigned as suggested in research sources

fully connected/ linear layers = 4
hidden_layer1_size = 256, hidden_layer2_size = 128, hidden_layer3_size = 64
Model([3, 64, 128, 256, 512], 4, 6) # input channels = 3, N = 4, k = 6, input feature sizes = 64, 128, 256, 512

learning_rate = 0.001 for the optimizer
warmup_epochs = 5 for the scheduler
total_epochs = 50
steps_per_epoch = len(trainloader.dataset) // trainloader.batch_size (Note: // is floor division)
get_cosine_schedule_with_warmup(optimizer, num_warmup_steps=warmup_epochs * steps_per_epoch, num_training_steps=total_epochs * steps_per_epoch)
weight_decay=1e-4

```
loss 0.035, train acc 0.988, test acc 0.896
225.4 examples/sec on cuda
```



Task 5) Training and testing accuracy.

The model achieves a high training accuracy (98.8%) and a relatively good test accuracy (89.6%). This is a strong performance on the CIFAR-10 classification task when measured relatively with other models found online such as the one provided in the PyTorch CIFAR-10 tutorial. However, there is a notable difference between the training and test accuracies, which might suggest overfitting. Some modifications to improve this might be to implement more regularization techniques and experimentation with different hyperparameters and architectural changes.

Sources:

Park, S. and Kwak, N. (n.d.). Analysis on the Dropout Effect in Convolutional Neural Networks Sungheon Park and Nojun Kwak. *Graduate School of Convergence Science and Technology, Seoul National University, Seoul, Korea*. [online] doi:https://doi.org/10.1007/978-3-319-54184-6.

Lu, Z., Xia, W., Arora, S. and Hazan, E. (2022). Adaptive Gradient Methods with Local Guarantees. [online] Available at: arxiv:2203.01400v1.

transformers.optimization — transformers 2.9.1 documentation (huggingface.co)

https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html