CIFAR-10 Classification Coursework

ECS6P9U - Neural Networks and Deep Learning

Christian Juresh - 210517307

1 Dataset and Dataloaders: 5%

Two sets of transformations are defined. One is for dataset augmentation: used for the training dataset, and one for standard processing: used for the validation and testing datasets. To prevent overfitting, the dataset augmentation transformation includes several steps to artificially increase the diversity of the training data:

- **RandomHorizontalFlip:** Mirrors images horizontally at random.
- **RandomCrop:** Randomly crops the images and adds padding.
- **ToTensor:** Converts the images to PyTorch tensors.
- **Cutout:** Uses my custom implementation of a random masking technique that "cuts out" one or more rectangular patches from the input image.
- **Normalize:** Standardises the colors of the images.

The testing dataset is a batch size of 4, and only includes the ToTensor and Normalize transformations. This is so that the model is tested on the original images for accurate results. A batch size of 64 for training gives a balance of speed and accuracy.

2 Model: 40%

2.1 Block

The `Block` class represents one instance of many blocks, which are responsible for a sequence of operations that map an input tensor to a transformed output. It uses an `AdaptiveAvgPool2d` layer to perform spatial averaging, reducing each channel to a vector of d. This vector is then transformed by a `Linear` layer, which outputs a vector a of length K, the number of convolutional layers, where each element is a weight. The convolutional layers consist of a `Conv2d` layer, batch normalization, and a non-linear `ReLU` activation function. In the forward pass, a weight vector a is calculated from the linear layer's output, softmax is applied to obtain a distribution of weights, and the outputs of the convolutional layers are combined based on these weights to produce a single tensor O.

2.2 Backbone

The `Backbone` class is responsible for constructing N Blocks. It does this by sequentially stacking multiple different stages of Block instances in two `for` loops.

2.3 Classifier

The `Classifier` class creates a classifier that takes the output from the final Block in the Backbone and computes a mean feature vector by applying a SpatialAveragePool. This mean feature vector is then passed to a multi-layer perceptron (MLP) classifier, consisting of one fully connected layer, `ReLU` activation, dropout for regularization, and then another fully connected layer.

2.4 Model

The `Model` class combines the Backbone and Classifier to create a Convolutional Neural Network (CNN). The forward pass puts tensor X through the Backbone, receives the O_N output from the N_{th} Block, and then puts it through the Classifier.

3 Loss and Optimizer: 5%

`CrossEntropyLoss` is used as the loss function. It applies a softmax function to the output of the network to compute probabilities, and then calculates the loss between the predicted probabilities and the true class labels. `optim.SGD` is used as a Stochastic Gradient Descent optimiser, which updates the weights of the model based on the gradient of the loss function with respect to the weights. It finds the set of weights that minimise the loss function. Learning rate controls how much the weights are adjusted by, and momentum accelerates the optimiser in the right direction. Weight decay helps in preventing overfitting by penalising large weights. `lr_scheduler.CosineAnnealingLR` is used to adjust the learning rate following a cosine annealing schedule. It reduces the learning rate between an upper and lower boundary over the number of epochs. This improves performance and allows the model to find better local minima.

4 Training Script: 30%

4.1 Training Loop

The training loop simply runs the training function over a set number of epochs.`lr_scheduler.step` is called to adjust the learning rate at the end of each epoch.

4.1.1 Function `train_epoch`

The `train_epoch` function begins by setting the model to training mode. It iterates through the *loader*, which provides batches of *inputs* and their *labels*. The data is then moved to the correct device.
`optimizer.zero_grad` clears any old gradients from the last step so that they don't accumulate. Then the model computes the prediction for the input data. `criterion` calculates the difference between the model's predictions and the actual labels, and produces a scalar value to represent the model's performance. `loss.backward` computes the gradient of the loss with respect to the model's parameters, which are then used to adjust and update the weights with `optimizer.step`

4.2 Hyperparameters

Dataset Augmentation Transformation

- `RandomHorizontalFlip`
  - `Probability` = 0.5
- `RandomCrop`
  - `Size` = 32
  - `Padding` = 4
- `Cutout`
  - `Number of holes` = 1
  - `Length of each hole` = 16
- `Normalize`
  - `Mean & SD` = $(0.5, 0.5, 0.5)$

Dataset Loading

- `training batch size` = 128
- `vallidation batch size` = 4

Model Architecture

- `number of classes` = 10
- `input channels` = 3 (RGB)
- `convolutions` = 2
- `layer output channels` = [64, 128, 256, 512]
- `blocks` = [2, 2, 2, 2]

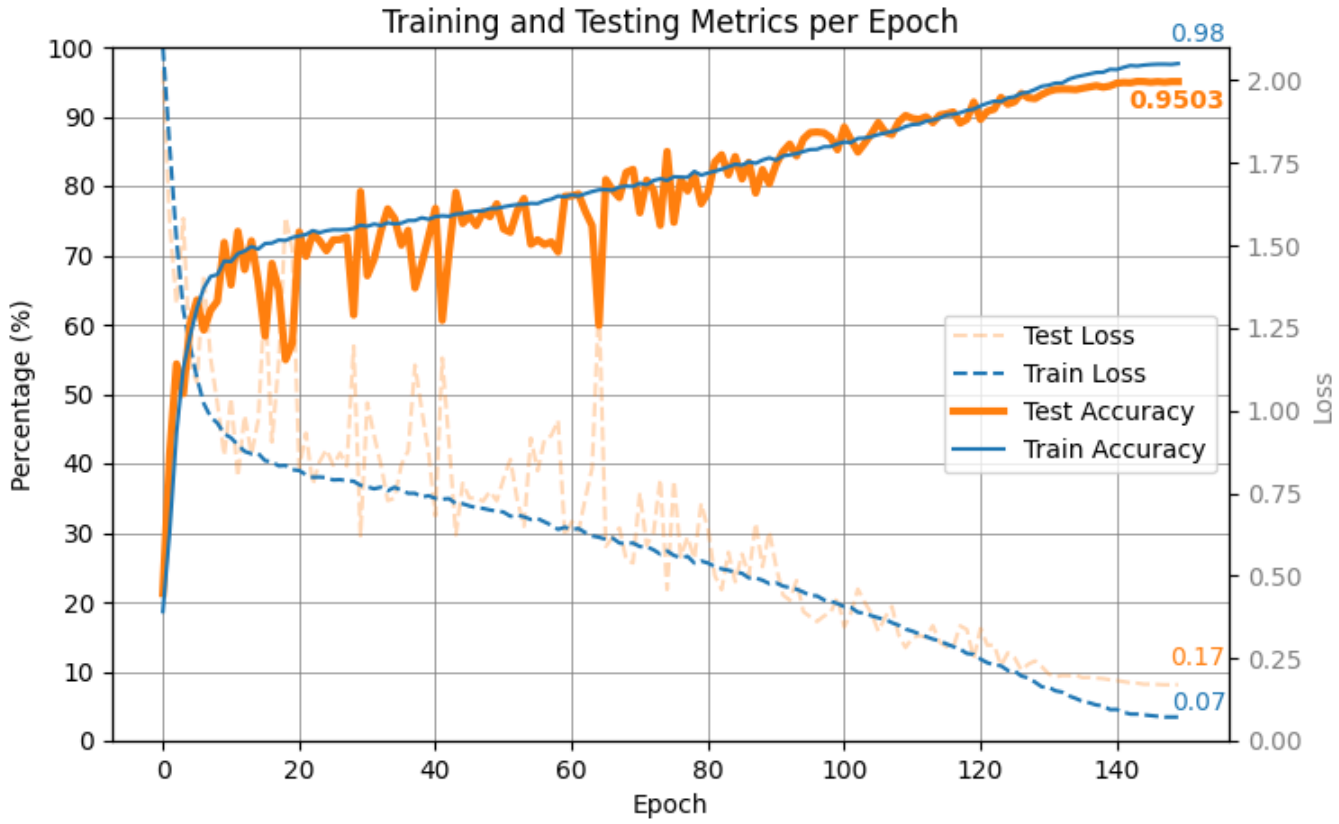
Training

- `epochs` = 150

Optimisers

- `learning rate` = 0.2
- `momentum` = 0.9
- `weight decay` = 5×10^{-4}

4.3 Plot



5 Final Model Accuracy: 20%

Final model accuracy: 95.03%