Report for Assignment 1 - Roberto Carminati, Robert Sauerborn

What is the purpose of a loss function? What does the cross-entropy measure? Which criteria must the ground-truth labels and predicted class-scores fulfill to support the cross-entropy loss, and how is this ensured?

The loss function measures the performance of a model, it quantifies the difference between the values predicted by the model and the actual values of the dataset. The purpose of the loss function is to optimize the parameters during the training process. Minimizing the value of the loss function and thereby improving the ability of the model to make accurate predictions. It also provides a metric to evaluate the performance of the trained model.

The Cross-entropy is a common loss function used primarily in classification tasks. It measures the difference between the predicted probability distribution and the actual probability distribution.
In binary classification the cross-entropy loss function quantifies the difference between the predicted probability of the positive class and the actual label. It penalizes the model more when it predicts a high probability for the wrong class.
In multi-class classification, cross-entropy extends to measure the difference between the predicted probabilities for multiple classes and the actual distribution of class labels. It encourages the model to assign higher probabilities to the correct classes and lower probabilities to incorrect ones.
The cross-entropy loss function is defined as:

$$H(y,\hat{y}) = -\sum_i y_i \log(\hat{y}_i)$$

Where:

- y is the true probability distribution (the ground truth)
- $\hat{y}$ is the predicted probability distribution
- y_i and $\hat{y}_i$ are the probabilities of the i-th class

The cross-entropy loss function is minimized when the predicted probability distribution matches the true distribution, making it a suitable choice for training classification models.

The ground-truth labels should represent the true distribution of classes for each sample in the dataset and should be a probability distribution, where one class is assigned a probability of 1 (indicating it as the true class) and all other classes have a probability of 0.
The predicted class scores are the probability estimates for each class.They should sum up to 1 for each sample.
We can ensure this criteria: training with appropriate loss function, using Soft-max activation and with One-hot encoding.

What is the purpose of the training, validation, and test sets and why do we need all of them?

The training set is used to train the model. It should be the largest part of the dataset. During training, the model learns patterns and relationships from the training data. The models parameters are adjusted to minimize the chosen loss function.
The validation set is used to fine tune the models' hyper-parameters and to assess its performance during training. After a certain number of epochs the performance is evaluated on the validation set. Using a validation set helps to prevent on our training data.
And the test Set is used to evaluate the final performance of the trained model. It's only used in the end not during training process to measures the performance on unseen data.

What are the goals of data augmentation, regularization, and early stopping? How exactly did you use these techniques (hyper-parameters, combinations) and what were your results (train, val and test performance)? List all experiments and results, even if they did not work well, and discuss them.

Data augmentation diversifies the training data by applying transformations, enhancing the model's ability to generalize. In our project we applied a normalization after loading the data.
Regularization techniques constrain model complexity to prevent overfitting by penalizing overly complex solutions during training. (In our code weight decay = 0.0001 prevent it)
Early stopping monitors the model's performance on a validation set, halting training when performance starts to degrade to preventing overfitting and ensuring optimal generalization. A situation that didn't occur in our experiments.

Experiments

In this project we just try to modify learning rate and the number of epoch. The other parameter were set to the default values: batch size = 128, momentum = 0.9, weight decay = 0.0001, learning rate step size = 5, learning rate gamma= 0.1.

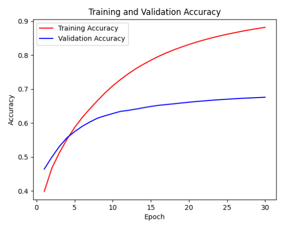
Resnet

With different learning rates(LR). LR = 0.03, LR = 0.06 and LR = 0.09. Also LR = 0.06 with 50 epochs.

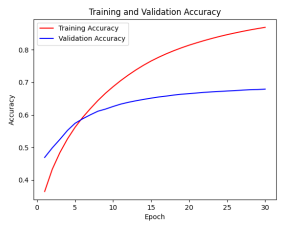
With LR = 0.03 we achieved Train Accuracy: 0.8821, Validation Accuracy: 0.6762 and a Test Accuracy of: 0.7059 in epoch 30.
With LR = 0.06 we achieved Train Accuracy: 0.8688, Validation Accuracy: 0.6792 and a Test Accuracy of: 0.7205 in epoch 30.
With LR = 0.09 we achieved Train Accuracy: 0.8475, Validation Accuracy: 0.6693 and a Test Accuracy of: 0.6907 in epoch 30.
We then used 50 epochs for LR = 0.06 and achieved Train Accuracy: 0.9048, Validation Accuracy: 0.6775 and a Test Accuracy of: 0.6957 in epoch 50. (See Figure 1)
We obtain the best performance with LR = 0.06 and 30 epoch. It's interesting to note that increasing the epoch to 50 the accuracy on the training raises, but the one on the test falls, highlighting an overfitting scenario.

CNN

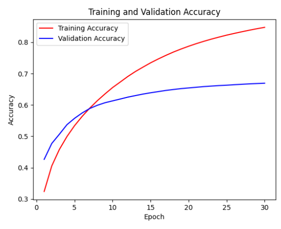
With LR = 0.06 we achieved Train Accuracy: 0.5171, Validation Accuracy: 0.6019 and a Test Accuracy of: 0.6537 in epoch 30.



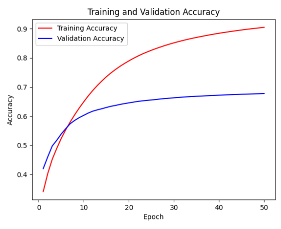
(a)



(b)

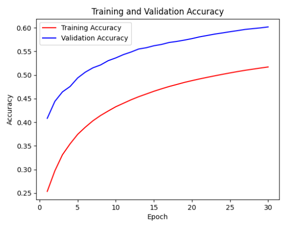


(c)

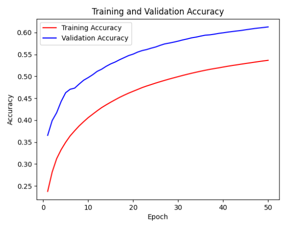


(d)

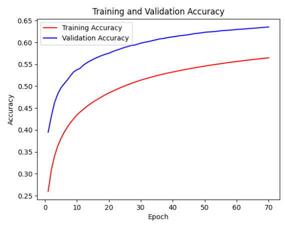
Figure 1: (a) LR = 0.03, (b) LR = 0.06, (c) LR = 0.09 each 30 epochs, (d) LR = 0.06 with 50 epochs



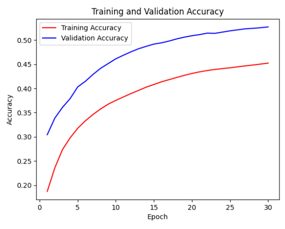
(a)



(b)



(c)



(d)

Figure 2: (a) LR = 0.06, 30 epochs. (b) LR = 0.06, 50 epochs. (c) LR = 0.06, 70 epochs. (d) LR = 0.09, 30 epochs.

With LR = 0.06 we achieved Train Accuracy: 0.5366, Validation Accuracy: 0.6128 and a Test Accuracy of: 0.6692 in epoch 50.
With LR = 0.06 we achieved Train Accuracy: 0.5648, Validation Accuracy: 0.6353 and a Test Accuracy of: 0.6860 in epoch 70.
With LR = 0.09 we achieved Train Accuracy: 0.4524, Validation Accuracy: 0.5272 and a Test Accuracy of: 0.5682 in epoch 30.(See Figure 2)
The best result is obtained with a LR of 0.06 and 70 epoch. The best model was the one obtained at the last epoch so we suspect that we could also have trained the model longer to get a better performance.

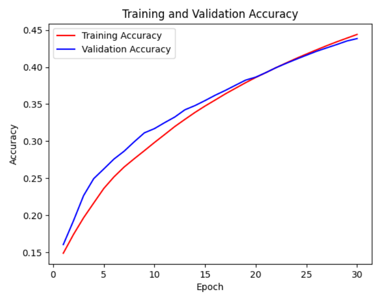
VIT

With LR = 0.06 we achieved Train Accuracy: 0.4302, Validation Accuracy: 0.4204 and a Test Accuracy of: 0.5510 in epoch 30.
With LR = 0.06 we achieved Train Accuracy: 0.5055, Validation Accuracy: 0.4867 and a Test Accuracy of: 0.6268 in epoch 50.
With LR = 0.09 we achieved Train Accuracy: 0.4440, Validation Accuracy: 0.4384 and a Test Accuracy of: 0.5397 in epoch 30. (See Figure 3)
The best result is obtained with a LR of 0.06 and 50 epoch. As for CNN, we suspect that we should train the model longer to improve the performance.

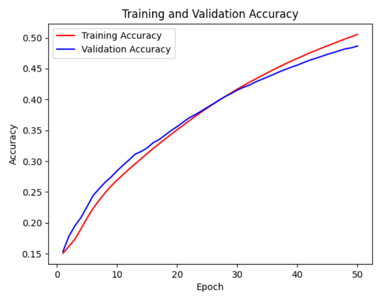
In (Figure 4) we compare the performance of the 3 models. As expected the best one is the Resnet already implemented in Pytorch. There is a weird behaviour of CNN, which has an higer validation accuracy than the training one during the hole process.

What are the key differences between ViTs and CNNs? What are the underlying concepts, respectively?

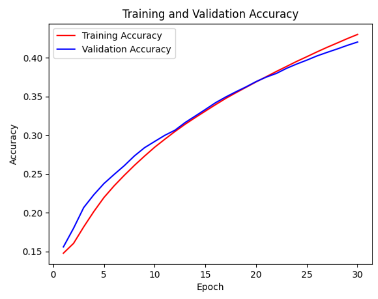
CNNs use convolutional and pooling layers to find local features from images, focusing on spatial hierarchies. ViTs utilize self-attention mechanisms inspired by Transformers, enabling them to capture both global and local



(a)



(b)



(c)

Figure 3: (a) LR = 0.06, 30 epochs. (b) LR = 0.06, 50 epochs. (c) LR = 0.09, 30 epochs.

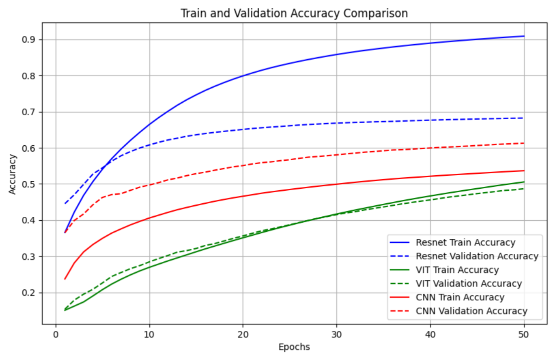


Figure 4: Comparison between the model with LR = 0.06 and 50 epoch on training and validation

information by attending to all image patches simultaneously.
CNNs are great for tasks like edge detection and texture recognition, ViTs are better for tasks that require understanding global context like scene understanding and fine-grained recognition.

The implementation of the Vision Transformer (ViT) model used in this project is based on the code provided by lucidrains on GitHub `https://github.com/lucidrains/vit-pytorch`.

The following code (Listing 1) illustrate how the ViT model processes input images by dividing them into patches, transforming these patches into embeddings, and applying dropout regularization. This approach allows ViTs to capture spatial information from images and learn global dependencies between patches.

Listing 1: Python code from vit.py

```
patches = x.unfold(2, self.patch_size, self.patch_size).unfold(3,
                                   self.patch_size, self.patch_size)
patches = patches.contiguous().view(patches.size(0),
                         patches.size(2) * patches.size(3), -1)
embeddings = self.patch_to_embedding(patches)
embeddings = self.dropout(embeddings)
```

In these lines, the input image X is divided into patches. This operation unfolds the image into overlapping patches based on the patch size. Then the patches are reshaped into a 2D tensor where each row represents a patch. This tensor is afterwards given into a linear layer to obtain embeddings for each patch. Finally a dropout regularization is applied to the patch embeddings to prevent overfitting during training.