Report 1: Training CNN
How to train a CNN on MNIST and CIFAR10

Niccolò Arati
niccolo.arati@edu.unifi.it

Abstract

This project is about finding and exploring optimal ways to train a CNN to fulfil the Image Classification task. Three main strategies will be analyzed: train a CNN from scratch, train a predefined model and fine-tune a pretrained model. Lots of configurations obtained by varying hyperparameters, both related to training and to the model considered, will be presented and discussed through some tables, where there will be reported the main results in terms of classification accuracy. From the experiments, it is shown that the better results (both from convergence speed and accuracy) are achieved with the fine-tuning process.

1. Introduction

A **Convolutional Neural Network** (**CNN**) is a deep learning model designed to handle and interpret visual information. In this work, CNNs will be used for the image classification task, thanks to their ability to learn spatial hierarchies of features directly from images.
A CNN usually has a base structure which is made of:

- **Convolutional Layers**: extract important features from an image while preserving spatial relationships between pixels. A convolutional layer in a CNN works by applying filters (kernels) to an input image or feature map to extract patterns and features. Each filter slides across the image, performing at each position an element-wise multiplication and then summing the results to produce a feature map in output. In Sec. 2.2 parameters that determines how the convolutional layer operates will be discussed.
- **Pooling Layers**: reduce the spatial dimensions of the feature maps, providing some translation invariance.
- **Fully Connected Layers**: combine the features extracted by convolutional and pooling layers to make a final prediction.
- **Activation Functions**: non-linear functions, make the network learn complex patterns and relationships.

To carry out the image classification task, fully connected layers are paired with a **Softmax Layer**, which converts the network output scores into a normalized probability distribution across all different classes that have been taken into account.
The goal of this project is to train CNNs and then analyze the progress of training and the results as certain techniques and choices of hyperparameters related to training are varied. These variations will be explained in more detail along the next sections (mainly in Sec. 2.2 and Sec. 2.3).

1.1. Training a CNN

Training a Convolutional Neural Network (CNN) is the process of guiding the model to learn how to detect patterns and features in data, images in this case, by adjusting its internal weights over a series of iterations.
The final goal of training for image classification is to predict a class for each image. To evaluate training, the prediction is confronted with the label associated to data (the so called ground truth), referring to the field of Supervised Learning.
Here is an overview on how this training process usually progresses:

- **Dataset Preparation**: from input labeled dataset, images are usually resized, normalized, and sometimes augmented to improve model generalization and reduce overfitting. The images are also split between training dataset and test dataset, and sometimes training dataset is split again into training and validation dataset.
- **Forward Pass**: input images from training dataset are passed as batch through the network, where intermediate layers extract features which are then mapped to a score for each class by the fully connected layers.
- **Loss Calculation**: measures the difference between the predicted class probabilities after the final softmax layer and the true labels. For classification tasks, the most common loss function (and the one used in this project) is **Categorical Cross-Entropy**. In order

to improve the training results, loss need to be minimized.

- **Backpropagation**: calculate the gradient of the loss function with respect to each weight in the network. This information shows how to update weights to reduce the loss and consequently improve the model's performance.
  An **optimization algorithm** uses the gradients to adjust the weights, and the update quantity is scaled by a parameter called **Learning Rate**.
  These concepts are further explored in Sec. 2.3.

The process described between forward pass and weight update is repeated over multiple epochs (an epoch is complete when all the training dataset have been processed).
Sometimes, after some batches or after each epoch, the model's performance could be checked on a separate **validation** set to monitor its generalization capability, for example to detect an eventual problem of overfitting (model performs well on training data but poorly on new validation data).
After a specified number of epochs, training is complete and the CNN is evaluated on a **test** set (made of data that the model has never seen). To evaluate the performance, in image classification the number of correct predicted label over the total predictions are usually considered as an accuracy metric.
To conclude this brief training explanation, it is important to specify that the gradient computation, and consequently weight update, is disabled during validation and testing of the trained model, because the objective of these two processes is to evaluate the model performance on unseen images, so the model should not learn anything from these ones.

2. Experiments Parameters

The project have been developed with **Pytorch** framework, and to specify the different setups employed I will refer to parameters that are required by this framework in the following subsections.
All the experiments setup and results are contained in this github repository: https://github.com/Sossio699/ComputerVision

2.1. Datasets

CNN models have been trained on two different datasets:

- **MNIST**: large collection of grayscale images of handwritten digits.
  The dataset contains 70000 images split into 60000 training images and 10000 test images.
  Each image has a resolution of 28x28 grayscale pixels and there is a label from 0 to 9 associated with every individual image, indicating which digit is represented in the corresponding image.

- **CIFAR10**: collection of images from 10 different classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.
  The dataset contains 60000 images split into 50000 training images and 10000 test images. Each image is 32x32 pixel in size and has 3 color channels (RGB).
  A label is associated to every image, specifying from which class the image belongs to.
  A CNN training with this dataset is harder than a training with MNIST, mainly due to the images, that are in full colors, the relatively small size of images and the diversity between classes. These factors provides more diversity and a more challenging classification task.

In the subsequent analysis (presented in Sec. 3), we will look only at the results obtained by training with CIFAR10 dataset, because MNIST is an extremely easy dataset to learn image classification. Indeed, results (that are still available at the previous link) shows very high accuracies even after the first few epochs of training.

2.2. Model and hyperparameters

Hyperparameters are adjustable settings that have to be decided before the training starts. Depending on the value assigned to each hyperparameters, training could have different results. A subdivision of the hyperparameters could be done by distinguishing the ones that have to be specified within the CNN model and the ones related to the training process (explained in Sec. 2.3).
In this section the model hyperparameters will be presented.
A base model is considered within this project, which has a structure of two Convolutional Layers and two Linear Layers, and the principal hyperparameters are mainly related to the Convolutional Layers:

- **in_channels** and **out_channels**: the first one specifies the number of channels in the input image or feature map, while the second one indicates the number of kernels used by the layer (controls the number of different feature maps produced).

- **kernel_size**: specifies the height and width of the convolution filter (kernel). It is set as 5x5 in the base model.

- **stride**: controls the step size of the convolution filter as it moves across the input image. Useful to reduce output spatial dimensions, it is set with the default value of 1 in the base model.

- **padding**: indicates how many layer of zeros are added around the input image. This parameter helps retaining more information in correspondence of the edges

of the images, and it also prevents excessive spatial reduction by simply introducing a border around the image without informative content. It is set with the default value of 0 in the base model.

Other hyperparameters include those associated with the kernel_size for the Max Pooling opration associated with Convolutional Layer and the number of both Convolutional and Fully Connected Layers (the experiments will also consider models with only fully connected layers).
in_channels and out_channels are also related to fully connected layers, but in two cases their value is not arbitrary: first Linear Layer after Max Pooling have in_channels set as the dimension of the output of the convolutional block, and the last Linear Layer have out_channels set as the number of classes.

2.3. Optimizers and Schedulers

The hyperparameters related to the training process are mainly the number of epochs, the batch size and the learning rate. This last one in particular is also related to the chosen optimization algorithm that updates the weights of the network, and it is a crucial parameter because if it's too high, the updates might overshoot the optimal values, while if it's too low convergence might be slow. In this project two optimizers are considered:

- **Stochastic Gradient Descent** (**SGD**): iteratively adjusts the model's weights based on the gradients of the loss with respect to those weights. In each iteration, SGD updates the model's weights ω by moving in the direction opposite to the gradient of the loss function:
  $\omega = \omega - \eta \cdot \nabla L(\omega)$
  where η is the learning rate and $\nabla L(\omega)$ is the gradient of the loss with respect to ω.
  An extension of this algorithm called SGD with **Momentum** is also considered, which incorporates past gradient information to smooth out updates and speed up convergence.

- **Adam**: adapts the learning rate for each parameter based on both past gradients (momentum) and the average of squared gradients (from another optimizer, RMSprop, which aims to adjust the learning rate based on recent changes to avoid large steps that could overshoot the minimum). This is the weight update rule:
  $\omega = \omega - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon}$
  where η is the learning rate, ϵ is a small constant to avoid division by zero, $\hat{m}_t$ is the first moment (mean of gradients), and $\hat{v}_t$ is the second moment (uncentered variance of gradients).

Furthermore, **schedulers** can also be used to adjust the value of the learning rate during training. These schedulers are particularly useful for avoiding plateaus and, consequently, local minima. In the experiments presented in Sec. 3, two schedulers are considered:

- **MultiStepLR**: used to decrease the learning rate at specified points during training by indicating the epoch(s) where to apply the reduction and a multiplicative factor.

- **ReduceLROnPlateau**: dynamically reduces the learning rate when a specific performance metric stops improving (typically on a plateau).

3. Results

In the following subsections, the results of the experiments conducted will be presented. Each of the subsequent tables will describe the values of the main hyperparameters and the techniques used for the specific experiment.
The different impact between SGD and Adam optimizer was analized with the MNIST dataset, in the following experiments SGD with momentum will always be the training optimizer, and where it is not specified differently, its associated hyperparameters will always be lr=0.001 and momentum=0.9.
Three main scenarios will be considered: training a model from scratch, training a predefined model from torchvision, and fine-tuning a pretrained model. For each scenario, most of the times the main hyperparameters will be varied individually with respect to the ones of an initial base model.
Some considerations on convergence will be based on the training loss values, available at the previously mentioned GitHub link.

3.1. Training a CNN from scratch

In this first experiment, the images are normalized without any form of data augmentation. The training parameters are varied (as showed Tab. 1) while keeping the same base model, consisting of 2 Convolutional Layers (kernel_size=5, pooling=0, stride=1) with Max Pooling and 2 Linear Layers. The focus will be on the utilize of learning rate Schedulers.
From the first table, we can observe how the Schedulers are impactful when there are fewer epochs and when the learning rate is increased by a right value. This is probably due to the fact that whit an higher learning rate the convergence speed of the training increases, as well as the risk of a plateau, and the Scheduler helps to deal with that.
In the second table, the impact of data augmentation on the training dataset is considered. The training parameters are set like the ones in the first row of Tab. 1, and the model is still the previous base one.
As we can see from the results in Tab. 2, with a low number of epochs the process of data augmentation seems irrelevant, this is because the training is harder and slower (as

Training Parameters	Test Accuracy
epochs=10, lr=0.001, momentum=0.9	0.57
epochs=10, MultiStepLR, lr=0.01, momentum=0.9	**0.59**
epochs=10, ReduceLROnPlateau, lr=0.1, momentum=0.9	0.50
epochs=50, lr=0.001, momentum=0.9	**0.63**
epochs=50, MultiStepLR, lr=0.01, momentum=0.9	0.61
epochs=50, ReduceLROnPlateau, lr=0.1, momentum=0.9	0.57

Table 1. Table with classification accuracy values varying with the training hyperparameters, focusing on learning rate Schedulers.

Epochs, Transformation	Test Accuracy
epochs=10, no Transform	**0.57**
epochs=10, Strong Transform	0.49
epochs=10, Stronger Transform	0.47
epochs=10, Base Transform	0.52
epochs=50, no Transform	0.63
epochs=50, Strong Transform	**0.67**
epochs=50, Base Transform	**0.67**

Table 2. Table with classification accuracy values varying with the augmentation associated with the training dataset. Strong transform indicates data augmentations that impact the training speed, while base transform indicates the image transforms that are usually done with image classification task (crop and flip).

the transforms become stronger). With an higher number of epochs, the beneficts starts to show, as the classification accuracy with test set is better if compared with the training without data augmentation.
In the next table there will be results related to the variation of the model hyperparameters. All the training results are done with 10 epochs.

Model	Test Accuracy
FC Model	0.48
FC Model with 8 layers	0.10
Base CNN	0.57
CNN, kernel_size=3	0.55
CNN, kernel_size=1	0.48
CNN, kernel_size=7	0.51
CNN, kernel_size=5, padding=2	**0.62**
CNN, kernel_size=7, padding=3	0.59
CNN, stride=2	0.46

Table 3. Table with classification accuracy values varying with the model hyperparameters configuration.

First of all, in Tab. 3 the FC model with 8 layers stands out, as it has very low accuracy. This is due to the vanishing gradient problem, which can be addressed by implementing skip connections between the layers.
Looking at CNNs, changing the stride does not have a significant impact on training, likely because CIFAR-10 images have low resolution, and increasing the stride risks to compress the feature maps too much. Similarly, changing the kernel size does not improve training compared to the base model. Better results are only achieved if the change in kernel size is paired with a padding value that allows the kernel to be centered on the edges of the image.
Now, there is an attempt to improve some of the previous results with the ReduceLROnPlateau scheduler, again over 10 epochs.

Model, Scheduler	Test Accuracy
FC Model, No Scheduler	0.48
FC Model, Scheduler lr=0.001	0.50
CNN, kernel_size=5, padding=2, No Scheduler	0.62
CNN, kernel_size=7, padding=3, No Scheduler	0.59
CNN, kernel_size=5, padding=2, Scheduler lr=0.001	0.49
CNN, kernel_size=7, padding=3, Scheduler lr=0.001	0.61
CNN, kernel_size=5, padding=2, Scheduler lr=0.01	**0.66**
CNN, kernel_size=7, padding=3, Scheduler lr=0.01	0.63
CNN, stride=2, No Scheduler	0.46
CNN, stride=2, Scheduler lr=0.001	0.47
CNN, stride=2, Scheduler lr=0.01	0.52

Table 4. Table with classification accuracy values varying with the model hyperparameter configuration and the utilize of a Scheduler.

The results in Tab. 4 further confirm what was observed in Tab. 1, namely that using a learning rate scheduler improves training convergence with fewer epochs and a higher initial learning rate. The best model is again the one with kernel_size=5 paired with padding=2.

3.2. Training a torchvision predefined CNN model

The predefined models from torchvision considered are AlexNet, ResNet18, and VGG16. Each of the three models is trained for 30 epochs, and since their architecture is designed for the ImageNet dataset, the final FC layer of each model is modified to produce a probability distribution over 10 classes (the CIFAR10 oens).
As showed in Tab. 5, AlexNet and VGG16 have the best performance, but AlexNet is trained with the training images resized to (469, 387), like ImageNet, because otherwise the images would have been comprimed too much, resulting in an error, so I will not consider it as the best model. Further-

Model	Test Accuracy
Best CNN Model	0.68
Best CNN Model, Scheduler lr=0.01	0.73
AlexNet	**0.77**
ResNet18	0.67
ResNet18, Scheduler lr=0.01	0.76
VGG16	**0.77**

Table 5. Table with classification accuracy values varying with the predefined model trained. In the second and fifth row, a ReduceLROnPlateau is used to improve training results. In the first row there is the better model configuration came from the results in Sec. 3.1 and trained also for 30 epochs with the training dataset augmented by Base Transform.

more, with a ReduceLROnPlateau scheduler, also the Best CNN Model and ResNet18 model have comparable performances with VGG16.

3.3. Fine-tuning pretrained models

Finally, in these last experiments, we will evaluate the performance of the ResNet18 model pretrained on ImageNet and fine-tuned for the image classification task on CIFAR10.

Fine-tuning is a technique in deep learning where a pretrained model is adapted to perform well on a new, often smaller, dataset or a similar task. The fine-tuning is done by freezing some layer weights, while the other layer weights are updated again depending on the value of the loss function with the new dataset. In this experiment, the fine-tuning is done for 10 epochs, the objective is to understand which and how many layers of the pretrained ResNet18 model to fine-tune.

Fine-tuned layers	Test Accuracy
FC Layer	0.35
FC Layer and Layer4	0.63
FC Layer, Layer4 and Layer3	0.75
Fc Layer, Layer4, Layer3 and Layer2	0.798
All Layers	**0.81**

Table 6. Table with classification accuracy values varying with the fine-tuned Layers. For the FC Layer the learning rate was set to 0.1, while the learning rate for the other inner Layers considered was set to 0.001.

By observing the results in Tab. 6, it is clear that increasing the number of fine-tuned layers does improve classification accuracy on the new dataset. However, it also increases the number of parameters to update and, consequently, the complexity and duration of the training process. Since significant improvements are only seen up to the third row, the next and final experiment will consider a fine-tuning process involving the FC Layer and Layers 3 and 4 of the pretrained ResNet18 model.

The next table will evaluate the impact of varying the fine-tuning hyperparameters.

Fine-tuning hyperparameters	Test Accuracy
Base parameters	0.75
All 3 lr=0.01	0.747
FC head lr=0.001, inner layers lr=0.1	**0.781**
All 3 lr=0.1	0.747
3 Linear Layers in FC head	0.782
4 Linear Layers in Fc heaf	**0.786**

Table 7. Table with classification accuracy values varying with the fine-tuning setup.

Looking at Tab. 7, the best performance is achieved by setting the learning rate for the FC Layer to 0.1 and the learning rate for the two inner Layers to 0.001. This is likely because, over 10 epochs, we encourage the new classification head to learn quickly, while the two inner layers, which were already pretrained, can adjust their weights more conservatively and still adapt to the new task.

From the last two row it's clear that modifying the last FC Layer by adding more Linear Layers improves slightly the performances, and thus it is not required.

4. Conclusions

In this project, various configurations and strategies to train a CNN model with the CIFAR10 dataset have been explored, in order to accomplish the image classification task. The three main strategies are training a CNN from scratch (with the optimal parameters being the ones described in the second row of Tab. 5), training a predefined torchvision model (the best one according to the results is VGG16) and fine-tuning a pretrained ResNet18 (in Tab. 7 there is the best configuration).

Overall, fine-tuning seems to have better performance in terms of classification accuracy, and it is also the faster strategy in terms of training time, but the other two strategies remains valid (their performance is just slightly worse than the fine-tuning). Obviously, the experiments conducted in this project are only a small portion of all possible alterations associated with hyperparameters, so there may be configurations not considered that could lead to better results.

In the code provided by the link in Sec. 2, I also explored the early stopping technique, which exploits a validation dataset to avoid overfitting during the last epochs of training. This could be crucial in the configurations where the learning rate is set with higher value and the optimizer is paired with a scheduler, because there is no security on avoiding an eventual local minimum, but I did not find a configuration in which it improved the performance.