Homework 3 Report **Piero Macaluso s252894**
Machine Learning and Artificial Intelligence 2018/2019 Due Date: 19/01/2019
Prof. Barbara Caputo

Deep Learning Setup

For this homework I set up the environment on my personal laptop: the durations shown in the report refer to the following specifications. The dataset used in this homework is **CIFAR 100**.

Hardware

- **Laptop Model** Dell Inspiron 7559
- **CPU** Intel® Core™ i7-6700HQ CPU @ 2.60GHz x 8
- **Video Card** NVIDIA GeForce GTX 960M
- **RAM** 16GB

Software

- **OS** Ubuntu 18.04.1 LTS
- **PyTorch** v. 1.0.0
- **TorchVision** v. 0.2.1

1 Old Neural Networks

Algorithm 1: Old Neural Network class (`NN1.py`)

```
119 class old_nn(nn.Module):
120     def __init__(self):
121         super(old_nn, self).__init__()
122         self.fc1 = nn.Linear(32 * 32 * 3, 4096)
123         self.fc2 = nn.Linear(4096, 4096)
124         self.fc3 = nn.Linear(4096, n_classes) # last FC for classification
125 
126     def forward(self, x):
127         x = x.view(x.shape[0], -1)
128         x = F.sigmoid(self.fc1(x))
129         x = F.sigmoid(self.fc2(x))
130         x = self.fc3(x)
131         return x
```

In the first part of this homework I trained a traditional neural network with 2 hidden FC (Fully connected) layers and a last FC for classification. The parameters were: **256 batch size**, **20 epochs**, **32x32 resolution**, **0.0001 Adam Solver learning rate**. The class of this network is in algorithm 1.

The training lasted about 5 minutes and I obtained an accuracy of **27%** and a loss of 2.682 as wee can see in fig. 1 on the next page. These are not good results: the accuracy is very low and the loss is high. I expected these results because of the NN (Neural Network) provided is implemented using only FC (Fully Connected) layers.

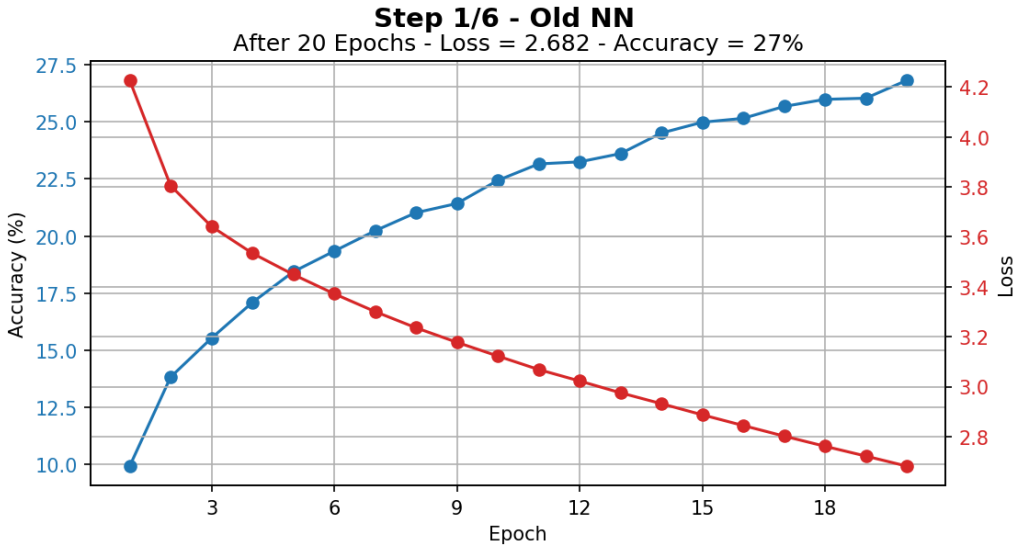

Figure 1: Old NN Loss Accuracy graph

An FC Layer links each pixel to all neurons, so it produces a large number of metric that the network has to learn. This NN can end up with **overfitting**, because the network will learn too much without acquiring the ability to generalize on the test set and perhaps this is what could have led to these results.

2 Simple CNN

In the second part of this homework I trained a CNN (Convolutional Neural Network) architecture. The parameters were: **256 batch size**, **20 epochs**, **32x32 resolution**, **0.0001 Adam Solver learning rate**. The class of this network is in algorithm 2.

Algorithm 2: Simple Convolutional Neural Network class (`CNN2.py`)

```
135 class CNN(nn.Module):
136     def __init__(self):
137         super(CNN, self).__init__()
138         self.conv1 = nn.Conv2d(3, 32, kernel_size=5, stride=2, padding=0)
139         self.conv2 = nn.Conv2d(32, 32, kernel_size=3, stride=1, padding=0)
140         self.conv3 = nn.Conv2d(32, 32, kernel_size=3, stride=1, padding=0)
141         self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
142         self.conv_final = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=0)
143         self.fc1 = nn.Linear(64 * 4 * 4, 4096)
144         self.fc2 = nn.Linear(4096, n_classes)
145 
146     def forward(self, x):
147         x = F.relu(self.conv1(x))
148         x = F.relu(self.conv2(x))
149         x = F.relu(self.conv3(x))
150         x = F.relu(self.pool(self.conv_final(x)))
151         x = x.view(x.shape[0], -1)
152         x = F.relu(self.fc1(x))
153         x = self.fc2(x)
154         return x
```

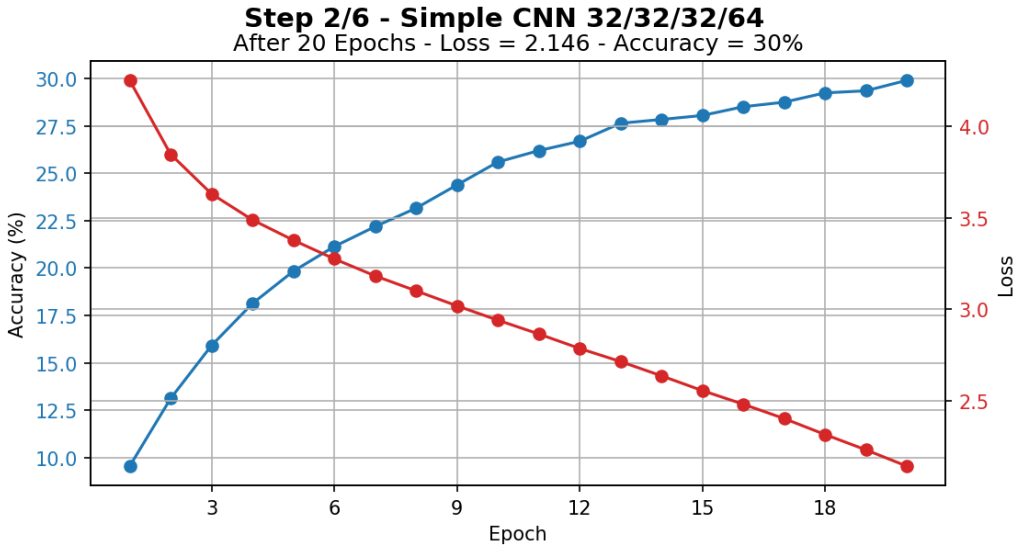

Figure 2: Simple CNN Loss Accuracy graph

The training lasted about 2 minutes and I obtained an accuracy of **30%** and a loss of 2.146 as wee can see in fig. 2. This is better than the previous NN, but not enough: the loss is still too high. This may be due, once againg, to overfitting, because we are using a CNN without any type of regularization or improving techniques which can led to better results as we will see in next sections.

3 CNN and convolutional filters

In the third part of this homework I trained a CNN architecture using different numbers of convolutional filters. The parameters were: **256 batch size**, **20 epochs**, **32x32 resolution**, **0.0001 Adam Solver learning rate**. The classes of these networks are in algorithms 3 to 5 on pages 3–4 .

Algorithm 3: Convolutional Neural Network 128/128/128/256 class (`CNN3a.py`)

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 128, kernel_size=5, stride=2, padding=0)
        self.conv2 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=0)
        self.conv3 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=0)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.conv_final = nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=0)
        self.fc1 = nn.Linear(256 * 4 * 4, 4096)
        self.fc2 = nn.Linear(4096, n_classes)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        x = F.relu(self.pool(self.conv_final(x)))
        x = x.view(x.shape[0], -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Algorithm 4: Convolutional Neural Network 256/256/256/512 class (`CNN3b.py`)

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 256, kernel_size=5, stride=2, padding=0)
        self.conv2 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=0)
        self.conv3 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=0)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.conv_final = nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=0)
        self.fc1 = nn.Linear(512 * 4 * 4, 4096)
        self.fc2 = nn.Linear(4096, n_classes)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        x = F.relu(self.pool(self.conv_final(x)))
        x = x.view(x.shape[0], -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Algorithm 5: Convolutional Neural Network 512/512/512/1024 class (`CNN3c.py`)

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 512, kernel_size=5, stride=2, padding=0)
        self.conv2 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=0)
        self.conv3 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=0)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.conv_final = nn.Conv2d(512, 1024, kernel_size=3, stride=1, padding=0)
        self.fc1 = nn.Linear(1024 * 4 * 4, 4096)
        self.fc2 = nn.Linear(4096, n_classes)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        x = F.relu(self.pool(self.conv_final(x)))
        x = x.view(x.shape[0], -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

The training lasted about 9 minutes for the first CNN (**32%** accuracy, **0.131** loss) , 25 minutes for the second one (**35%** accuracy, **0.078** loss) and 1 hour and 15 minutes for the last one (**36%** accuracy, **0.047** loss) as wee can see in figs. 3 to 5 on the following page. These are better than the previous CNN not only in terms of accuracy, but above all of loss. Accuracies are not so high, while in all three cases the loss is very close to 0. The parameters that the network has to learn increase with the number of filters. I noticed also that all lines (accuracy and loss) reach an asymptote. This may be due, once againg, to overfitting or co-adaptation, because, as in the previous section, we are using a CNN without any type of regularization or improving techniques which can led to better results as we will see in next sections.

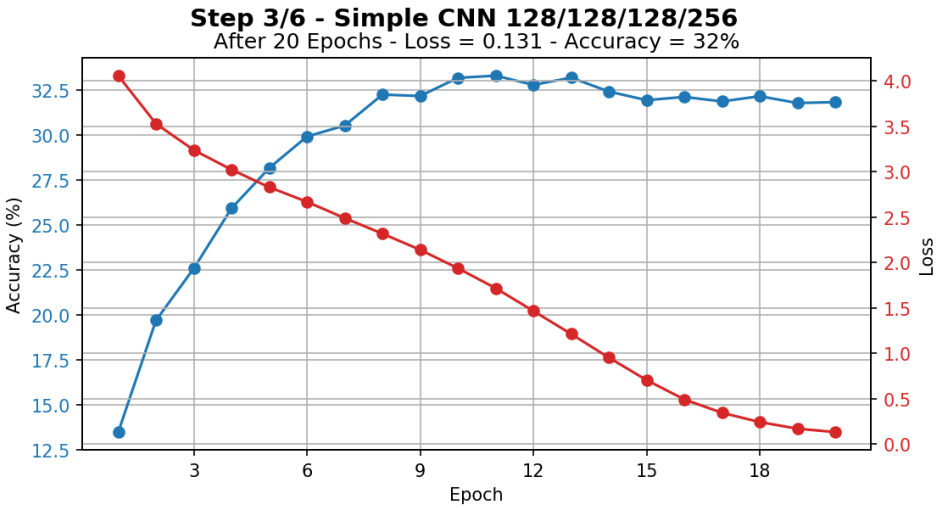

Figure 3: CNN 128/128/128/256 Loss Accuracy graph

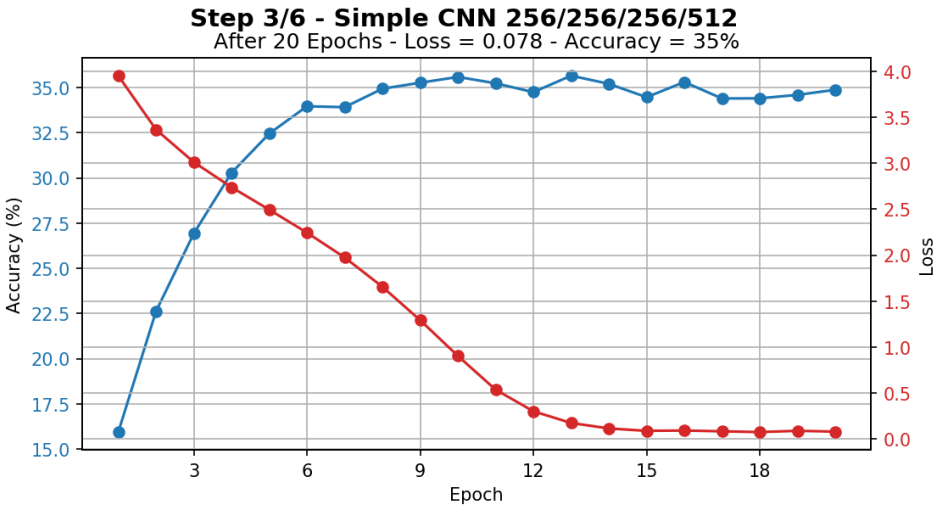

Figure 4: CNN 256/256/256/512 Loss Accuracy graph

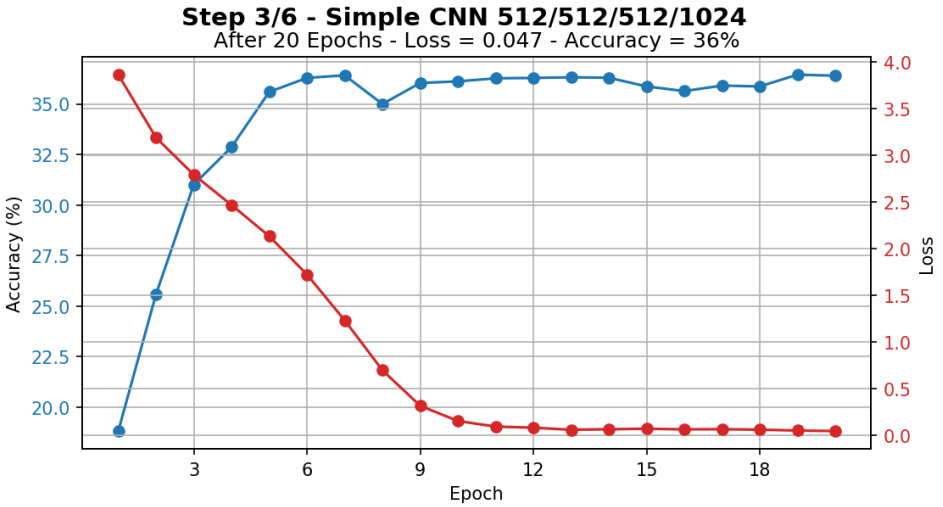

Figure 5: CNN 512/512/512/1024 Loss Accuracy graph

4 Regularization and Improving techniques

Algorithm 6: Convolutional Neural Network with BN (Batch Normalization) (`CNN4a.py`)

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 128, kernel_size=5, stride=2, padding=0)
        self.conv1_bn = nn.BatchNorm2d(128)
        self.conv2 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=0)
        self.conv2_bn = nn.BatchNorm2d(128)
        self.conv3 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=0)
        self.conv3_bn = nn.BatchNorm2d(128)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.conv_final = nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=0)
        self.conv_final_bn = nn.BatchNorm2d(256)
        self.fc1 = nn.Linear(256 * 4 * 4, 4096)
        self.fc2 = nn.Linear(4096, n_classes)

    def forward(self, x):
        x = F.relu(self.conv1_bn(self.conv1(x)))
        x = F.relu(self.conv2_bn(self.conv2(x)))
        x = F.relu(self.conv3_bn(self.conv3(x)))
        x = F.relu(self.pool(self.conv_final_bn(self.conv_final(x))))
        x = x.view(x.shape[0], -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Algorithm 7: Convolutional Neural Network with BN and FC1 wider (`CNN4b.py`)

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 128, kernel_size=5, stride=2, padding=0)
        self.conv1_bn = nn.BatchNorm2d(128)
        self.conv2 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=0)
        self.conv2_bn = nn.BatchNorm2d(128)
        self.conv3 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=0)
        self.conv3_bn = nn.BatchNorm2d(128)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.conv_final = nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=0)
        self.conv_final_bn = nn.BatchNorm2d(256)
        self.fc1 = nn.Linear(256 * 4 * 4, 8192)
        self.fc2 = nn.Linear(8192, n_classes)

    def forward(self, x):
        x = F.relu(self.conv1_bn(self.conv1(x)))
        x = F.relu(self.conv2_bn(self.conv2(x)))
        x = F.relu(self.conv3_bn(self.conv3(x)))
        x = F.relu(self.pool(self.conv_final_bn(self.conv_final(x))))
        x = x.view(x.shape[0], -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Algorithm 8: Convolutional Neural Network with BN and Dropout 0.5 on FC1 (`CNN4c.py`)

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 128, kernel_size=5, stride=2, padding=0)
        self.conv1_bn = nn.BatchNorm2d(128)
        self.conv2 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=0)
        self.conv2_bn = nn.BatchNorm2d(128)
        self.conv3 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=0)
        self.conv3_bn = nn.BatchNorm2d(128)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.conv_final = nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=0)
        self.conv_final_bn = nn.BatchNorm2d(256)
        self.fc1 = nn.Linear(256 * 4 * 4, 4096)
        self.fc2 = nn.Linear(4096, n_classes)

    def forward(self, x):
        x = F.relu(self.conv1_bn(self.conv1(x)))
        x = F.relu(self.conv2_bn(self.conv2(x)))
        x = F.relu(self.conv3_bn(self.conv3(x)))
        x = F.relu(self.pool(self.conv_final_bn(self.conv_final(x))))
        x = x.view(x.shape[0], -1)
        x = F.relu(self.fc1(x))
        x = F.dropout2d(x, p=0.5)
        x = self.fc2(x)
        return x
```

In the fourth part of this homework I trained a CNN architecture using different types of regularization and improving techniques starting from the network with 128/128/128/256 filters. The parameters were: **256 batch size**, **20 epochs**, **32x32 resolution**, **0.0001 Adam Solver learning rate**. The classes of these networks are in algorithms 6 to 8 on pages 6–7 .

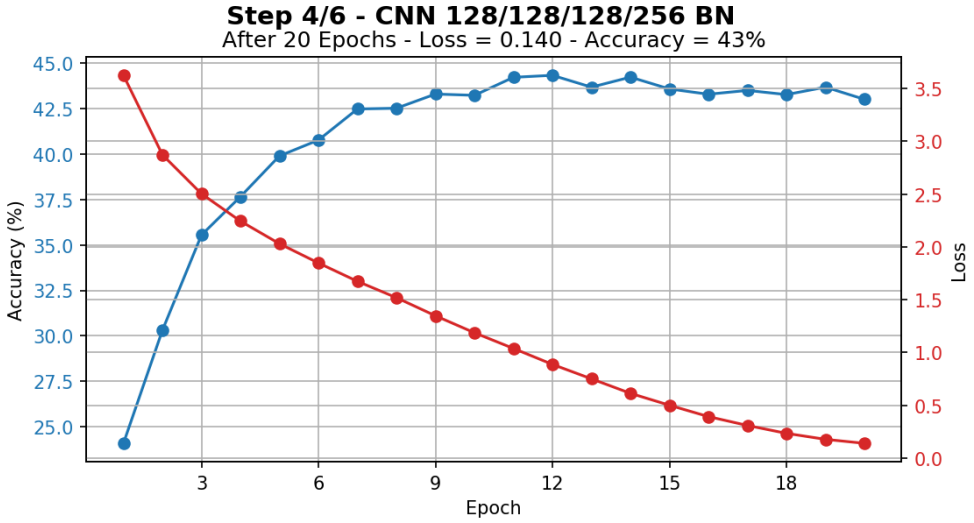

Figure 6: CNN with Batch Normalization Loss Accuracy graph

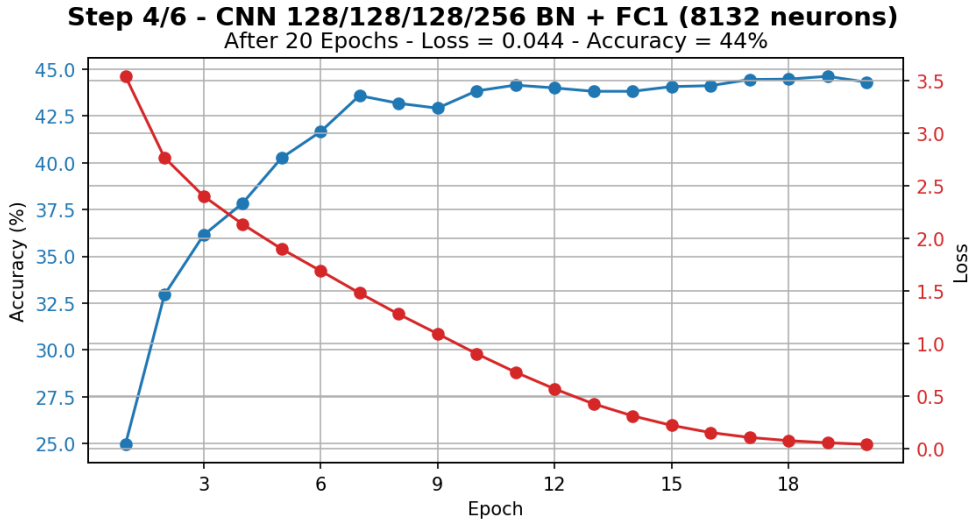

Figure 7: CNN with Batch Normalization and FC1 wider Loss Accuracy graph

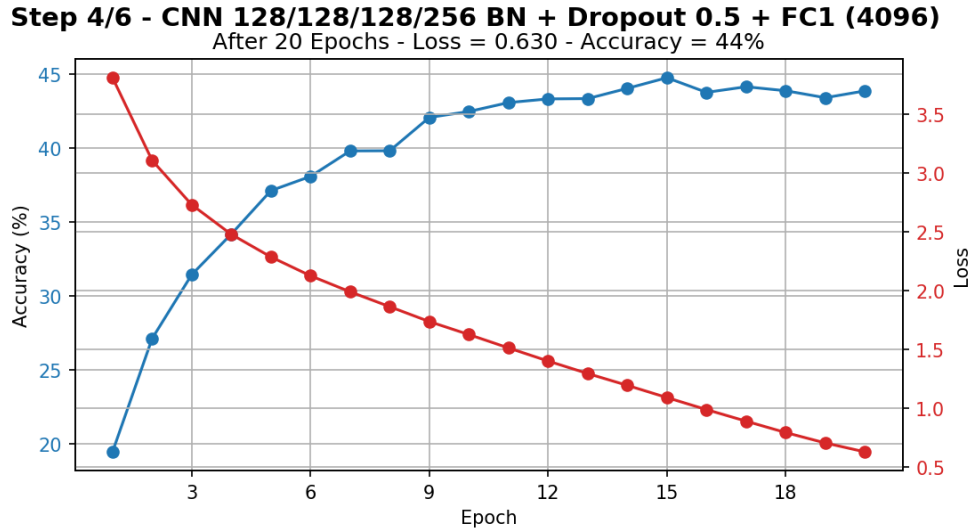

Figure 8: CNN with Batch Normalization and Dropout 0.5 on FC1 Loss Accuracy graph

The training lasted about 10 minutes for the first CNN (**43%** accuracy, **0.140** loss) , 12 minutes for the second one (**44%** accuracy, **0.044** loss) and 10 minutes for the last one (**44%** accuracy, **0.630** loss) as we can see the results in figs. 6 to 8 on pages 7–8. These are better than the previous CNN in terms of accuracy (10% higher), while the loss is almost as low as before. These results are better because we used regularization and improving techniques.

Batch Normalization is useful to normalize the data after each convolutional layer. We have to do this because after each convolutional layer the data is not normalized anymore and this slows down the process bringing convergence and the **Vanishing Gradient** problems. The mean and the variance is not know, but can be calculated on each batch of data at training time. Thanks to this the network learns means and variances and it uses them to approximate true mean and variance of the dataset.

Dropout is useful to prevent overfitting and co-adaptation. It consists in randomly dropping-out some neurons at training time with a specified rate. This is done usually on FC layers which are the ones where co-adaptation is more frequent.

5 Data Augmentation techniques

In the fifth part of this homework I trained a CNN architecture using different types of data augmentation the network with 128/128/128/256 filters.The parameters were: **256 batch size**, **20 epochs**, **32x32 resolution** for random horizontal flipping, **40x40 resolution** for random crop, **0.0001 Adam Solver learning rate**. We can see training transformation composition on these networks in algorithms 9 and 10.

Algorithm 9: Random Horizontal Flipping in training data (`CNN5a.py`)

```
162 transform_train = transforms.Compose(
163     [
164         transforms.RandomHorizontalFlip(),
165         transforms.Resize((32, 32)),
166         transforms.ToTensor(),
167         transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
168     ])
```

Algorithm 10: Random Crop on training data (`CNN5b.py`)

```
162 transform_train = transforms.Compose(
163     [
164         transforms.Resize((40, 40)),
165         transforms.RandomCrop(32),
166         transforms.ToTensor(),
167         transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
168     ])
```

The training lasted about 10 minutes for **Random Horizontal Flipping** (**36%** accuracy, **0.977** loss) and 10 minutes for **Random Crop** (**33%** accuracy, **2.107** loss) and the results are in figs. 9 and 10 on the following page. These are better than the CNN of section 2 in terms of accuracy, because with data augmentation the CNN applies random variations on the original dataset which can be seen as noise useful to prevent overfitting.

Random Horizontal Flipping is a simple modification that increases accuracy almost without affecting training complexity, as we can see from the low final loss value in our experiment. Instead, **Random Crop** is still a simple modification, but it affects training complexity, as we can see from the high final loss value, in order to obtain a better accuracy in test time. We can conclude that, in this case, **Random Horizontal Flipping** is the type of data augmentation which leads to better results.

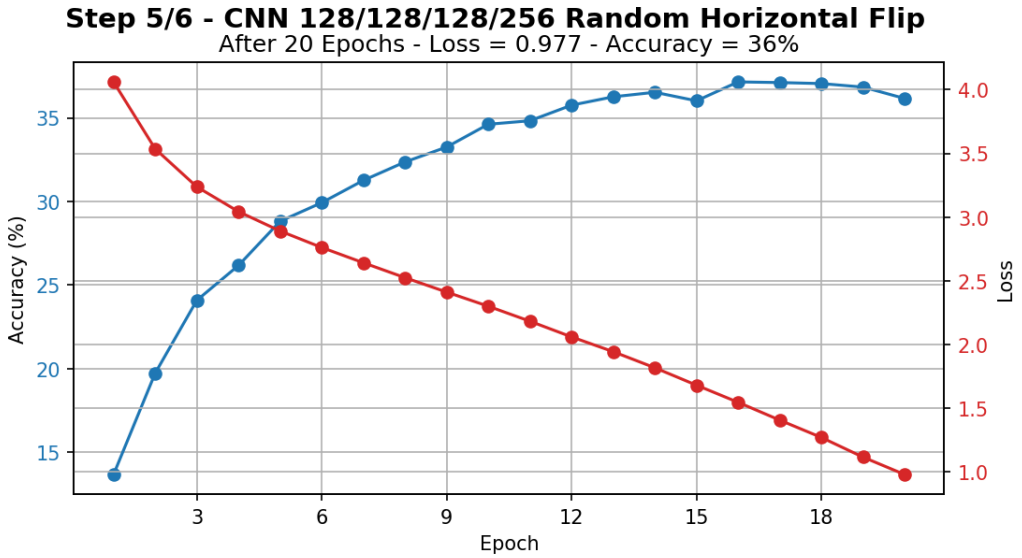

Figure 9: CNN with Random Horizontal Flipping Loss Accuracy graph

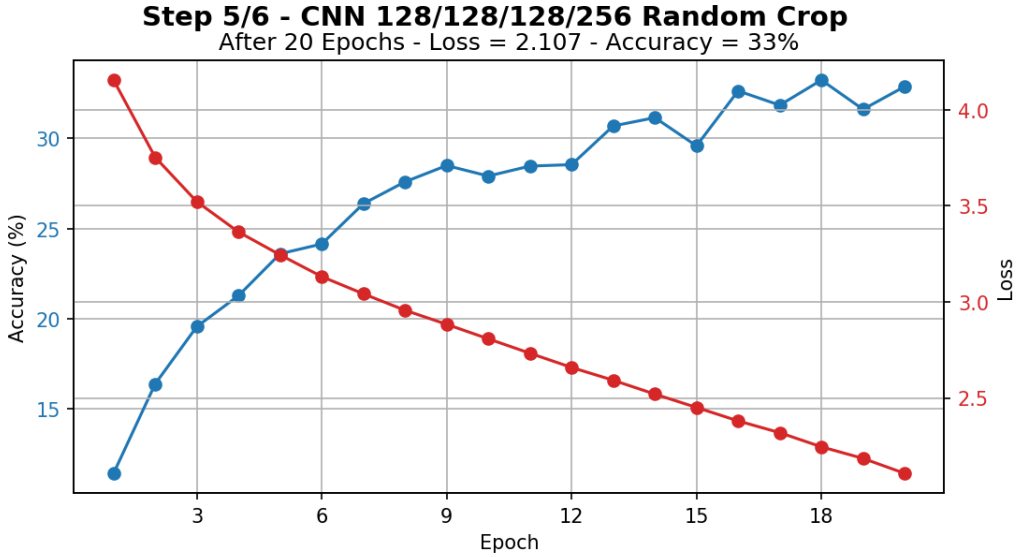

Figure 10: CNN with Random Crop Loss Accuracy graph

6 ResNet18

In the last part of this homework I performed finetuning using Random Horizontal Flipping on **ResNet18** pretrained on *ImageNet*. The parameters were: **128 batch size**, **10 epochs**, **224x224 resolution** and **0.0001 Adam Solver learning rate**.

The training lasted about 1 hour and 40 minutes with an accuracy of **80%** and a loss of **0.049** as we can see in fig. 11 on the next page. As I expected, the accuracy is very high and the loss is near to zero, so the ResNet18 produced very valuable results.

I noticed that the curve of loss goes to zero faster than other ones in all graphs of this report. This is a characteristics of ResNet18 which probably derives by its model (see fig. 12 on the following page). A Residual Network is able to keep the gradient clean and avoid **Vanishing Gradient Problem** and thanks to this it can leads to better results.

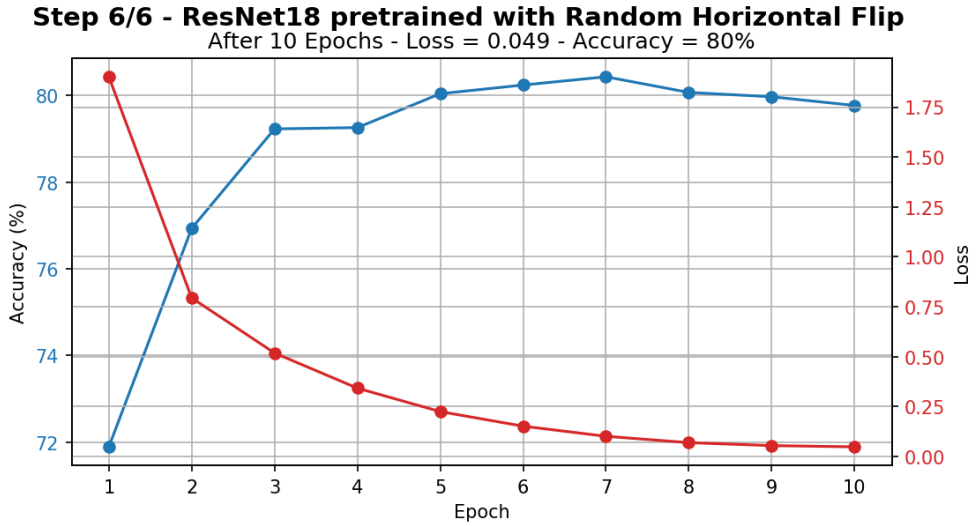

Figure 11: ResNet18 pretrained with Random Horizontal Flipping Loss Accuracy graph

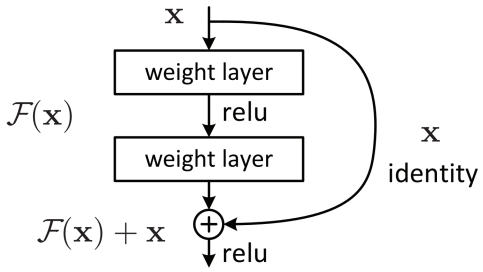

Figure 12: Residual Network: connection skipping

7 Code Execution

7.1 Requirements

- Python 3
- All dependencies in `requirements.txt`.

  `$ pip install -r requirements.txt` to install them

7.2 Usage

- `$ python main.py <python_script_cnn>`

  Execute the script of the specified `<python_script_cnn>` path among these:
  STEP 1: NN1.py
  STEP 2: CNN2.py
  STEP 3: CNN3a.py or CNN3b.py or CNN3c.py
  STEP 4: CNN4a.py or CNN4b.py or CNN4c.py
  STEP 5: CNN5a.py or CNN5b.py
  STEP 6: CNN6.py

Attachments

- `source_code` folder:
  - `main.py`
  - `requirements.txt`
  - `NN1.py` - First Step Source Code
  - `CNN2.py` - Second Source Code
  - `CNN3a.py`, `CNN3b.py`, `CNN3c.py` - Third Step Source Codes
  - `CNN4a.py`, `CNN4b.py`, `CNN4c.py` - Fourth Step Source Codes
  - `CNN5a.py`, `CNN5b.py` - Fifth Step Source Codes
  - `CNN6.py` - Sixth Step Source Codes
- `report` folder:
  - `hw3_report.pdf`
  - `img` folder - Images of the report
  - `output` folder - Output of codes epoch by epoch