Contents

Section 1

Overview

The objective of this report is to evaluate the performances obtained by solving an image classification task. In particular, the challenge is to come up with results against small datasets, training different network architectures and observing what changes if we change some element. This last process consists of testing optimizers and hyper-parameters like learning rate and weight decay, in order to find the way to reach the highest performance. To evaluate the obtained results we can use a simple accuracy metric for balanced datasets, while for unbalanced ones we need balanced accuracy, that is the arithmetic mean between sensitivity (true positive rate) and specificity (true negative rate)

Accuracy $A = (TP + TN) \,/\, (TP + FP + TN + FN)$

Sensitivity $TPR = TP \,/\, (TP + FN)$

Specificity $TNR = TN \,/\, (FP + TN)$

Balanced Accuracy $BA = (TPR + TNR) \,/\, 2$

where

True Class	Predicted Class: Yes	Predicted Class: No
Yes	True Positive (TP)	False Negative (FN)
No	False Positive (TP)	True Negative (TN)

The dataset used for our scope is ciFAIR-10, in which we find images of dimension $32 \times 32 \times 3$ belonging to ten classes. There are 50 training images per class and 1000 testing images per class. We expect that ensembles

of small-scale convolutional networks offer us good accuracy in small-data settings.

The base architectures exploited are ResNet, Wide ResNet and EfficientNet. The differences in the respective configuration give us the possibility of doing some comparisons and having insights on how the networks behave in the specific problem presented. For what concerns the choice of the optimizer, we take into account some popular alternatives such as SGD, Adam, Adadelta, Adagrad and RMSprop. Each of these optimizers tries to search on local minima of our loss function (Cross Entropy Loss) with different mathematical re-weighting patterns. Thus, aside from the common parameters, we can tweak some specific hyper-parameters, as detailed later.

Section 2

Architecture: ResNet

ResNet is the architecture that we can take as a baseline to make a lot of testing on optimizers and hyper-parameters. The great advantage of ResNet dwells in the fact that, even if the training phase is the most time-consuming, a complete execution can be carried out spending way fewer resources. This means that we can execute more tests in a reasonable amount of time, and make effective comparisons between one configuration and another. It can be observed that it takes about five minutes to train the network *rn-20* with a value of *epoch* equals to 100.

ResNet, residual neural network, is a neural network in which we add skip connections to jump over some layers in order to make more effective the phase of weight learning. Residual blocks, joint with ReLU (used as non-linearity function) and batch normalization, let us to fight the problems of "exploding gradient" and "vanishing gradient". After the convolutional layers, the last part of the network is simply an average pooling layer and a fully connected one. This architecture has proven to be of great interest in image classification.

```
def __init__(self, block, layers, num_classes=10, input_channels=3,
    shortcut_downsampling='pad', groups=1):
    ...

def forward(self, x):
    out = F.relu(self.bn1(self.conv1(x)))
    out = self.layer1(out)
    out = self.layer2(out)
    out = self.layer3(out)
    out = F.adaptive_avg_pool2d(out, 1).flatten(1)
    out = self.fc(out)
    return out
```

```
@staticmethod
def get_classifiers():
    return ['rn20', 'rn32', 'rn44', 'rn56', 'rn110', 'rn1202']

@classmethod
def build_classifier(cls, arch: str, num_classes: int, input_channels):
    _, depth = arch.split('rn')

    CONFIG = {20   : {'block' : BasicBlock, 'layers' : [3, 3, 3]},
              32   : {'block' : BasicBlock, 'layers' : [5, 5, 5]},
              44   : {'block' : BasicBlock, 'layers' : [7, 7, 7]},
              56   : {'block' : BasicBlock, 'layers' : [9, 9, 9]},
              101  : {'block' : BasicBlock, 'layers' : [18, 18, 18]},
              1202 : {'block' : BasicBlock, 'layers' : [200, 200, 200]},
              }

    cls_instance = cls(**CONFIG[int(depth)], num_classes=num_classes,
        input_channels=input_channels)
    return cls_instance
```

Section 3

Optimizer: SGD

To compute the loss of the network output we use the well-known Cross Entropy Loss function. The problem of founding the minimum of this function plays a central role because gives us the measure by which the learner provides an actual good classification of input images. The most famous optimizing method is *Gradient Descent*, which iteratively updates the weights of the network using partial derivates. It tries to reach a local minimum proceeding in the opposite direction with respect to the gradient of the function.

$$w_t \leftarrow w_{t-1} - \gamma \nabla_w f_t(w_{t-1})$$

The *Stochastic Gradient Descent* (SGD) is an effective variant to overcome some limitations of a "blind" gradient computation. In any case, a crucial factor is the correct setting of the learning rate (lr, γ), the coefficient that determines the updates. If the value is too small or too high, this can easily end up in a bad optimization process. The other parameter is weight decay (wd, λ), a regularization coefficient able to limit the value of the weights and, by doing this, to avoid the problem of over-fitting

$$w_t \leftarrow w_{t-1} - \gamma \nabla_w f_t(w_{t-1}) - \gamma \lambda w_{t-1}$$

At last, SGD has one more interesting hyper-parameters: momentum (μ). This is used to use the previous updates when we have to execute a new update to the weights

$$w_t \leftarrow w_{t-1} - \gamma \nabla_w f_t(w_{t-1}) + \gamma \mu \Delta w_{t-1}$$

Obviously, we can set a value of weight decay without momentum and vice versa, or keep both parameters.

Here are the results obtained with different values of the learning rate, weight decay and momentum (the number of epochs is fixed to 150)

Learning rate	Weight decay	Momentum	Nesterov	Accuracy
1e−3	5e−3	0.9	False	0.42
4e−3	5e−3	0.9	False	0.49
4e−3	5e−3	0.7	False	0.45
4e−3	5e−3	0.3	False	0.41
4e−3	1e−2	0.9	False	0.51
1e−2	1e−3	0.9	False	0.48
1e−2	1e−2	0.9	False	0.53
1e−2	1e−2	0.9	True	0.55
1e−2	1e−2	0.7	False	0.49
1e−2	1e−2	0.3	False	0.46
1e−2	1e−2	0.3	True	0.47

The code related to this optimizer is the following

```
@staticmethod
def default_hparams() -> dict:
    return {
        **super(CrossEntropyClassifier, CrossEntropyClassifier).
            default_hparams(),
        'momentum': 0.9,
        'nesterov': True
    }

def get_optimizer(self, model: nn.Module, max_epochs: int, max_iter: int):
  optimizer = torch.optim.SGD(model.parameters(),
             lr=self.hparams['lr'],
             weight_decay=self.hparams['weight_decay'],
             momentum=self.hparams['momentum']
             nesterov=self.hparams['nesterov'])

  scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer,
             T_max=max_iter)
  return optimizer, scheduler
```

Section 4

Optimizer: Adagrad

AdaGrad (adaptive gradient algorithm) is another optimization method based on the computation of the gradient. This is used in some contexts to improve the convergence of stochastic gradient descent, especially in a situation where data is sparse. Both learning rate and weight decay have the same function as before, except for the fact that the additional parameter lr decay (η) can progressively modify the value of learning rate

$$\hat{\gamma} \leftarrow \frac{\gamma}{1 + (t-1)\eta}$$

The difference in Adagrad is the update rule, a more structured than before

$$g_t \leftarrow \nabla_w f_t(w_{t-1}) + \lambda w_{t-1}$$
$$G_{i,t} \leftarrow \sum_{\tau=1}^{t} g_{i,\tau}^2$$
$$w_{i,t} \leftarrow w_{i,t-1} - \frac{\hat{\gamma}}{\sqrt{G_{i,t}} + \epsilon} g_{i,t}$$

Here are the results obtained with different values of learning rate, weight decay and lr decay (the number of epochs is fixed to 150)

Learning rate	Weight decay	LR decay	Accuracy
1e−3	1e−1	0	0.38
1e−3	1e−1	0.1	< 0.30
1e−2	1e−2	0	0.47
1e−2	1e−2	0.1	0.34
1e−2	1e−1	0	0.52
1e−2	1e−1	0.6	< 0.30

The code related to Adagrad is the following

```
@staticmethod
def default_hparams() -> dict:
    return {
        **super(CrossEntropyClassifier, CrossEntropyClassifier).
            default_hparams(),
        'lr_decay': 0
    }

def get_optimizer(self, model: nn.Module, max_epochs: int, max_iter: int):
  optimizer = torch.optim.Adagrad(model.parameters(),
            lr=self.hparams['lr'],
            weight_decay=self.hparams['weight_decay'],
            lr_decay=self.hparams['lr_decay'])

  scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer,
            T_max=max_iter)
  return optimizer, scheduler
```

Section 5

Optimizer: RMSprop

RMSprop (Root Mean Square Propagation) is again a gradient-based optimizer. It shares with Adagrad the idea of changing the learning rate with respect to each weight of the network. What changes is how we compute the factor of this modification

$$
\begin{aligned}
g_t &\leftarrow \nabla_w f_t(w_{t-1}) + \lambda w_{t-1} \\
v_{i,t} &\leftarrow \alpha\, v_{i,t-1} + (1-\alpha) g_{i,t}^2 \\
w_{i,t} &\leftarrow w_{i,t-1} - \frac{\gamma}{\sqrt{v_{i,t}} + \epsilon} g_{i,t}
\end{aligned}
$$

The presented formulas are the core of the optimizer, but it is important to know that we can enrich them with other manipulations. For example, we can use momentum in the update rule, in the same way as we have done for SGD. It is also possible to add a corrective computation to $v_{i,t}$ using the average of the previous gradients for a certain weight.

Here are the results obtained with different values of learning rate, weight decay, momentum and the parameter α (the number of epochs is fixed to 150)

Learning rate	Weight decay	Momentum	α	Accuracy
1e−3	1e−3	0.1	0.99	0.49
1e−3	1e−2	0.1	0.99	0.52
1e−3	1e−2	0.1	0.33	0.49
1e−3	1e−2	0.5	0.99	0.44
1e−3	1e−2	0.9	0.99	0.30
1e−3	1e−2	0.9	0.33	0.34
1e−2	1e−3	0.1	0.99	0.40

The code related to RMSprop is the following

```
@staticmethod
def default_hparams() -> dict:
    return {
        **super(CrossEntropyClassifier, CrossEntropyClassifier).
            default_hparams(),
        'alpha': 0.99,
        'momentum': 0.1
    }

def get_optimizer(self, model: nn.Module, max_epochs: int, max_iter: int):
  optimizer = torch.optim.RMSprop(model.parameters(),
             lr=self.hparams['lr'],
             weight_decay=self.hparams['weight_decay'],
             alpha=self.hparams['alpha'],
             momentum=self.hparams['momentum'])

  scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer,
             T_max=max_iter)
  return optimizer, scheduler
```

Section 6

Optimizer: Adam

Adam (Adaptive Moment Estimation) is the last optimizer used. The principle is all way similar to RMSprop. The differences are that, together with $v_{i,t}$, we use another adaptation term, $m_{i,t}$, each with its parameter β, and both are rescaled

$$
\begin{aligned}
g_t &\leftarrow \nabla_w f_t(w_{t-1}) + \lambda w_{t-1} \\
m_{i,t} &\leftarrow \frac{\beta_1 \, m_{i,t-1} + (1-\beta_1) g_{i,t}}{1-\beta_1} \\
v_{i,t} &\leftarrow \frac{\beta_2 \, v_{i,t-1} + (1-\beta_2) g_{i,t}^2}{1-\beta_2} \\
w_{i,t} &\leftarrow w_{i,t-1} - \frac{\gamma}{\sqrt{v_{i,t}} + \epsilon} m_{i,t}
\end{aligned}
$$

Even in this case, it exists a corrective action (*amsgrad*) useful to obtain a better convergence. It slightly changes the value of $v_{i,t}$ taking the maximum between the actual value and the previous ones.

Here are the results obtained with different values of learning rate, weight decay and the beta parameters (the number of epochs is fixed to 150)

Learning rate	Weight decay	(β_1, β_2)	amsgrad	Accuracy
1e−3	5e−3	$(0.9, 0.999)$	False	0.49
4e−3	5e−3	$(0.9, 0.999)$	False	0.47
1e−3	1e−2	$(0.9, 0.999)$	False	0.50
1e−3	1e−2	$(0.9, 0.999)$	True	0.52
1e−3	1e−2	$(0.6, 0.6)$	False	0.49
1e−3	1e−2	$(0.3, 0.3)$	False	0.48

The code related to Adam is the following

```
@staticmethod
def default_hparams() -> dict:
    return {
        **super(CrossEntropyClassifier, CrossEntropyClassifier).
            default_hparams(),
        'betas': (0.9, 0.999),
        'amsgrad': True
    }

def get_optimizer(self, model: nn.Module, max_epochs: int, max_iter: int):
  optimizer = torch.optim.Adam(model.parameters(),
             lr=self.hparams['lr'],
             weight_decay=self.hparams['weight_decay'],
             betas=self.hparams['betas'],
             amsgrad=self.hparams['amsgrad'])

  scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer,
             T_max=max_iter)
  return optimizer, scheduler
```

Section 7

Architecture: Wide ResNet

As the name itself suggests, Wide ResNet is a variation of the described ResNet architecture. It tries to get the better of ResNet, improving the block structures and then the results obtainable in classifying objects. We know that a wider network can find better solutions if we tune correctly the hyper-parameters. Wide ResNet is built to achieve this condition. Different configurations can be obtained by changing the depth and the widen factor, coming up with more layers and convolutional filters

```
def __init__(self, depth, num_classes, input_channels=3, widen_factor=1,
    dropRate=0.0, block=BasicBlock):
    ...

def forward(self, x):
    out = self.conv1(x)
    out = self.block1(out)
    out = self.block2(out)
    out = self.block3(out)
    out = self.relu(self.bn1(out))
    out = F.adaptive_avg_pool2d(out, 1).flatten(1)
    return self.fc(out)

@staticmethod
def get_classifiers():
    return ['wrn-16-8', 'wrn-16-10', 'wrn-22-8',
           'wrn-22-10', 'wrn-28-10', 'wrn-28-12']

@classmethod
def build_classifier(cls, arch: str, num_classes: int, input_channels):
    _, depth, widen_factor = arch.split('-')
    cls_instance = cls(int(depth), num_classes,
             input_channels=input_channels, widen_factor=int(widen_factor))
    return cls_instance
```

Here are the best results obtained with different optimizers and the network *wrn-16-8* (the number of epochs is fixed to 500 and the training time is more than 2 hours)

Optimizer	Learning rate	Weight decay	Hyper-params	Accuracy
SGD	1e−2	1e−2	$\mu = 0.9, \text{nest.} = T$	0.65
Adagrad	1e−2	1e−1	lr decay $= 0$	< 0.30
RMSprop	1e−3	1e−2	momentum $= 0.1$	0.52
Adam	1e−3	1e−2	$(\beta_1, \beta_2) = (0.9, 0.9)$	0.47

Best Model

The highest result can be observed in the first row of the table. It is useful to write the complete list of parameter used in the configuration:

architecture = wrn-16-8	optimizer = SGD
learning rate = 1e−2	weight decay = 1e−2
momentum = 0.9	nesterov = $True$
rand-shift = 4	epochs = 750
batch size = 10	evaluation interval = 10
normalize = $True$	hflip = $True$

All the parameters that are not in this list have the default value.

We can also draw the related plot