Low-resource Image Classification

Borghini Alessia

December 2021

Abstract

Image Classification is the task of assigning labels to images. Thanks to recent progresses in Computer Vision, with the introduction of Convolutional Neural Networks and Residual Blocks, high performance has been achieved in images tasks. However, the bottleneck is the amount of data because the more data are available for training the model the higher will be the accuracy. The goal of this work is to obtain an image classification model, trained on a small dataset, to be merged with other models in order to create an ensemble which achieves better performance than single models. I will compare architectures which use Convolutional Neural Networks with others that use Residual Blocks and study which methodologies are best for training on small dataset.

1 Introduction

Image Classification is the task of analyzing images, extracting their features and then assigning them specific labels. Every image represent an object to which a label corresponds.

A large amount of data is required to train a Deep Neural Networks that labels well the images. In this case, however, it will be assumed that the data available are few. In particular, the Machine Learning Models will be trained on a subset of ciFAIR-10 dataset [Barz and Denzler 2020]. ciFAIR-10 is a variant of ciFAR-10 dataset where images in the test set similar to those present in the train have been replaced with new ones. The dataset is divided into 10 classes and there are 50 images for each class for both training and evaluation.

The state-of-the-art architecture on this dataset is the Wide ResNet with standard cross-entropy [Brigato et al. 2021]. This model is obtained using Asynchronous HyperBand with Successive Halving (ASHA) to fine tune the hyper-parameters and achieves an accuracy value of 58,22%.

In Section 2 different Data Preprocessing methodologies will be shown, in particular some Data Augmentation techniques and the Optimizer used in the experiments. Subsequently, in Section 3 the Neural Architecture will be introduced and the results achieved from them will be presented and discussed in Section 4.

2 Data Preprocessing

2.1 Data Augmentation

Data Augmentation is an essential technique which aims to increase the amount of data adding images similar to those already existing.

For the ciFAIR-10 dataset the Data Augmentation strategy that fit better, to the best of my knowledge, is AutoAugment [Cubuk et al. 2018] combined with other standard methods. AutoAugment is an automated approach to find a successful policy, i.e. a combination of Data Augmentation methods for a given dataset. On ciFAR-10 dataset, color-based transformations are the mostly used (e.g. Equalize, AutoContrast, Color, Brightness) geometric transformations, instead, are not very effective. AutoAugment give a great increment of performance but alone is not enough, so I used this method in addition to the following Pytorch implemented ones[1]:

- Random Horizontal Flip: which flips the image with a given probability, in this case 0.5;
- Random Shift: which firstly Resize the image and then make a Random Crop in order to obtain a shift of 4 pixels;
- Normalization: which standardizes the data so that they would have a zero mean and a unit variance.

To achieve this I defined a subclass of `LearningMethod`, `My_DataAugmentation`, and I redefined the method `get_data_transforms` adding to the preexisting one the line showed in Listing 1.

```python
class My_DataAugmentation(LearningMethod):
    def get_data_transforms(self, dataset) -> Tuple[Callable, Callable]:
        # rest of the lines taken from the original "get_data_transforms" function
        ...
        transforms.append(tf.AutoAugment(tf.AutoAugmentPolicy.CIFAR10))
        ...
```

Listing 1: Extract of My Data Augmentation class from the file xent.py

2.2 Optimizer

The optimizer is the tools used to minimize the loss function with the purpose of obtain a model which converges to an optimal solution. In the experiments I used firstly Adam optimizer and then I compered the results obtained with ones obtained with SGD optimizer. Both the optimizer implementations are taken from Pytorch[2].

Adam optimizer [Kingma and Ba 2017] is defined as a fast optimizer even if not always converges to an optimal solution. It computes individual learning rates for different parameters and estimates the momentum. The parameters with a default value are the `betas` – that are the exponential decay rate for the first and second moment estimates – and `epsilon` – a value needed to prevent division by zero in the implementation. Then there are the `learning-rate` and the

[1] `https://pytorch.org/vision/stable/transforms.html#torchvision.transforms.AutoAugment`
[2] `https://pytorch.org/docs/stable/optim.html`

weights-decay whose values vary accordingly to the experiments. The learning-rate determines the size of the updating step of the weights, while the weights-decay is an additional term of the loss function which penalize the complexity of the function itself. The implementation of the class CrossEntropyClassifier in which Adam optimizer is defined is shown in the Listing 2.

```
class CrossEntropyClassifier(My_DataAugmentation):
    def get_optimizer(self, model: nn.Module, max_epochs: int, max_iter: int) ->
    Tuple[torch.optim.Optimizer, torch.optim.lr_scheduler._LRScheduler]:

        optimizer = torch.optim.Adam(
            model.parameters(),
            lr=self.hparams["lr"],
            betas=(0.9, 0.999),
            eps=1e-08,
            weight_decay=self.hparams["weight_decay"],
        )
        ...
```

Listing 2: Extract of CrossEntropyClassifier class from xent.py

SGD optimizer [Sutskever et al. 2013], on the other hand, for some architectures converges to a better solution and in a faster way. This is a stochastic approximation of gradient descent optimization which is faster than the classical gradient descent optimizer. Additionally it uses momentum like Adam optimizer. The parameters in this case are learning-rate, weights-decay and the momentum, which provides an update rule motivated from the physical perspective of optimization getting a speed boost. Further details about the results are provided in Section 4.

3 Architectures

In recent years, the advances in Machine Learning have lead to the formulation of Convolutional Neural Networks (CNN) first and then of the Residual Neural Networks. The CNN takes three-dimensional input, for example the RGB version of an image divided in three color planes, and returns in output an image easier to process without losing important features. This network is able of capturing the spatial and temporal dependencies in an image because it process the images selecting matrices of pixels instead of a sequence of pixels. A filter or kernel is applied to the pixels through a matrix multiplication, it is possible to chose the dimension of the filer, the stride with which the the filter is moved and the type of padding. After that the features are dimensionally changed a nonlinear function, a pooling layer and a fully-connected layer can be applied in order to classify the input. Training networks with ever more convolutional layers does not lead to infinite improvement, so to overcome the problem of using very deep neural networks the Residual Blocks were introduced. The Residual Blocks avoid information loss making skip connections between the shallow layers and the deep ones.

I decided to compare Deep Neural Networks, like VGG and AlexNet, with one that use Residual Blocks, like and PyramidNet. I did not include ResNet architecture in my experimental study on purpose because of the goal of this work that is to make an ensemble of classification models. ResNet is one of the best known architecture and therefore I decided to use different ones trying to make a contribution for the final ensemble. All the architecture implementations are taken from `https://pytorch.org/vision/stable/models.html#torchvision-models` excepting for PyramidNet (further details are provided in Section 3.3). In order to adapt the implementation to the code provided by the Professor I implemented for every architecture the following methods:

1. `get_classifiers()`
2. `build_classifier((cls, arch: str, num_classes: int, input_channels: int)`

The first one is to specify the names of the architectures implemented in the file. Every name correspond to a given implementation of the architecture, for example with a certain depth or width. The second one, instead, needs to initialize the architecture. After the definition of the parameters of the architecture it can be created and returned in output.

3.1 AlexNet

AlexNet [Krizhevsky, Sutskever, and Hinton 2012] is the first Deep Neural Network which stacked convolutional layer without pooling layer in between. It is composed by five convolutional layers and three fully-connected layers and then uses ReLU Nonlinearity and Local Response Normalization for a faster training and better performance. To avoid overfitting is used the Dropout [Hinton et al. 2012] which randomly zeroes some of the elements of the input tensor with a given probability during the train. The implementation provided by Pytorch does not use Local Response Normalization but uses the Average Pooling before the fully-connected layers. After some experiment I found that Local Response Normalization speed up training, but also the Average Pooling is helpful to reach better performance with Adam optimizer. With SGD, instead, the Pytorch version is the best one. With AlexNet architecture the images need to be resized to a dimension of 227x227 pixels. The following lines (Listing 3) shows the implementation above. The implementation is taken from Pytorch and then it was adapted to the code given by the Professor.

```python
class AlexNet(nn.Module):
    def __init__(self, num_classes: int = 10) -> None:
        super(AlexNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=96, kernel_size=11, stride=4),
            nn.ReLU(),
            nn.LocalResponseNorm(size=5, alpha=0.0001, beta=0.75, k=2),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(96, 256, 5, padding=2),
            nn.ReLU(),
            nn.LocalResponseNorm(size=5, alpha=0.0001, beta=0.75, k=2),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(256, 384, 3, padding=1),
            nn.ReLU(),
            nn.Conv2d(384, 384, 3, padding=1),
            nn.ReLU(),
            nn.Conv2d(384, 256, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )
        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes),
        )
```

Listing 3: AlexNet architecture

3.2 VGG

VGG [Simonyan and Zisserman 2015] is a very deep convolutional network used for the classification of large-scale images, nevertheless it works good also with 32x32 images. It makes an improvement over AlexNet by reducing the kernel sizes and by using Batch Normalization. Batch Normalization layer has the aim of normalize the activation function in order to converge quickly and improve performance. Among all the VGG networks, that differ between them in depth, I used the VGG 11 one with Batch Normalization. This is the smaller network, indeed it has eight convolutional layer, each one followed by batch normalization layer and max pooling layer, and three fully-connected layer with ReLU nonlinearity and dropout. In the Listing 4 there is an extract of the implementation, the complete version is available in the file vgg.py. The function `make_layers` creates a layer for every element of the `"cfg"` array. In particular, for every `"M"` which appears a Max Pooling layer (with a 3x3 kernel) is added, in the other cases a Convolutional Layer of the same dimension of the value, a Batch Normalization layer and ReLU is added.

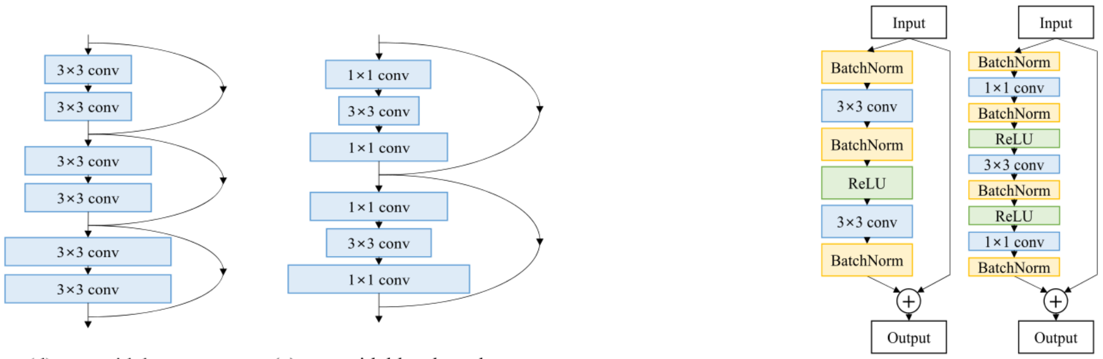
```
class VGG(nn.Module):
    def __init__(
        self, features: nn.Module, num_classes: int = 10, init_weights: bool =
        ↪  True
    ) -> None:
        super(VGG, self).__init__()
        self.features = features
        self.avgpool = nn.AdaptiveAvgPool2d((7, 7))
        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(4096, num_classes),
        )
        if init_weights:
            self._initialize_weights()
        ...

    @staticmethod
    def get_classifiers():
        return ["vgg11_bn", ...]

    @classmethod
    def build_classifier(cls, arch: str, num_classes: int, input_channels: int):
        kwargs = {}

        VGG_CONFIG = {
            "vgg11_bn": {"cfg": [64, "M", 128, "M", 256, 256, "M", 512, 512, "M",
            ↪  512, 512, "M"], "batch_norm": True},
            ...
        }
        model = VGG(make_layers(cfgs[VGG_CONFIG[arch]["cfg"]], batch_norm =
        ↪  VGG_CONFIG[arch]["batch_norm"]), **kwargs)
        return model
```

Listing 4: VGG with Batch Normalization architecture

3.3 PyramidNet

PyramidNet [Han, Jiwhan Kim, and Junmo Kim 2016] is an residual network in which the channel size gradually increases, adding a factor or multiplying it. This increment method is similar to the one used for VGG (Section 3.2) and ResNet [He et al. 2015]. Through the increasing of the feature map dimension diversified highlevel attributes are extracted from the images obtain advantage for classification tasks. Moreover, this design has proven to be an effective means of improving generalization ability.

In the paper, the authors use both Simple Block and the Bottleneck Block for their experiments. The main difference between them is that the bottleneck one uses `conv 1x1` layers, as shown in Figure 1, to reduce the number of channels encouraging the network to compress feature representations to best fit in the available space and getting the best loss during training. Both the blocks uses Rectified Linear Units (ReLUs) and Batch Normalization (BN) layers. ReLUs are necessary for nonlinearity but putting too much of them or putting them in the wrong location can reduce performance. Indeed, ReLUs are place before the convolutional layers instead of after. The Batch Normalization layer, instead, as described in Section 3.2 normalizes the activation function. In this case are placed after convolutional layers and at the end of the building blocks. Zero-padded identity-mapping shortcuts are chosen for all residual units because does not lead to overfitting as there are not additional parameters and additionally increase the generalization ability.



(d) pyramidal (e) pyramidal bottleneck

Figure 1: The simple blocks and bottleneck blocks structure (left) and the layers (right).

ShakeDrop [Yamada, Iwamura, and Kise 2018] is a regularization method inspired from the classical RandomDrop – also known as Stochastic Depth – and Shake-Shake – which can applied only to ResNeXt. The main role of the regularization is to improve generalization and thus also the performance. To achieve this, the strategy adopted is to disturb the learning process by multiplying even a negative factor to the output of a convolutional layer in the forward training pass. The ShakeDrop function is reported in Listing 5.

```
class ShakeDropFunction(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, training=True, p_drop=0.5, alpha_range=[-1, 1]):
        if training:
            gate = torch.cuda.FloatTensor([0]).bernoulli_(1 - p_drop)
            ctx.save_for_backward(gate)
            if gate.item() == 0:
                alpha = torch.cuda.FloatTensor(x.size(0)).uniform_(*alpha_range)
                alpha = alpha.view(alpha.size(0), 1, 1, 1).expand_as(x)
                return alpha * x
            else:
                return x
        else:
            return (1 - p_drop) * x

    @staticmethod
    def backward(ctx, grad_output):
        gate = ctx.saved_tensors[0]
        if gate.item() == 0:
            beta = torch.cuda.FloatTensor(grad_output.size(0)).uniform_(0, 1)
            beta = beta.view(beta.size(0), 1, 1, 1).expand_as(grad_output)
            beta = Variable(beta)
            return beta * grad_output, None, None, None
        else:
            return grad_output, None, None, None
```

```python
class ShakeDrop(nn.Module):
    def __init__(self, p_drop=0.5, alpha_range=[-1, 1]):
        super(ShakeDrop, self).__init__()
        self.p_drop = p_drop
        self.alpha_range = alpha_range

    def forward(self, x):
        return ShakeDropFunction.apply(x, self.training, self.p_drop,
        ↪  self.alpha_range)
```

Listing 5: ShakeDrop function

I compared the following configurations of the PyramidNet architecture:

1. **PyramidNet with SimpleBlocks**: implementation adapted from `https://github.com/dyhan0920/PyramidNet-PyTorch`;
2. **PyramidNet with BottleneckBlocks**: implementation adapted from `https://github.com/dyhan0920/PyramidNet-PyTorch`;
3. **PyramidNet with SimpleBlocks and ShakeDrop**: implementation is adapted by `https://github.com/owruby/shake-drop_pytorch`;
4. **PyramidNet with BottleneckBlocks and ShakeDrop**: the implementation was not provided, so I implemented this architecture by myself following the specifications of the paper and taking inspiration by the repositories above.

The implementation of the previously explained architecture can be found in the files "pyramidnet_shake.py" and "pyramidnet.py". The code of the Bottleneck block with ShakeDrop that I implemented is reported in the Listing 6.

```python
class ShakeBottleneckBlock(nn.Module):
    outchannel_ratio = 4

    def __init__(self, in_ch, out_ch, stride, p_shakedrop=1.0):
        super(ShakeBottleneckBlock, self).__init__()
        self.downsampled = stride == 2
        self.branch = self._make_branch(in_ch, out_ch, stride=stride)
        self.shortcut = (
            not self.downsampled and None
            or nn.AvgPool2d((2, 2), stride=(2, 2), ceil_mode=True)
        )
        self.shake_drop = ShakeDrop(p_shakedrop)

    def _make_branch(self, in_ch, out_ch, stride=1):
        return nn.Sequential(
            nn.BatchNorm2d(in_ch),
            nn.Conv2d(in_ch, out_ch, kernel_size=1, bias=False),
            nn.BatchNorm2d(out_ch),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_ch, (out_ch * 1), kernel_size=3, padding=1,
            ↪  stride=stride, bias=False),
            nn.BatchNorm2d((out_ch * 1)),
            nn.ReLU(inplace=True),
            nn.Conv2d(
                (out_ch * 1),
                out_ch * ShakeBottleneckBlock.outchannel_ratio,
                kernel_size=1,
                bias=False,
            ),
            nn.BatchNorm2d(out_ch * ShakeBottleneckBlock.outchannel_ratio),
        )

    def forward(self, x):
        h = self.branch(x)
        h = self.shake_drop(h)
        h0 = x if not self.downsampled else self.shortcut(x)
        pad_zero = Variable(
            torch.zeros(
                h0.size(0), h.size(1) - h0.size(1), h0.size(2), h0.size(3)
            ).float()
        ).cuda()
        h0 = torch.cat([h0, pad_zero], dim=1)
        return h + h0
```

Listing 6: Bottleneck Block with Shake Drop

4 Results

For all the tests I used the notebook provided by the Professor to launch the code. In particular modifying the lines in the Listing 7. In particular, in every experiment I chose the `architecture` to train, the `learning-rate` and the `weight-decay`. As data augmentation the best results are obtained, as introduced in Section 2.1, with AutoAugment and setting the parameter `normalize`, `hflip` and `rand-shift = 4`. A small `batch-size` lead to a better generalization, so I set this to 10. For the firsts epochs I used a `eval-interval` of 10 but then, in order to find the best model, I decrease this value to 1 and also the value of the `learning-rate`. Furthermore, it was possible to specify the path in which saves the model and the training history.

```
!python scripts/train.py cifair10 \
    --architecture ARCHITECTURE_NAME \
    --init-weights PATH \
    --normalize \
    --rand-shift 4 \
    --hflip \
    --epochs #EPOCHS \
    --batch-size 10 \
    --lr LEARNING_RATE \
    --weight-decay  WEIGHT_DECAY \
    --eval-interval 10 \
    --history HISTORY_PATH \
    --save SAVE_PATH
```

Listing 7: Experiment launcher

I made experiments with both Adam optimizer and SGD optimizer and I noticed that with Adam a lesser value of learning rate is needed compared to SGD. For this reason the number of epochs needed with Adam are more than SGD to achieve the same results, however SGD reaches higher performance so the total number of epochs are similar.

The results of the best experiments are reported in the Table 1.

Architecture	**optimizer**	**learning rate**	**weight decay**	**accuracy**
Wide ResNet (baseline)	SGD	4.55e-3	5.29e-3	0.582
VGG11 with BatchNormalization	Adam	0.0001	0.0001	0.571
VGG11 with BatchNormalization	SGD	0.001	0.001	**0.602**
AlexNet	Adam	0.0001	0.001	0.543
AlexNet	SGD	0.001	0.001	0.539
PyramidNet Simple	Adam	0.0001	0.0001	0.54
PyramidNet Simple ShakeDrop	Adam	0.0001	0.0001	0.57
PyramidNet Bottleneck	Adam	0.0001	0.0001	0.545
PyramidNet Bottleneck ShakeDrop	Adam	0.0001	0.0001	**0.59**
PyramidNet Bottleneck ShakeDrop	SGD	0.001	0.001	**0.61**

Table 1: Results of the experiments

5 Conclusions

In conclusion, the residual are important but in this case with a deep neural network without residual is possible to achieve similar results. The VGG network achieves indeed results close to those reached by PyramidalNet, especially thanks to the Batch Normalization layers. The Pyramidal architecture in both the configurations, with the basic blocks and the bottleneck blocks, is a good baseline. During the training the accuracies on the training set and the validation set rise together up to a certain point then the accuracy on the validation set slow down. The ShakeDrop combined with SGD optimizer gives another good improvement to the performance.

The chose of the optimizer is crucial because with this dataset and this architectures Adam does not converge, in the training, at the best solution. SGD, instead is fast and give an increment on all the experiment. With both the optimizer is a good choice to start the training with a higher learning and then lower it to find the best configuration of the weights. Having a small dataset, it is essential to increase it through data augmentation to avoid overfitting. From the experiments it emerged that using only the standard methods (e.g. horizontal flip and pixel shift) is not enough, however even using only AutoAugment the performance is low. On the other hand, when these methods are combined, performance increases significantly (by about 10 accuracy points). Weight decay also is a good option to make the model generalize better. I found that for VGG and PyramidNet with Adam the maximum value to make the model converge is 0.0001, with SGD 0.001. For Alexnet, instead, also with Adam the maximum possible value is 0.001.

References

Barz, Björn and Joachim Denzler (2020). "Do We Train on Test Data? Purging CIFAR of Near-Duplicates". In: *Journal of Imaging* 6.6. ISSN: 2313-433X. DOI: `10.3390/jimaging6060041`. URL: `https://www.mdpi.com/2313-433X/6/6/41`.

Brigato, Lorenzo et al. (Oct. 2021). "Tune It or Don't Use It: Benchmarking Data-Efficient Image Classification". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) Workshops*, pp. 1071–1080.

Cubuk, Ekin Dogus et al. (2018). "AutoAugment: Learning Augmentation Policies from Data". In: *CoRR* abs/1805.09501. arXiv: `1805.09501`. URL: `http://arxiv.org/abs/1805.09501`.

Han, Dongyoon, Jiwhan Kim, and Junmo Kim (2016). "Deep Pyramidal Residual Networks". In: *CoRR* abs/1610.02915. arXiv: `1610.02915`. URL: `http://arxiv.org/abs/1610.02915`.

He, Kaiming et al. (2015). "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385. arXiv: `1512.03385`. URL: `http://arxiv.org/abs/1512.03385`.

Hinton, Geoffrey E. et al. (2012). *Improving neural networks by preventing co-adaptation of feature detectors.* arXiv: `1207.0580 [cs.NE]`.

Kingma, Diederik P. and Jimmy Ba (2017). *Adam: A Method for Stochastic Optimization.* arXiv: `1412.6980 [cs.LG]`.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton (2012). "ImageNet Classification with Deep Convolutional Neural Networks". In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1.* NIPS'12. Lake Tahoe, Nevada: Curran Associates Inc., pp. 1097–1105.

Simonyan, Karen and Andrew Zisserman (2015). *Very Deep Convolutional Networks for Large-Scale Image Recognition.* arXiv: `1409.1556 [cs.CV]`.

Sutskever, Ilya et al. (2013). "On the importance of initialization and momentum in deep learning". In: *International conference on machine learning.* PMLR, pp. 1139–1147.

Yamada, Yoshihiro, Masakazu Iwamura, and Koichi Kise (2018). "ShakeDrop regularization". In: *CoRR* abs/1802.02375. arXiv: `1802.02375`. URL: `http://arxiv.org/abs/1802.02375`.