# Reproducing a ResNet based on G-CNN's evaluated through CIFAR10 and CIFAR10+ data

Casper van Engelenburg (4237080)
Philip de Rijk (4375017)

In this notebook we reproduce the ResNet44 results found in table 2 of the paper on Group Equivariant Convolutional Networks by T. Cohen (2016). In the experiment, conventional (Z2) and group equivariant (e.g. p4m) CNN's are compared on CIFAR10 and CIFAR10+. Qualitatively we could reproduce their most important finding: group equivariant convolutions in residual networks significantly reduces the test error rates. On the other hand, we were not able to get the error rates as low as they were originally mentioned. We think our way of training can be improved to reduce some apparant overfitting which may lead to better performance.

## Introduction

Deep Convolutional Neural Networks have proven to be very powerful models of sensory data such as images, video, and audio. Even though there is no solid theory available (yet) on the design of neural networks, large amount of empirical evidence indicates that (among others) **convolutional weight sharing** and **depth** are important for good predictive performance.

Convolutional weight sharing is very effective, because of the **translation symmetry** in most perception tasks: The label function and data distribution are both approximately **invariant to shifts**. This means that when using the same weights to analyze or model each part of an image, a convolution layer uses far fewer parameters than its fully connected counterpart, while still preserving the capacity to learn many useful transformations.

A Convolutional Neural network is very effective in deep neural nets, because all the layers are translation equivariant. This means that first shifting an image and then feeding it through the networks layer gives the same result as first feeding an image through the layers and then shifting it. Therefore symmetry is preserved by each layer, making it possible to exploit in not just the first, but in all higher layers of the network.

In his paper, Cohen (2016) makes smart use of these two properties by introducing **Group Equivariant Convolutional Networks (G-CNNs)** [1]. G-CNNs use **G-convolutions**, a new type of layer that enjoys a substantially higher degree of weight sharing than regular convolution layers. G-convolutions increase the expressive capacity of the network without increasing the number of parameters. Explicitly, they exploit rotational and mirror reflection symmetries. Group convolution layers are easy to use and can be implemented with negligible computational overhead for discrete groups generated by translations, reflections and rotations.

Tha aim of this notebook is to reproduce the ResNet44 results found in table 2 of the paper on Group Equivariant Convolutional Networks by Cohen. In the experiment, Z2 and p4m CNN's are compared on CIFAR10 and CIFAR10+.

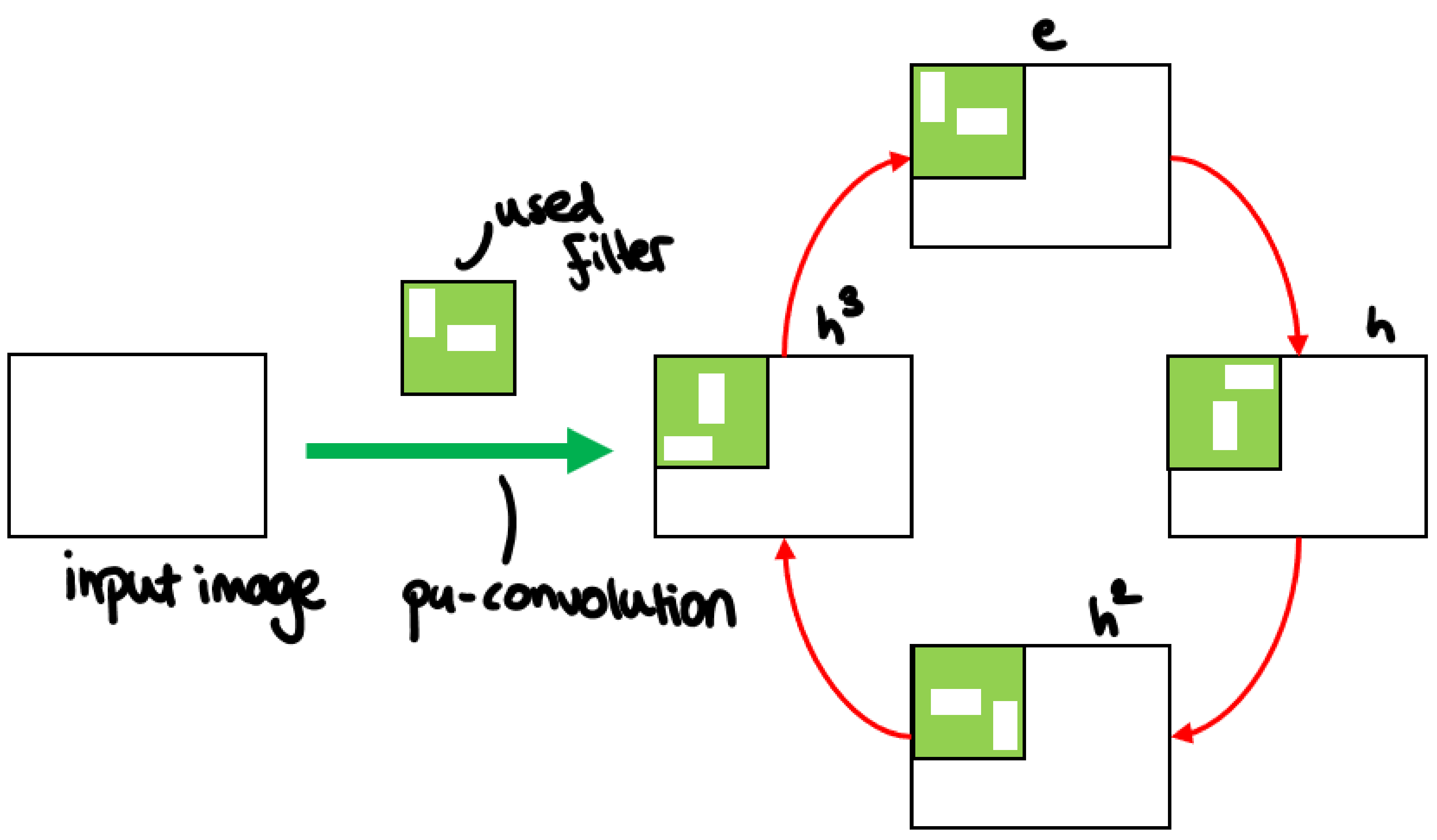
1. Cohen, T., & Welling, M. (2016, June). Group equivariant convolutional networks. In International conference on machine learning (pp. 2990-2999).

## 1. Theory and concept

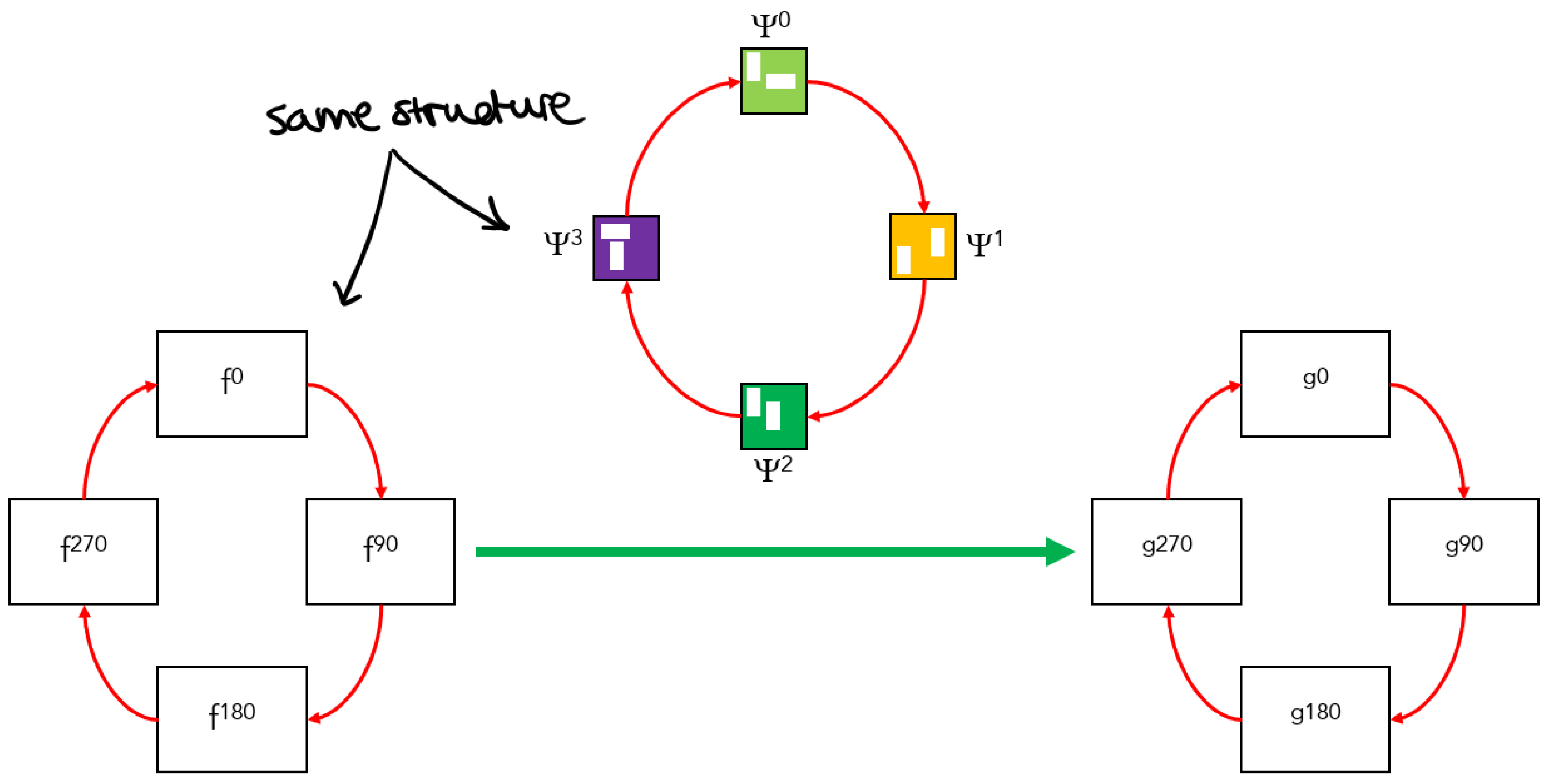
It should be known that convolutions are translation invariant: translated inputs result in translated outputs. The idea of G-CNN's is to exploit other symmetries such as rotations and reflections as well. This enables the network to see objects in different poses while still classify it as the same. Taco Cohen and Max Welling provide clear mathematical proves showing why their G-CNN's are rotational and/or reflection equivariant. Here, the prove is not paraphrased but merely the implementation clarified in detail.

### Principle of group-equivariant convolutions

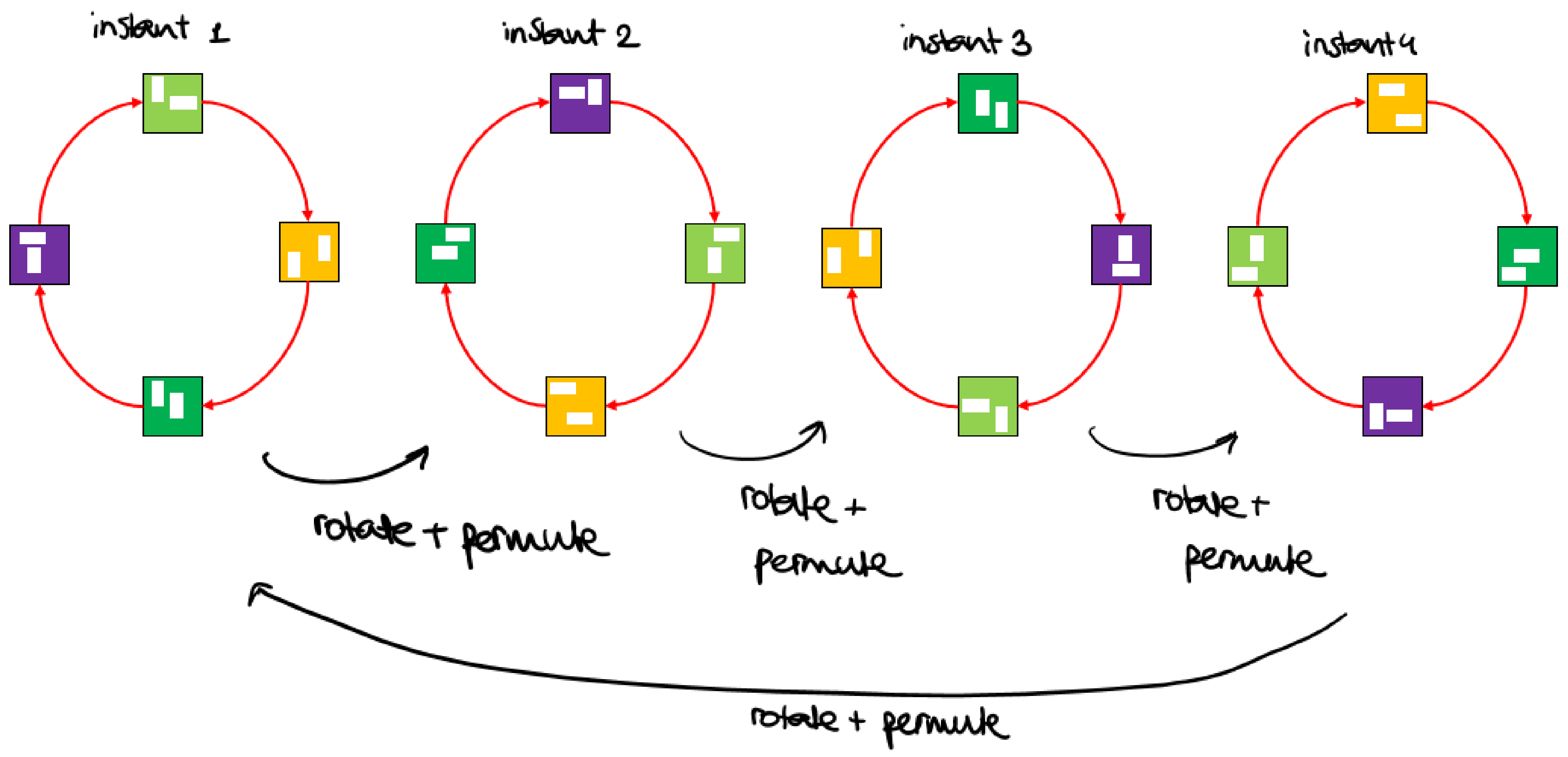
A group-equivariant convolution's output is different compared to a normal convolutions while the operations within are equivalent, namely convolutions. Let me explain what is different using the p4 group as an example. This group exists of all 4 rectangular rotations (0, 90, 180 and 270 degrees). The group-equivariant convolution produces 4 patches (for all 4 possible transformations) that are the result of the input convoluted with the **same** filter **rotated** in all the groups transformations}. In other words, the first patch is reserved for the convolution of the input with the unrotated filter ($e$), the second patch for the 90deg-rotated filter, the third for the 180deg-rotated filter ($h^2$) and the fourth for the 270deg-rotated filter ($h^3$). The outputs of these different convolutions are **place-fixed** (they belong to a particular patch). Please have a look at the figure below.



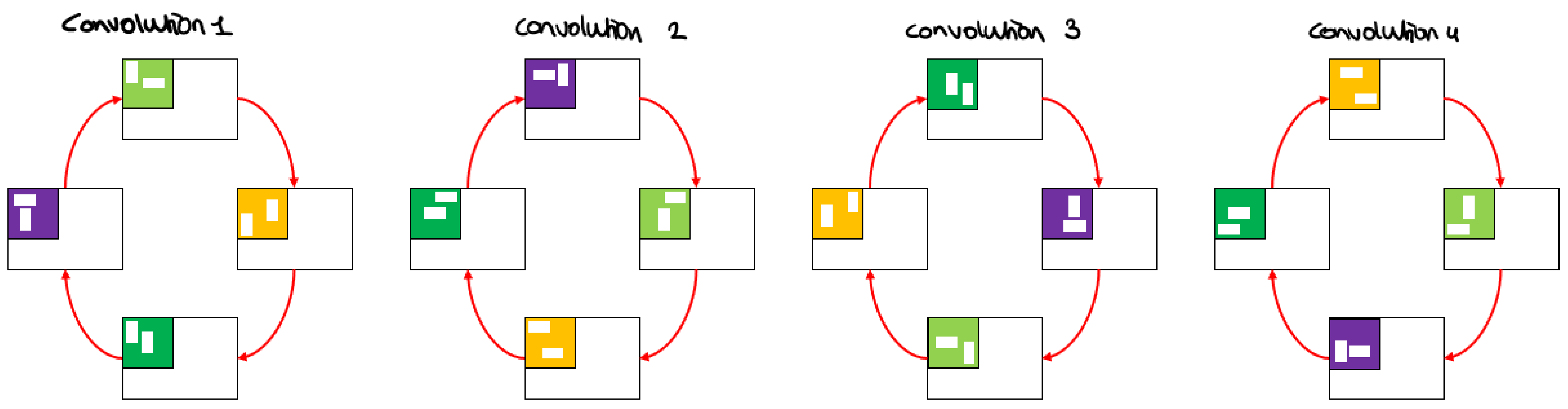
If the input is rotated in rectangular fashion two things are observed: The group-equivariant convolution (1) produces the same output only rotated and (2) differently placed along the patches. The p4-group is thus **equivariant** for its inherent transformations (not invariant since the placement along the patches is different). That is all nice and dandy but a major problem arises if the output, which is what I call 'patched', of this layer is the input for the next layer. Fun fact 1, this is the case for all ResNet layers except the first one. Fun fact 2, it is not trivial what should be done. Let me explain why this is not trivial. As mentioned earlier, the output of a group-equivariant convolution should always be organized along 4 patches. Naively, there is two 'simple' options: (1) perform the same convolution for all patches or (2) rotate the filter equivalently to the transformation belonging to the specific patch (e.g. rotate the filter by 90 degrees for patch $h$). Both are not equivariant for rotations and the thus its goal is not achieved. The following describes how it should be done for G-CNN's. Let us first create the filter. The filter's structure or representation is equivalent to that of the input: it consists the same amount and same ordering of patches (see the next figure). The individual filters (differently colored) at every location are independent: they do not share parameters.



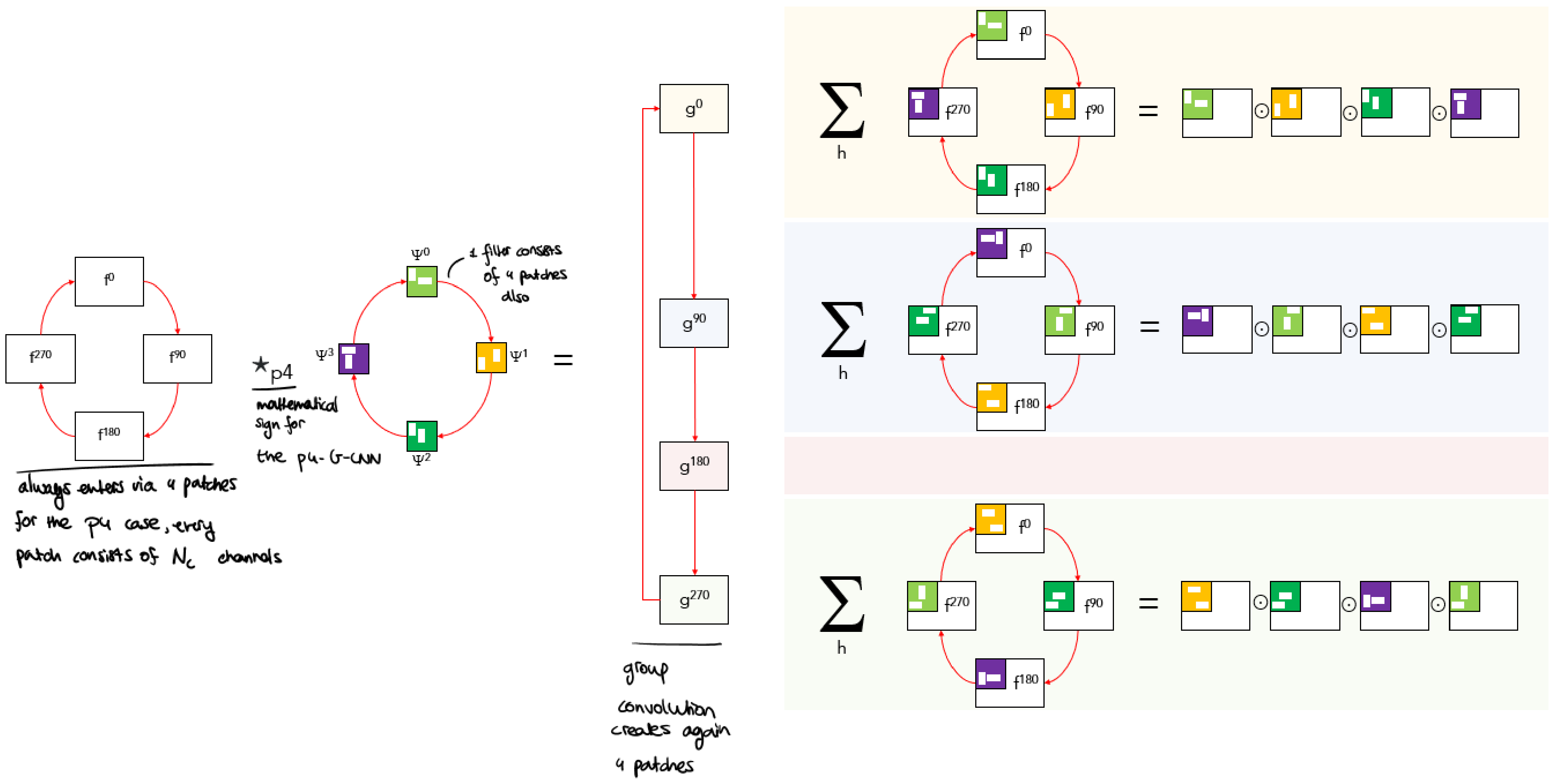
The filters we will be explicitly working with later on are those that both rotate and permute the individual filters at the same time. We will refer to these newly created filters as instances of the original filter (note that the first instant is the filter itself). The possible instances are depicted below:



Three steps remain. The first steps are 4 'normal' convolutions for the input for every instant individually:



The second step is a point-wise summation over all patches of every of these outputs. We end up with 4 outputs that are single-patched and do not have a particular place. Let us now assign every output a place in a 4-patched overall output. The place is determined as follows. For patch $e$ we take output that we created using the first instant of the filter, for patch $h$ we take the output that was created using the second instant of the filter, etc. Let me clarify using a visualization:


f0
f90
f180
f270
always enters via 4 patches for the p4 case, every patch consists of $N_c$ channels
$\star_{p4}$
mathematical sign for the p4-G-CNN
Ψ0
Ψ1
Ψ2
Ψ3
1 filter consists of 4 patches also
=
g0
g90
g180
g270
group convolution creates again 4 patches
Σ h
f0
f90
f180
f270
=
⊙

In addition we provide an explanation through the tranformation of arrays: in normal convolutions the dimension of the data is 4-fold: batch size $(BS)$, number of channels (or features, $N_c$), height $(H)$ and width $(W)$. In group convolutions, we have to add an extra dimension to accompany the possible transformations $(P)$.

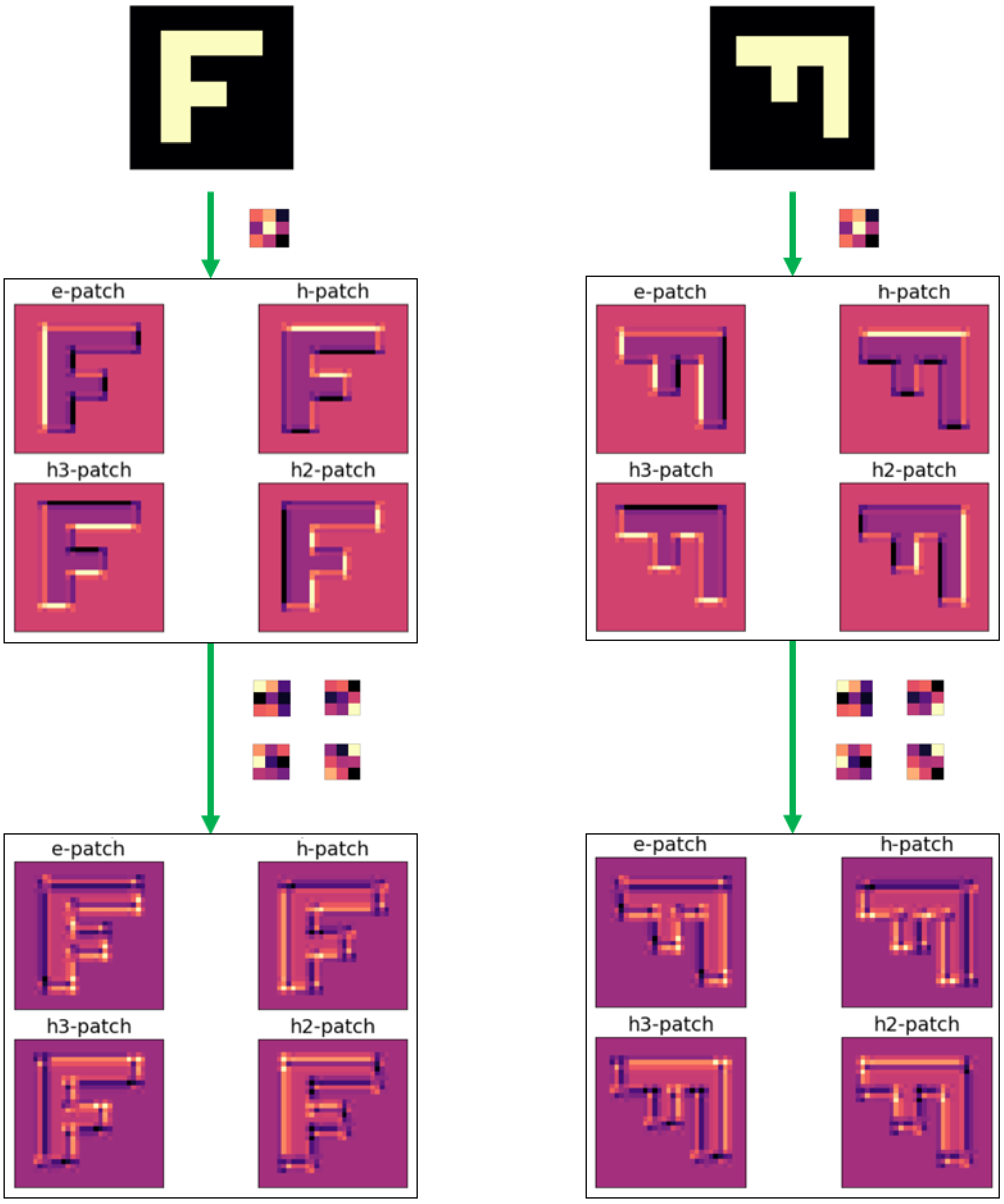
From $\mathbb{Z}^2$ to P4(M):

$$\underbrace{[BS \times N_{c,in} \times H_{in} \times W_{in}]}_{\text{un-patched input}} \Longrightarrow \underbrace{[BS \times N_{c,out} \times P \times H_{out} \times W_{out}]}_{\text{patched output}}$$

From P4(M) to P4(M):

$$\underbrace{[BS \times N_{c,in} \times P \times H_{in} \times W_{in}]}_{\text{patched input}} \Longrightarrow \underbrace{[BS \times N_{c,out} \times P \times H_{out} \times W_{out}]}_{\text{patched output}}$$

The following shows a prove that the P4-convolution is equivariant for 90 degree rotations:



## 2. Organization in Google Colab

The reproduction has been conducted on the Google Colab Platform, therefore the drive needs to be mounted using the following code. Feel free to adjust this code according to the platform you are using.

```
import sys
from google.colab import drive
import importlib.util

# Mounting Google Drive
```

```
drive.mount('/content/gdrive')
```

## 3. Programming tools used throughout

The main library used in this implementation is Pytorch. Furthermore some additional supporting libraries are imported.

```
import torch
import torchvision
import torchvision.transforms as transforms

import torch.nn as nn
import torch.nn.functional as F

import torch.optim as optim

from torchsummary import summary

import matplotlib.pyplot as plt
import numpy as np

import time

import math

from functools import partial
from dataclasses import dataclass
from collections import OrderedDict

#use GPU is available
use_gpu = torch.cuda.is_available()
```

### Groupy

Finally, the GrouPy library is imported. GrouPy is a python library that implements group equivariant convolutional neural networks, and has been adjusted by Adam Bielski to be compatible with Pytorch [2]. Again, feel free to adjust the directories according to your needs.

[2] https://github.com/adambielski/GrouPy

```
!pip install nose
!pip install chainer

%cd /content/gdrive/My Drive/Deep Learning/Reproduction project
! git clone https://github.com/adambielski/GrouPy.git

%cd /content/gdrive/My Drive/Deep Learning/Reproduction project/GrouPy
! python setup.py install

!nosetests -v

from groupy.gconv.pytorch_gconv.splitgconv2d import P4ConvZ2, P4ConvP4, P4MConvZ2, P4MConvP4M
```

## 4. Architecture of the networks

### ResNet44

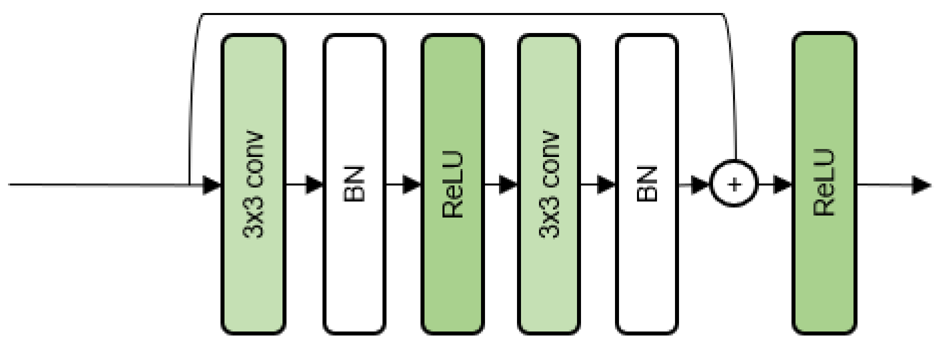
The paper evaluates 2 architecture modalities: residual and all-CNN networks. We will focus on the residual networks and especially on what is referred to as ResNet44, which consists of 43 convolutional layers and 1 fully connected layer in total.

ResNet44's encoder consists of an initial convolutional stage and 3 residual stages. The decoder consists of 1 fully connected layer. What these stages look like and how they are implemented will become clear throughout this section.

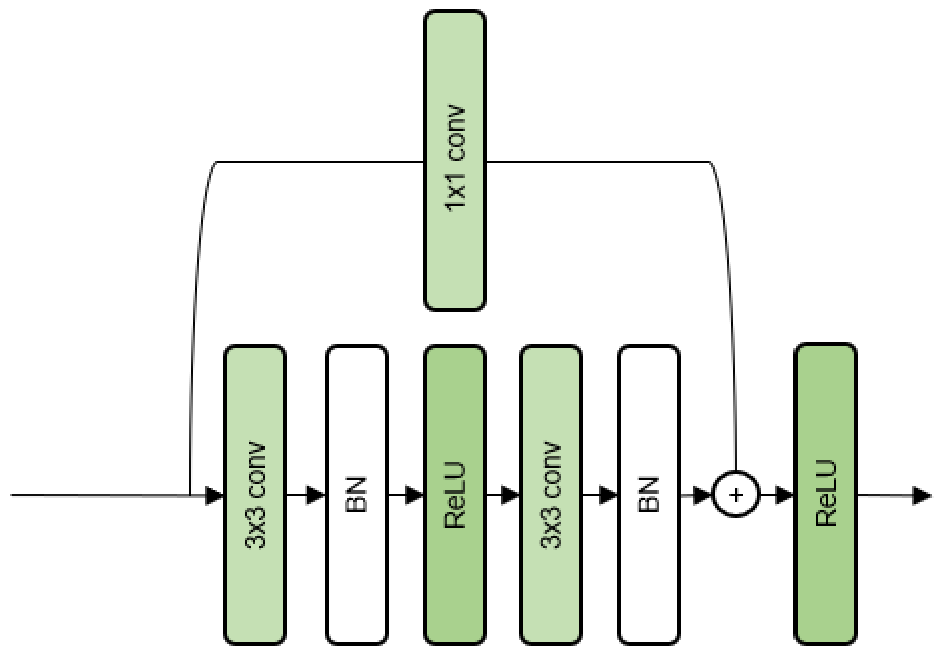
One should know beforehand that this high-level architecture holds for all three convolution types. The only difference in the $\mathbb{Z}^2$, P4 and P4M implementations are the specific convolutions used and the amount of filters per convolution layer. We will talk you through the exact implementation now. Be aware that the specific parts of the network are build-up from scratch throughout the story.

### The residual block

Residual networks consist of (mostly) many residual blocks. Every residual block consists of two distinct paths, or branches: the shortcut connection and the convolutional branch. These branches split at the start of the residual block and merge together towards the end. The shortcut connection enables the input to directly travel to the output. The other branch consists of the usual convolutional layer(s). The output of both paths are added and form the input for the next residual block. The first class, *ResidualBlock*, creates the most simple residual block in which two identity initialized paths are added in the definition of the foward operator.



The second class, *ResNetResidualBlock*, specifies the shortcut connection. If the input equals the output size it uses the identity mapping. (Note that two things are able to change the size: either the size of the image or the number of channels (features) changes.) If the input and output size differ, 1x1 convolution and batch normalization are subsequently used instead of the identity mapping. The 1x1 convolution (, or kernel size is 1,) uses the stride and expansion needed. *ResNetResidualBlock* is a subclass of *ResidualBlock* since it generalizes the block.



*ResNetBasicBlock* creates the specific block used in our implementation. It specifies the architecture of the convolutional branch: convolution, batch normalization, activation, convolution and batch normalization are subsequently used in our model. Consequently we use 2 successive convolution in every basic block. Since the basic block is a generalization of *ResNetResidualBlock* it is a subclass of that class.

Two things should be mentioned: the implementation shows the one for P4. One could simply replace these convolutions by either *nn.Conv2D* or *P4MConvP4M* to create the other networks. In addition for creating the normal network, *nn.BatchNorm2d* should be used instead of *BatchNorm2d*. These also hold for the next explanatory sections.

```
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.in_channels, self.out_channels =  in_channels, out_channels
        self.blocks   = nn.Identity() #initialize block as identity
        self.shortcut = nn.Identity() #initialize skip as identity

    def forward(self, x):
        #shortcut
        residual = x
        if self.should_apply_shortcut: residual = self.shortcut(x)
        x = self.blocks(x)

        #combination
        x += residual
        x = F.relu(x)
        return x

    @property
    def should_apply_shortcut(self):
        return self.in_channels != self.out_channels
```

```
class ResNetResidualBlock(ResidualBlock):
    def __init__(self, in_channels, out_channels, expansion=1, downsampling=1,
                 conv=P4ConvP4, *args, **kwargs):
        super().__init__(in_channels, out_channels)
        self.expansion, self.downsampling, self.conv = expansion, downsampling, conv

        self.shortcut = nn.Sequential(OrderedDict({
            'conv' : P4ConvP4(self.in_channels,
                                self.expanded_channels, kernel_size=1,
                                stride=self.downsampling, bias=False),
            'bn'   : nn.BatchNorm3d(self.expanded_channels)

        })) if self.should_apply_shortcut else None


    @property
    def expanded_channels(self):
        return self.out_channels * self.expansion

    @property
    def should_apply_shortcut(self):
        return self.in_channels != self.expanded_channels
```

```
def bn_conv(in_channels, out_channels, conv, activation = nn.ReLU, *args, **kwargs):
    return nn.Sequential(OrderedDict({'conv': conv(in_channels, out_channels,
                                                    kernel_size = 3, padding = 1,
                                                    *args, **kwargs),
                                      'bn': nn.BatchNorm3d(out_channels)}))
```

```
class ResNetBasicBlock(ResNetResidualBlock):
    expansion = 1
    def __init__(self, in_channels, out_channels, *args, **kwargs):
        super().__init__(in_channels, out_channels, *args, **kwargs)
        self.blocks = nn.Sequential(
            bn_conv(self.in_channels, self.out_channels, conv=self.conv,
                    bias=False, stride=self.downsampling),
            nn.ReLU(),
            bn_conv(self.out_channels, self.expanded_channels,
                    conv=self.conv, bias=False),
        )
```

## Residual layers

Most residual networks consist of several residual layers which consist of consecutive residual blocks. Except for the first residual block, the size of the data does not change throughout. In other words, only the first residual block uses the 1x1 convolution in the shortcut connection and stride and expansion in the first convolution in the convolutional branch. The amount of residual blocks used per layer is called the depth of that layer which in our case is 7 for every layer. It can be concluded that the amount of convolutions used per layer is 7 x 2 = 14. *ResNetLayer* is the class that encodes these layers.

```
class ResNetLayer(nn.Module):
    def __init__(self, in_channels, out_channels, block=ResNetBasicBlock, n=1, *args, **kwargs):
        super().__init__()
        # 'We perform downsampling directly by convolutional layers that have a stride of 2.'
        downsampling = 2 if in_channels != out_channels else 1

        #first introduce block(... )
        self.blocks = nn.Sequential(
            block(in_channels , out_channels, *args, **kwargs, downsampling=downsampling),
            # in the next the n-1 blocks are subsequently stacked (_ underscore means
            # that the do_something will be executed the prescribed amount of times
            *[block(out_channels * block.expansion,
                    out_channels, downsampling=1, *args, **kwargs) for _ in range(n - 1)]
        )

    def forward(self, x):
        x = self.blocks(x)
        return x
```

## The encoder

The encoder consists of an initial convolutional layer followed up by several residual layers. The initial convolutional layer consists of a convolution, batch normalization, activation and subsequently pooling. The initial convolutional layer is followed by three residual layers. In our network the size of the data transforms as follows:

Dimensions can be:

1. 4D ~ [batch size, features, height, width]
2. 5D ~ [batch size, features, transformations, height, width]

The specific sizes transform as follows through the network: Input:

1. $\mathbb{Z}^2$, P4 and P4M: [~, 3, 32, 32]

After first initial convolutional layer we observe that the sizes of the entrees and the dimensions change for different use of convolutions. The group convolutions create an extra dimension (for all its inherent transformation as explained earlier) while the normal convolutions remain the same dimension, logically. The feature size is different for every type of convolutions to make sure that approximately the same amount of parameters is used:

1. $\mathbb{Z}^2$: [~, 32, 32, 32]
2. P4: [~, 16, 4, 32, 32]
3. P4M: [~, 11, 8, 32, 32]

The first residual layer does not change the size or the dimension!

Within the 2nd residual layer (remind yourself that the size does only change in the first residual block) the image sizes are reduced by 2 in both spatial directions and the features sizes increased:

1. $\mathbb{Z}^2$: [~, 64, 16, 16]
2. P4: [~, 32, 4, 16, 16]
3. P4M: [~, 23, 8, 16, 16]

The same happens in in the 3rd residual layer:

1. $\mathbb{Z}^2$: [~, 128, 8, 8]
2. P4: [~, 64, 4, 8, 8]
3. P4M: [~, 45, 8, 8, 8]

*ResNetEncoder* does the job. The change in feature size is encoded in *blocks_sizes* . The outcome of the encoder serves as input for the decoder which will be explained next.

```
class ResNetEncoder(nn.Module):
    def __init__(self, in_channels=3, blocks_sizes=[32, 64, 128], depths=[7,7,7],
                 activation=nn.ReLU, block=ResNetBasicBlock, *args,**kwargs):
        super().__init__()

        self.blocks_sizes = blocks_sizes

        self.initconv = nn.Sequential(OrderedDict({
            'conv'    : P4ConvZ2(in_channels, self.blocks_sizes[0],
                        kernel_size=7, stride = 1, padding=3, bias=False),
            'bn'      : nn.BatchNorm3d(self.blocks_sizes[0]),
            'act'     : activation()})
        )

        self.in_out_block_sizes = list(zip(blocks_sizes, blocks_sizes[1:]))
        self.blocks = nn.ModuleList([
            ResNetLayer(blocks_sizes[0], blocks_sizes[0], n=depths[0], activation=activation,
                        block=block,  *args, **kwargs),
            *[ResNetLayer(in_channels * block.expansion,
                          out_channels, n=n, activation=activation,
                          block=block, *args, **kwargs)
              for (in_channels, out_channels), n in zip(self.in_out_block_sizes, depths[1:])]
        ])


    def forward(self, x):
        x = self.initconv(x)
        for block in self.blocks:
            x = block(x)
        return x
```

## The decoder

The decoder is a fully connected layer that connects the flattened output of the encoder to the amount of classes one wants to distinguish. Before the fully connected layer, average pooling is used. The class *ResNetDecoder* is defined below.

```
class ResNetDecoder(nn.Module):
    def __init__(self, in_features, n_classes, *args, **kwargs):
        super().__init__()
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.decoder = nn.Linear(in_features, n_classes)

    def forward(self, x):
        x = self.avgpool(x)
        x = x.view(x.size(0), -1) #flatten
        x = self.decoder(x)
        return x
```

## Connecting the dots

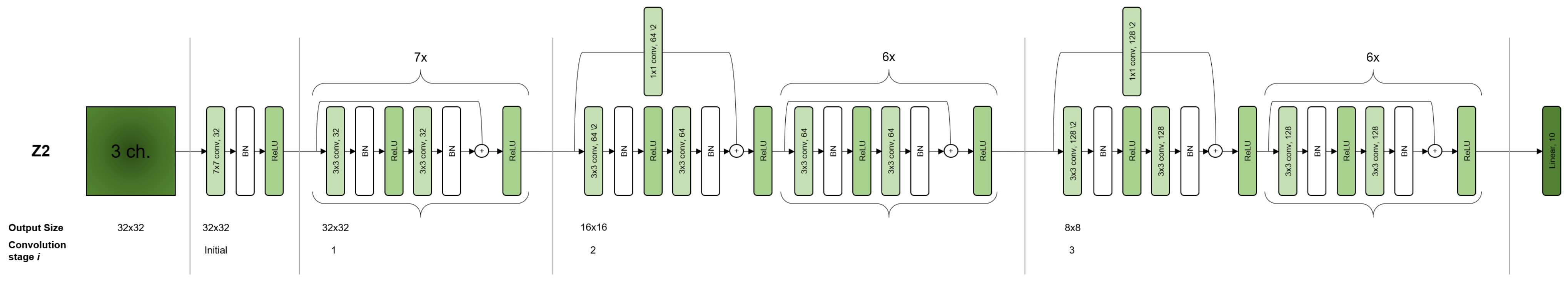
*ResNet* connects the encoder and decoder and finalizes the network.

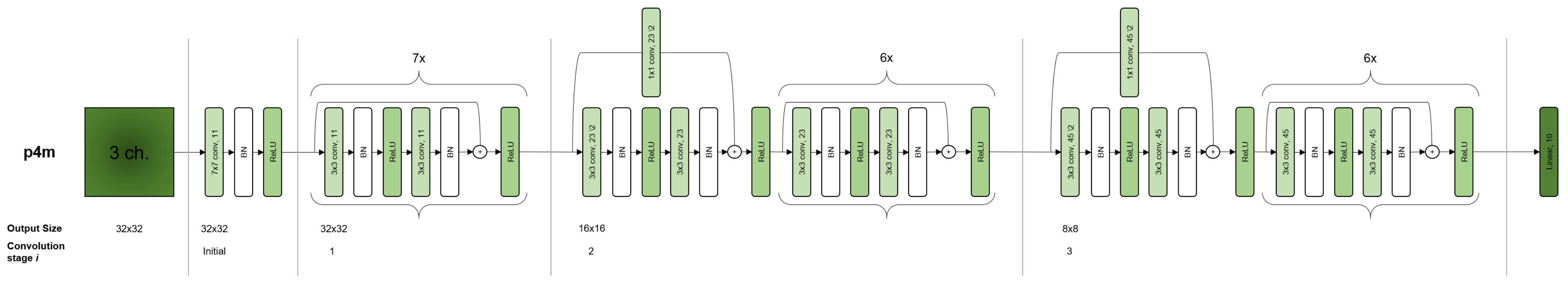
Note that *x.view* is only needed if group convolutions are used.

```
class ResNet(nn.Module):
    def __init__(self, in_channels, n_classes, *args, **kwargs):
        super().__init__()

        self.encoder = ResNetEncoder(in_channels, *args, **kwargs)
        self.decoder = ResNetDecoder(self.encoder.blocks[-1].blocks[-1].expanded_channels*4, n_classes)

    def forward(self, x):
        x = self.encoder(x)
        xs = x.size()
        x = x.view(xs[0], xs[1]*xs[2], xs[3], xs[4])
        x = self.decoder(x)
        return x
```

### Visualizing the architectures

Detailed architectures for $\mathbb{Z}^2$ (first) and P4M (second)





## 5. Validation and training

Training is done using the function *train_network*. Stochastic gradient descent and cross entropy are used as optimizer and loss function respectively. The learning rate decreases every 50 epochs by 10 uptill 150 from which it stays unchanged until 300 epochs. The momentum value is 0.9. Furthermore, the function saves the model and curves (learning and loss) every 50 epochs and prints time, train and test accuracy and loss value every epoch.

```
def train_network(net, train_loader, test_loader, device, title,
                  lr = 0.05, momentum = 0.9, gamma = 0.1
```

```
              lr = 0.05, momentum = 0.9, gamma = 0.1,
              n_epochs = 300, epoch_start = 0,
              criterion = nn.CrossEntropyLoss().cuda()):
"""
Training and evaluation of a connected network which should be written
in Pytorch fashion.
"""

print('training on', device)

optimizer = optim.SGD(net.parameters(), lr=lr, momentum=momentum)

#Create vectors with information on loss, training accuracy and test accuracy
loss_curve   = np.zeros((n_epochs))
train_curve  = np.zeros((n_epochs))
test_curve   = np.zeros((n_epochs))

#vector with epochs
epoch_vec = np.arange(epoch_start,n_epochs)

# ---------------------------- START TRAINING ------------------------------
for epoch in epoch_vec:

    net.train()
    n, start = 0, time.time()

    train_l_sum   = torch.tensor([0.0], dtype=torch.float32, device=device)
    train_acc_sum = torch.tensor([0.0], dtype=torch.float32, device=device)

    for i, data in enumerate(train_loader, 0):

        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        with torch.no_grad():
            outputs = outputs.long()
            train_l_sum += loss.float()
            n += outputs.shape[0]

    #update learning rate
    if epoch <= 300:
        if epoch % 50 == 49:
            lr *= gamma
            optimizer = optim.SGD(model.parameters(), lr=lr, momentum=momentum)
            print('We got ourselves a change in learning rate:')
            print('Epoch: '+ str(epoch))
            print('Learning rate changed to: ' + str(lr))

    # ------------------- START VALIDATION and SAVING ----------------------
    loss_value = train_l_sum/n

    stop1 = time.time()
    train_acc  = evaluate_accuracy(train_loader, net, device)
    stop2 = time.time()
    test_acc   = evaluate_accuracy(test_loader, net, device)

    loss_curve[epoch]  = loss_value
    train_curve[epoch] = train_acc
    test_curve[epoch]  = test_acc

    curves = np.array([loss_curve, train_curve, test_curve])

    #print updated scores
    print('epoch %d [#]  |  loss %.4f [-]  |  train acc %.3f [-]  |  test acc %.3f [-]  |  time %.1f-%.1f-%.1f [s]'\
    % (epoch + 1, loss_value, train_acc, test_acc, stop1-start, stop2-stop1, time.time()-stop2))

    #save every 5 epochs
    if epoch % 10 == 9:    # save every 10 epochs
        #model
        name_to_save = title + '__test=' + str(test_acc) + '%_train=' + str(train_acc) + '%_epoch=' + str(epoch+1)
        directory = '/content/gdrive/My Drive/Deep Learning/Reproduction project/Saved Networks/Preliminary/'
        path = '/content/gdrive/My Drive/Deep Learning/Reproduction project/Saved Networks/Preliminary/' + name_to_save + '.pth'
        torch.save(model.state_dict(), path)
```

```
            #learning curves
            path2 = directory + name_to_save + '_start:' + str(epoch_start)
            np.save(path2, curves)

    print('Finished Training')
    return loss_curve, train_curve, test_curve



# ---------------------------------- VALIDATION --------------------------------
def evaluate_accuracy(data_iter, net, device=torch.device('cpu')):
    """Evaluate accuracy of a model on the given data set."""
    net.eval()  # Switch to evaluation mode for Dropout, BatchNorm etc layers.
    acc_sum, n = torch.tensor([0], dtype=torch.float32, device=device), 0
    for X, y in data_iter:
        # Copy the data to device.
        X, y = X.to(device), y.to(device)
        with torch.no_grad():
            y = y.long()
            acc_sum += torch.sum((torch.argmax(net(X), dim=1) == y))
            n += y.shape[0]
    return acc_sum.item()/n
```

## 6. CIFAR10 and CIFAR10+ evaluation

To compare dataset augmentation performance, CIFAR10 will be compared to augmented CIFAR10+. Regular CIFAR10 only needs to be normalized using the following code:

```
transform_train = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
```

CIFAR10+ consists of horizontal flips and small translations, which are apllied to the training data using the following code:

```
#Normalize a tensor image with mean and standard deviation.
# Given mean: (M1,...,Mn) and std: (S1,..,Sn) for n channels, this transform will normalize each channel of the input torch.*Tensor i
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])
```

```
# Load training sets
batch_size  = 128
num_workers = 2
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform_train)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=num_workers)

# Load test sets
testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform_test)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=num_workers)
```

## 7. Discussion and conclusion

The aim of this blog was to reproduce the results of the original paper on Group Equivariant Convolutional Networks. In order to achieve this, we recreated the network in Pytorch for Z2, p4 and p4m convolutions with an equal ResNet44 architecture and parameter count. After training the network, we can conclude that based on our network and specific way of loading, group convolutions show significant decrease in test error for both the CIFAR and CIFAR10+ dataset (upto 1.5 - 2% depending on the batch size used). The results of our findings are shown in Table 7.1. Additionally, figures 7.1 & 7.2 show the learning curves for all three evaluated networks for batch sized 64.

Qualitatively, the result of this reproduction is in line with the results of the paper, proving its thruth and thus worth. The results clearly show an improvement in accuracy when using p4m convolutions over Z2 convolutions, as the paper claims.

Quantitatively we were not able to achieve such low errors for all the networks as the original paper proposed. As the paper did not specify batch sizes, we tested many different ones, and it seems as if the way we train the networks tends to overfit the data. It can be seen that after 50 epochs for CIFAR10 and 100 epochs for CIFAR10+ the training data already approaches 100% accuracy. Furthermore, the use of group convolutions significantly slows down training in our implementation, contrary to the suggestion of the paper that "Group convolution layers can be implemented with negligible computational overhead". It is approximately 3 (P4) and 7 (P4M) times slower in both updating its parameters as conducting its accuracy compared to the normal residual network.

Table 7.1: Reproduction results

| | CIFAR10 [%] | | CIFAR10+ [%] | | Params [#] | |
|---|---|---|---|---|---|---|
| | paper | **ours** | paper | **ours** | paper | **ours** |
| $\mathbb{Z}^2$ | 9.45 | **13.39** | 5.62 | **8.34** | 2.64 | **2.64** |
| p4m | 6.45 | **9.1** | 4.94 | **6.97** | 2.62 | **2.62** |

Figure 7.1

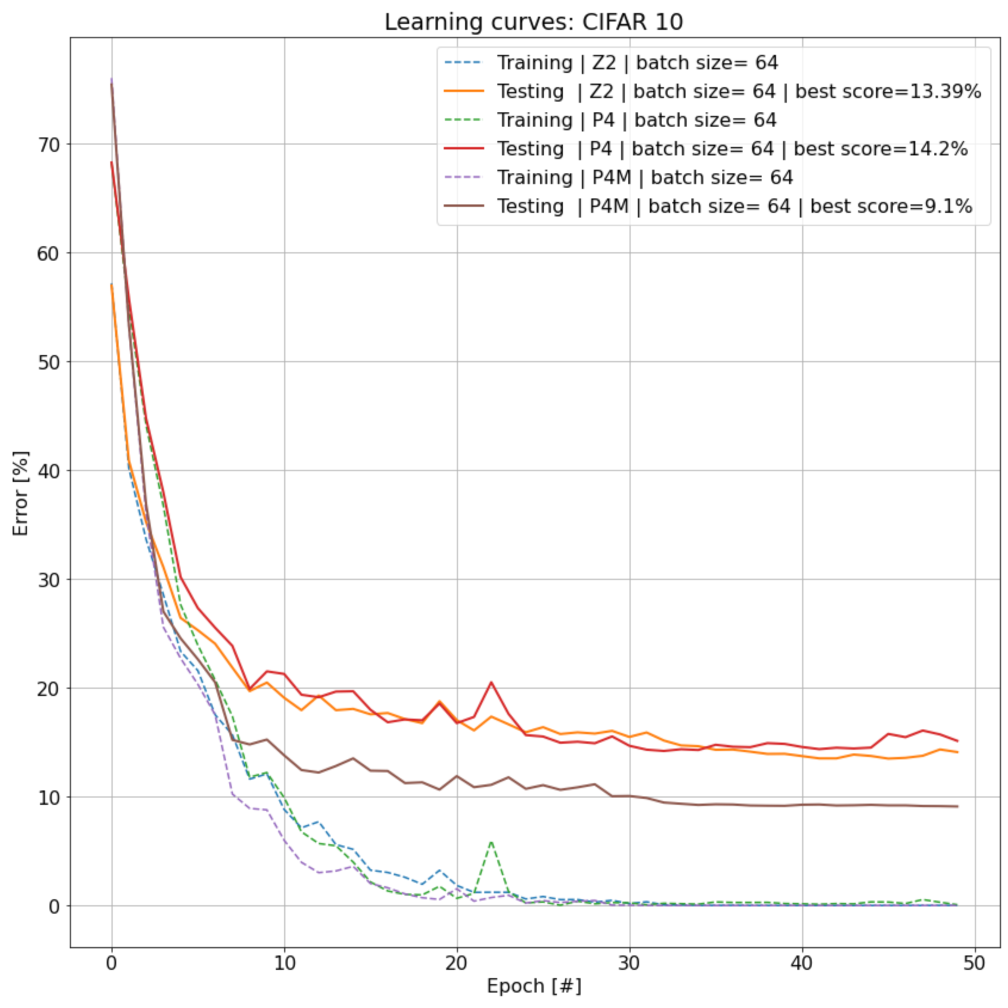
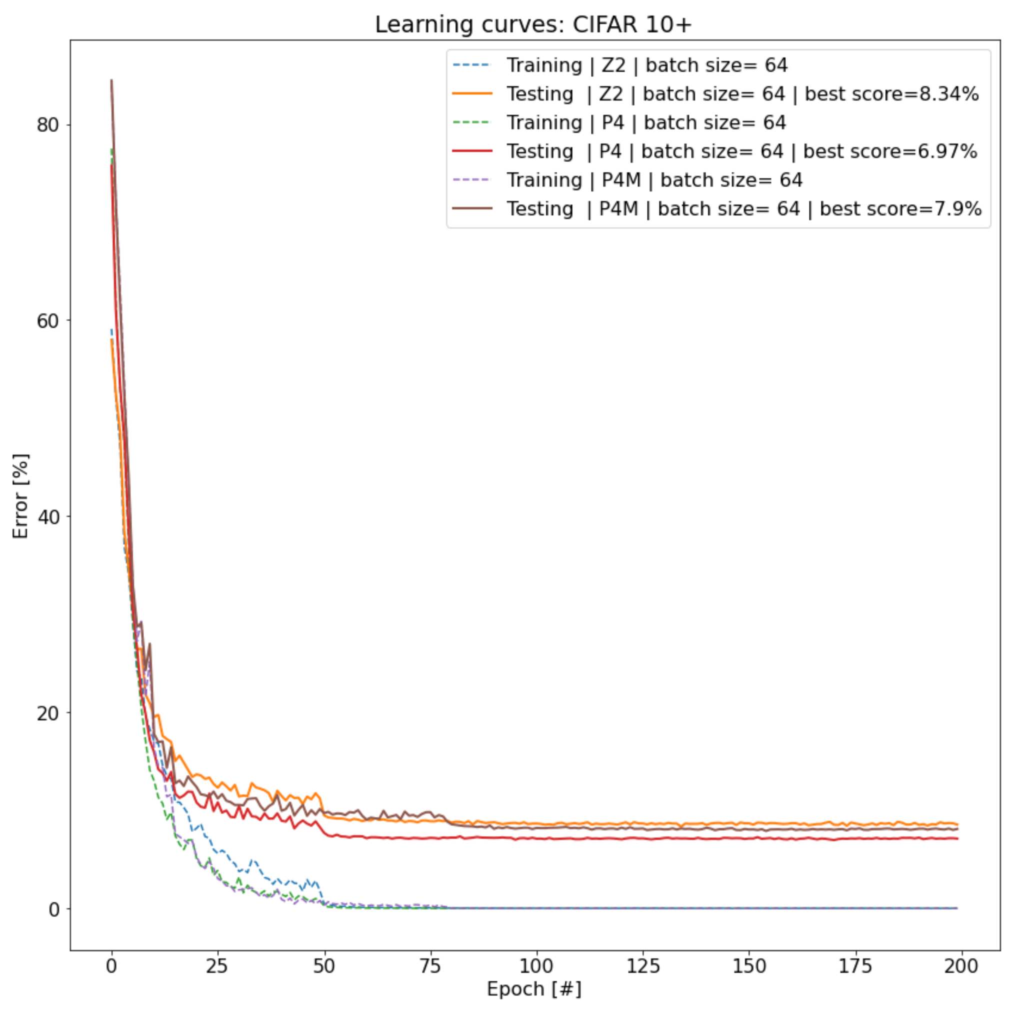


Figure 7.2


Learning curves: CIFAR 10+
Training | Z2 | batch size= 64
Testing | Z2 | batch size= 64 | best score=8.34%
Training | P4 | batch size= 64
Testing | P4 | batch size= 64 | best score=6.97%
Training | P4M | batch size= 64
Testing | P4M | batch size= 64 | best score=7.9%
Error [%]
Epoch [#]
0
20
40
60
80
0
25
50
75
100
125
150
175
200