Detect targets in radar signals

Tudor Andrei Dumitrascu

December 22, 2021

1 Introduction

2 Approaches

3 CNN

4 ADNet

5 ViT

6 ConvMixer and ConvRepeater

7 Cross Validation

Model	Validation MAE
CNN	1.93
Vit	1.76
ADNet	1.93
ConvMixer	1.77
ConxRepeater	1.66

{'model': <class 'lightning_models.CNN_lightning'>, 'hyps': {'size': [3, 16, 32, 64], 'num_blocks': 1, 'kernel_size': 3, 'strides': 2, 'padding': 1, 'op': 'regression'}} : 1.9370444059371947 {'model': <class 'lightning_models.ConvMixerLightning'>, 'hyps': {'size': 32, 'num_blocks': 4, 'kernel_size': 11, 'patch_size': 5, 'num_classes': 5, 'lr': 0.3, 'res_type': 'cat', 'op': 'classification'}} : 1.7744715690612793 {'model': <class 'lightning_models.ConvMixerLightning'>, 'hyps': {'size': 32, 'num_blocks': 4, 'kernel_size': 11, 'patch_size': 5, 'num_classes': 5, 'lr': 0.3, 'res_type': 'add', 'op': 'classification'}} : 1.6684766292572022 {'model': <class 'lightning_models.ADNet_lightning'>, 'hyps': {'lr': 0.003, 'op': 'regression', 'optim': 'adam'}} : 1.9370444059371947 {'model': <class 'lightning_models.ViTLigthning'>, 'hyps': {'num_classes': 5, 'image_size': 128, 'patch_size': 32, 'lr': 0.03, 'dim': 256, 'depth': 20, 'heads': 9, 'mlp_dim': 256, 'dropout': 0.2, 'emb_dropout': 0.2}} : 1.7659256219863892

Bibliography

Appendix

```
import os
import warnings
```

```python
import pretty_errors
import torch
from pytorch_lightning import Trainer, seed_everything
from pytorch_lightning.callbacks.early_stopping import EarlyStopping
from pytorch_lightning.callbacks.lr_monitor import LearningRateMonitor
from pytorch_lightning.callbacks.model_checkpoint import ModelCheckpoint
from pytorch_lightning.loggers import TensorBoardLogger
from sklearn.metrics import classification_report
from torchvision import transforms
from torchvision.transforms import Pad, ToTensor

from config import config
from data_utils import data_generator
from lightning_models import get_regression_output

warnings.filterwarnings("ignore")
seed_everything(42)

REGRESSION = "regression"
CLASSIFICATION = "classification"
BATCH_SIZE = 64
NW = 8
EPOCHS = 100
classif_report = True


if __name__ == "__main__":

    model = config["adnet"]
    trans_ = transforms.Compose(
        [
            ToTensor(),
            Pad(
                [5, 0, 4, 0],
            ),
        ]
    )
    train_, val_ = data_generator(
        img_dir="./train",
        csv_path=r"train.csv",
        bs=BATCH_SIZE,
        nw=NW,
        transform=trans_,
    )

    logger = TensorBoardLogger("tb_logs", name=model["name"])
    hyp = model["hyp"]
    module = model["model"](**hyp)
    trainer = Trainer(
        # fast_dev_run=True,
        enable_model_summary=False,
        # benchmark=True,
        logger=logger,
        detect_anomaly=True,
        gpus=1,
        # precision=16,
        max_epochs=EPOCHS,
        callbacks=[
            ModelCheckpoint(
                monitor="val/val_loss",
                mode="min",
                dirpath=f"models/{model['name']}",
                filename="radar-epoch{epoch:02d}-val_loss{val/val_loss:.2f}",
                auto_insert_metric_name=False,
            ),
            # EarlyStopping(monitor="val/val_loss", patience=5),
            LearningRateMonitor(logging_interval="step"),
        ],
    )
    trainer.fit(module, train_, val_)
    os.system("notify-send 'Done Training'")
    if classif_report:
        module.eval()
        module.freeze()
        preds = []
        gt = []
        for (
            img,
            label,
        ) in val_:

            if module.op == "classification":
                output = module(img)
                predict = output.argmax(1)
            else:
                label = label.float()
                output = module(img)
                output = torch.squeeze(output, dim=1)
                predict = get_regression_output(output)
            predict = predict.int()
            label = label.int()
            for p, l in zip(predict, label):
                preds.append(p)
                gt.append(l)
        print(classification_report(gt, preds))
```

```
from lightning_models import (
    ADNet_lightning,
    CNN_lightning,
    ConvMixerLightning,
    ViTLigthning,
)

REGRESSION = "regression"
config = {
    "cnn": {
        "name": "cnn",
        "model": CNN_lightning,
        "hyps": {
            "size": [3, 16, 32, 64],
            "num_blocks": 1,
            "kernel_size": 3,
            "strides": 2,
            "padding": 1,
            "op": "regression",
        },
    },
    "conv_mixer": {
            'name': 'conv_mixer',
        "model": ConvMixerLightning,
        "hyps": {
            "size": 32,
            "num_blocks": 4,
            "kernel_size": 11,
            "patch_size": 5,
            "num_classes": 5,
            "lr": 0.3,
            "res_type": "cat",
            # "op": "regression",
            "op": "classification",
        },
    },
    "conv_repeater": {
            'name': 'conv_repeater',
        "model": ConvMixerLightning,
        "hyps": {
            "size": 32,
            "num_blocks": 4,
            "kernel_size": 11,
            "patch_size": 5,
            "num_classes": 5,
            "lr": 0.3,
            "res_type": "add",
            "op": "classification",
        },
    },
    "adnet": {
        "model": ADNet_lightning,
        "hyps": {"lr": 0.003, "op": REGRESSION, "optim": "adam"},
    },
    "vit": {
        "model": ViTLigthning,
        "hyps": {
            "num_classes": 5,
            "image_size": 128,
            "patch_size": 32,
            "lr": 0.03,
            "dim": 256,
            "depth": 20,
            "heads": 9,
            "mlp_dim": 256,
            "dropout": 0.2,
            "emb_dropout": 0.2,
        },
    },
}
```

```python
import logging

import numpy as np
import pandas as pd
import pretty_errors
from pytorch_lightning import Trainer, seed_everything
from pytorch_lightning.callbacks.early_stopping import EarlyStopping
from pytorch_lightning.loggers import TensorBoardLogger
from sklearn.model_selection import KFold
from torch.utils.data.dataloader import DataLoader
from torch.utils.data.dataset import Subset
from torchvision import transforms
from torchvision.transforms import Pad, ToTensor

from data_utils import kfold_generator
from config import config

logging.getLogger("pytorch_lightning").setLevel(logging.WARNING)

seed_everything(42)

REGRESSION = "regression"
CLASSIFICATION = "classification"
BATCH_SIZE = 64
NW = 1
EPOCHS = 100
trans_ = transforms.Compose(
    [
        ToTensor(),
        Pad(
            [5, 0, 4, 0],
        ),
    ]
)

if __name__ == "__main__":
    csv_path = r"train.csv"
    img_dir = "./train"
    train_dataset = kfold_generator(img_dir, csv_path, trans_)
    for i, (name, model) in enumerate(config.items()):
        mmae = []
        kf = KFold(n_splits=5, random_state=42, shuffle=True).split(
            range(len(train_dataset))
        )
        for i, (train_index, valid_index) in enumerate(kf):
            trainer = Trainer(
                # fast_dev_run=False,
                enable_progress_bar=False,
                # logger=TensorBoardLogger("tb_logs", name=f"kfold_{name}{i}"),
                enable_model_summary=False,
                max_epochs=EPOCHS,
                # benchmark=True,
                detect_anomaly=True,
                gpus=1,
                callbacks=[
                    EarlyStopping(monitor="val/val_loss", patience=5),
                ],
            )
            train_data = Subset(train_dataset, train_index)
            valid_data = Subset(train_dataset, valid_index)
            trainer.fit(
                model=model["model"](**model["hyps"]),
                train_dataloaders=DataLoader(
                    train_data, batch_size=BATCH_SIZE, shuffle=True, num_workers=4
                ),
                val_dataloaders=DataLoader(
                    valid_data, batch_size=BATCH_SIZE, num_workers=4
                ),
            )
            val_result = trainer.validate(
                model=model["model"](**model["hyps"]),
                dataloaders=DataLoader(
                    valid_data, batch_size=BATCH_SIZE, num_workers=4
                ),
                verbose=False,
            )
            mmae.append(val_result[0]["val/mae"])
        print(f"{model} : {np.mean(mmae)}")
        # break
```

```
import torch
import torch.nn as nn
from torch.nn import (
    AdaptiveMaxPool2d,
    Conv2d,
    Flatten,
    Linear,
    MaxPool2d,
    ReLU
)
from torch.nn.modules.activation import GELU


class PrintLayer(torch.nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        print(x.shape)
        return x


class DoubleConvBlock(torch.nn.Module):
    def __init__(self, in_ch, out_ch, kernel_size, strides, padding):
        super().__init__()
        self.layers = nn.Sequential(
            Conv2d(
                in_ch,
                out_ch,
                kernel_size=kernel_size,
                stride=strides,
                padding=padding,
            ),
            GeluBatch(out_ch),
            Conv2d(
                out_ch,
                out_ch,
                kernel_size=kernel_size,
                stride=strides,
                padding=padding,
            ),
        )

    def forward(self, x):
        return self.layers(x)


class ResidualBlock(nn.Module):
    def __init__(self, layers):
        super().__init__()
        self.layers = nn.Sequential(*layers)

    def forward(self, x):
        return self.layers(x) + x


class ConcatBlock(nn.Module):
    def __init__(self, layers):
        super().__init__()
        self.layers = nn.Sequential(*layers)

    def forward(self, x):
        out = torch.cat([self.layers(x), x], dim=1)
        return out


class PatchEmbedding(nn.Module):
    def __init__(self, size, patch_size) -> None:
        super().__init__()
        self.patch = nn.Conv2d(3, size, kernel_size=patch_size, stride=patch_size)

    def forward(self, x):
        return self.patch(x)


class GeluBatch(nn.Module):
    def __init__(self, size) -> None:
        super().__init__()
        # self.gelu = GELU()
        self.gelu = ReLU()
        # self.drop = nn.Dropout(0)
        self.norm = nn.BatchNorm2d(size)

    def forward(self, x):
        out = self.gelu(x)
        # out= self.drop(out)
        return self.norm(out)
```

```python
class DepthConv2d(nn.Module):
    def __init__(self, channels, kernel_size, p=0.3, padding="same"):
        super().__init__()
        self.conv = nn.Conv2d(
            in_channels=channels,
            out_channels=channels,
            kernel_size=kernel_size,
            groups=channels,
            padding=padding,
        )
        self.drop = nn.Dropout(p)

    def forward(self, x):
        return self.drop(self.conv(x))


class PointConv2d(nn.Module):
    def __init__(self, in_channels, out_channels, p=0.3):
        super().__init__()
        self.conv = nn.Conv2d(
            in_channels,
            out_channels,
            kernel_size=1,
        )
        self.drop = nn.Dropout(p)

    def forward(self, x):
        return self.drop(self.conv(x))


class ConvMixerLayer(nn.Module):
    def __init__(self, size, num_blocks, kernel_size=9):
        super().__init__()

        self.conv_mixer = nn.Sequential(
            *[
                nn.Sequential(
                    ResidualBlock(
                        [
                            DepthConv2d(size, kernel_size),
                            GeluBatch(size),
                        ]
                    ),
                    PointConv2d(size, size),
                    GeluBatch(size),
                )
                for _ in range(num_blocks)
            ]
        )

    def forward(self, x):
        return self.conv_mixer(x)


class ConvMixer(nn.Module):
    def __init__(
        self,
        size,
        num_blocks,
        kernel_size,
        patch_size,
        num_classes,
        op="classification",
    ):
        super().__init__()
        self.layers = nn.Sequential(
            PatchEmbedding(size, patch_size=patch_size),
            GeluBatch(size),
            ConvMixerLayer(size, num_blocks, kernel_size),
            nn.AdaptiveAvgPool2d((1, 1)),
            nn.Flatten(),
        )

        if op == "classification":
            self.op = nn.Linear(size, num_classes)
        elif op == "regression":
            self.op = nn.Sequential(nn.Linear(size, 1))

    def forward(self, x):
        out = self.layers(x)
        # print(out.shape)
        return self.op(out)


class ConvRepeaterLayer(nn.Module):
    def __init__(self, size, num_blocks, kernel_size=9):
        super().__init__()
        self.residual = ConcatBlock
        size = [size]

        for i in range(1, num_blocks + 1):
            size.append(2 * size[i - 1])

        # print(size)
        self.conv_mixer = nn.Sequential(
            *[
                nn.Sequential(
                    self.residual(
                        [
                            DepthConv2d(size[i], kernel_size),
                            GeluBatch(size[i]),
                        ]
                    ),
                    PointConv2d(size[i + 1], size[i + 1]),
                    GeluBatch(size[i + 1]),
                )
                for i in range(num_blocks)
            ]
        )

    def forward(self, x):
        return self.conv_mixer(x)


class ConvRepeater(nn.Module):
    def __init__(
        self,
        size,
        num_blocks,
        kernel_size,
        patch_size,
        num_classes,
        op="classification",
    ):
        super().__init__()
        self.layers = nn.Sequential(
            PatchEmbedding(size, patch_size=patch_size),
            GeluBatch(size),
            ConvRepeaterLayer(size, num_blocks, kernel_size),
            nn.AdaptiveAvgPool2d((1, 1)),
            nn.Flatten(),
        )

        if op == "classification":
            self.op = nn.Linear(size * (2 ** num_blocks), num_classes)
        elif op == "regression":
            self.op = nn.Linear(size * (2 ** num_blocks), 1)

    def forward(self, x):
        out = self.layers(x)
        return self.op(out)


class CNN(torch.nn.Module):
    def __init__(
        self,
        num_blocks,
        size,
        kernel_size,
        strides,
        padding,
        op='classification',
        num_classes=5
    ):
        super().__init__()
        self.num_blocks = num_blocks
        self.residual = ResidualBlock
        # print(size)
        self.layers = nn.Sequential(
            *[
                nn.Sequential(
                    Conv2d(
                        size[i],
                        size[i + 1],
                        kernel_size=kernel_size,
                        stride=strides,
                        padding=padding,
                    ),
                    GeluBatch(size[i + 1]),
                )
                for i in range(len(size) - 1)
            ],
            Flatten(),
        )
        self.flattened = 64 * 16 * 8
        self.linear = Linear(self.flattened, 1024)

        if op == "classification":
            self.op = nn.Linear(1024, num_classes)
        elif op == "regression":
            self.op = nn.Linear(1024, 1)

    def forward(self, x):
        out = self.layers(x)
        out = self.linear(out)
        out = self.op(out)
        return out


class ADNet(torch.nn.Module):
    def __init__(
        self,
        num_classes=5,
    ):
        super().__init__()
        self.in_size = 3
        self.residual = ResidualBlock
        self.layers = nn.Sequential(
            DoubleConvBlock(3, 128, kernel_size=3, strides=1, padding=1),
            MaxPool2d(2, stride=(2, 1)),
            DoubleConvBlock(128, 128, kernel_size=3, strides=1, padding=1),
            MaxPool2d(2, stride=(2, 1)),
            PointConv2d(128, 128, p=0),
            AdaptiveMaxPool2d((1, 1)),
            Flatten(),
        )
        self.flattened = 128
        self.classifier = Linear(self.flattened, num_classes)

    def forward(self, x):
        out = self.layers(x)
        out = self.classifier(out)
        return out
```

```
import pytorch_lightning as pl
import torch
import torchmetrics as tm
from vit_pytorch import ViT

from models import CNN, ADNet, ConvMixer, ConvRepeater

REGRESSION = "regression"
CLASSIFICATION = "classification"
```

```python
def get_means(array, keys):
    means = []
    for key in keys:
        means.append(torch.stack([x[key] for x in array]).mean())
    return means


def get_regression_output(output):
    output = torch.round(output)
    output = torch.clamp(output, min=0, max=4)
    return output


class BaseLightning(pl.LightningModule):
    def __init__(self, num_classes=5, lr=0.03, op=CLASSIFICATION, optim="adam"):
        super().__init__()
        self.lr = lr
        self.op = op
        self.save_hyperparameters()
        self.num_classes = num_classes
        self.optim = optim
        self.num_outputs = num_classes
        if self.op == REGRESSION:
            self.num_outputs = 1

        self.lossf = torch.nn.CrossEntropyLoss()
        if self.op == "regression":
            self.lossf = torch.nn.MSELoss()
        # print(f"Operation is {self.op}")

    def configure_optimizers(self):
        if self.optim == "sgd":
            optimizer = torch.optim.SGD(
                self.parameters(), momentum=0.9, lr=self.lr, nesterov=True
            )
            return {"optimizer": optimizer}

        else:
            optimizer = torch.optim.Adam(self.parameters(), lr=self.lr)
            scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
                optimizer, threshold=5, factor=0.5
            )
            return {
                "optimizer": optimizer,
                "lr_scheduler": {
                    "scheduler": scheduler,
                    "monitor": "train/train_loss",
                    "interval": "epoch",
                },
            }

    def training_step(self, batch, _):
        img, label = batch
        if self.op == "classification":
            label = label
            output = self(img)
            predict = output.argmax(1)
        else:
            label = label.float()
            output = self(img)
            output = torch.squeeze(output, dim=1)
            predict = get_regression_output(output)

        loss = self.lossf(output, label)
        predict = predict.int()
        label = label.int()
        acc = tm.functional.accuracy(predict, label)
        self.log("train/train_accuracy", acc, on_step=False, on_epoch=True)
        self.log(
            "train/train_loss",
            loss,
            # prog_bar=True,
            on_step=False,
            on_epoch=True,
        )
        return loss

    def validation_step(self, batch, _):
        img, label = batch

        if self.op == "classification":
            output = self(img)
            predict = output.argmax(1)
        else:
            label = label.float()
            output = self(img)
            output = torch.squeeze(output, dim=1)
            predict = get_regression_output(output)

        loss = self.lossf(output, label)

        predict = predict.int()
        label = label.int()
        acc = tm.functional.accuracy(predict, label)
        f1 = tm.functional.f1(predict, label)
        mae = tm.functional.mean_absolute_error(predict, label)
        weighted_acc = tm.functional.accuracy(
            predict, label, average="weighted", num_classes=self.num_classes
        )
        return {
            "val_loss": loss,
            "val_accuracy": acc,
            "val_f1": f1,
            "w_acc": weighted_acc,
            "mae": mae
        }

    def validation_epoch_end(self, out):
        acc, loss, f1, w_acc, mae = get_means(
            out, ["val_accuracy", "val_loss", "val_f1", "w_acc", "mae"]
        )
        self.log("val/val_accuracy", acc, on_step=False, on_epoch=True)
        self.log("val/f1", f1, on_step=False, on_epoch=True)
        self.log("val/w_acc", w_acc, on_step=False, on_epoch=True)
        self.log("val/mae", mae, on_step=False, on_epoch=True)
        self.log(
            "val/val_loss",
            loss,
            # prog_bar=True,
            on_step=False,
            on_epoch=True,
        )


class ConvMixerLightning(BaseLightning):
    def __init__(
        self,
        size=5,
        num_blocks=2,
        kernel_size=15,
        patch_size=15,
        num_classes=5,
        lr=0.003,
        res_type="add",
        op="classification",
    ):
        super().__init__(lr=lr, num_classes=num_classes, op=op)
        self.op = op
        if res_type == "add":
            # print("using ConvMixer")
            self.model = ConvMixer(
                size, num_blocks, kernel_size, patch_size, num_classes, self.op
            )
        elif res_type == "cat":
            # print("using ConvRepeater")
            self.model = ConvRepeater(
                size, num_blocks, kernel_size, patch_size, num_classes, self.op
            )
        else:
            print(f"{res_type} is not a valid option")

    def forward(self, x):
        return self.model(x)


class CNN_lightning(BaseLightning):
    def __init__(
        self,
        size=5,
        num_blocks=2,
        kernel_size=15,
        num_classes=5,
        strides=2,
        padding=1,
        lr=0.003,
        op="classification",
    ):
        super().__init__(lr=lr, num_classes=num_classes, op=op)
        self.op = op
        self.model = CNN(
            num_blocks=num_blocks,
            size=size,
            kernel_size=kernel_size,
            strides=strides,
            padding=padding,
            op=self.op,
        )

    def forward(self, x):
        return self.model(x)


class ADNet_lightning(BaseLightning):
    def __init__(self, optim, num_classes=5, lr=0.03, op=CLASSIFICATION):
        super().__init__(lr=lr, num_classes=num_classes, op=op, optim=optim)
        self.model = ADNet(num_classes=self.num_outputs)

    def forward(self, x):
        return self.model(x)


class ViTLigthning(BaseLightning):
    def __init__(
        self,
        num_classes=5,
        image_size=128,
        lr=0.03,
        patch_size=32,
        dim=1024,
        depth=6,
        heads=16,
        mlp_dim=1025,
        dropout=0.1,
        emb_dropout=0.1,
        op="classification",
    ):
        super().__init__(lr=lr, num_classes=num_classes, op=op)
        self.save_hyperparameters(ignore=["image_size", "lr"])
        self.model = ViT(
            image_size=image_size,
            patch_size=patch_size,
            num_classes=self.num_outputs,
            dim=dim,
            depth=depth,
            heads=heads,
            mlp_dim=mlp_dim,
            dropout=dropout,
            emb_dropout=emb_dropout,
            pool="mean",
        )
    def forward(self, x):
        return self.model(x)
```

```
import os

import pandas as pd
import pretty_errors
import pytorch_lightning as pl
import torch
from torchvision import transforms
from torchvision.transforms import ConvertImageDtype, Normalize, Pad, ToTensor

from data_utils import get_submission_dataloader
from lightning_models import ADNet_lightning, ViTLigthning

BATCH_SIZE = 128
NW = 1
sub_dir = "submissions"

best_model_path = f"models/adnet/radar-epoch22-val_loss1.28.ckpt"


trans_ = transforms.Compose(
    [
        ToTensor(),
        Pad(
            [5, 0, 4, 0],
        ),
        ConvertImageDtype(torch.float),
    ]
)

hyps = {
    "num_classes": 5,
    "dropout": 0.1,
    "emb_dropout": 0.1,
}

hyp_print = ""
for key, value in hyps.items():
    hyp_print += f"_{key}_{value}"

if __name__ == "__main__":

    img_dir = "test/*"
    file_name = lambda x: x.split("/")[-1]

    model_name = best_model_path.split("/")[-2]
    model = ADNet_lightning().load_from_checkpoint(best_model_path)

    model.eval()
    model.freeze()
    preds = []
    data = get_submission_dataloader(img_dir, BATCH_SIZE, NW, transform=trans_)
    for batch in data:
        img, img_path = batch
```

```python
        out = model(img)
        predictions = torch.argmax(out, 1).numpy()
        # print(predictions)
        # preds = [(file_name(img_path[i]), pred + 1) for i,pred in enumerate(predictions)]
        for i, pred in enumerate(predictions):
            preds.append((file_name(img_path[i]), pred + 1))
    result_df = pd.DataFrame(preds, columns=["id", "label"])
    result_df.to_csv(os.path.join(sub_dir, f"{model_name}{hyp_print}.csv"), index=False)
    print("submission created")
```

```python
import os
from glob import glob

import cv2
import pandas as pd
from sklearn.model_selection import train_test_split
from torch.utils.data import DataLoader, Dataset


def get_submission_dataloader(img_dir, bs, nw, transform=None):
    dataset = RadarSubmission(img_dir, transform=transform)
    return DataLoader(dataset, batch_size=bs, num_workers=nw)


def kfold_generator(img_dir, csv_path, transform=None):
    train_samples = pd.read_csv(os.path.abspath(csv_path))
    return RadarDataset(img_dir, train_samples, transform=transform)


def data_generator(img_dir, csv_path, bs=4, nw=1, transform=None):

    file_path = pd.read_csv(os.path.abspath(csv_path))

    train_samples, val_samples = train_test_split(
        file_path, test_size=0.2, shuffle=True, stratify=file_path["label"]
    )

    return DataLoader(
        RadarDataset(img_dir, train_samples, transform=transform),
        batch_size=bs,
        shuffle=True,
        num_workers=nw,
        pin_memory=True,
    ), DataLoader(
        RadarDataset(img_dir, val_samples, transform=transform),
        batch_size=bs,
        # shuffle=True,
        num_workers=nw,
        pin_memory=True,
    )


class RadarSubmission(Dataset):
    def __init__(self, img_dir, transform):
        super().__init__()
        self.img_dir = img_dir
        self.img_paths = glob(img_dir)
        self.transform = transform

    def __getitem__(self, idx):
        img_path = self.img_paths[idx]
        img = cv2.imread(img_path)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        # img = img / 255
        if self.transform:
            img = self.transform(img)
        return img, self.img_paths[idx]

    def __len__(self):
        return len(self.img_paths)


class RadarDataset(Dataset):
    def __init__(self, img_dir: str, data_csv: pd.DataFrame, transform=None):
        super().__init__()
        self.image_dir = img_dir
        self.labels = dict(data_csv.values)
        self.imgs = data_csv["id"].values
        self.n_samples = len(self.imgs)
        self.transform = transform

    def __getitem__(self, idx):
        img_name = self.imgs[idx]
        # print(img_name)
        img_path = os.path.join(self.image_dir, img_name)
        img = cv2.imread(img_path)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        # img = img / 255
        if self.transform:
            img = self.transform(img)
        label = self.labels[img_name] - 1

        # print(label)
        return img, label

    def __len__(self):
        return self.n_samples
```

```
import pretty_errors
import pytorch_lightning as pl
import torch
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint
from pytorch_lightning.callbacks.early_stopping import EarlyStopping
from sklearn.metrics import classification_report

from data_utils import data_generator
from lightning_models import ConvMixerLightning

# pl.seed_everything(42)

BATCH_SIZE = 4
NW = 8
```

```
EPOCHS = 10
classif_report = False
file_name = lambda x: x.split("/")[-1]

search_space = {
    "size": [32, 64, 128],
    "num_blocks": [3, 4, 5, 6],
    "kernel_size": [3, 5, 7, 9],
    "patch_size": [20, 25, 30],
}

search_space = {
    "size": [32],
    "num_blocks": [4],
    "kernel_size": [5],
    "patch_size": [11],
}


if __name__ == "__main__":

    train_, val_ = data_generator(
        img_dir="./train", csv_path=r"train.csv", bs=BATCH_SIZE, nw=NW
    )

    for size in search_space["size"]:
        for num_blocks in search_space["num_blocks"]:
            for kernel_size in search_space["kernel_size"]:
                for patch_size in search_space["patch_size"]:
                    hyps = {
                        "size": size,
                        "num_blocks": num_blocks,
                        "kernel_size": kernel_size,
                        "patch_size": patch_size,
                        "num_classes": 5,
                        "lr": 0.3,
                        # "res_type": "add",
                        "res_type": "cat",
                        # "op": "regression",
                        "op": "classification",
                    }
                    hyp_print = ""
                    for key, value in hyps.items():
                        hyp_print += f"_{key}_{value}"
                    print(f"Training {hyp_print}")
                    model = ConvMixerLightning(**hyps)
                    trainer = pl.Trainer(
                        fast_dev_run=True,
                        # benchmark=True,
                        gpus=1,
                        # precision=16,
                        max_epochs=EPOCHS,
                        callbacks=[
                            # ModelCheckpoint(
                            #     monitor="val/val_loss",
                            #     mode="min",
                            #     dirpath=f"models/model{hyp_print}",
                            #     filename="radar-epoch{epoch:02d}-val_loss{val/val_loss:.2f}",
                            #     auto_insert_metric_name=False,
                            # ),
                            EarlyStopping(monitor="val/val_loss", patience=5),
                            LearningRateMonitor(logging_interval="step"),
                        ],
                    )
                    trainer.fit(
                        model,
                        train_dataloaders=train_,
                        val_dataloaders=val_,
                    )

                    if classif_report:
                        model.eval()
                        model.freeze()
                        preds = []
                        gt = []
                        for batch in val_:
                            img, labels = batch
                            img = img.permute(0, 3, 1, 2).float()
                            out = model(img)
                            predictions = torch.argmax(out, 1).numpy()
                            # print(predictions)
                            # preds = [(file_name(img_path[i]), pred + 1) for i,pred in enumerate(predicti
                            for pred, label in zip(predictions, labels):
                                preds.append(pred)
                                gt.append(label)
                        print(classification_report(gt, preds))
```