Detect targets in radar signals

Tudor Andrei Dumitrascu

December 11, 2021

1 Begin

Anonymous (2022)

@ anonymous2022patches

Bibliography

Anonymous. 2022. “Patches Are All You Need?” In *Submitted to the Tenth International Conference on Learning Representations*. https://openreview.net/forum?id=TVHS5Y4dNvM.

Appendix

```
import pytorch_lightning as pl
import numpy as np
import torch
import torch.nn as nn
import torchmetrics as tm


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 = nn.GELU()
        # 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)


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 ConvMixerModule(pl.LightningModule):
    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__()
        self.lr = lr
        self.op = op
        self.save_hyperparameters()
        self.num_classes = num_classes
        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")

        self.lossf = torch.nn.CrossEntropyLoss()
        if self.op == "regression":
            # self.lossf = torch.nn.MSELoss()
            self.lossf = torch.nn.L1Loss()
        print(f"Operation is {self.op}")

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters(), lr=self.lr)
        scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
            optimizer, threshold=5
        )
        return {
            "optimizer": optimizer,
            "lr_scheduler": {
                "scheduler": scheduler,
                "monitor": "val/val_loss",
                "interval": "epoch",
            },
        }

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

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

        img = img.float().permute(0, 3, 1, 2)

        if self.op == "classification":
            label = label
            output = self.model(img)
            predict = output.argmax(1)
```

```
        else:
            label = label.float()
            output = self.model(img)
            output = torch.squeeze(output, dim=1)
            output = torch.round(output)
            output = torch.clamp(output, min=0, max=4)
            predict = output

        loss = self.lossf(output, label)
        # print(output)

        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
        img = img.float().permute(0, 3, 1, 2)

        if self.op == "classification":
            label = label
            output = self.model(img)
            predict = output.argmax(1)
        else:
            label = label.float()
            output = self.model(img)
            output = torch.squeeze(output, dim=1)
            output = torch.round(output)
            output = torch.clamp(output, min=0, max=4)
            predict = output

        loss = self.lossf(output, label)

        # the metrics needs int values
        predict = predict.int()
        label = label.int()
        # metrics that are to be tracked
        acc = tm.functional.accuracy(predict, label)
        f1 = tm.functional.f1(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,
```

```
        }

    def validation_epoch_end(self, out):
        acc = torch.stack([acc["val_accuracy"] for acc in out]).mean()
        loss = torch.stack([acc["val_loss"] for acc in out]).mean()
        f1 = torch.stack([acc["val_f1"] for acc in out]).mean()
        w_acc = torch.stack([acc["w_acc"] for acc in out]).mean()
        self.log("val/val_accuracy", acc, on_step=False, on_epoch=True)
        self.log(
            "val/val_loss", loss, prog_bar=True, 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)
```

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

from data_utils import data_generator

# pl.seed_everything(42)

BATCH_SIZE = 128
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 = ConvMixerModule(**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))
```

```
import os

import pandas as pd
import pretty_errors
import pytorch_lightning as pl
import torch

from convmixer import ConvMixerModule
from data_utils import get_submission_dataloader

pl.seed_everything(42)

BATCH_SIZE = 128
NW = 1
sub_dir = "submissions"

best_model_path = f"models/model_size_512_num_blocks_5_kernel_size_3_patch_size_11_num_classes_5_lr_0.003_

search_space = {
    "size": [512],
    "num_blocks": [5],
    "kernel_size": [3],
    "patch_size": [11],
}

hyps = {
    "size": search_space["size"][0],
    "num_blocks": search_space["num_blocks"][0],
    "kernel_size": search_space["kernel_size"][0],
    "patch_size": search_space["patch_size"][0],
    "num_classes": 5,
    "lr": 0.003,
    "res_type": "add",
    # "op": 'regression'
    "op": "classification",
}

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("/")[-1]
    model = ConvMixerModule(**hyps).load_from_checkpoint(
        best_model_path, **hyps
    )

    model.eval()
    model.freeze()
```

```
    preds = []
    data = get_submission_dataloader(img_dir, BATCH_SIZE, NW)
    for batch in data:
        img, img_path = 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(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")
```

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


def get_submission_dataloader(img_dir, bs, nw):
    dataset = RadarSubmission(img_dir)
    return DataLoader(dataset, batch_size=bs, num_workers=nw)


def data_generator(img_dir, csv_path, bs=4, nw=1):

    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),
        batch_size=bs,
        shuffle=False,
        num_workers=nw,
        pin_memory=True,
    ), DataLoader(
        RadarDataset(img_dir, val_samples),
        batch_size=bs,
        shuffle=False,
        num_workers=nw,
        pin_memory=True,
    )


class RadarSubmission(Dataset):
    def __init__(self, img_dir):
        super().__init__()
```

```
        self.img_dir = img_dir
        self.img_paths = glob(img_dir)

    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
        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):
        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)

    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
        label = self.labels[img_name] - 1
        # print(label)
        return img, label

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