# به نام خدا

# شرح پروژه و کد

پروژه در پایتون و به کمک کتابخانه پایتورچ پیاده سازی شده است.

در سلول اول کتاب خانه پایتورچ و زیر کتاب خانه های مورد نیاز به همراه تعدادی کتابخانه عمومی پایتون به پروژه افزوده شده است.

در سلول بعد حالت های مختلف اجرای پروژه قرار داده شده است: 8 حالت به قرار زیر:

* fomaml-5way-1shot (بخش دوم)
* maml-5way-1shot (بخش اول)
* fomaml-5way-5shot (بخش دوم)
* maml-5way-5shot (بخش اول)
* fomaml-5way-1shot\_COSINE Scheduler (امتیازی)
* maml-5way-1shot\_COSINE Scheduler (امتیازی)
* fomaml-5way-5shot\_COSINE Scheduler (امتیازی)
* maml-5way-5shot\_COSINE Scheduler (امتیازی)

با تنظیم پارامتر های سلول دوم هر یک از 8 حالت بالا را میتوان ایجاد کرد.

در سلول سوم هایپر پارامتر های متالرنینگ تعریف شده است. شامل:

meta\_train\_batch = 4

meta\_test\_batch = 20

meta\_epochs = 20000

base\_train\_epochs = 5

base\_test\_epochs = 10

meta\_train\_rep\_freq = 50

meta\_test\_freq = 200

meta\_save\_freq = 1000

alpha = 0.01 # base model learning rate

beta = 0.01 # meta model learning rate

سپس دو تسک ست تعریف کردیم، یکی برای ترین و دیگری برای تست.

هر تسک ست شامل 4 تسک برای 1shot و دو تسک برای ترین به صورت 5shot میشود. برای تسک ست تست 20 تسک را در نظر گرفتیم (حل تمرین مربوطه در گروه 5 تسک را کافی دانستند و مقدار بیشتر را اختیاری در نظر گرفتند)

سلول بعد برای مشخص کردن دستگاهی که شبکه ها قرار است بر روی آن ترین شوند است. ما ترجیه استفاده را به GPU Cuda دادیم و در صورت موجود نبودن به روی cpu اجرا میشود.

در دو سلول بعدی دو تابع کمکی تعریف کرده ایم، تابع اول برای محاسبه دقت و f1 score است و تابع بعدی برای بخش امتیازی (تعیین نرخ یادگیری در هر ایپاک به کمک زمانبندی کسینوسی که استاد در کلاس اموزش دادند.)

سلول بعد معماری شبکه عصبی کانولوشن را تعیین میکند، این معماری مطابق با مقاله مرجع قرار گرفته شده ایجاد گردیده است. شامل 4 بلوک کانولوشن و یک لایه پرسپترون میشود. هر بلوک شامل یک لایه کانولوشن به دنبال آن Batch Norm و سپس Relu و در اخر مکسپول.

دو سلول بعد دو تابع اصلی این پیاده سازی را شامل میشوند، سلول اول شامل تابع اموزش مدل داخلی (حلقه داخلی) و سلول دوم شامل تابع آموزش متا مدل است. (شرح کامل دو تابع در ادامه امده است)

در سلول بعد حلقه اصلی برنامه قرار دارد، این حلقه به تعداد متا ایپاک ها که در این مسئله 20000 تا فرض شده است میچرخد.

در هر گردش حلقه، ابتدا تعدادی تسک از تسک ست آموزش انتخاب می شود (4 تا برای 1 شات و 2 تا برای 5 شات) و سپس به ازای هر تسک مدلی اموزش داده میشود. معماری این مدل ها با معماری که بالاتر توضیح داده شد یکسان است. برای اموزش هر مدل تابع اموزش مدل داخلی train\_base\_model صدا زده میشود. این تابع تسک، معماری مدل، ایپاک حلقه داخلی، دستگاه، الفا و مرتبه مشتق را گرفته و متناسبا مدلی را اموزش میدهد. برای اموزش مدل ابتدا داده ی ساپورت و کوئری تسک ورودی استخراج شده و سپس مدل به مدت ایپاک حلقه داخلی روی داده ساپورت ترین میشود. (5 ایپاک طبق داده های مسئله)

سپس مدل بر روی داده های کوئری پیش بینی انجام داده و مقدار لاس و گرادیانش را بر روی کوئری ذخیره می کنیم.

بعد از بدست اوردن گرادیان و لاس مدل های مختلف ( که متناظر با تسک های مختلف ترین هستند) این لاس و گرادیان ها با هم ترکیب شده و میانگین گرفته میشود. از گرادیان حاصل برای اپدیت پارامتر های متا مدل استفاده میشود (در تابع train\_meta\_model)

در این تابع ما یکی یکی پارامتر های متا مدل را با گرادیان هم ارزش آپدیت می کنیم.

در انتها مدل جدید به جای متا مدل فعلی نشسته و به گردش بعدی حلقه میرویم. برای توضیحات بیشتر، خط به خط کد نوشته شده کامنت گذاری شده است.

# %%

import torch

import torchvision

from torchvision import transforms

from torch import nn, optim

import numpy as np

import random

from PIL import Image

import matplotlib.pyplot as plt

from sklearn.metrics import accuracy\_score, f1\_score

import os, copy, time, math

import MiniImageNetTaskSet as ts

# %%

# fomaml-5way-5shot.ipynb

nway = 5

support\_samples = 5

query\_samples = 10

dv\_mode = 'fo' # fo: first order (FOMAML) - so: second order (MAML)

use\_lr\_scheduler = False

#-- Running on 1650

# %%

meta\_train\_batch = 4

meta\_test\_batch = 20

meta\_epochs = 20000

base\_train\_epochs = 5

base\_test\_epochs = 10

meta\_train\_rep\_freq = 50

meta\_test\_freq = 200

meta\_save\_freq = 1000

alpha = 0.01 # base model learning rate

beta = 0.01 # meta model learning rate

ts\_train = ts.MiniImageNetTaskSet(mode='train', nway=nway, support\_samples=support\_samples, query\_samples=query\_samples)

#ts\_val = ts.MiniImageNetTaskSet(mode='val', nway=nway, support\_samples=support\_samples, query\_samples=query\_samples)

ts\_test = ts.MiniImageNetTaskSet(mode='test', nway=nway, support\_samples=support\_samples, query\_samples=query\_samples)

# %%

if torch.cuda.is\_available():

    device = torch.device('cuda')

    print('OK CUDA was avaialbe.')

else:

    device = torch.device('cpu')

print(device)

# %%

def acc\_f1(y, y\_hat):

    p = y\_hat.max(1, keepdim=True)[1]

    acc, f1 = accuracy\_score(y.cpu(), p.cpu()), f1\_score(y.cpu(), p.cpu(), average='macro')

    return acc, f1

# %%

def cosine\_decay(epoch, beta):

    a = 0.5 \* beta \* (1 + math.cos(epoch \* math.pi / meta\_epochs))

    return a

# %%

meta\_model = nn.Sequential(

    nn.Conv2d(3, 32, (3,3), (1,1)),

    nn.BatchNorm2d(32),

    nn.ReLU(),

    nn.MaxPool2d((2,2)),

    nn.Conv2d(32, 32, (3,3), (1,1)),

    nn.BatchNorm2d(32),

    nn.ReLU(),

    nn.MaxPool2d((2,2)),

    nn.Conv2d(32, 32, (3,3), (1,1)),

    nn.BatchNorm2d(32),

    nn.ReLU(),

    nn.MaxPool2d((2,2)),

    nn.Conv2d(32, 32, (3,3), (1,1)),

    nn.BatchNorm2d(32),

    nn.ReLU(),

    nn.MaxPool2d((2,2)),

    nn.Flatten(),

    nn.Linear(288, nway)

)

print(meta\_model)

# %%

def train\_base\_model(base\_model, episode, base\_epochs, device, alpha, dv\_mode):

    base\_model\_grads = None # variable to store base model grads after backward propagating Query data loss

    D\_sup, D\_que = episode # extracting Support and Query data from episode

    X\_sup, y\_sup = D\_sup # extracting Support inputs (X) and labels (y)

    X\_que, y\_que = D\_que # extracting Query inputs (X) and labels (y)

    # assigning Support and Query tensors to device

    X\_sup, y\_sup = X\_sup.to(device), y\_sup.to(device)

    X\_que, y\_que = X\_que.to(device), y\_que.to(device)

    base\_criterion = nn.CrossEntropyLoss() # base model loss function

    base\_optimizer = optim.SGD(base\_model.parameters(), lr=alpha) # base model optimizer

    # training base model on Support data

    base\_model.train() # telling pytorch we want to train this base model

    for i in range(base\_epochs):

        base\_optimizer.zero\_grad() # telling pytorch to clear calculated grads from previous computations

        y\_sup\_hat = base\_model(X\_sup) # feed forward of all Support inputs for current episode

        base\_loss = base\_criterion(y\_sup\_hat, y\_sup) # computing the loss of current task Support

        # backward propagate derivatives for base model

        base\_loss.backward()

        # update base model parameters

        base\_optimizer.step()

    # testing trained base model on Query data

    else:

        base\_model.train() # telling pytroch we want to train this base model

        base\_optimizer.zero\_grad() # telling pytorch to clear calculated grads from previous computations

        y\_que\_hat = base\_model(X\_que) # feed forward of all Query inputs for current episode

        base\_que\_loss = base\_criterion(y\_que\_hat, y\_que) # calculating query loss for base model

        if dv\_mode == 'fo': # first order grad calc

            base\_que\_loss.backward() # backward propagating the query loss to calculate parameters gradients w.r.t. query loss

        elif dv\_mode == 'so': # second order grad calculation

            base\_que\_loss.backward(retain\_graph=True)

            base\_que\_loss.backward()

        # storing base model grads w.r.t. Query data loss

        base\_model\_grads = [param.grad for param in list(base\_model.parameters())]

        # storing base model loss over Query data

        base\_model\_loss = base\_que\_loss.item()

        # calculating base model's accuracy and F1 score over Query data

        base\_model\_acc, base\_model\_f1 = acc\_f1(y\_que, y\_que\_hat)

        # removing the base model from gpu to free vram

        del base\_que\_loss

        del base\_loss

        del base\_criterion

        del base\_optimizer

        del base\_model

        torch.cuda.empty\_cache()

    return base\_model\_loss, base\_model\_grads, base\_model\_acc, base\_model\_f1

# %%

def train\_meta\_model(meta\_model, beta, meta\_batch, dv\_mode, device, ls\_base\_models\_grads, ls\_base\_models\_losses, ls\_base\_models\_acc, ls\_base\_models\_f1):

    # training meta model over a batch of episodes

    with torch.no\_grad():

        \_ = meta\_model.to(device) # assigning meta model to device

        # a list to store meta model parameters grads

        meta\_model\_grads = [torch.zeros\_like(param) for param in ls\_base\_models\_grads[0]]

        # creating meta model parameters grads

        for base\_model\_grads in ls\_base\_models\_grads: # for each base model

            for grad\_index in range(len(base\_model\_grads)): # for each base model's parameter's grad

                base\_model\_grad = base\_model\_grads[grad\_index] # get base model grad

                # create meta model parameter grad

                meta\_model\_grads[grad\_index] += (1/meta\_batch) \* base\_model\_grad

        # updating meta model parameter

        for param, grad in zip(meta\_model.parameters(), meta\_model\_grads):

            new\_param = param - beta \* grad

            param.copy\_(new\_param)

    # calculate meta training loss, acc, f1 by averaging base models' losses, acc, f1

    meta\_loss = sum(ls\_base\_models\_losses) / meta\_batch

    meta\_acc = sum(ls\_base\_models\_acc) / meta\_batch

    meta\_f1 = sum(ls\_base\_models\_f1) / meta\_batch

    return meta\_model, meta\_loss, meta\_acc, meta\_f1

# %%

ls\_meta\_train\_metrics = []

ls\_meta\_test\_metrics = []

time\_begin = time.time()

for meta\_i in range(1, meta\_epochs+1):

    # sample meta\_batch tasks

    train\_episodes = ts\_train.sample\_episodes(meta\_train\_batch)

    ls\_base\_models\_grads = []

    ls\_base\_models\_losses = []

    ls\_base\_models\_acc = []

    ls\_base\_models\_f1 = []

    for episode, ep\_i in zip(train\_episodes, range(len(train\_episodes))):

        base\_model = copy.deepcopy(meta\_model) # copy parameters of meta model to base model

        \_ = base\_model.to(device) # assign base model to device

        # training base model on Support and Calculate it's Gradients w.r.t. Query data

        base\_model\_loss, base\_model\_grads, base\_model\_acc, base\_model\_f1 = train\_base\_model(base\_model,

                                                                                            episode,

                                                                                            base\_train\_epochs,

                                                                                            device,

                                                                                            alpha,

                                                                                            dv\_mode)

        ls\_base\_models\_losses.append(base\_model\_loss)

        ls\_base\_models\_grads.append(base\_model\_grads)

        ls\_base\_models\_acc.append(base\_model\_acc)

        ls\_base\_models\_f1.append(base\_model\_f1)

    # training meta model w.r.t. base models' gradients

    lr = beta if not use\_lr\_scheduler else cosine\_decay(meta\_i, beta)

    meta\_model, meta\_train\_loss, meta\_train\_acc, meta\_train\_f1 = train\_meta\_model(meta\_model,

                                                                lr,

                                                                meta\_train\_batch,

                                                                dv\_mode,

                                                                device,

                                                                ls\_base\_models\_grads,

                                                                ls\_base\_models\_losses,

                                                                ls\_base\_models\_acc,

                                                                ls\_base\_models\_f1)

    ls\_meta\_train\_metrics.append((meta\_train\_loss, meta\_train\_acc, meta\_train\_f1))

    # storing model checkpoint

    if meta\_i%meta\_save\_freq == 0:

        torch.save(meta\_model.state\_dict(), os.path.join('models', f'model\_{meta\_i}\_{nway}way\_{support\_samples}shot\_{dv\_mode}{"\_COS" if use\_lr\_scheduler else ""}.pt'))

    # reporting back to user

    if meta\_i%meta\_train\_rep\_freq == 0:

        print(f'- Meta Epoch: {meta\_i}, meta loss: {meta\_train\_loss}, meta acc: {round(meta\_train\_acc, 3)}, meta f1: {round(meta\_train\_f1, 3)}, took: {round(time.time()-time\_begin, 1)} secs')

        time\_begin = time.time()

    # meta testing

    if meta\_i%meta\_test\_freq == 0:

        time\_begin = time.time()

        test\_episodes = ts\_test.sample\_episodes(meta\_test\_batch)

        ls\_test\_models\_losses = []

        ls\_test\_models\_acc = []

        ls\_test\_models\_f1 = []

        for episode, ep\_i in zip(test\_episodes, range(len(test\_episodes))):

            test\_model = copy.deepcopy(meta\_model) # copy parameters of meta model to val model

            \_ = test\_model.to(device) # assign base model to device

            # training val model on Support and Calculate it's loss, f1 and acc w.r.t. Query data

            test\_model\_loss, \_, test\_model\_acc, test\_model\_f1 = train\_base\_model(test\_model, episode, base\_test\_epochs, device, alpha, dv\_mode)

            ls\_test\_models\_losses.append(test\_model\_loss)

            ls\_test\_models\_acc.append(test\_model\_acc)

            ls\_test\_models\_f1.append(test\_model\_f1)

        # calculate meta validation loss, acc, f1 by averaging validation models' losses, acc, f1

        meta\_test\_loss = sum(ls\_test\_models\_losses) / meta\_test\_batch

        meta\_test\_acc = sum(ls\_test\_models\_acc) / meta\_test\_batch

        meta\_test\_f1 = sum(ls\_test\_models\_f1) / meta\_test\_batch

        ls\_meta\_test\_metrics.append((meta\_test\_loss, meta\_test\_acc, meta\_test\_f1))

        print(f'[META TEST]- Meta Epoch: {meta\_i}, meta loss: {meta\_test\_loss}, meta acc: {round(meta\_test\_acc, 3)}, meta f1: {round(meta\_test\_f1, 3)}, took: {round(time.time()-time\_begin, 1)} secs')

        time\_begin = time.time()

# %%

ls\_meta\_train\_losses = [item[0] for item in ls\_meta\_train\_metrics]

plt.plot(ls\_meta\_train\_losses)

plt.xlabel('Meta Training Epoch')

plt.ylabel('Meta Training Loss')

plt.title('Meta Training Loss Plot')

# %%

ls\_meta\_train\_acc = [item[1] for item in ls\_meta\_train\_metrics]

plt.plot(ls\_meta\_train\_acc)

plt.xlabel('Meta Training Epoch')

plt.ylabel('Meta Training Accuracy')

plt.title('Meta Training Accuracy Plot')

# %%

ls\_meta\_train\_f1 = [item[2] for item in ls\_meta\_train\_metrics]

plt.plot(ls\_meta\_train\_f1)

plt.xlabel('Meta Training Epoch')

plt.ylabel('Meta Training F1-Score')

plt.title('Meta Training F1-Score Plot')

# %%

ls\_meta\_test\_losses = [item[0] for item in ls\_meta\_test\_metrics]

plt.plot(ls\_meta\_test\_losses)

plt.xlabel('Meta Testing Epoch')

plt.ylabel('Meta Testing Loss')

plt.title('Meta Testing Loss Plot')

# %%

ls\_meta\_test\_acc = [item[1] for item in ls\_meta\_test\_metrics]

plt.plot(ls\_meta\_test\_acc)

plt.xlabel('Meta Testing Epoch')

plt.ylabel('Meta Testing Accuracy')

plt.title('Meta Testing Accuracy Plot')

# %%

ls\_meta\_test\_f1 = [item[2] for item in ls\_meta\_test\_metrics]

plt.plot(ls\_meta\_test\_f1)

plt.xlabel('Meta Testing Epoch')

plt.ylabel('Meta Testing F1-Score')

plt.title('Meta Testing F1-Score Plot')

# %%

import pandas as pd

# %%

df\_test = pd.DataFrame(ls\_meta\_test\_metrics, columns=['loss', 'acc', 'f1'])

df\_train = pd.DataFrame(ls\_meta\_train\_metrics, columns=['loss', 'acc', 'f1'])

df\_test.to\_csv('test\_metrics.csv', index=False)

df\_train.to\_csv('train\_metrics.csv', index=False)

# %%

در ادامه نمودار های 8 حالت مختلف آمده است، داده های ترین 20000 نقطه با رنگ آبی و داده های 100 بار تست کردن ( هر 200 بار ترین یک تست ) با رنگ نارنجی رسم شده است.

همانگونه که از نمودار ها مشخص است، نوسان نمودار آبی (20000 ایپاک متا ترین) بسیار بیشتر ازنمودار نارنجی (متا تست) است. از نظر بنده دلیل این اتفاق میتواند تلاش مدل برای فرار از مینیمم های محلی باشد. مدل به دنبال پارامترهایی است که بتواند برای رنج وسیعی از تسک ها خوب عمل کند، پس طبیعی است که وزن های مختلفی را مورد بررسی قرار دهد. از طرفی دلیل نوسانات کمتر روی تسک های تست، میتواند به دلیل پیدا شدن وزن های مناسب توسط بخش ترین باشد.

جداول زیر مقایسه نتایج مدل های مختلف روی تسک های تست و ترین است. اعداد به درصد هستند.



نتایج بر روی تسک های تست

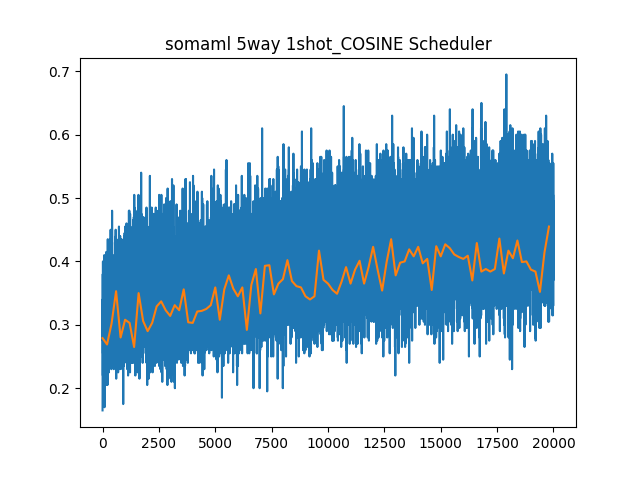
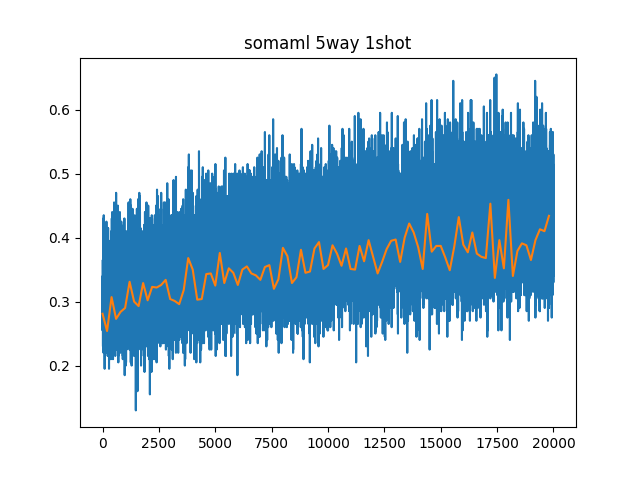
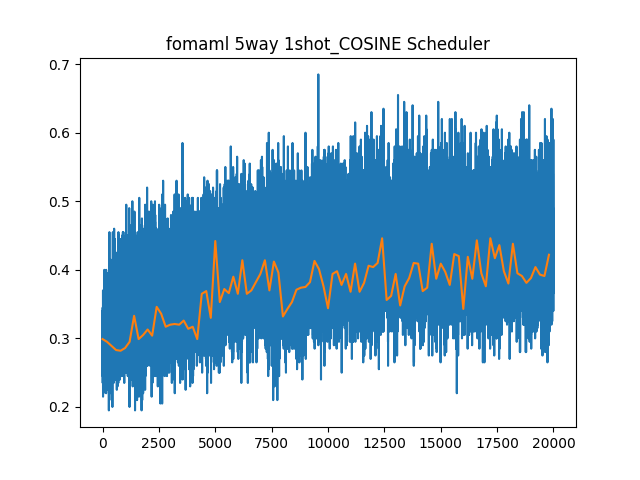
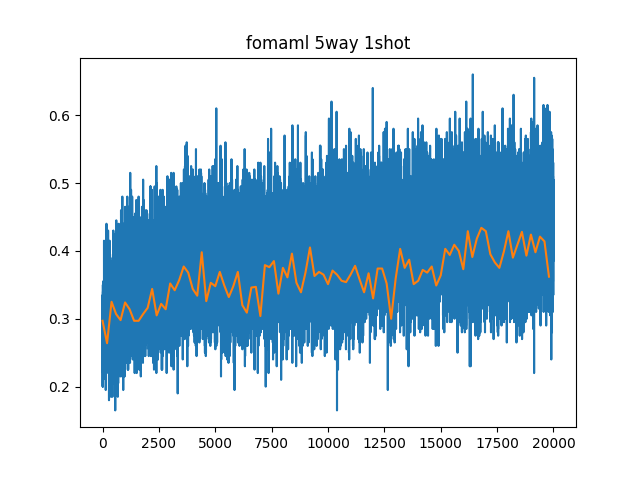


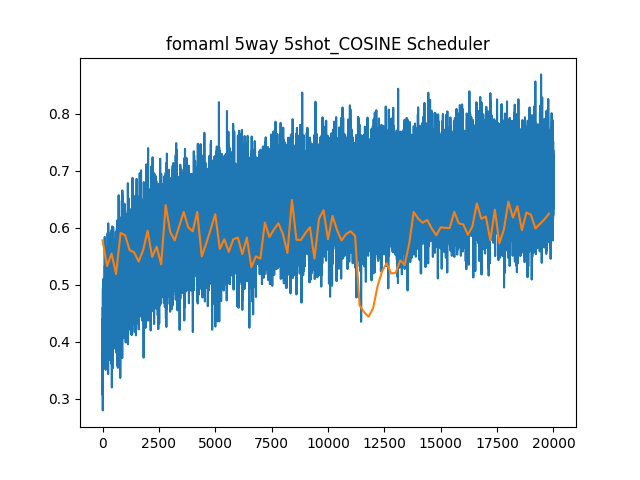
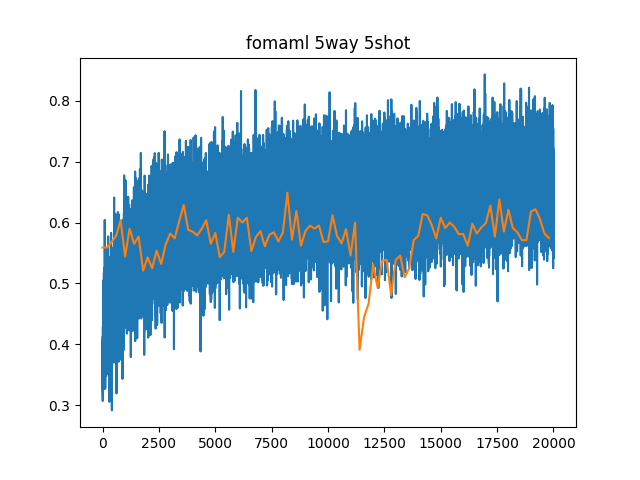
نتایج بر روی تسک های ترین

برای دیدن نتایج کامل ( شامل میانگین، انحراف معیار، مینیمم، مکسیمم) به فایل analysis.xlsx بروید.

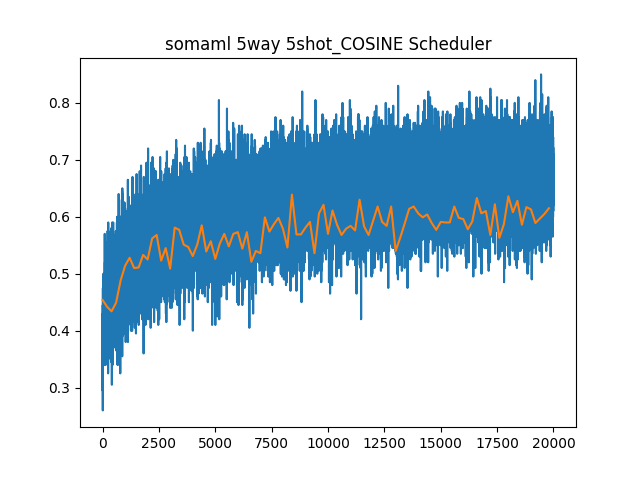
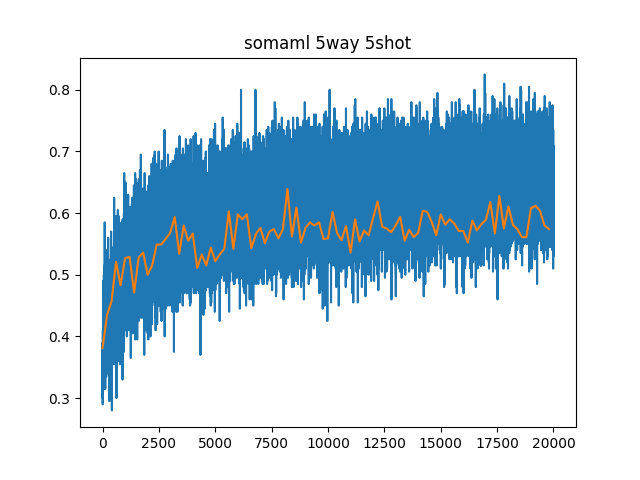
مدل هایی که به صورت First order MAML ترین شدند دقت بالاتری نسبت به مدل های عادی MAML دارند و از طرفی مدل های 5 شات نسبت به 1 شات دقت بالاتری نشان داده اند.

توجه داشته باشید، که دقت ها و F1 score ها به صورت میانگین هستند.



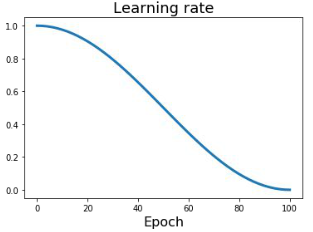


این دوتا نمودار به خاطر یک باگ در کد در زمان ترین، یک شیفت در نتایج تست خورده است، اون تکه که افت شدید کرده در واقع در ابتدای نمودار بوده است. بسیار علاقه مند بودم که این مشکل رو درست کنم اما Google Colab دیگه بهم GPU نداد. داده های تست دقیق و درست هستند به همین دلیل از نظر میانگین و ماکسیمم و مینیمم تفاوتی ندارند (نتایج جداول چند صفحه قبل صحیح است) اما چون ترتیب قرار گیری این نتایج اشتباه شده است، نمودار ها دچار این آشفتگی شده اند.



به صورت 5 شات چون دیتای بیشتری برای ترین در اختیار مدل بوده است، مدل توانسته به دقت بالاتری برسد. چه در حالت ترین و چه در حالت تست.

استفاده از نرخ یادگیری داینامیک و متغیر نیز به مدل برای یادگیری بهتر کمک می کند. برای متغیر کردن نرخ یادگیری، بنده از یادگیری کسینوسی که در کلاس توسط استاد تدریس شد استفاده کردم. در این نوع یادگیری ما نرخ یادگیری را به عنوان حاصل ضرب یک نرخ یادگیری پایه در یک عبارت کسینوسی در نظر میگیریم، فرمول محاسبه نرخ یادگیری در این حالت به شرح زیر میباشد.





که در آن آلفا صفر نرخ یادگیری پایه، تی بزرگ تعداد ماکسیمم ایپاک ها و تی کوچک ایپاک فعلی را نشان میدهد. نرخ یادگیری متغیر با کوچکتر کردن تدریجی گام های بروزرسانی پارامتر های مدل، باعث جلوگیری از تغییر ناگهانی مدل و رفتار واریانسی آن میشود. به این صورت که در ابتدا به مدل آزادی عمل بیشتری در یادگیری میدهد و به تدریج این ازادی کاهش یافته و مدل را به ثبات میرساند. از دید فضای حل مسئله، در ابتدا ما فضای بیشتری را مورد کاوش قرار میدهیم و با گذشت زمان فضا مورد جست و جو کاهش بیشتری پیدا میکند و در نهایت به اطراف نقطه ای محدود می شود.

از طرف دیگر با توجه به نتایج بدست آمده که در جدول قابل مشاهده است، مدل ها در حالت first order maml بهتر ترین شده اند و نتایج بهتری را ارائه داده اند که این امر در مقاله مرجع نیز مورد اشاره واقع شده بود.

برای اجرای پروژه فایل maml.ipynb را به کمک نرم افزاری مناسب مانند Google Colab باز کنید و سپس بعد از تنظیم پارامتر های مورد نیاز (ایجاد هرکدام از 8 حالت) در سلول دوم، کل سلول ها را با هم دیگر اجرا کنید. توجه داشته باشید در هنگام اجرا در کنار فایل نوت بوک، فولدری به نام models موجود باشد تا کد نوشته شده بتواند به ازای هر 1000 ایپاک یک مدل ذخیره کند که در صورت از دست رفتن اجرا، بتوانید از نزدیک ترین اجرا مجدد از سر بگیرید.

همچنین فایل MiniImageNetTaskSet.py نیز باید در کنار فایل نوت بوک موجود باشد. این فایل شامل کلاس و توابع مورد نیاز برای پارس کردن تصاویر و ایجاد تسک ست مناسب است. تمام خطوط کد این فایل درون فایل توضیح داده شده است. برای راحتی شما این کد ها در ادامه نیز پیوست شده است.

import torch

from torchvision import transforms

import random

from PIL import Image

import matplotlib.pyplot as plt

import os

class MiniImageNetTaskSet():

    '''A task set helper class

    '''

    def \_\_init\_\_(self, root='miniimagenet/', mode='train', nway=5, support\_samples=1, query\_samples=10) -> None:

        self.root = root

        self.data\_dir = os.path.join(root, 'data')

        self.mode = mode

        self.nway = nway

        self.sup\_count = support\_samples

        self.que\_count = query\_samples

        with open(os.path.join(root, 'splits', f'{mode}.txt')) as file:

            self.C = [line.strip().split(',') for line in file.readlines()]

        self.transform = transforms.Compose([

            #transforms.Resize((32,32)),

            transforms.ToTensor(),

            transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225)),

        ])

    def \_\_len\_\_(self):

        'Denotes the total number of classes'

        return len(self.C)

    def sample\_episode(self):

        # sample nway class from all classes avaialabe to this task set

        C = random.sample(self.C, k=self.nway)

        ls\_tensors\_sup = []

        ls\_tensors\_que = []

        ls\_labels\_sup = []

        ls\_labels\_que = []

        # load all nway \* (sup\_count + que\_count) samples

        for cc, ci in zip(C, range(len(C))):

            c = cc[0]

            # get all images of class c

            all\_c\_images = os.listdir(os.path.join(self.data\_dir, c))

            # get a sup\_count + que\_count sample out of all

            selected\_images = random.sample(all\_c\_images, k=self.sup\_count+self.que\_count)

            # load images to torch tensors

            tensors = [self.transform(Image.open(os.path.join(self.data\_dir, c, image\_name))).unsqueeze(0) for image\_name in selected\_images]

            ls\_tensors\_sup += tensors[:self.sup\_count]

            ls\_labels\_sup += [ci] \* self.sup\_count

            ls\_tensors\_que += tensors[self.sup\_count:]

            ls\_labels\_que += [ci] \* self.que\_count

        #

        X\_sup = torch.cat(ls\_tensors\_sup, 0)

        y\_sup = torch.tensor(ls\_labels\_sup)

        #

        X\_que = torch.cat(ls\_tensors\_que, 0)

        y\_que = torch.tensor(ls\_labels\_que)

        # shuffling Support and Query data

        shuff\_indices\_sup = random.sample(range(X\_sup.shape[0]), X\_sup.shape[0])

        X\_sup = X\_sup[shuff\_indices\_sup]

        y\_sup = y\_sup[shuff\_indices\_sup]

        shuff\_indices\_que = random.sample(range(X\_que.shape[0]), X\_que.shape[0])

        X\_que = X\_que[shuff\_indices\_que]

        y\_que = y\_que[shuff\_indices\_que]

        D\_sup = (X\_sup, y\_sup) # nway \* sup\_count

        D\_que = (X\_que, y\_que) # nway \* que\_count

        return (D\_sup, D\_que) # a sampled task

    def sample\_episodes(self, batch):

        return [self.sample\_episode() for \_ in range(batch)]

    def visualize\_episode(self, task):

        D\_sup, D\_que = task

        X\_sup, y\_sup = D\_sup

        X\_que, y\_que = D\_que

        X = torch.cat([X\_sup, X\_que], 0)

        noi = 4

        num\_of\_images = noi \* noi

        for index in range(1, num\_of\_images+1):

            img = X[index-1]

            \_ = plt.subplot(noi,noi, index)

            \_ = plt.axis('off')

            \_ = plt.imshow(img.moveaxis(0, -1))

پیاده سازی این فایل دقیقا مشابه با مقاله مرجع که هم راستا با تدریس استاد در کلاس بوده است انجام شده است.

فایل کمکی run-settings.ipynb برای راحتی ایجاد هر یک از 8 حالت گفته شده ساخته شده است. با اجرای این کد و کپی کردن هر یک از 8 سلول موجود در این فایل در سلول دوم فایل maml.ipynb میتوانید یکی از 8 حالت را ترین کنید.