به نام خدا

گزارش کار پروژه چهارم

مهدی فقهی

۴۰۱۷۲۲۱۳۶

در ابتدای کار بایستی لینک‌هایی که از آن‌ها برای زدن این پروژه استفاده کردم را در اختیار شما بگذارم .

برای زدن این پروژه برای ساخت data loader از لینک :

https://github.com/dragen1860/MAML-Pytorch/blob/master/MiniImagenet.py

و برای ساخت کلاس meta learner که بتواند روی داده classification ما کار کند از لینک

https://github.com/tristandeleu/pytorch-maml:

بخش اول :

خوب در ابتدا به توضیح کد Data loader می‌پردازم .

```
def __init__(self, root='miniimagenet/', mode='train', n_way=5, k_shot=5, k_query=10,resize=84,
             transforms_normalization=transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))):
    self.root = root
    self.mode = mode
    self.n_way = n_way
    self.k_shot = k_shot
    self.k_query = k_query
    self.resize = resize

    if transforms_normalization is None:
      self.transform = transforms.Compose([
          transforms.Resize((resize,resize)),
          transforms.ToTensor(),
      ])
    else :
      self.transform = transforms.Compose([
          transforms.Resize((resize,resize)),
          transforms.ToTensor(),
          transforms_normalization
      ])

    self.data_folder = os.path.join(root, 'data')
    with open(os.path.join(root, 'splits/ravi-larochelle', f'{mode}.txt')) as file:
        self.calsses_mode_name = [line.strip().split(',') for line in file.readlines()]
```

همانطور که مشاهده می‌کنید یک کلاس به نام `MiniImagenet` که ورودی دایرکتوری که فایل دیتا ما در آن قرار گرفته است را می‌گیرد و اینکه این Data loader قرار است روی کدام بخش از داده‌های ما (train, val ,test)

کار کند تعداد کلاس‌های هر تسک که با n way مشخص شده و اینکه از این کلاس‌ها چه تعداد به عنوان support و چه تعداد برای query در نظر گرفته شود که به ترتیب با k_shot و k_query مشخص شده است.

سپس با اطلاعات داده شده یک transforms می‌سازیم و یک لیست از اسم تمام folder هایی که در آن عکس مربوط به مکانی که می‌خواهیم از folder های آن استفاده کنیم یعنی (train, val ,test) بر میداریم .

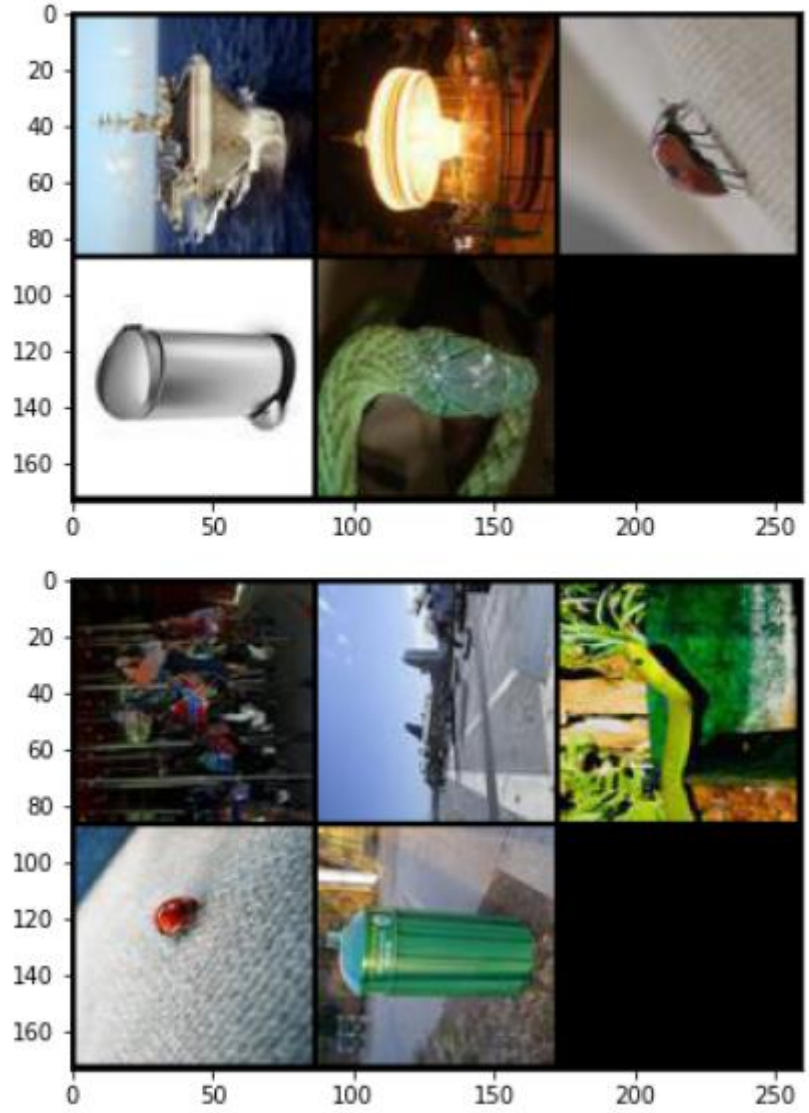
حال در قدم بعد تلاش می کنیم که دیتاهای یک task را بسازیم پس از یک تابع درون کلاس خود استفاده می کنیم که به ما یک تسک بدهد به اسم one_task
همانطور که قبلا گفته شد ما یک کلاس می‌سازیم که یا فقط تسک های train را بدهد یا val و یا test.
حال به کمک این تابعی یکی از تسک‌های ممکن را صدا می‌زنیم .

```
random_classes_name,_ = self.shuffeling(self.calsses_mode_name, self.n_way)
number_of_samples_for_each_class = self.k_shot+self.k_query

for index,name_folder_image_in_list in enumerate(random_classes_name):
    name_folder_image = name_folder_image_in_list[0]

    images = os.listdir(os.path.join(self.data_folder, name_folder_image))

    selected_images,_ = self.shuffeling(images,number_of_samples_for_each_class)

    all_images = [self.transform(Image.open(os.path.join(self.data_folder, name_folder_image, sample_image))).unsqueeze(0)
    for sample_image in selected_images]

    list_of_support.extend(all_images[:self.k_shot])
    class_labels_support = [index for _ in range(self.k_shot)]
    list_labels_support+=class_labels_support

    list_of_query.extend(all_images[self.k_shot:])
    class_labels_query =  [index for _ in range(self.k_query)]
    list_labels_query.extend(class_labels_query)

random_list_of_support,random_list_labels_support = self.shuffeling(list_of_support,len(list_of_support),list_labels_support)
random_list_of_query ,random_list_labels_query = self.shuffeling(list_of_query,len(list_of_query),list_labels_query)

support_x =torch.cat(random_list_of_support, 0)
support_y = torch.tensor(random_list_labels_support)
query_x = torch.cat(random_list_of_query, 0)
query_y =torch.tensor(random_list_labels_query)

return support_x, support_y, query_x, query_y
```

ابتدا به صورت تصادفی تعدادی کلاس به تعداد n way گفته شده اسم‌های folder آنان را جدا می کنم .
سپس در یک حلقه به تعداد مجموع داده‌های لازم برای query و support از این folder عکس به صورت tensor ذخیره می کنیم عکس‌ها اینبار نیز به صورت رندوم انتخاب شده اند سپس عکس‌ها و label ها را به دو دسته به تعداد مورد نیاز برای support و query تقسیم می کنیم و با این لیست‌های بدست آمده را با لیست‌های بدست آمده از مراحل قبل تر ادغام می کنیم و در نهایت به دلیل اینکه عکس‌ها و label های یکسان پشت سرهم قرار نگیرند دوباره این این دو را متناظر جاها به صورت رندوم تغییر می‌دهیم تا یک task در نهایت با چهار آیتم support x , support y , query x query y , ساخته شود و به عنوان خروجی داده شود.
اگر به تعداد تسک‌های خواسته شده این عملیات را صدا بزنیم به تعداد عملیات موردنظر تسک داریم که می‌توانیم درون یک لیست به عنوان خروجی به کاربر بدهیم کاری که تابع list_of_taskانجام می‌دهد .

در انتها برای اینکه عملکرد data loader را مشاهده کنیم یک data loader train ساخته ام که پنج کلاس و هر کلاس یک دیتای support و یک دیتای query دارد که در کد به تعداد تسک‌هایی که می‌خواستم این شکل‌ها را ساخته‌ام و می‌توانید بررسی کنید برای نمونه :

```
support_y : tensor([3, 1, 0, 2, 4])
query_y : tensor([0, 4, 3, 1, 2])
```



بخش دوم :

در بخش دوم به بررسی class ModelAgnosticMetaLearning می‌پردازیم .

قبل از آن باید اشاره به تفاوت روش Maml و Fomaml در این است که Maml برای update کردن گرادیان از مشتق مرتبه دوم استفاده می‌کند و Fomaml از مشتق مرتبه اول .

```
def __init__(self, model,number_task,f1_score,inner_lr,outer_lr,opt_func,device,first_order,
             scheduler=None):
    self.model = model.to(device=device)
    self.lr_outer = outer_lr
    self.lr_inner = inner_lr
    self.device = device
    self.optimizer = opt_func(model.parameters(), self.lr_outer,weight_decay=0.0005)
    self.first_order = first_order
    self.num_adaptation_steps_train = 5
    self.num_adaption_stepss_val = 10
    if scheduler is not None:
      self.scheduler = scheduler(self.optimizer, gamma=0.9)
    else :
      self.scheduler = scheduler

    self.loss_function = F.cross_entropy
    self.number_task = number_task
    self.f1_calculate = f1_score
```

کلاس ModelAgnosticMetaLearning من از این موارد تشکیل می‌شود یک مدل که در واقع اینجا همان CNN ما خواهد شد ، تعداد تسک‌هایی که ماشین با آن آموزش می‌بیند، تابعی که به کمک آن f1_score را حساب می‌کنیم ، میزان یادگیری ماشین در loop داخلی (learning rate) ، میزان یادگیری ماشین در loop بیرونی (learning rate)، اینکه روی کدام یکی از device ها در حال انجام عملیات موردنظر هستیم (cpu , gpu) با first order مشخص می‌کنیم که می‌خواهیم از fomaml استفاده کنیم یا maml و همچنین یک scheduler نیز در صورتی که کاربر به ما بدهد می‌گیریم برای update کردن میزان learning rate بیرونی .

همچنین تعداد تکرارSGDرا در فرا آموزش meta training برابر 5 و meta testing برابر 10 در num adaption step train و num adaption step val قرار میدهیم .

همچنین loss function را برابر با cross entropy و optimizer را هم متناسب با ورودی بر روی learning rate موردنظر با weight decay برابر 0.0005 قرار دادهیم و اگر scheduler داشتیم با همین optimizer و gamma برابر با 0.9 آن را نیز می‌سازم.

حال سراغ بخش آموزش `ModelAgnosticMetaLearning` می‌رویم .

در این بخش هدف ما این است که به تعداد تسک‌هایی که داریم براساس مدل پایه ، همان تعداد ماشین بسازیم ، سپس ماشین ها را براساس داده‌های query ارزیابی کنیم و میزان خطای بدست آمده را جمع کنیم و در انتها براساس میانگین آن مدل اصلی را به کمک gradient descent یا همان optimizer sgd خود آپدیت نماییم .

```
def train(self, mini,ep,save_item):
        train_res = {}
        acc = 0.
        f1 = 0.

        self.optimizer.zero_grad()
        mean_outer_loss = torch.tensor(0.)
        mean_outer_loss = mean_outer_loss.to(self.device)
        for i, set_train in enumerate(mini):
                # print(set_train)
                support_x, support_y, query_x, query_y = set_train

                support_x = support_x.to(self.device)
                support_y = support_y.to(self.device)
                # print(self.device)
                query_x = query_x.to(self.device)
                query_y = query_y.to(self.device)

                params = self.adapt(support_x, support_y,self.num_adaptation_steps_train,self.first_order)
                with torch.set_grad_enabled(self.model.training):
                    test_logits = self.model(query_x, params=params)
                    if ep%save_item ==0:
                              acc += accuracy(test_logits, query_y)
                              _, preds = torch.max(test_logits, dim=1)
                              f1 = f1+self.f1_calculate(preds.cpu(),query_y.cpu())
                    query_y.to(self.device)
                    test_logits.to(self.device)
                    outer_loss = self.loss_function(test_logits, query_y)
                    mean_outer_loss += outer_loss
        mean_outer_loss.div_(self.number_task)

        if ep%save_item ==0:
              loss_train = mean_outer_loss.item()
              acc = acc/self.number_task
              f1 = f1/self.number_task
              print(f"train_loss : {loss_train:.2f} train_acc : {acc:.4f} train_F1: {f1:.4f}")
              train_res ={'train_loss': loss_train, 'train_acc': acc,'train_F1':f1}

        mean_outer_loss.backward()
        self.optimizer.step()

        if self.scheduler is not None:
                self.scheduler.step()

        return train_res
```

همانطور که می‌بینید در حلقه خارجی که به تعداد تسک‌هاست در هر مرحله به کمک تابع adapt پارامترهایی که همان پارامترهای آپدیت شده مدل‌اصلی براساس train روی داده‌های support هر تسک هست را می گیرم و سپس به کمک پارامترهای داده شده و ماشین base داده‌های query را پیش بینی می‌نمایم و در صورتی که مثلا تعداد ۲۰۰ مرتبه شده باشد مقادیر acc , f را حساب می کنیم و همچنین در همه حال میزان loss را بدست می‌آوریم و با میزان loss ماشین‌های قبل بر روی تسک‌های خودشان جمع میزنیم و از این loss یک میانگین میگیریم و پارامترهای اصلی ماشین را آپدیت می کنیم به کمک backward , step optimizer واگر scheduler برای آپدیت کردن learning rate outer loop داشته باشم آن را آپدیت می کنیم .

و در نهایت مقیاس های اعتبار acc , f ,loss1 را در صورت وجود هریک برمیگردانم .

خوب شاید سوال پیش بیاید که تابع adapt چکار می کند .

```
def adapt(self, inputs, targets,num_adaptation_steps_train,first_order=False):

    params = None


    for step in range(num_adaptation_steps_train):
        logits = self.model(inputs, params=params)

        inner_loss = self.loss_function(logits, targets)

        self.model.zero_grad()
        params = gradient_update_parameters(self.model, inner_loss,
            step_size=self.lr_inner, params=params,
            first_order=(not self.model.training) or first_order)

    return params
```

مثلا پنج بار به ماشین خود داده‌های support را می‌دهیم و loss هر مرتبه را حساب می کنیم و سپس به صورت دستی پارامترهای ماشین را به کمک فرمول gradient آپدیت می کنیم .

این آپدیت کردن به کمک gradient نیز اگر maml باشد مشتق مرتبه دوم و اگر fomaml باشد از مشتق مرتبه اول گرادیانت استفاده می کنیم .

تابع gradient update parameters را باهم بررسی می کنیم .

```
    if not isinstance(model, MetaModule):
        raise ValueError('The model must be an instance of `torchmeta.modules.'
                         'MetaModule`, got `{0}`'.format(type(model)))

    if params is None:
        params = OrderedDict(model.meta_named_parameters())

    grads = torch.autograd.grad(loss,
                                params.values(),
                                create_graph=not first_order)

    updated_params = OrderedDict()

    if isinstance(step_size, (dict, OrderedDict)):
        for (name, param), grad in zip(params.items(), grads):
            updated_params[name] = param - step_size[name] * grad

    else:
        for (name, param), grad in zip(params.items(), grads):
            updated_params[name] = param - step_size * grad

    return updated_params
```

در بخش اول این تابع گفته شده که باید مدل ما یعنی حالا cnn که ساختیم یا مدل خطی و یا هر نوع مدل ماشین یادگیری که می‌سازیم باید از نوع Meta model باشد که در انتها درباره آن صحبت می کنیم .

در صورتی که پارامترهای مدل را به ما نداده باشند (اولین بار صدا زدن در لوپ داخلی پارامترها نمی‌دهیم)، پارامترهای مدل را به کمک meta named parameters می گیریم .

سپس در مرحله بعد به کمک torch.auto.grad و مقادیر ورودی loss و پارامترهای مدل و اینکه برحسب مشتق اول کارکند یا مشتق دوم (maml , fomaml) یک میزان گرادیان به ازای هر پارامتر بدست می‌آوریم که باید این مقدار ضرب در learning rate داخلی که اینجا به اسم step size هست آنان را آپدیت کنیم . اگر ما یک دیکشنری که مشخص کند هر کدام از attribute ها پارامترهای مدل ما چه میزان learning rate باید داشته باشند ، در اختیار داشته باشیم هر آیتم متناسب با ضریب یادگیری خودش و گرادیانش مقدارش آپدیت می‌شود در غیر اینصورت که روشی هست که من از این تابع استفاده می کنم میزان نرخ یادگیری همه یکسان در نظر گرفته می‌شود .

نکته قابل توجه این است که اگر create graph برابر با false باشد که همان گرادیان معمولی هست که می‌شود fomaml و اگر true باشد می‌شود می‌توانیم مشتق‌های بالاتر برای گرادیانت حساب کنیم.

توجه : کدهای زده شده در این قسمت مستقیما از لایبری مذکور گرفته شده و هیچ تغییری در آن داده نشده صرفا به توضیح آن پرداختم .

در نهایت خروجی ما پارامترهای آپدیت شده است .

در قدم بعد سراغ قسمت تابع ارزیابی ModelAgnosticMetaLearning می‌رویم .

```
def evaluate(self, mini_val):

    mean_accuracy, mean_f1 = 0., 0.
    mean_outer_loss = torch.tensor(0.)
    mean_outer_loss = mean_outer_loss.to(self.device)

    machine = copy.deepcopy(self.model)
    self.model.eval()

    for i, set_val in enumerate(mini_val):

            count = i
            support_x, support_y, query_x, query_y = set_val
            loss_val,acc_val,f1 = self.evaluate_task(support_x, support_y, query_x, query_y)
            mean_outer_loss += loss_val
            mean_accuracy +=acc_val
            mean_f1 += f1

    mean_outer_loss.div_(count)
    mean_accuracy =mean_accuracy/count
    mean_f1 = mean_f1/count
    print(f"val_loss : {mean_outer_loss.item():.2f} val_acc : {mean_accuracy:.4f} val_F1: {mean_f1:.4f}")
    resut ={'val_loss': mean_outer_loss.item(), 'val_acc': mean_accuracy,'val_F1':mean_f1}

    self.model = machine
    return resut
```

```
def evaluate_task(self, support_x, support_y, query_x, query_y):

    f1 = 0.
    x_spt = support_x.to(self.device)
    y_spt = support_y.to(self.device)
    x_qry = query_x.to(self.device)
    y_qry = query_y.to(self.device)
    params = self.adapt(x_spt, y_spt,self.num_adaption_stepss_val,self.first_order)
    with torch.set_grad_enabled(self.model.training):
            test_logits = self.model(x_qry, params=params)
            outer_loss = self.loss_function(test_logits, y_qry)
            acc = accuracy(test_logits, y_qry)
            _, preds = torch.max(test_logits, dim=1)
            f1 = self.f1_calculate(y_qry.cpu(), preds.cpu())

            return outer_loss, acc,f1
```

در اینجا ابتدا یک copy از ماشین اصلی می‌سازیم .

سپس همانند قبل به ازای تسک‌های مختلف آموزش می‌دهیم، سپس loss و acc و 1f را به ازای تسک‌های مختلف بدست می‌آوریم و به صورت جمعی نگه میداریم و درنهایت میانگین‌های را برمی گردانیم .

قسمت نهایی کد :

```
def train_F_maml(epoch,number_task,n_way,k_shot,k_query,save_item,Fomaml,scheduler):

    device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
    # print(device)
    train = MiniImagenet(mode='train',n_way=n_way,k_shot=k_shot,k_query=k_query)
    val = MiniImagenet(mode='test',n_way=n_way,k_shot=k_shot,k_query=k_query)

    model_base = ModelConvMiniImagenet(n_way,32)
    history = {'train':[],'val':[]}
    opt_func=torch.optim.SGD
    inner_lr=0.01
    outer_lr = 0.01
    save_item = save_item
    f1 = MulticlassF1Score(num_classes=n_way,average = 'macro')
    metalearner = ModelAgnosticMetaLearning(model_base,number_task,f1,inner_lr,outer_lr,opt_func,
                                            device,first_order=Fomaml,scheduler=scheduler)
    for ep in range(epoch):

        res = metalearner.train(train.list_of_task(number_task),ep,save_item)


        if ep%save_item == 0 :
            history['train'].append(res)
            res_val = metalearner.evaluate(val.list_of_task(number_task))
            history['val'].append(res_val)
            print("##########################################################################################

    return history,metalearner
```

در فانکشن train f maml ما تعداد epoch برای انجام meta training ، تعداد تسک‌هایی که مدل meta training ما براساس آن مدل را در هر مرحله بسازد، تعداد کلاس‌های هرتسک ، تعداد مثال به ازی هر تسک درquery , support و اینکه بعد هر چند epoch نتایج ماشین را بررسی ثبت کند برای مثال توی پروژه ما ۲۰۰ و اینکه به صورت maml آموزش بدهد یا fomaml و درنهایت اینکه scholder آن چی باشد را مشخص می کنیم .

براساس دیتاست train و test کارمیکنیم. یک مدل CNN می‌سازیم که خروجی آن تعداد کلاس‌ها هست .

مدل ModelAgnosticMetaLearning خود را می‌سازم براساس داده‌های موجود و در هر epoch آن را train می‌کنم براساس داده‌های موجود در train set و در هر ۲۰۰ مرتبه میزان عملکرد آن را بر روی داده‌های تست می‌سنجم و ذخیره می‌کنم .

```
def test_F_maml(model,repid,number_task,n_way,k_shot,k_query):

    all_loss = 0.
    all_acc = 0.
    all_f1 = 0.
    test = MiniImagenet(mode='test',n_way=n_way,k_shot=k_shot,k_query=k_query)
    for _ in range(repid):
            res_val = model.evaluate(test.list_of_task(number_task))
            all_loss += res_val['val_loss']
            all_acc += res_val['val_acc']
            all_f1  += res_val['val_F1']
    return (all_loss/repid,all_acc/repid,all_f1/repid)
```

جدا از مراحل آموزش ModelAgnosticMetaLearning بعد از آنکه مراحل آموزش تمام شد ، داده‌های تست را برمیداریم و به تعداد تکرار ۱۰ مرتبه مثلا بر روی تعداد task خواسته شده مدل گرفته شده را evaluate می‌کنیم و از میانگین 1loss , acc , f بدست آمده در این ۱۰ مرحله میانگین میگیریم و برمی‌گردانیم .

کدهای پیوست :

مدل که از "MetaModule" ارث بری کند کاملاً با ماژول های PyTorch از "torch.nn.Module" سازگار هستند.

آرگومان «params» یک دیکشنری از تانسورها با پشتیبانی کامل از نمودار محاسباتی (مشتق) است.

یکی از مشکلاتی که من توی این پروژه با آن دست بگریبان بودم آپدیت کردن پارامترها مدل اصلی به صورت دستی و بدون backward خود torch بود ، در حین گشتن ها با لایبرری دوم آشنا شدم و متوجه شدم همانکاری که من می‌خواستم انجام دهم را دارد انجام می‌دهد با این تفاوت که مدل CNN خود را به گونه‌ای تعریف کرده بود که عملیات gradiant decent دستی را روی آن انجام می‌داد .

این مدل که از MetaModule ارث بری کرده بود را درادامه باهم بررسی می کنیم و همچنین code مربوط به MetaModule نیز در کدها برای بررسی بیشتر آورده‌ام .

در ادامه ساخت class meta Model مجبوریم یک کلاس به نام `MetaSequential` که بتوانیم عمل forwarding برای meta Model را نیز انجام دهیم .

```
class MetaSequential(nn.Sequential, MetaModule):
    __doc__ = nn.Sequential.__doc__

    def forward(self, input, params=None):
        for name, module in self._modules.items():
            if isinstance(module, MetaModule):
                input = module(input, params=self.get_subdict(params, name))
            elif isinstance(module, nn.Module):
                input = module(input)
            else:
                raise TypeError('The module must be either a torch module '
                    '(inheriting from `nn.Module`), or a `MetaModule`. '
                    'Got type: `{0}`'.format(type(module)))
        return input
```

همانطور که در اینجا می‌بینیم تابع forward را در اینجا به دو شکل جلو می‌بریم همانند خود تابع forward nn.sequential ابتدا لایه به لایه جلو می‌رویم اگر مدل از نوع nn.Module اصلی باشد که مشکلی ندارم براساس ورودی هرلایه ورودی لایه بعد را تولید می کنیم به کمک یکی از attributeهای آن به نام module و اینکار را تا لایه آخر ادامه می‌دهیم و خروجی را بیرون میدهیم .

اما اگر از نوع MetaModole باشد مدل ما علاوه بر مدل باید براساس پارامترهای در آن لایه و نوع فانکشن عمل کننده که مدل است مقدار خروجی را بدست آوریم و عمل forwarding را انجام دهیم در ادامه علت این امر را توضیح می‌دهم .

```
class MetaConv2d(nn.Conv2d, MetaModule):
    __doc__ = nn.Conv2d.__doc__

    def forward(self, input, params=None):
        if params is None:
            params = OrderedDict(self.named_parameters())
        bias = params.get('bias', None)
        return self._conv_forward(input, params['weight'], bias)
```

مدل MetaConv2d را می‌سازیم که همان Conv2d خودمان هست با این تفاوت که از MetaModule ارث بری کرده است ، اگر پارامترها مدل در این لایه را داشته باشیم که مشکلی نیست اگر نه که با صدا زدن named_parameters از

فانکشن‌های MetaModule به params ها دسترسی پیدامی‌کنیم و در نهایت به کمک تابع con_forward خود conv2d مقدار ورودی را از این لایه عبور میدهیم براساس پارامتر وزن و بایاس که در قدم قبل به آنان دسترسی پیدا کردیم .

```
class MetaLinear(nn.Linear, MetaModule):
    __doc__ = nn.Linear.__doc__

    def forward(self, input, params=None):
        if params is None:
            params = OrderedDict(self.named_parameters())
        bias = params.get('bias', None)
        return F.linear(input, params['weight'], bias)
```

همانند 2MetaConv برای یک لایه خطی از نرون‌ها نیز MetaLinear به همان سبک بالا تعریف می‌کنیم .

حال از مدل های بالا یک بلاک کانولوشنال درست می‌کنیم به نام `conv_block` که با یک تابع آن را فراخوانی می‌کنیم .

```
def conv_block(in_channels, out_channels, **kwargs):
    return MetaSequential(OrderedDict([
        ('conv', MetaConv2d(in_channels, out_channels, **kwargs)),
        ('norm', nn.BatchNorm2d(out_channels, momentum=1.,
            track_running_stats=False)),
        ('relu', nn.ReLU()),
        ('pool', nn.MaxPool2d(2))
    ]))
```

شامل یک لایه conv یک نرمالیزشن و activition function relu و در نهایت یک MaxPool داریم .

سه لایه آخر همانطور که مشاهده می‌کنید از مدل اصلی ارث بری کرده و تنها لایه Conv هست که از Metaconv2d خود برای تعریف آن استفاده کرده ایم .

همانطور که که می‌بینید فلسفه کلاس Metal Sequential در اینجا خودش را نشان داده هست زیر ما بلاکی را تعریف کردیم که عملا برای پردازش forward آن بایستی بقیه قسمت‌ها را در نظر بگیریم و حساب کنیم و به صورت ساده خود مدل کار را جلو نمی‌بریم .

در نهایت می‌آییم و مطابق چیزی که تا حالا یاد گرفتیم می‌توانیم یک مدل ساده CNN به شکل زیر تعریف کنیم .

```
class MetaConvModel(MetaModule):

    def __init__(self, in_channels, out_features, hidden_size=64, feature_size=64):
        super(MetaConvModel, self).__init__()
        self.in_channels = in_channels
        self.out_features = out_features
        self.hidden_size = hidden_size
        self.feature_size = feature_size

        self.features = MetaSequential(OrderedDict([
            ('layer1', conv_block(in_channels, hidden_size, kernel_size=3,
                                  stride=1, padding=1, bias=True)),
            ('layer2', conv_block(hidden_size, hidden_size, kernel_size=3,
                                  stride=1, padding=1, bias=True)),
            ('layer3', conv_block(hidden_size, hidden_size, kernel_size=3,
                                  stride=1, padding=1, bias=True)),
            ('layer4', conv_block(hidden_size, hidden_size, kernel_size=3,
                                  stride=1, padding=1, bias=True))
        ]))

        self.classifier = MetaLinear(feature_size, out_features, bias=True)

    def forward(self, inputs, params=None):
        features = self.features(inputs, params=self.get_subdict(params, 'features'))
        features = features.view((features.size(0), -1))
        logits = self.classifier(features, params=self.get_subdict(params, 'classifier'))
        return logits
```

این مدل از چهار لایه تشکیل شده است که هر لایه شامل یک Conv_block که همانطور که در بالا دیدم شامل یک کانولوشن ، نرمالیزشن ، اکتیویشن فانکشن relu و در نهایت یک MaxPool می باشد ما چهار تا از این لایه را در کنار هم قرار می‌دهیم با Hidensize که مد نظر خودمان که برای تمامی لایه‌ها یکسان است و در نهایت باید از قبل حساب کنیم با توجه به تعداد این لایه‌ها میزان feature size ما چه مقدار خواهد شد و خوب در نهایت با این مقدار و مقدار تعداد کلاس‌های خود یک لایه classification را می‌سازیم .

همانند بالا می‌توانیم هرنوع مدل CNN که می‌خواهیم را به کمک MetaModule بسازیم و در نهایت هم forward آن را همانند قبل تعریف می کنیم با این تفاوت که یک فراخوانی get_subdict از مدل MetaModule برای دست یابی به پارامتر ها انجام می‌دهیم .

از مدل بالا استفاده کردم و مدل CNN اشاره شده در مقاله را پیاده سازی کردم .

مدل اشاره شده CNN در مقاله برای Mini Imagenet برابر بود با ۴ لایه Conv,bach,relu,maxPool همانند بالا و یک لایه برای classification که چهار لایه دارای عمق ۳۲ بودند .

```
def ModelConvMiniImagenet(out_features, hidden_size=32):
    return MetaConvModel(3, out_features, hidden_size=hidden_size,
                         feature_size=5 * 5 * hidden_size)
# def ModelConvMiniImagenet(out_features, hidden_size=64):
#     return MetaConvModel(3, out_features, hidden_size=hidden_size,
#                          feature_size=10 * 10 * hidden_size)
```

در نهایت مدل bais براساس تعداد کلاس‌های تسک خود را به کمک این فانکشن می‌ساختم که ورودی اول تعداد کلاس‌ها بود خوب hidden_size که به صورت عام ۳۲ ست شده بود .

بعد از این تعریف حال با این سه خط کد تمامی مدل‌های که در پایین نتایج آن را می‌بینید می‌سازم .

```
Fomaml = False
# scheduler = torch.optim.lr_scheduler.ExponentialLR
scheduler = None
history,model_5_way_shot_1_maml = train_F_maml(Epochs,4,5,1,10,save_epoch,Fomaml,scheduler)
```

در خط اول مشخص می کنم از روش Fomaml برای train استفاده کند مدل یا Maml.
و در نهایت اگر بخواهیم نرخ یادگیری بیرونی تغییر کند در روند آموزش scheduler برای مدل تعریف می کنیم و به کمک تابع train_F_maml که قبلا درباره آن صحبت کردیم مدل و تاریخچه یادگیری آن را بدست می‌آوریم و نمودارها را نیز براساس این تاریخچه‌ها ساخته‌ام که در ادامه می‌بینید . (تمامی تاریخچه‌ها و مدل‌ها به صورت فایل در کنار فایل کد قرار گرفته اند برای صحت سنجی شما)

برای ران کردن پروژه کافی است است فایل main.ipynp را در محیط colab ران کنید . سعی شده است تمامی کتابخانه‌های موردنظر import و دیتاست نیز با دستورات قرار داده شده دانلود شود و سپس مراحل به ترتیب انجام شود .
دیتا ست را برای کاهش لود بر روی dropbox ذخیره کردم و به کمک دستور wget دریافت می کردم .
برای احتیاط دستورات سیستم را کامنت کردم برای استفاده از این دستورات کافی ایست این دستورات را از حالت کامنت خارج کنید.

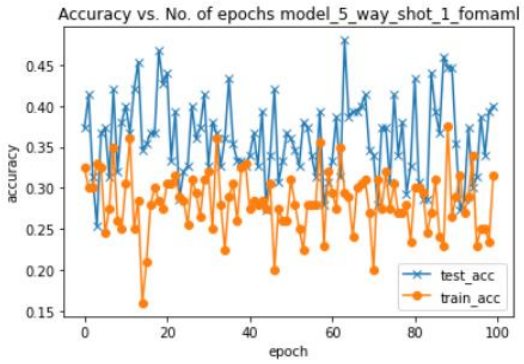
خروجی :(هر epoch در نمودار نشان دهنده ۲۰۰ iteration همانند خواست سوال است =>۲۰,۰۰۰ = ۱۰۰ epoch

نمودار قرمز رنگ آموزش و نمودار آبی رنگ تست است.

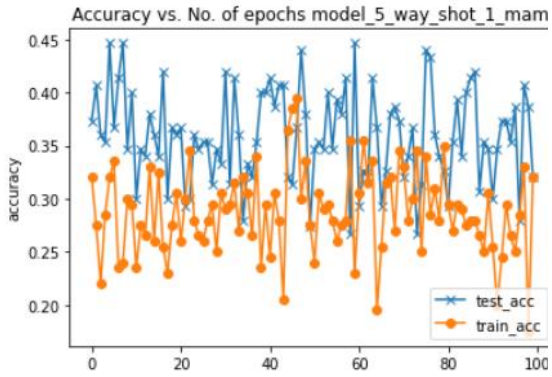
5 way one shot:

دقت

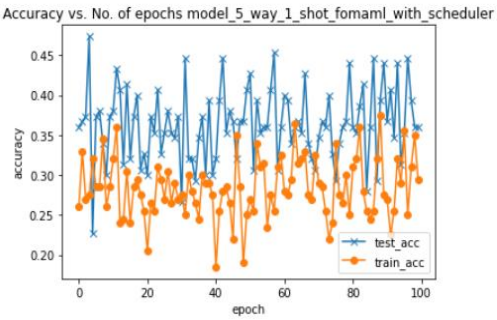
```
plot_accuracies(history3,'model_5_way_shot_1_fomaml')
```



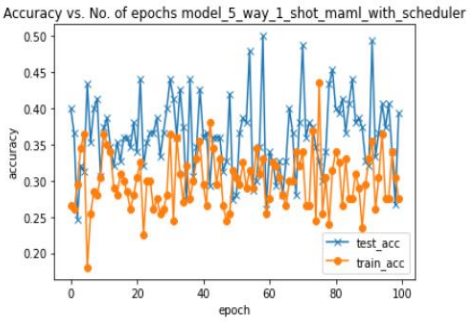

```
plot_accuracies(history,'model_5_way_shot_1_maml')
```




```
plot_accuracies(history7,'model_5_way_1_shot_fomaml_with_scheduler')
```




```
plot_accuracies(history5,'model_5_way_1_shot_maml_with_scheduler')
```

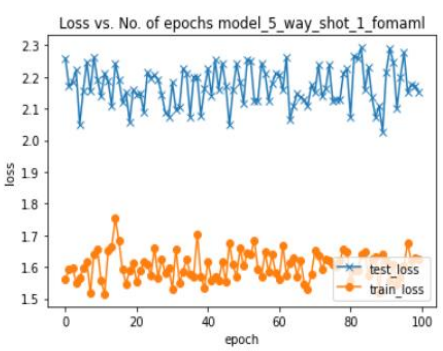


در اینجا نمودار دقت برای مدل مذکور 1 shot 5 way در ۲۰۰۰۰ epochs را مشاهده می‌نماییم و همانطور که مشاهده می‌کنیم مدل fomaml در تمامی حالات نسبت به نمودار maml از پیوستگی و تفاوت کمتری بین سقف و کف دقت در هر دوره‌ها را تجربه کرده است و نمودار حرکت هموارتر و کم نوسان‌تری نسبت به مدل maml دارد در هر دو مدل دقت در بعضی از مدل‌ها از ۴۵ درصد نیز گذر کرده است همچنین همانطور که می‌بینیم بیشینه دقت در مدل دارای scheduler برای تنظیم نرخ یادگیری بیرونی از دو مدل قبل بیشتر از است اما از طرفی تعداد مدل‌ها با دقت کمینه‌آن نیز بیشتر است، به عبارتی می‌توانیم انتظار داشته باشیم مدل با sheduler برای تنظیم نرخ یادگیری با افزایش تعداد epochها بتواند مدل را بهتر از دو مدل قبل به یک حالت پایدار و با دقت بالا برساند و در میانگین با این تعداد epoch احتمالا هنوز کارایی خودش را به خوبی نتواند نشان دهد . همانطور که مشاهده می‌کنیم کمینه میزان دقت در مدل fomaml نسبت به maml با افزایش epochها در حال پیشرفت بهتری هست.

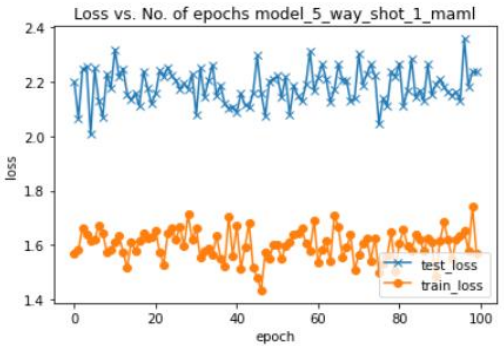
اینکه دقت داده test نیز از train بیشتر است هم خوب طبیعی است چونکه برای اینکه ماشین را به ازای تسک‌های این داده بسازیم هربار از 10 بار adaptation استفاده می‌کنیم به جای 5 بار adaptation پس می‌توانیم انتظار داشته باشیم که دقت بالاتری داشته باشد .

میزان خطا :

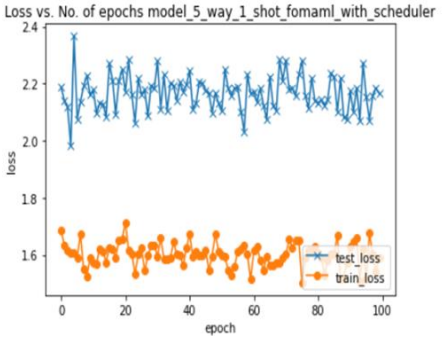
```
plot_losses(history3,'model_5_way_shot_1_fomaml')
```



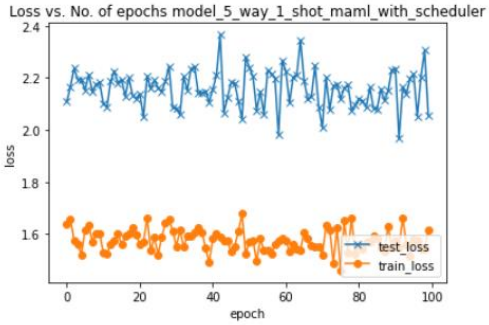

```
plot_losses(history,'model_5_way_shot_1_maml')
```




```
plot_losses(history7,'model_5_way_1_shot_fomaml_with_scheduler')
```



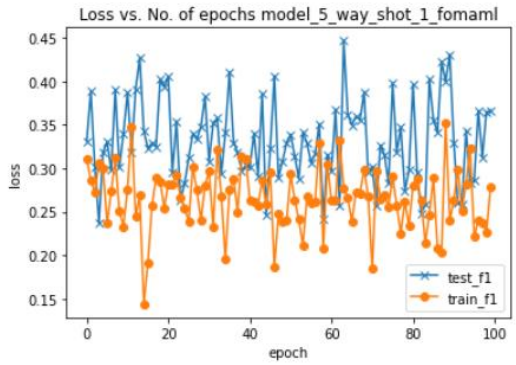

```
plot_losses(history5,'model_5_way_1_shot_maml_with_scheduler')
```



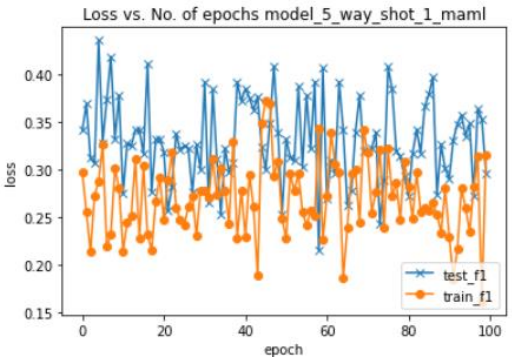
در بعضی از epochها همانطورکه مشاهده می‌کنیم نرخ خطای ماشین درحالت scheduler بیشتر از حالت بدون آن کاهش پیدا کرده‌است و قله‌های آن پایین‌تر هستند به طور کلی با این حال احتمالا اگر به آن فرصت بیشتری داده شود در کل به سمت بهبود بهتر مدل (خطای کمتر) حرکت کند.

میزان F1

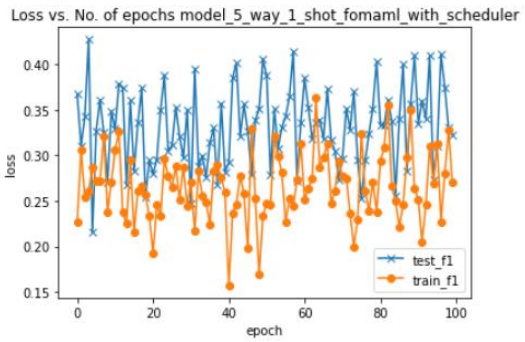
```
plot_f1(history3,'model_5_way_shot_1_fomaml')
```



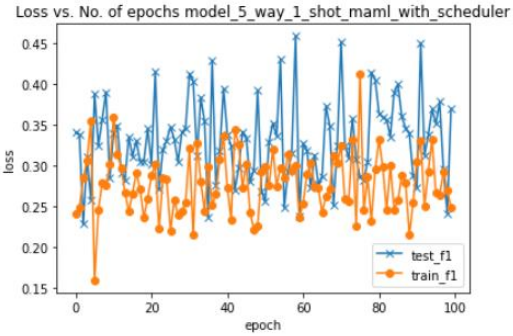

```
plot_f1(history,'model_5_way_shot_1_maml')
```




```
plot_f1(history7,'model_5_way_1_shot_fomaml_with_scheduler')
```




```
plot_f1(history5,'model_5_way_1_shot_maml_with_scheduler')
```

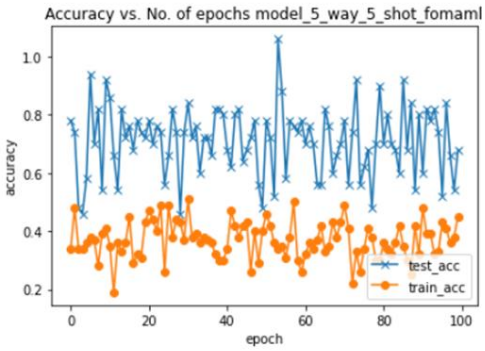


همانطور که مشاهده می‌کنیم دو مدل fomaml تصویر شده میزان قله‌های بیشتری نسبت به معیار 1F دارند و در جایگاه و وضعیت بهتری قرار دارند..

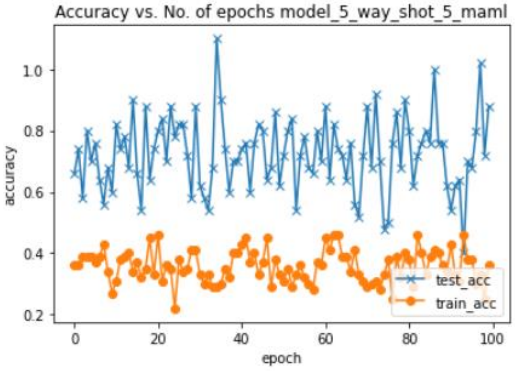
5 way 5 shot:

دقت

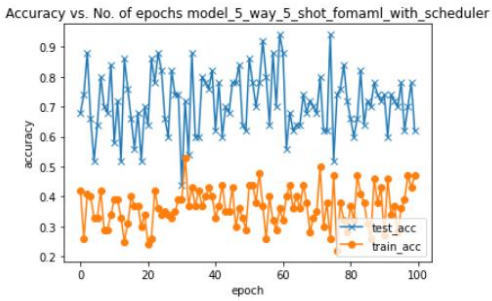
```
plot_accuracies(history4,'model_5_way_5_shot_fomaml')
```



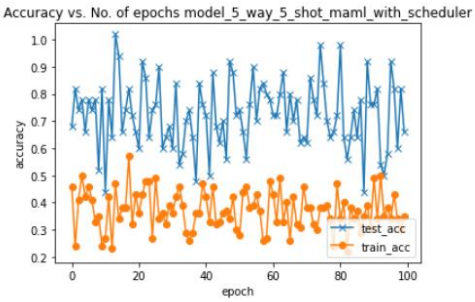

```
plot_accuracies(history2,'model_5_way_shot_5_maml')
```




```
plot_accuracies(history8,'model_5_way_5_shot_fomaml_with_scheduler')
```



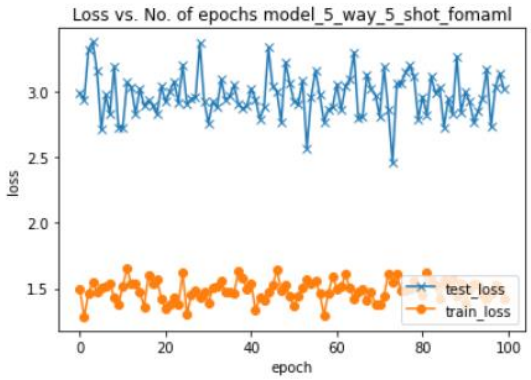

```
plot_accuracies(history6,'model_5_way_5_shot_maml_with_scheduler')
```



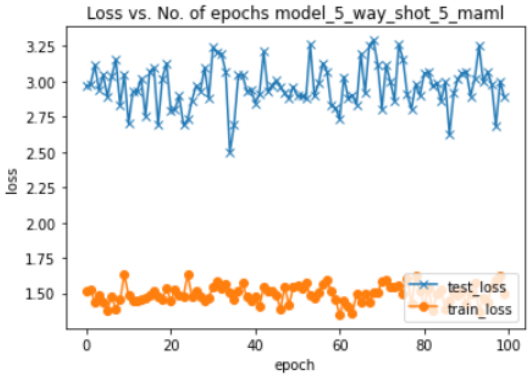
در مدل‌های 5 shot نسبت به one shot همانطور که مشاهده می‌کنیم دقت‌ها بسیار بالاتر است به‌گونه‌ای که مدل تست ما حتی به دقت صد در صد نیز در بعضی از موارد رسیده است . هرچند که خود مدل train نرخ یادگیری آن هنوز به صورت عمومی زیر ۵۰ درصد است اما میزان دقت بر روی بعضی از epochها روی داده تست فوق العاده است و در کل نسبت به One shot عملکرد بهتری دارد در اینجا نیز مدل fommalدامنه نوسان کمتری نسبت به maml دارد و باز مدل با scheduler از قله‌های بیشتری برخور دار هست اما به علت دامنه نوسان بالایی که دارد احتمالا به صورت میانگین از از حالت بدون scheduler ضعیف‌تر عمل کند اما باید دقت داشت که داده‌های train ما مرز ۵۰ درصد دقت را در این نوع پیاده سازی نسبت به حالت بدون آن، شکسته است .

میزان خطا:

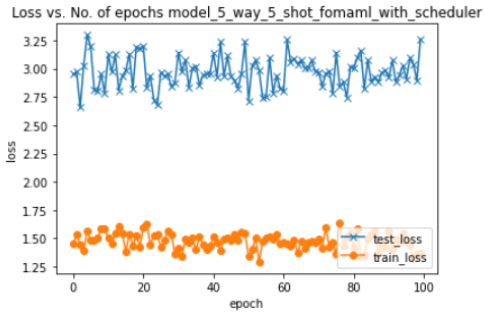
```
plot_losses(history4,'model_5_way_5_shot_fomaml')
```



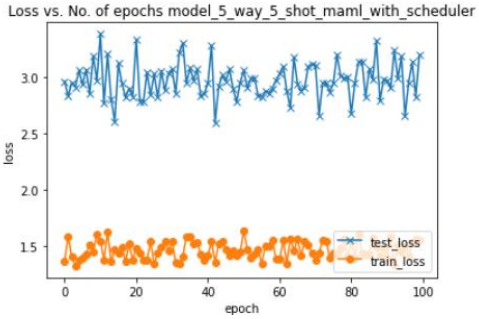

```
plot_losses(history2,'model_5_way_shot_5_maml')
```




```
plot_losses(history8,'model_5_way_5_shot_fomaml_with_scheduler')
```



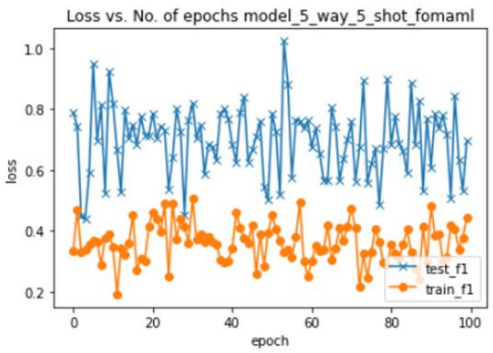

```
plot_losses(history6,'model_5_way_5_shot_maml_with_scheduler')
```



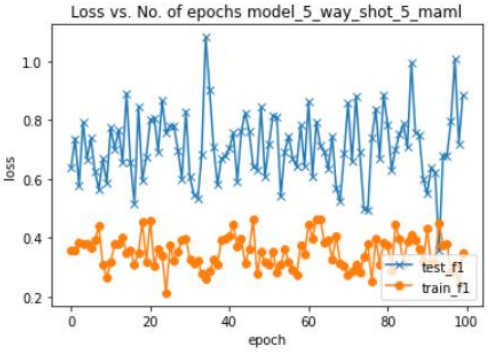
میزان خطاها در این چهار مدل به همدیگر نزدیک هستند و نمی‌توان به صورت دقیق درباره آنان بحث کرد .

میزان F1

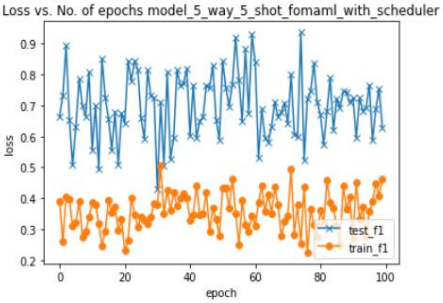
```
plot_f1(history4,'model_5_way_5_shot_fomaml')
```



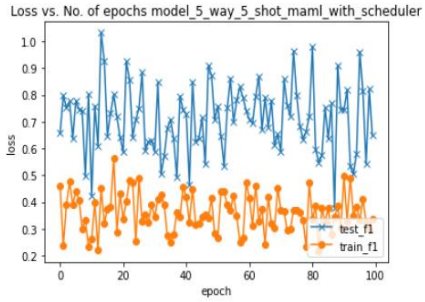

```
plot_f1(history2,'model_5_way_shot_5_maml')
```




```
plot_f1(history8,'model_5_way_5_shot_fomaml_with_scheduler')
```




```
plot_f1(history6,'model_5_way_5_shot_maml_with_scheduler')
```



همانطور که در درباره میزان دقت در one shot 5 way گفتم در اینجا نیز مدل fomaml بدون scheduler از دامنه نوسانی کمتری برخوردار هست ولی خوب مدل دارای scheduler داره قله‌های بیشتر و به همان نسبت دره‌های بیشتری است ولی به صورت کلی در۱۰۰۰۰ تعداد تکرار ما مدل بدون scheduler به صورت میانگین بهتر عمل کرده است شاید اگر تعداد epochها در اینجا بیشتر می‌شد مدل با scheduler در ادامه بهتر نیز عمل می‌کرد .

نتیجه‌نهایی :

برای بدست آوردن نتایج موجود بدست آمده در جدول زیر به کمک تابعی به نام test_F_maml به تعداد بارهای خواسته شده در متغیر repid برای مدل تابع evaluate را صدا زده با شرایطی ورودی و از حاصل‌های بدست آمده میانگین می گیریم.

جدول حاصل از repid = 10 بدست آمده است یعنی مدل ما بعد از ساخت که در ۲۰۰۰۰ epoch به انجام رسیده برای ارزیابی به اندازه ۱۰ بار مورد آزمون قرار گرفته و نتایج به این شکل ذخیره شده است.

	name	loss	acc	f1
0	model_5_way_shot_1_maml	2.028873	0.3550	0.332438
1	model_5_way_5_shot_maml	1.845626	0.4475	0.436394
2	model_5_way_1_shot_fomaml	1.999163	0.3565	0.334361
3	model_5_way_5_shot_fomaml	1.807084	0.4745	0.467698
4	model_5_way_1_shot_maml_with_scheduler	2.028186	0.3290	0.299164
5	model_5_way_5_shot_maml_with_scheduler	1.866307	0.4470	0.437437
6	model_5_way_1_shot_fomaml_with_scheduler	2.039116	0.3370	0.312835
7	model_5_way_5_shot_fomaml_with_scheduler	1.882448	0.4395	0.431333

همانطور که از جدول مشخص است بهترین دقت را در بین مدل‌ها، مدل 5 way 5 shot بدست آمده از روش fomaml است.

همانطور که می‌بینیم در تمامی موارد مدل fomaml از مدل متناظر maml خودش بهتر عمل کرده است و در جایگاه بالاتری قرار گرفته است .

برای آپدیت کردن نرخ یادگیری خارجی از torch.optim.lr_scheduler. با گامای 0.9 استفاده کردم و آن را آزمودم و از آنجا که خود نوع scheduler و میزان گامای آن یک فراپارمتر برای مدل محسوب می‌شود که باید بهینه‌ترین و بهترین میزان آن با آزمایش بدست بیاید، همانطور که می‌بینید در اینجا با این انتخابات برخلاف تصور که دنبال بهبود عملکرد کلی مدل بر روی داده‌های تست بودیم، جواب مثبت نگرفتیم و به دقت بالاتری نرسیدم، اما نمی‌توانیم با دیدن این نتیجه، نتیجه کلی درباره اینکه آپدیت کردن نرخ یادگیری در طول آزمایش باعث کاهش نرخ یادگیری می‌شود، برسیم و برای دیدن نتایج به روز رسانی نرخ یادگیری خارجی، می‌بایست گاماهای مختلف آزمایش و نتایج بررسی شوند

$$\text{Learning rate}(epoch) = \text{Initial learning rate} * (1 - \frac{\text{Decay rate}}{100})^{\text{epoch}}$$

تابع موردنظر همانند فرمول بالا نرخ، یادگیری حلقه بیرونی را به روزرسانی می کند .