Lecture_13_pytorch_hyperparameter_tuning_best_practices_code

October 11, 2024

PyTorch: Hyperparameter Tuning and Best Practices

1 0. Hyperparameter Tuning

For hyperparameter tuning, we will use the code at the end of the previous lecture's notebook.

2 1. The Adam Optimizer

The `torch.optim` contains many optimizers and so far, we have been using the `torch.optim.SGD` optimizer. Another extremely popular optimizer is known as Adam, or `torch.optim.Adam`.

If you want to use Adam, you only need to change `torch.optim.SGD` to `torch.optim.Adam` when you create the optimizer, and the rest of the code should be the same.

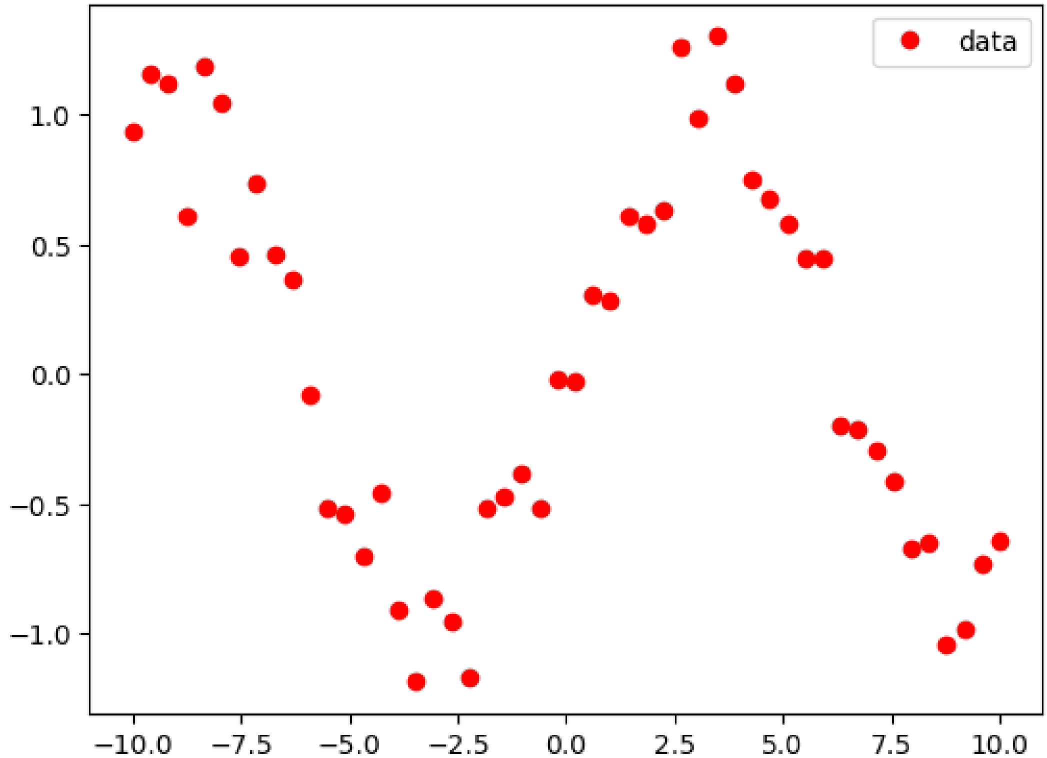
Below is the same nerual network fitting example from Lecture 18, but we now change the optimizer to Adam. The code will also save the training process as a GIF file and you can compare Adam and SGD.

[1]:
```
# Create Neural Network
import torch
from torch import nn
import numpy as np
from torch.utils.data import Dataset,DataLoader

class myMultiLayerPerceptron(nn.Module):
    def __init__(self,input_dim,output_dim):
        super().__init__()
        self.sequential = nn.Sequential(  # here we stack multiple layers␣
 ↪together
            nn.Linear(input_dim,20),
            nn.ReLU(),
            nn.Linear(20,20),
            nn.ReLU(),
            nn.Linear(20,20),
            nn.ReLU(),
            nn.Linear(20,20),
            nn.ReLU(),
```

```
            nn.Linear(20,output_dim)
        )
    def forward(self,x):
        y = self.sequential(x)
        return y
```

[2]:

```
# Create some simple synthetic data

import matplotlib.pyplot as plt

N_samples = 50
x = torch.linspace(-10,10,N_samples,dtype=torch.float)
x = x[:,None]
y = torch.sin(0.5*x) + np.random.randn(N_samples,1)*0.2


plt.plot(x,y,'ro')
plt.legend(['data'])


import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader

class MyDataset(Dataset):
    def __init__(self,x,y):
        self.x = x
        self.y = y

    def __len__(self):
        return self.x.shape[0]

    def __getitem__(self, idx):
        return (self.x[idx],self.y[idx])

mydataset = MyDataset(x,y) # generate a Dataset based on x,y

# Randomly split dataset into train and validate dataset
dataset_len = len(mydataset)
train_dataset_len = round(dataset_len*0.8)
validate_dataset_len = dataset_len - train_dataset_len
train_dataset,validate_dataset = torch.utils.data.
 ↪random_split(mydataset,[train_dataset_len, validate_dataset_len])
```



[3]:
```
# Training loops with Adam optimizer
import io
import imageio
from matplotlib.backends.backend_agg import FigureCanvasAgg as FigureCanvas
from matplotlib.figure import Figure

mymodel = myMultiLayerPerceptron(1,1) # creating a model instance with input␣
 ↪dimension 1

# Three hyper parameters for training
lr = .04
batch_size = 10
N_epochs = 160

# Create dataloaders for training and validation
train_dataloader = DataLoader(train_dataset, batch_size = batch_size, shuffle =␣
 ↪True)
validate_dataloader = DataLoader(validate_dataset,batch_size =␣
 ↪batch_size,shuffle = True)

# Create optimizer - choose between SGD or Adam
# optimizer = torch.optim.SGD(mymodel.parameters(), lr = lr)
```

```
optimizer = torch.optim.Adam(mymodel.parameters(), lr = lr)


frames = [] # This variable stores all images to be saved to the GIF file

losses = [] # training losses of each epoch
validate_losses = [] # validation losses of each epoch
losses_all = [] # training losses of each SGD iteration

gd_steps = 0
N_batches = len(train_dataloader)

for epoch in range(N_epochs):
    batch_loss = []
    for batch_id, (x_batch, y_batch) in enumerate(train_dataloader):
        gd_steps+=1
        # pass input data to get the prediction outputs by the current model
        prediction = mymodel(x_batch)

        # compare prediction and the actual output and compute the loss
        loss = torch.mean((prediction - y_batch)**2)

        # compute the gradient
        optimizer.zero_grad()
        loss.backward()

        # update parameters
        optimizer.step()


        # Generate visualization plots
        fig, ax = plt.subplots(nrows = 1, ncols = 3)
        canvas = FigureCanvas(fig)
        ax[0].plot(x,y,'ro')
        prediction_full = mymodel(x)
        ax[0].plot(x,prediction_full.detach(),linewidth = 2)
        ax[0].legend(['data','prediction of mymodel'],loc = 'upper left')
        ax[0].set_title(f"Batch size = {batch_size}, Learning rate = {lr},␣
↪Epoch #{epoch}, Batch #{batch_id}", fontsize = 20)
        ax[0].set_xlim((-10,10))
        ax[0].set_ylim((-2,2))
        losses_all.append(loss.detach().numpy())
        ax[1].plot(np.arange(gd_steps),np.array(losses_all).
↪squeeze(),linewidth=2 )
        ax[1].set_xlim((0,(N_epochs+1)*(N_batches)))
        ax[1].set_ylim((0,2))
        ax[1].set_title("Train loss per iteration", fontsize = 20)
```

```
        ax[1].set_xlabel("# of SGD Iterations", fontsize = 20)

        batch_loss.append(loss.detach().numpy())
        if epoch>0:
            ax[2].plot(np.arange(epoch),np.array(losses).squeeze(),linewidth=2, label = 'train loss' )
            ax[2].plot(np.arange(epoch),np.array(validate_losses).squeeze(),linewidth=2, label = 'validate loss')
            ax[2].legend(fontsize = 20)

        ax[2].set_xlim((0,N_epochs-1))
        ax[2].set_ylim((0,2))
        ax[2].set_title("Train/validate loss per epoch", fontsize = 20)
        ax[2].set_xlabel("# of Epochs", fontsize = 20)
        fig.set_size_inches(27,9)
        canvas.draw()       # draw the canvas, cache the renderer

        image = np.frombuffer(canvas.tostring_rgb(), dtype='uint8')

        image = image.reshape(fig.canvas.get_width_height()[::-1] + (3,))

        frames.append(image)
        plt.close(fig)

    # Calculate Validation Loss
    validate_batch_loss = []
    for x_batch, y_batch in validate_dataloader:
        # pass input data to get the prediction outputs by the current model
        prediction = mymodel(x_batch)

        # compare prediction and the actual output and compute the loss
        loss = torch.mean((prediction - y_batch)**2)
        validate_batch_loss.append(loss.detach())

    validate_losses.append( np.mean(np.array(validate_batch_loss)))
    losses.append(np.mean(np.array(batch_loss)))

print("Saving GIF file")
with imageio.get_writer("MLPADAM.gif", mode="I") as writer:
    for frame in frames:
        writer.append_data(frame)
```

```
Saving GIF file
```

3 2. The NSL-KDD Example: Using built-in loss functions, activations beyond ReLU, and best practices in training loops

The `torch.nn` libarary contains a lot of built-in layers for various neural architectures, including multi-layer perceptron, convolutional neural networks, etc. It also provides many built-in activation functions and loss functions.

Let's now use the NSL-KDD as an example, where we will build a neural network with a non-ReLU activation function, and train it using a built-in loss function. We will also summarize what is the best practice in writing training loops.

3.1 2.1 Prepare data

We are going to use the NSL-KDD dataset, and the following is some necessary preprocessing steps that we went through in previous lectures.

[4]:
```
import pyspark
from pyspark.sql import SparkSession, SQLContext
from pyspark.ml import Pipeline,Transformer
from pyspark.ml.feature import␣
↪Imputer,StandardScaler,StringIndexer,OneHotEncoder, VectorAssembler

from pyspark.sql.functions import *
from pyspark.sql.types import *
import numpy as np

col_names = ["duration","protocol_type","service","flag","src_bytes",
"dst_bytes","land","wrong_fragment","urgent","hot","num_failed_logins",
"logged_in","num_compromised","root_shell","su_attempted","num_root",
"num_file_creations","num_shells","num_access_files","num_outbound_cmds",
"is_host_login","is_guest_login","count","srv_count","serror_rate",
"srv_serror_rate","rerror_rate","srv_rerror_rate","same_srv_rate",
"diff_srv_rate","srv_diff_host_rate","dst_host_count","dst_host_srv_count",
"dst_host_same_srv_rate","dst_host_diff_srv_rate","dst_host_same_src_port_rate",
"dst_host_srv_diff_host_rate","dst_host_serror_rate","dst_host_srv_serror_rate",
"dst_host_rerror_rate","dst_host_srv_rerror_rate","class","difficulty"]

nominal_cols = ['protocol_type','service','flag']
binary_cols = ['land', 'logged_in', 'root_shell', 'su_attempted',␣
↪'is_host_login',
'is_guest_login']
continuous_cols = ['duration' ,'src_bytes', 'dst_bytes', 'wrong_fragment'␣
↪,'urgent', 'hot',
'num_failed_logins', 'num_compromised', 'num_root' ,'num_file_creations',
'num_shells', 'num_access_files', 'num_outbound_cmds', 'count' ,'srv_count',
'serror_rate', 'srv_serror_rate' ,'rerror_rate' ,'srv_rerror_rate',
'same_srv_rate', 'diff_srv_rate', 'srv_diff_host_rate' ,'dst_host_count',
'dst_host_srv_count' ,'dst_host_same_srv_rate' ,'dst_host_diff_srv_rate',
```

```
'dst_host_same_src_port_rate' ,'dst_host_srv_diff_host_rate',
'dst_host_serror_rate' ,'dst_host_srv_serror_rate', 'dst_host_rerror_rate',
'dst_host_srv_rerror_rate']

class OutcomeCreater(Transformer): # this defines a transformer that creates␣
 ↪the outcome column

    def __init__(self):
        super().__init__()

    def _transform(self, dataset):
        label_to_binary = udf(lambda name: 0.0 if name == 'normal' else 1.0)
        output_df = dataset.withColumn('outcome',␣
 ↪label_to_binary(col('class'))).drop("class")
        output_df = output_df.withColumn('outcome', col('outcome').
 ↪cast(DoubleType()))
        output_df = output_df.drop('difficulty')
        return output_df

class FeatureTypeCaster(Transformer): # this transformer will cast the columns␣
 ↪as appropriate types
    def __init__(self):
        super().__init__()

    def _transform(self, dataset):
        output_df = dataset
        for col_name in binary_cols + continuous_cols:
            output_df = output_df.withColumn(col_name,col(col_name).
 ↪cast(DoubleType()))

        return output_df
class ColumnDropper(Transformer): # this transformer drops unnecessary columns
    def __init__(self, columns_to_drop = None):
        super().__init__()
        self.columns_to_drop=columns_to_drop
    def _transform(self, dataset):
        output_df = dataset
        for col_name in self.columns_to_drop:
            output_df = output_df.drop(col_name)
        return output_df

def get_preprocess_pipeline():
    # Stage where columns are casted as appropriate types
    stage_typecaster = FeatureTypeCaster()

    # Stage where nominal columns are transformed to index columns using␣
 ↪StringIndexer
```

```
    nominal_id_cols = [x+"_index" for x in nominal_cols]
    nominal_onehot_cols = [x+"_encoded" for x in nominal_cols]
    stage_nominal_indexer = StringIndexer(inputCols = nominal_cols, outputCols␣
↪= nominal_id_cols )

    # Stage where the index columns are further transformed using OneHotEncoder
    stage_nominal_onehot_encoder = OneHotEncoder(inputCols=nominal_id_cols,␣
↪outputCols=nominal_onehot_cols)

    # Stage where all relevant features are assembled into a vector (and␣
↪dropping a few)
    feature_cols = continuous_cols+binary_cols+nominal_onehot_cols
    corelated_cols_to_remove =␣
↪["dst_host_serror_rate","srv_serror_rate","dst_host_srv_serror_rate",
                     ␣
↪"srv_rerror_rate","dst_host_rerror_rate","dst_host_srv_rerror_rate"]
    for col_name in corelated_cols_to_remove:
        feature_cols.remove(col_name)
    stage_vector_assembler = VectorAssembler(inputCols=feature_cols,␣
↪outputCol="vectorized_features")

    # Stage where we scale the columns
    stage_scaler = StandardScaler(inputCol= 'vectorized_features', outputCol=␣
↪'features')


    # Stage for creating the outcome column representing whether there is␣
↪attack
    stage_outcome = OutcomeCreater()

    # Removing all unnecessary columbs, only keeping the 'features' and␣
↪'outcome' columns
    stage_column_dropper = ColumnDropper(columns_to_drop =␣
↪nominal_cols+nominal_id_cols+
        nominal_onehot_cols+ binary_cols + continuous_cols +␣
↪['vectorized_features'])
    # Connect the columns into a pipeline
    pipeline =␣
↪Pipeline(stages=[stage_typecaster,stage_nominal_indexer,stage_nominal_onehot_encoder,
        stage_vector_assembler,stage_scaler,stage_outcome,stage_column_dropper])
    return pipeline

# if you installed Spark on windows,
# you may need findspark and need to initialize it prior to being able to use␣
↪pyspark
# Also, you may need to initialize SparkContext yourself.
```

```python
# Uncomment the following lines if you are using Windows!
#import findspark
#findspark.init()
#findspark.find()

spark = SparkSession.builder \
    .master("local[*]") \
    .appName("GenericAppName") \
    .getOrCreate()

nslkdd_raw = spark.read.csv('./NSL-KDD/KDDTrain+.txt',header=False).
 ↪toDF(*col_names)
nslkdd_test_raw = spark.read.csv('./NSL-KDD/KDDTest+.txt',header=False).
 ↪toDF(*col_names)

preprocess_pipeline = get_preprocess_pipeline()
preprocess_pipeline_model = preprocess_pipeline.fit(nslkdd_raw)

nslkdd_df = preprocess_pipeline_model.transform(nslkdd_raw)
nslkdd_df_test = preprocess_pipeline_model.transform(nslkdd_test_raw)


to_array = udf(lambda v: v.toArray().tolist(), ArrayType(FloatType()))

nslkdd_df_train = nslkdd_df
nslkdd_df_validate,nslkdd_df_test = nslkdd_df_test.randomSplit([0.5,0.5])

nslkdd_df_train_pandas = nslkdd_df_train.withColumn('features',␣
 ↪to_array('features')).toPandas()
nslkdd_df_validate_pandas = nslkdd_df_validate.withColumn('features',␣
 ↪to_array('features')).toPandas()
nslkdd_df_test_pandas = nslkdd_df_test.withColumn('features',␣
 ↪to_array('features')).toPandas()
```

```
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use
setLogLevel(newLevel).
23/10/29 10:56:36 WARN NativeCodeLoader: Unable to load native-hadoop library
for your platform… using builtin-java classes where applicable
23/10/29 10:56:52 WARN SparkStringUtils: Truncated the string representation of
a plan since it was too large. This behavior can be adjusted by setting
'spark.sql.debug.maxToStringFields'.
```

To convert the dataframes to tensors, a convenient function is `torch.from_numpy`, which converts numpy arrays to torch tensors.

[5]:
```
# Converting the pandas DataFrame to tensors
# Note we are using 3 data sets train, validate, test
import torch
from torch import nn
from torch.utils.data import Dataset, DataLoader

x_train = torch.from_numpy(np.array(nslkdd_df_train_pandas['features'].values.
 ↪tolist(),np.float32))
y_train = torch.from_numpy(np.array(nslkdd_df_train_pandas['outcome'].values.
 ↪tolist(),np.int64))

x_validate = torch.from_numpy(np.array(nslkdd_df_validate_pandas['features'].
 ↪values.tolist(),np.float32))
y_validate = torch.from_numpy(np.array(nslkdd_df_validate_pandas['outcome'].
 ↪values.tolist(),np.int64))

x_test = torch.from_numpy(np.array(nslkdd_df_test_pandas['features'].values.
 ↪tolist(),np.float32))
y_test = torch.from_numpy(np.array(nslkdd_df_test_pandas['outcome'].values.
 ↪tolist(),np.int64))
```

After obtaining the data as torch tensors, we further wrap them into a Dataset using the `MyDataset` class we created in lecture 17/18.

[6]:
```
# Defining MyDataset Class
class MyDataset(Dataset):
    def __init__(self,x,y):
        self.x = x
        self.y = y

    def __len__(self):
        return self.x.shape[0]

    def __getitem__(self, idx):
        return (self.x[idx],self.y[idx])

# Turning the data from tensors to datasets
train_dataset = MyDataset(x_train,y_train)
validate_dataset = MyDataset(x_validate,y_validate)
test_dataset = MyDataset(x_test,y_test)
```

3.2 2.2 Building Neural Network

We now create a neural network with tanh activation. Notice the change in the code compared with before.

[7]:
```
from torch import nn

class myMultiLayerPerceptron_TahnActivation(nn.Module):
    def __init__(self,input_dim,output_dim):
        super().__init__()
        self.sequential = nn.Sequential(  # here we stack multiple layers␣
 ↪together
            nn.Linear(input_dim,20),
            nn.Tanh(), # Using Tanh activation!
            nn.Linear(20,20),
            nn.Tanh(),
            nn.Linear(20,20),
            nn.Tanh(),
            nn.Linear(20,20),
            nn.Tanh(),
            nn.Linear(20,output_dim)
        )
    def forward(self,x):
        y = self.sequential(x)
        return y
```

[8]:
```
mymodel = myMultiLayerPerceptron_TahnActivation(x_train.shape[1],2) # creating␣
 ↪a model instance with input dimension 1 and output dimension 1

print(mymodel)
```

```
myMultiLayerPerceptron_TahnActivation(
  (sequential): Sequential(
    (0): Linear(in_features=113, out_features=20, bias=True)
    (1): Tanh()
    (2): Linear(in_features=20, out_features=20, bias=True)
    (3): Tanh()
    (4): Linear(in_features=20, out_features=20, bias=True)
    (5): Tanh()
    (6): Linear(in_features=20, out_features=20, bias=True)
    (7): Tanh()
    (8): Linear(in_features=20, out_features=2, bias=True)
  )
)
```

3.3 2.3 Training loop using built-in loss function and best practices

Now let's write the training loop. In the code, notice that we use a built-in loss function `nn.CrossEntropyLoss()`.

Also, we would like to apply some best practices in writing the training loops. These best practices include:

Before the training loop starts: - Put all training hyper-parameters in a single place. - Set `shuffle = True` in DataLoader.

During the training loop: - Include a validation loop in each epoch - Calculate and record the loss/metrics for train/validate for each epoch - Print out the progress for each epoch, including the epoch number, train/validate loss, train/validate metrics - When you encouter the best model so far (measured by validate loss/metrics), save the model.

After the training loop: - Plot the train/validate loss/metrics accross different epochs, and use this as an reference to tune training parameters - Load back the best model saved during the training process.

[9]:
```
mymodel = myMultiLayerPerceptron_TahnActivation(x_train.shape[1],2) # creating␣
↪a model instance with input dimension 1 and output dimension 1

# Three hyper parameters for training
lr = .005
batch_size = 64
N_epochs = 10

# Create loss function
loss_fun = nn.CrossEntropyLoss()

# Create dataloaders for training and validation
train_dataloader = DataLoader(train_dataset, batch_size = batch_size, shuffle =␣
↪True)
validate_dataloader = DataLoader(validate_dataset,batch_size =␣
↪batch_size,shuffle = True)

# Create optimizer
optimizer = torch.optim.Adam(mymodel.parameters(), lr = lr) # this line creates␣
↪a optimizer, and we tell optimizer we are optimizing the parameters in␣
↪mymodel


losses = [] # training losses of each epoch
accuracies = [] # training accuracies of each epoch

validate_losses = [] # validation losses of each epoch
validate_accuracies = [] # validation accuracies of each epoch

current_best_accuracy = 0.0

for epoch in range(N_epochs):
    # Train loop
    batch_loss = [] # keep a list of losses for different batches in this epoch
    batch_accuracy = [] # keep a list of accuracies for different batches in␣
↪this epoch
```

```
    for x_batch, y_batch in train_dataloader:
        # pass input data to get the prediction outputs by the current model
        prediction_score = mymodel(x_batch)

        # compute the cross entropy loss. Note that the first input to the␣
↪loss_func should be the predicted scores (not probabilities), and the second␣
↪input should be class labels as integers
        loss = loss_fun(prediction_score,y_batch)

        # compute the gradient
        optimizer.zero_grad()
        loss.backward()

        # update parameters with optimizer step
        optimizer.step()

        # append the loss of this batch to the batch_loss list
        batch_loss.append(loss.detach().numpy())

        # You can also compute other metrics (accuracy) for this batch here
        prediction_label = torch.argmax(prediction_score.detach(),dim=1).numpy()
        batch_accuracy.append( np.sum(prediction_label == y_batch.numpy())/
↪x_batch.shape[0])


    # Validation loop
    validate_batch_loss = [] # keep a list of losses for different validate␣
↪batches in this epoch
    validate_batch_accuracy = [] # same for the accuracy
    for x_batch, y_batch in validate_dataloader:
        # pass input data to get the prediction outputs by the current model
        prediction_score = mymodel(x_batch)

        # compare prediction and the actual output and compute the loss
        loss = loss_fun(prediction_score,y_batch)

        # append the loss of this batch to the validate_batch_loss list
        validate_batch_loss.append(loss.detach())

        # You can also compute other metrics (like accuracy) for this batch here
        prediction_label = torch.argmax(prediction_score.detach(),dim=1).numpy()
        validate_batch_accuracy.append( np.sum(prediction_label == y_batch.
↪numpy())/x_batch.shape[0])
```

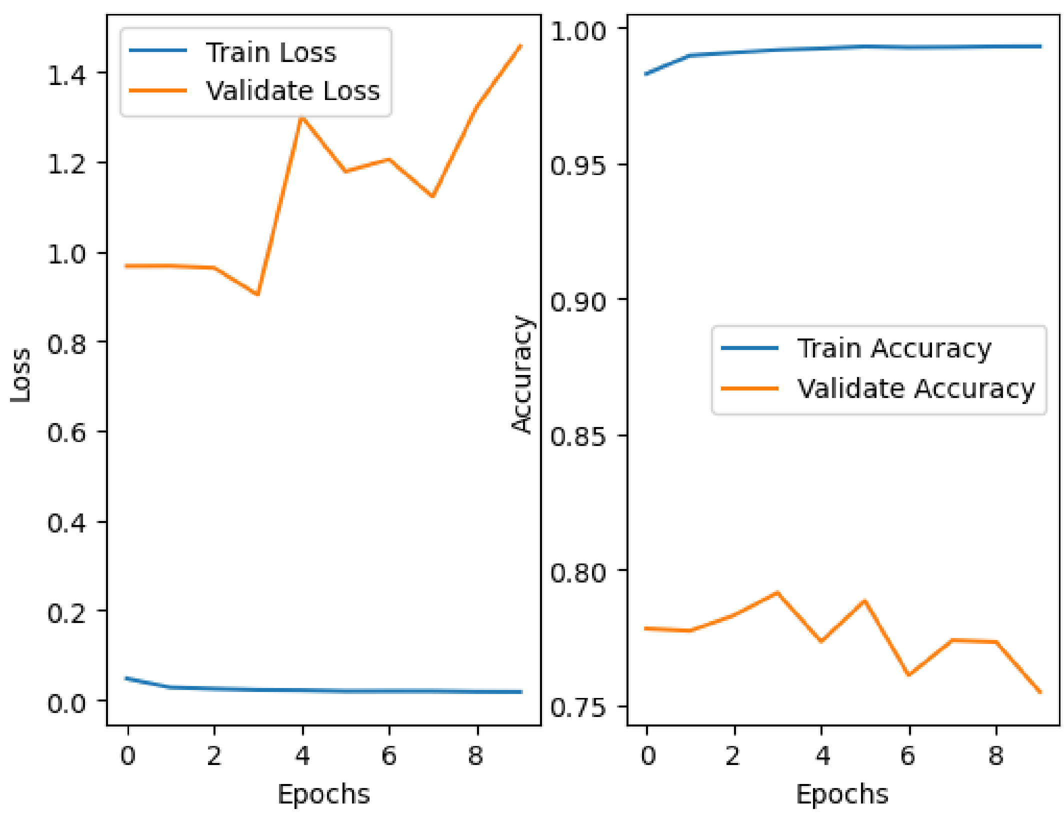
```
    # calculate the average train loss and validate loss in this epoch and␣
↪record them
    losses.append(np.mean(np.array(batch_loss)))
    validate_losses.append( np.mean(np.array(validate_batch_loss)))
    # You can also compute other metrics for this epoch here
    accuracies.append(np.mean(np.array(batch_accuracy)))
    validate_accuracies.append(np.mean(np.array(validate_batch_accuracy)))

    # Printing
    print(f"Epoch =␣
↪{epoch},train_loss={losses[-1]},validate_loss={validate_losses[-1]}")
    print(f"Train accuracy = {np.round(accuracies[-1]*100,2)}%, validate␣
↪accuracy = {np.round(validate_accuracies[-1]*100,2)}% ")

    # If the validate metric of this epoch is the best so far, save the model
    if validate_accuracies[-1]>current_best_accuracy:
        print("Current epoch is the best so far. Saving model...")
        torch.save(mymodel.state_dict(),'current_best_model')
        current_best_accuracy = validate_accuracies[-1]
```

```
Epoch = 0,train_loss=0.047911155968904495,validate_loss=0.9674395322799683
Train accuracy = 98.31%, validate accuracy = 77.82%
Current epoch is the best so far. Saving model…
Epoch = 1,train_loss=0.027876876294612885,validate_loss=0.967867374420166
Train accuracy = 98.97%, validate accuracy = 77.74%
Epoch = 2,train_loss=0.02473035827279091,validate_loss=0.9636149406433105
Train accuracy = 99.08%, validate accuracy = 78.31%
Current epoch is the best so far. Saving model…
Epoch = 3,train_loss=0.022321462631225586,validate_loss=0.9033389091491699
Train accuracy = 99.18%, validate accuracy = 79.15%
Current epoch is the best so far. Saving model…
Epoch = 4,train_loss=0.021562723442912102,validate_loss=1.3016053438186646
Train accuracy = 99.24%, validate accuracy = 77.35%
Epoch = 5,train_loss=0.019674159586429596,validate_loss=1.1784522533416748
Train accuracy = 99.31%, validate accuracy = 78.86%
Epoch = 6,train_loss=0.01978604681789875,validate_loss=1.2050907611846924
Train accuracy = 99.28%, validate accuracy = 76.1%
Epoch = 7,train_loss=0.0197481457144022,validate_loss=1.1221219301223755
Train accuracy = 99.29%, validate accuracy = 77.39%
Epoch = 8,train_loss=0.018451200798153877,validate_loss=1.321496605873108
Train accuracy = 99.31%, validate accuracy = 77.33%
Epoch = 9,train_loss=0.018177669495344162,validate_loss=1.4574381113052368
Train accuracy = 99.32%, validate accuracy = 75.49%
```

[10]:
```
# Plot train/validate loss and metrics across different epochs
from matplotlib import pyplot as plt
fig,axes = plt.subplots(nrows=1,ncols=2)
```

```
axes[0].plot(range(N_epochs),losses,label='Train Loss')
axes[0].plot(range(N_epochs),validate_losses,label='Validate Loss')
axes[0].set_xlabel("Epochs")
axes[0].set_ylabel("Loss")
axes[0].legend()
axes[1].plot(range(N_epochs),accuracies,label='Train Accuracy')
axes[1].plot(range(N_epochs),validate_accuracies,label='Validate Accuracy')
axes[1].set_xlabel("Epochs")
axes[1].set_ylabel("Accuracy")

axes[1].legend()
```

[10]: <matplotlib.legend.Legend at 0x7f906c20b8e0>



3.3.1 Load the best model back and conduct evaluation

[11]:
```
# create a new model with the same input-output dimension as before
mybestmodel = myMultiLayerPerceptron_TahnActivation(x_train.shape[1],2)

# load the "state_dict" from file into the new model
```

```
mybestmodel.load_state_dict(torch.load("current_best_model"))

# conduct testing via a test loop
test_dataloader = DataLoader(test_dataset,batch_size = batch_size,shuffle =␣
 ↪True)
test_batch_accuracy = []
for x_batch, y_batch in test_dataloader:
    # pass input data to get the prediction outputs
    prediction_score = mybestmodel(x_batch)

    # Compute metrics (like accuracy) for this batch here
    prediction_label = torch.argmax(prediction_score.detach(),dim=1).numpy()
    test_batch_accuracy.append( np.sum(prediction_label == y_batch.numpy())/
 ↪x_batch.shape[0])

# compute the mean accuracy across all batches
test_accuracy = np.mean(np.array(test_batch_accuracy))

print(f"Test accuracy = {np.round(test_accuracy*100,2)}%")
```

```
Test accuracy = 75.49%
```