Intro to ML - CS 419

Lecture 22: Deep Learning Tutorial

13 april 2023

Lecturer: Abir De *Scribe: Abhishek, Tanmay, Sunandinee & Naman*

1 Introduction

In this tutorial, we will

1. Train a dummy neural network on a classification dataset and learn things like: using dataloaders, learning rates, checkpointing the best model, training followed by inferencing, early stopping
2. Experiment with different hyperparameters that may increase/decrease the accuracy score:
   (a) Effect of different batch sizes on model convergence
   (b) Effect of different learning rates on task loss/accuracy
   (c) Effect of different activation functions on task loss/accuracy

2 Train a dummy neural network on a classification dataset

We first installed transformers and imported all important libraries. Then we are loading mnist dataset using following code snippet.

```
# loading the mnist dataset, train and test are datasets containing tensors
mnist_train = datasets.MNIST('data', train = True, download = True, transform=transforms.ToTensor())
mnist_test = datasets.MNIST('data', train = False, download = True, transform=transforms.ToTensor())
```

The first line loads the training set of the MNIST dataset and stores it in a variable named mnist_train, and the second line loads the test set and stores it in a variable named mnist_test. Both mnist_train and mnist_test are PyTorch datasets containing tensors.

```
print(mnist_train)
print(mnist_test)
print("Image tensor shape {}".format(list(mnist_train)[0][0].shape))
print(type(list(mnist_train)[0][0]))
```

Here the third line prints out the shape of the first image tensor in the mnist_train dataset. It does this by converting the first sample in the dataset to a list and then accessing the first element of that list, which contains the tensor representing the first image. The shape of the tensor is printed out using the shape attribute of the tensor. This line provides information about the size of the image tensor, which is (1, 28, 28) representing that each image is a grayscale image of size 28x28 pixels. The fourth line provides information about the data type of the tensor, which is a PyTorch tensor.

```
1 evens = list(range(0, len(mnist_train), 10))
2 odds = list(range(1, len(mnist_test), 10))
3 mnist_train = torch.utils.data.Subset(mnist_train, evens)
4 mnist_test = torch.utils.data.Subset(mnist_test, odds)
5 print("Final train and test sizes are {}, {}".format(len(mnist_train), len(
      mnist_test)))
```

Here, we select only a few samples for our training to have a lesser training time.
We are creating lists called evens and odds containing the indices of every 10th sample in the MNIST training set and test set respectively. We use torch.data.subset to select a subset of the mnist data
Line 3 creates a new subset of the MNIST training dataset called mnist_train, containing only the samples whose indices are listed in the evens list. Similiar working is of line 4.
Final train and test sizes are 6000, 1000

```
def set_seed(args):
    random.seed(args["seed"])
    np.random.seed(args["seed"])
    torch.manual_seed(args["seed"])
    torch.cuda.manual_seed_all(args["seed"])
```

Here we are defining a new function setseed. The function first sets the seed for the built-in "random" module using the value of "seed" key from the "args" dictionary. This ensures that any random number generation performed by the "random" module is reproducible, i.e., given the same seed, the module will generate the same sequence of random numbers.

Next, it sets the seed for the NumPy random number generator using the same seed value. This ensures that any random number generation performed by NumPy is also reproducible.

Then, it sets the seed for the PyTorch random number generator using the same seed value. This ensures that any random number generation performed by PyTorch is also reproducible.

Finally, it sets the seed for all CUDA devices using the same seed value.

```
args = {"seed": 42}
device = torch.device("cpu")
args["device"] = device
print(args["device"])

set_seed(args)
```

Here dictionary args is created with "seed" key of value 42. The device is set to the CPU. Then new key devices is added to args. Then finally setseed function is called.

```
class Net(nn.Module):

    def __init__(self, args):
        super(Net, self).__init__()
```

```
        self.fc1 = nn.Linear(28*28, 80)
        self.fc2 = nn.Linear(80, 30)
        self.fc3 = nn.Linear(30, 10)

    def forward(self, x):
        x = x.view(-1, 28*28)
        x = args["activation"](self.fc1(x))
        x = args["activation"](self.fc2(x))
        x = self.fc3(x)
        return x
```

Here Net defines a simple fully connected neural network for classifying images of handwritten digits from the MNIST dataset. The init method defines the architecture of the network by creating three linear layers (fc1, fc2, and fc3) with different input and output sizes.

The forward method defines how input data is passed through the network during the forward pass. The input x is first flattened into a 1D tensor using the view method, then passed through the fc1 layer, followed by an activation function specified in the args dictionary (e.g. nn.ReLU() or nn.Tanh()). The output of fc1 is then passed through fc2 and another activation function, followed by the final fc3 layer to produce a 10-dimensional output tensor representing the logits for each class.

```
# training loop
def train(args, train_dataset, val_dataset, model):

    # Prepare train data
    train_sampler = RandomSampler(train_dataset)  # random sampling of training data

    train_dataloader = DataLoader(
        train_dataset, sampler=train_sampler, batch_size=args["train_batch_size"])
    train_batch_size = args["train_batch_size"]

    t_total = len(train_dataloader) * args["num_train_epochs"]
    optimizer = args["optimizer"](model.parameters(), lr=args["learning_rate"], eps=args["adam_epsilon"])

    # explain what is learning rate warmup
    scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=t_total // 10, num_training_steps=t_total)
    criterion = nn.CrossEntropyLoss()  # defining the loss function

    # Train!
    print("***** Running training *****")
    print("  Num examples = ", len(train_dataset))
    print("  Num Epochs = ", args["num_train_epochs"])
    print("  Instantaneous batch size per GPU = ", train_batch_size)

    global_step = 0
    train_losses, val_losses = [], []
```

```
    train_acc, val_acc = [], []
    tr_loss, logging_loss = 0.0, 0.0
    model.zero_grad()

    train_iterator = trange(int(args["num_train_epochs"]), desc="Epoch")

    best_f1_score = 0
    if not os.path.exists(args["output_dir"]):
        os.makedirs(args["output_dir"])

    patience = 3
    last_best_epoch = -1


    for epoch in train_iterator:
        epoch_iterator = tqdm(train_dataloader, desc="Iteration")

        for step, batch in enumerate(epoch_iterator):
            model.train()

            batch = tuple(t.to(args["device"]) for t in batch)  # bringing the examples on same device as the model
            input_, labels_ = batch
            outputs = model(input_)

            loss = criterion(outputs, labels_)

            loss.backward()

            # gradient clipping
            torch.nn.utils.clip_grad_norm_(
                model.parameters(), args["max_grad_norm"])

            tr_loss += loss.item()
            optimizer.step()
            scheduler.step()
            model.zero_grad()
            optimizer.zero_grad()
            global_step += 1

        print("Train loss: {}".format(tr_loss/global_step))
        train_losses.append(tr_loss/global_step)

        # get train accuracy
        print("Train accuracy stats: ")
        results = evaluate(args, train_dataset, model)
        print("Train accuracy: {}".format(results["acc"]))
        train_acc.append(results["acc"])

        # Recording validation f1 scores
        results = evaluate(args, val_dataset, model)
```

```python
        print("Validation accuracy: {}".format(results["acc"]))
        print("Validation loss: {}".format(results["eval_loss"]))

        val_losses.append(results["eval_loss"])
        val_acc.append(results["acc"])

        if results.get('f1') > best_f1_score and args["save_steps"] > 0:
            best_f1_score = results.get('f1')
            model_to_save = model.module if hasattr(model, "module") else model
            torch.save(model_to_save.state_dict(), args["output_dir"] + "clssnn.pth")
            torch.save(args, os.path.join(args["output_dir"], "training_args.bin"))
            last_best_epoch = epoch
            print("Last best epoch is {}".format(last_best_epoch))
        elif epoch - last_best_epoch > patience:
            print("Early stopped at epoch {}".format(epoch))
            break

    return train_losses, train_acc, val_losses, val_acc

def evaluate(args, val_dataset, model):

    eval_sampler = SequentialSampler(val_dataset)
    eval_dataloader = DataLoader(
        val_dataset, sampler=eval_sampler, batch_size=args["eval_batch_size"])

    results = {}
    criterion = nn.CrossEntropyLoss()

    print("  Num examples = ", len(val_dataset))
    print("  Batch size = ", args["eval_batch_size"])
    eval_loss = 0.0
    nb_eval_steps = 0
    preds = None
    out_label_ids = None

    for batch in tqdm(eval_dataloader, desc="Evaluating"):
        model.eval()
        batch = tuple(t.to(args["device"]) for t in batch)

        with torch.no_grad():
            inputs, labels_ = batch

            outputs = model(inputs)  # forward pass
            logits = outputs

            loss = criterion(outputs, labels_)
            eval_loss += loss.mean().item()

```

```
        nb_eval_steps += 1

        if preds is None:
            preds = logits.detach().cpu().numpy()
            out_label_ids = labels_.detach().cpu().numpy()
        else:
            preds = np.append(preds, logits.detach().cpu().numpy(), axis=0)
            out_label_ids = np.append(
                out_label_ids, labels_.detach().cpu().numpy(), axis=0)

    eval_loss = eval_loss / nb_eval_steps
    preds = np.argmax(preds, axis=1)
    result = acc_and_f1(preds, out_label_ids)
    results.update(result)
    results["eval_loss"] = eval_loss

    return results


def simple_accuracy(preds, labels):
    return (preds == labels).mean()

def acc_and_f1(preds, labels):
    acc = simple_accuracy(preds, labels)
    f1 = f1_score(y_true=labels, y_pred=preds, average='weighted')
    precision = precision_score(
        y_true=labels, y_pred=preds, average='weighted')
    recall = recall_score(y_true=labels, y_pred=preds, average='weighted')

    return{
        "acc": acc,
        "f1": f1,
        "acc_and_f1": (acc + f1) / 2,
        "precision": precision,
        "recall": recall
    }
```

This code defines a function called train that is responsible for training a deep learning model. The function takes several arguments including args which is a dictionary of various hyperparameters, train_dataset which is the training dataset, val_dataset which is the validation dataset, and model which is the deep learning model being trained.

The first step in the train function is to prepare the training data. This involves creating a RandomSampler object to randomly sample the training data, creating a DataLoader object to load the data in batches, and setting the batch size. The total number of training steps (t_total) is calculated based on the number of epochs and the number of steps per epoch. The optimizer and learning rate scheduler are also created.

The train function then enters a loop that iterates over the specified number of epochs. Within each epoch, the function iterates over the batches of data in the train_dataloader. For each batch, the model is set to training mode (model.train()) and the input data and labels are loaded onto the same device as the model using the to() method. The model is then run on the input data (outputs = model(input_)) and the loss is calculated using a cross-entropy loss function. The loss is then back-propagated through the model (loss.backward()) and the gradients are clipped to prevent them from becoming too large using torch.nn.utils.clip_grad_norm_(). The optimizer is then updated (optimizer.step()) and the gradients and loss are reset (model.zero_grad() and optimizer.zero_grad()).

After each epoch, the function calculates and prints the average training loss and accuracy (train_loss and results["acc"]), and the average validation loss and accuracy (results["eval_loss"] and results["acc"]). The function also saves the best model based on the validation accuracy and stops training early if the model does not improve for a specified number of epochs.

The evaluate function is called within the train function to evaluate the model on the validation dataset. This function is similar to the training loop, but it does not involve backpropagation or updating the model parameters. The function loads the data onto the device, runs the model on the input data, calculates the loss and accuracy, and returns the results.

Overall, the train function implements a standard training loop for a deep learning model with cross-entropy loss and gradient descent optimization. The function also includes early stopping and model checkpointing to prevent overfitting and improve model performance.

```
# defining training hyperparameters

args["train_batch_size"] = 60
args["eval_batch_size"] = 32
args["num_train_epochs"] = 5
args["optimizer"] = AdamW
args["learning_rate"] = 1.5e-3
args["adam_epsilon"] = 1e-8
args["output_dir"] = "./output/"
args["max_grad_norm"] = 1.0
args["save_steps"] = 1
args["activation"] = F.relu

model = Net(args)
model.to(args["device"])
```

In this section of the code, we are defining the hyperparameters for training our neural network model. These hyperparameters specify various settings for the training process, such as the batch size, number of epochs, and learning rate.

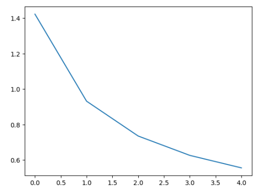
Finally, we create an instance of the Net class, which represents our neural network model. We pass in the args dictionary to configure the model's hyperparameters, and call model.to(args["device"]) to move the model to the specified device for computation.

Now we'll call train function

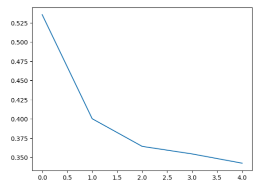
```
train_losses, train_acc, val_losses, val_acc = train(args, mnist_train,
    mnist_test, model)
```

The train function is responsible for training the model on the training set, evaluating the model on the validation set, and returning the training and validation losses and accuracies at each epoch.

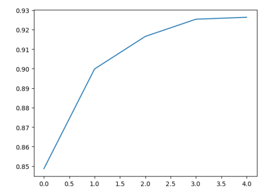
train_losses and val_losses are lists that contain the training and validation losses, respectively, for each epoch of training. The loss is a measure of how well the model is able to predict the



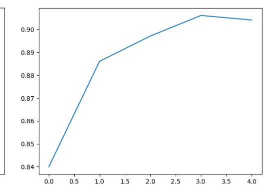
(a) train _losses



(b) val_losses



(c) train_acc



(d) val_acc

correct labels for the training and validation examples.

train_acc and val_acc are lists that contain the training and validation accuracies, respectively, for each epoch of training. The accuracy is a measure of how well the model is able to correctly classify the training and validation examples.

We then plotted all these four lists.

```
# Inference
def inference(model, sample):
    softmax = nn.Softmax(dim=-1)

    model.eval()

    with torch.no_grad():
        inputs, labels_ = sample

        logits = model(inputs)  # forward pass
        outputs = softmax(logits)
    print("Preds are {}".format(outputs))
    preds = outputs.detach().cpu().numpy()[0]
    logits = logits.detach().cpu().numpy()[0]
    print("Outputs are {}".format(preds))
    print("Logits are {}".format(logits))
    print("Predicted number is {}".format(np.argmax(preds)))
    print("Actual number is {}".format(labels_))


# load model
model = Net(args)
model.load_state_dict(torch.load("./output/clssnn.pth"))
model.to(args["device"])

sample = mnist_test[10]
inference(model, sample)
```

The code block above defines a function inference that takes a trained model and a sample as input and performs inference to predict the label of the input sample.

The inference function first initializes a softmax layer, sets the model to evaluation mode (using model.eval()) and then performs a forward pass through the model with the input sample to obtain the logits. The logits are then passed through the softmax layer to obtain the predicted probabilities for each class.

The function then prints the predicted probabilities and the actual label of the input sample, as well as the predicted number (which is the class with the highest probability). The function also returns the predicted probabilities as a numpy array.

The code then loads the trained model using model.load_state_dict(torch.load("./output/clssnn.pth")) and moves the model to the appropriate device using model.to(args["device"]). Finally, the function is called with a sample from the test set to perform inference and print the results.

3 Experiment with different Training Batch Sizes

```
from collections import defaultdict
```

Here we are importing Defaultdict: a sub-class of the dictionary class that returns a dictionary-like object, and which like dictionaries is a container and is present in the module collections. The functionality of both dictionaries and defaultdict are almost the same except for the fact that defaultdict never raises a KeyError. Instead, for the keys that do not exist it provides a default value. This solves the issues that crop up when KeyError is raised.

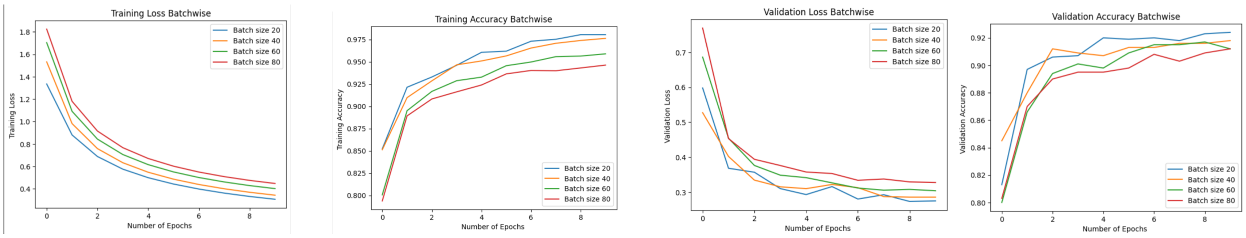
```
1 batch_sizes = [20,40,60,80]
2 train_loss_df = defaultdict()
3 train_acc_df = defaultdict()
4 val_loss_df = defaultdict()
5 val_acc_df = defaultdict()
6 
7 for bs in batch_sizes:
8     args["train_batch_size"] = bs
9     model = Net(args)
10     model.to(args["device"])
11     train_losses, train_acc, val_losses, val_acc = train(args, mnist_train, mnist_test, model)
12     train_loss_df[bs] = train_losses
13     train_acc_df[bs] = train_acc
14     val_loss_df[bs] = val_losses
15     val_acc_df[bs] =  val_acc
```

Batch size is a parameter that defines the number of samples that will be propagated through the network. It gives the number of data points we train our model over, in every iteration before the model is updated. While it controls the accuracy of the estimate of the error gradient, there is a tension between batch size and the speed and stability of the learning process.

Line 1 of this code defines a list of batch sizes to be used for training - 20, 40, 60 and 80.
Then we define four defaultdict objects for storing the training loss, training accuracy, validation loss and validation accuracy values for each batch size. The batch size argument in the "args" dictionary holds the current batch size value. An object of the class "Net" is instantiated using the updated "args" dictionary.
Then the "train" function with the updated "args" dictionary is called, along with the training and testing datasets, and the model object as arguments. This function trains the model on the training dataset, evaluates its performance on the testing dataset, and returns the training and validation loss

and accuracy values which are stored in the respective defaultdict lists using the current batch size as the key.

The resulting loss and accuracy values for each batch size can be used to compare their performance and select the best one.



(a) Training Loss Batchwise

(b) Training Accuracy Batchwise

(c) Validation Loss Batchwise

(d) Validation Accuracy Batchwise

4 Experiment with different Learning Rates

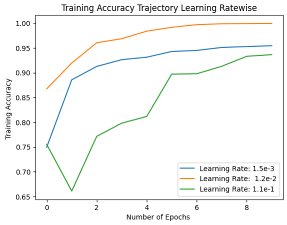
```
learning_rates = [1.5e-3,1.2e-2,1.1e-1]
train_loss_df = defaultdict()
train_acc_df = defaultdict()
val_loss_df = defaultdict()
val_acc_df = defaultdict()

for bs in learning_rates:
    args["learning_rate"] = bs
    model = Net(args)
    model.to(args["device"])
    train_losses, train_acc, val_losses, val_acc = train(args, mnist_train,
   mnist_test, model)
    train_loss_df[bs] = train_losses
    train_acc_df[bs] = train_acc
    val_loss_df[bs] = val_losses
    val_acc_df[bs] =  val_acc
```

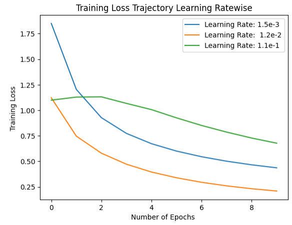
The learning rate is a hyper-parameter that determines the step size at which a machine learning model's parameters are updated during training. In other words, it controls the speed at which the model learns from the data. The learning rate determines how much the parameter is adjusted during each iteration of training.

The code then initializes empty dictionaries train_loss_df, train_acc_df, val_loss_df, and val_acc_df using the defaultdict function from the collections module. These dictionaries will be used to store the training and validation loss and accuracy for each learning rate.

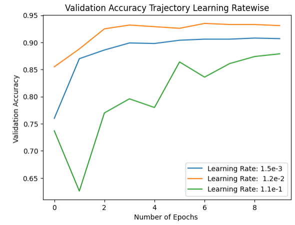
The code then loops over the learning rates using a for loop, setting the learning_rate parameter in the args dictionary to the current learning rate, and creates a new neural network model using the Net class defined elsewhere. The model is then moved to the device specified in the args dictionary which is the "Computer" in this case.



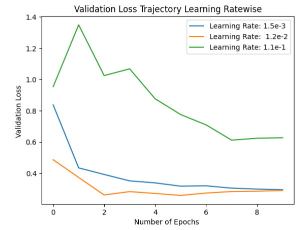
(a) Training accuracy



(b) Training loss



(c) Validation accuracy



(d) Validation loss

The train function is then called with the args dictionary, the MNIST training and test datasets, and the newly created model as arguments. The train function returns four lists: train_losses, train_acc, val_losses, and val_acc, which represent the training loss, training accuracy, validation loss, and validation accuracy over time during training.

Overall, this code is a simple example of how to train and evaluate a neural network model with different learning rates and record the training and validation performance for each learning rate.

Then we plot the Training accuracy and loss, and Validation accuracy and loss trajectory as a function of the learning rate:

5 Experiment with different Activation Functions

```
# defining training hyperparameters

args["train_batch_size"] = 60
args["eval_batch_size"] = 32
args["num_train_epochs"] = 5
args["optimizer"] = AdamW
args["learning_rate"] = 1.5e-3
args["adam_epsilon"] = 1e-8
args["output_dir"] = "./output/"
args["max_grad_norm"] = 1.0
args["save_steps"] = 1
```

In this section of the code, we are defining the hyperparameters for training our neural network model. These hyperparameters specify various settings for the training process, such as the batch size, number of epochs, and learning rate.

```
activationFunctions = [F.relu, F.tanh, F.sigmoid]
train_loss_df = defaultdict()
train_acc_df = defaultdict()
val_loss_df = defaultdict()
val_acc_df = defaultdict()

for bs in activationFunctions:
    args["activation"] = bs
    model = Net(args)
```

```
    model.to(args["device"])
    train_losses, train_acc, val_losses, val_acc = train(args, mnist_train,
mnist_test, model)
    train_loss_df[bs] = train_losses
    train_acc_df[bs] = train_acc
    val_loss_df[bs] = val_losses
    val_acc_df[bs] =  val_acc
```

The code involves training the Net model with different activation functions. Three activation functions: Rectified Linear Unit (ReLU), hyperbolic tangent (Tanh), and sigmoid are used.

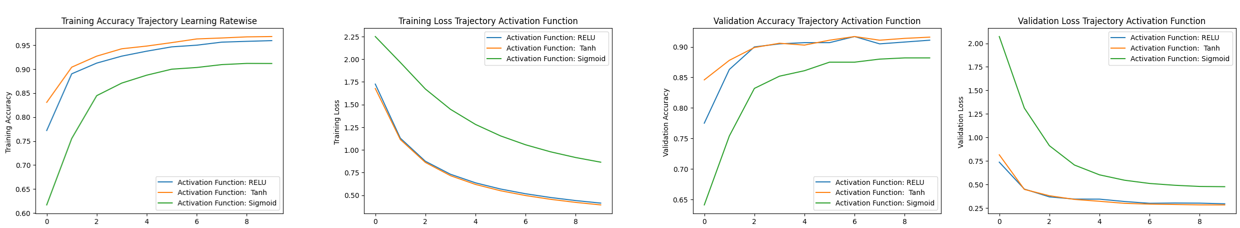
- activationFunctions: A list of activation functions to be experimented with
- train_loss_df, train_acc_df, val_loss_df, val_acc_df: Empty dictionaries to store the training loss, training accuracy, validation loss, and validation accuracy for each activation function

The code then enters a loop over the activationFunctions list. For each activation function bs in the list, the following steps are executed:

- The args dictionary is updated with the activation key set to bs
- A new neural network model is created using the updated args
- The model is moved to the device specified in args
- The train function is then called with the updated args, the training and testing datasets (mnist_train and mnist_test, respectively), and the created model. The train function performs training and evaluation on the model and returns four lists of values: train_losses, train_acc, val_losses, and val_acc
- These values are then added to their respective defaultdict objects using the bs activation function as the key

After the loop completes, the train_loss_df, train_acc_df, val_loss_df, and val_acc_df defaultdict objects contain lists of training and validation loss and accuracy values for each of the three activation functions.

Then we plot the Training accuracy and loss, and Validation accuracy and loss trajectory as a function of the activation functions:



(a) Training accuracy (b) Training loss (c) Validation accuracy (d) Validation loss