ECE655 Project 03

Author: Stewart Schuler

Due: 10/12/2025

Contents

1 Part A

The dataset used for this project was the *Real estate price prediction* data set download from *Kaggle*[1]. The dataset contained 6 features and 414 data points, the data was shuffled and split into training and validation datasets using an 80 : 20 split.

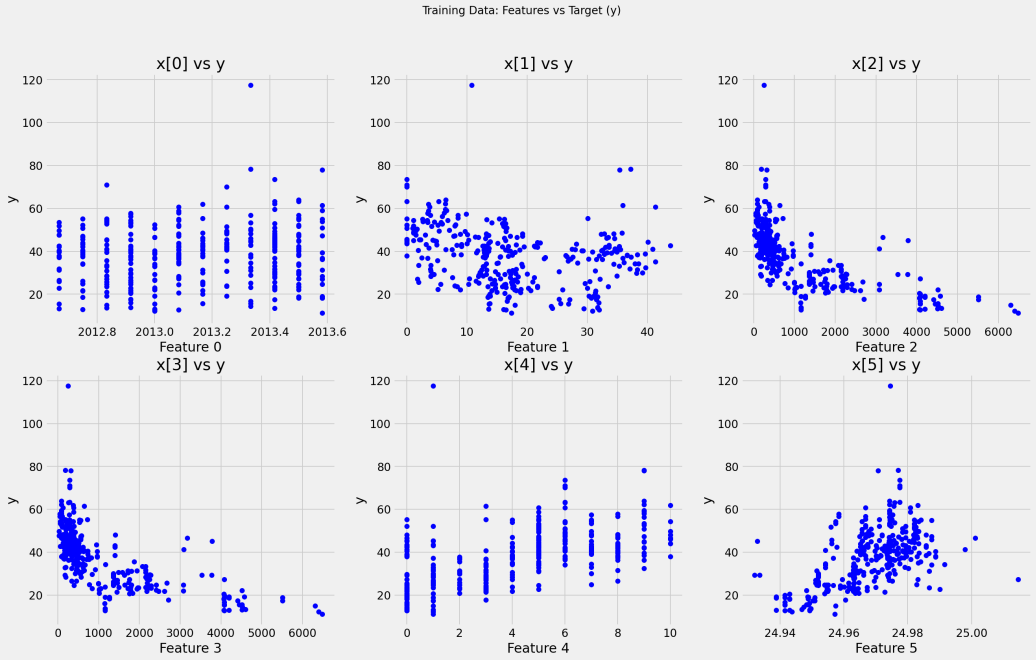

Figure 1: Training Dataset

[1] https://www.kaggle.com/datasets/quantbruce/real-estate-price-prediction

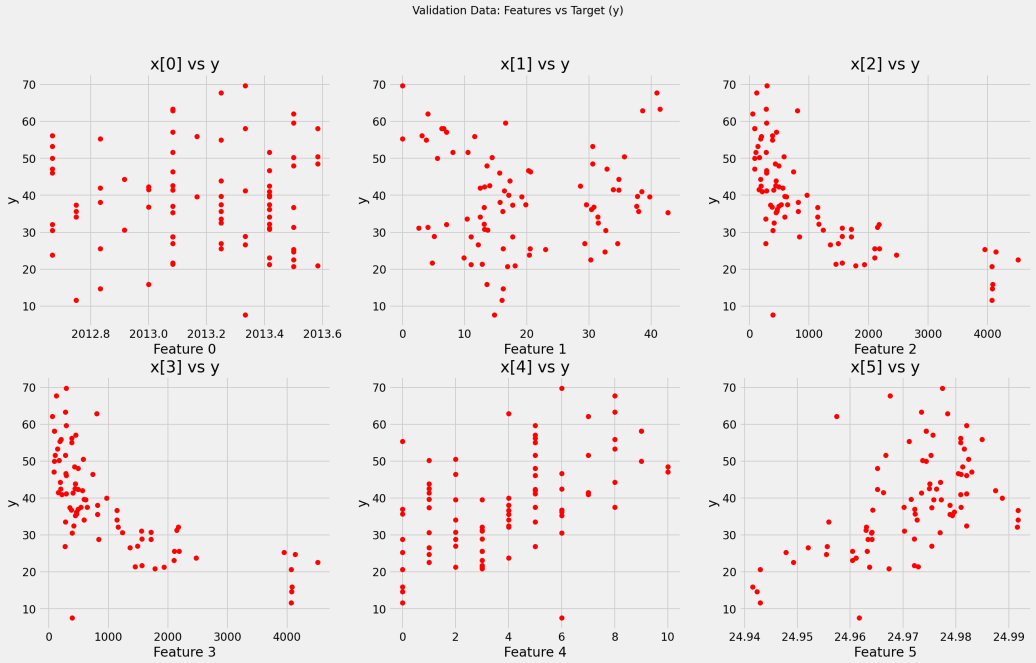

Figure 2: Validation Dataset

2 Part B

For learning the regression a single layer linear model was trained using the **SGD** optimizer, **MSE** loss, and **full** batch size. The loss curve is shown in figure 3. For this experiment a learning rate μ of 0.05 was used and stayed constant for the remainder of this project. As can be seen with the chosen parameters convergence is nearly completed after 20 epochs.

```
model=nn.Sequential(nn.Linear(num_features,1)).to(device)
optimizer = optim.SGD(model.parameters(), lr=lr)
loss_fn=nn.MSELoss(reduction='mean')
train_step_fn = make_train_step_fn(model, loss_fn, optimizer)
val_step_fn   = make_val_step_fn(model, loss_fn)

for epoch in range(n_epochs):
    loss = mini_batch(device, train_loader, train_step_fn)
    losses[epoch] = loss;
    with torch.no_grad():
        val_loss = mini_batch(device, val_loader, val_step_fn
    )
        val_losses[epoch] = val_loss;
```

Listing 1: part_b.py

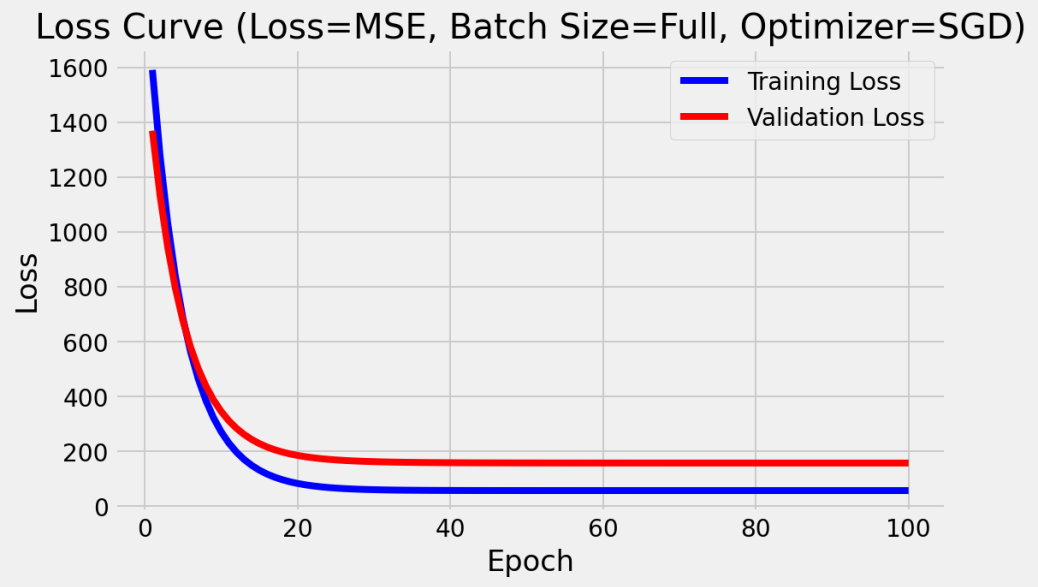

Figure 3: Loss Curve with (**SGD**, **MSE**, **full**)

3 Part C

For the next experiment the loss was changed to **L1** loss, with all other parameters saying constant.

```
model=nn.Sequential(nn.Linear(num_features,1)).to(device)
optimizer = optim.SGD(model.parameters(), lr=lr)
loss_fn=nn.L1Loss(reduction='mean')
train_step_fn = make_train_step_fn(model, loss_fn, optimizer)
val_step_fn   = make_val_step_fn(model, loss_fn)

for epoch in range(n_epochs):
    loss = mini_batch(device, train_loader, train_step_fn)
    losses[epoch] = loss;
    with torch.no_grad():
        val_loss = mini_batch(device, val_loader, val_step_fn
    )
        val_losses[epoch] = val_loss;
```

Listing 2: part_c.py

In Figure 4 it can be seen that the loss curve takes on a linear shape. This is because the MSE (or L2) loss is computed as square of the difference, giving it that exponential shape. Since L1 loss is computed simple as the difference between y and $\hat{y}$, which is a linear equation the loss curve has a linear shape.

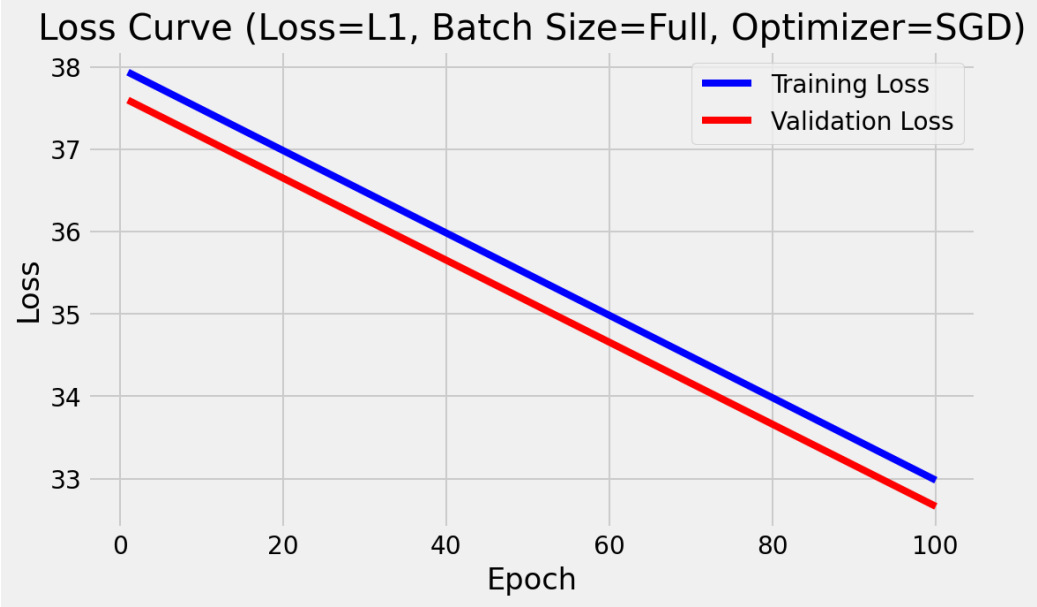

Figure 4: Loss Curve with (**SGD**, **L1**, **full**)

4 Part D

Next to consider the impact of changing the optimizer the loss was set back to **MSE** loss and the optimizer was changed to **Adam**.

```
model=nn.Sequential(nn.Linear(num_features,1)).to(device)
optimizer = optim.Adam(model.parameters(), lr=lr)
loss_fn=nn.L1Loss(reduction='mean')
train_step_fn = make_train_step_fn(model, loss_fn, optimizer)
val_step_fn   = make_val_step_fn(model, loss_fn)

for epoch in range(n_epochs):
    loss = mini_batch(device, train_loader, train_step_fn)
    losses[epoch] = loss;
    with torch.no_grad():
        val_loss = mini_batch(device, val_loader, val_step_fn
    )
        val_losses[epoch] = val_loss;
```

Listing 3: part_d.py

It can be seen in Figure 5 that the curve mains it's exponential shape that it had when using the **SGD** optimizer but it is approaching convergence much slower.

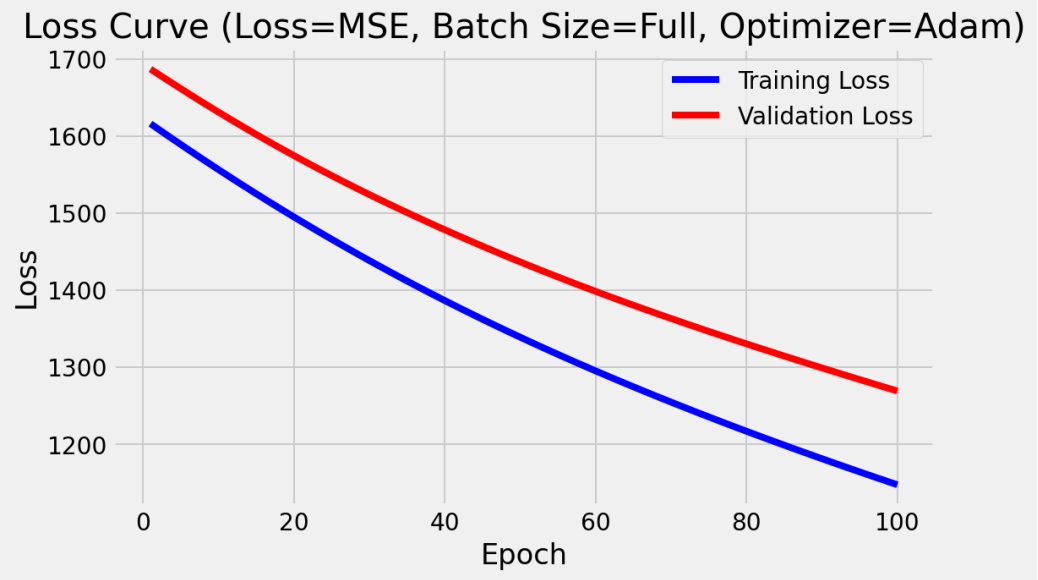

Figure 5: Loss Curve with (**Adam**, **MSE**, **full**)

5 Part E

In this experiment we consider how changing the batch size impacts the training.

```
plt_bs = np.array([4, 8, 16, 32, 64])
for batch_idx, batch_size in enumerate(plt_bs):
    train_loader = DataLoader(dataset=train_data, batch_size=int(batch_size), shuffle=True)
    val_loader = DataLoader(dataset=val_data, batch_size=int(batch_size))

    model=nn.Sequential(nn.Linear(num_features,1)).to(device)
    optimizer = optim.SGD(model.parameters(), lr=lr)
    loss_fn=nn.MSELoss(reduction='mean')
    train_step_fn = make_train_step_fn(model, loss_fn, optimizer)
    val_step_fn   = make_val_step_fn(model, loss_fn)

    losses = np.empty(n_epochs);
    val_losses = np.empty(n_epochs);
    for epoch in range(n_epochs):
        loss = mini_batch(device, train_loader, train_step_fn)
        losses[epoch] = loss;
        with torch.no_grad():
```

```
18              val_loss = mini_batch(device, val_loader,
    val_step_fn)
19              val_losses[epoch] = val_loss;
```

Listing 4: part_e.py

It can be seen in Figure 6 that the final training and validations losses are not the same depending on the batch sizes. For this specific dataset with 331 training samples after the split a batch size of 16 produced the best final loss values.

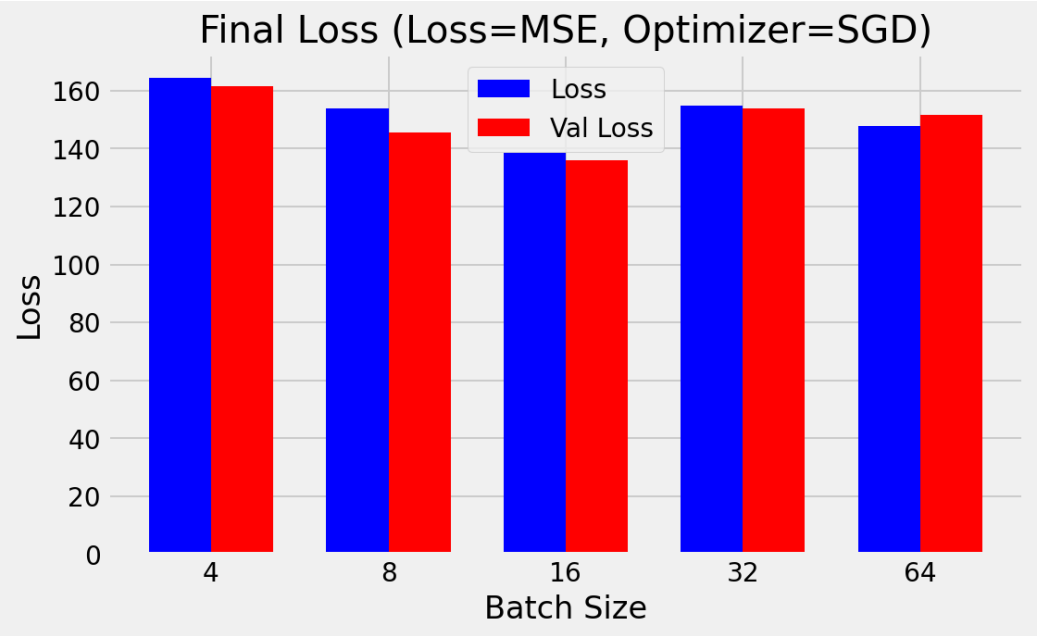

Figure 6: Fianl Losses with (**SGD**, **MSE**, **Variable Batch Size**)

6 Part F

In this experiment we sweep two different optimizers (**Adam** and **SGD**) across different number of training epochs. Figure 7 compares the final training loss after X epochs and the training time. For this experiment training was ran entirely on the CPU.

```
epochs_sweep = np.linspace(20,1000,50)
optim_sweep  = [optim.SGD, optim.Adam]
device='cpu'
for optim_idx, opt in enumerate(optim_sweep):
    for epoch_idx, n_epochs in enumerate(epochs_sweep):

        model=nn.Sequential(nn.Linear(num_features,1)).to(
    device)
        optimizer = opt(model.parameters(), lr=lr)
        loss_fn=nn.MSELoss(reduction='mean')
        train_step_fn = make_train_step_fn(model, loss_fn,
    optimizer)
        val_step_fn   = make_val_step_fn(model, loss_fn)

        start_ts = time.time()
        for epoch in range(int(n_epochs)):
            loss = mini_batch(device, train_loader,
    train_step_fn)

        stop_ts = time.time()
```

Listing 5: part_f.py

It can be seen that with a large number of Epochs the *Adam* optimizer does eventually converge towards the same values as the *SGD* optimizer. And the ADAM runtime is slightly slower.

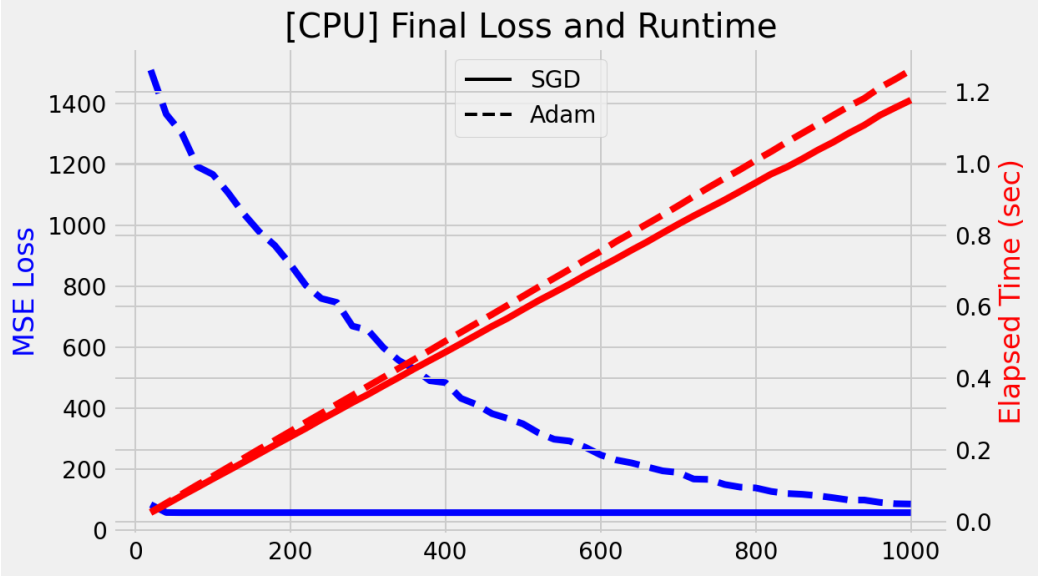

Figure 7: CPU Final Training loss and Runtime vs Epochs

7 Part G

The same experiment from the previous section was reran, with the modification of targeting the GPU. The hardware being tested on was an *AMD 7900 GRE* using *ROCm* support for *pytorch*.

```
epochs_sweep = np.linspace(20,1000,50)
optim_sweep  = [optim.SGD, optim.Adam]
device='cuda'
for optim_idx, opt in enumerate(optim_sweep):
    for epoch_idx, n_epochs in enumerate(epochs_sweep):

        model=nn.Sequential(nn.Linear(num_features,1)).to(
    device)
        optimizer = opt(model.parameters(), lr=lr)
        loss_fn=nn.MSELoss(reduction='mean')
        train_step_fn = make_train_step_fn(model, loss_fn,
    optimizer)
        val_step_fn   = make_val_step_fn(model, loss_fn)

        start_ts = time.time()
        for epoch in range(int(n_epochs)):
            loss = mini_batch(device, train_loader,
    train_step_fn)

```

```
17          stop_ts = time.time()
```

Listing 6: part_g.py

Surprisingly, as shown in Figure 8, when processing on the GPU the comparable runtimes were actually slower than on the CPU. This result is likely because the dataset is so small that the overhead associated with moving the data to/from the GPU and launching the kernels negates any performance improvements that come from processing on the GPU. Also since the Linear Regression model is only a single (6,1) neuron there isn't mass parallelization available for each parameter like there would be in a deep model. Additionally the ROCm framework is not as well optimized as the CUDA based implementations.

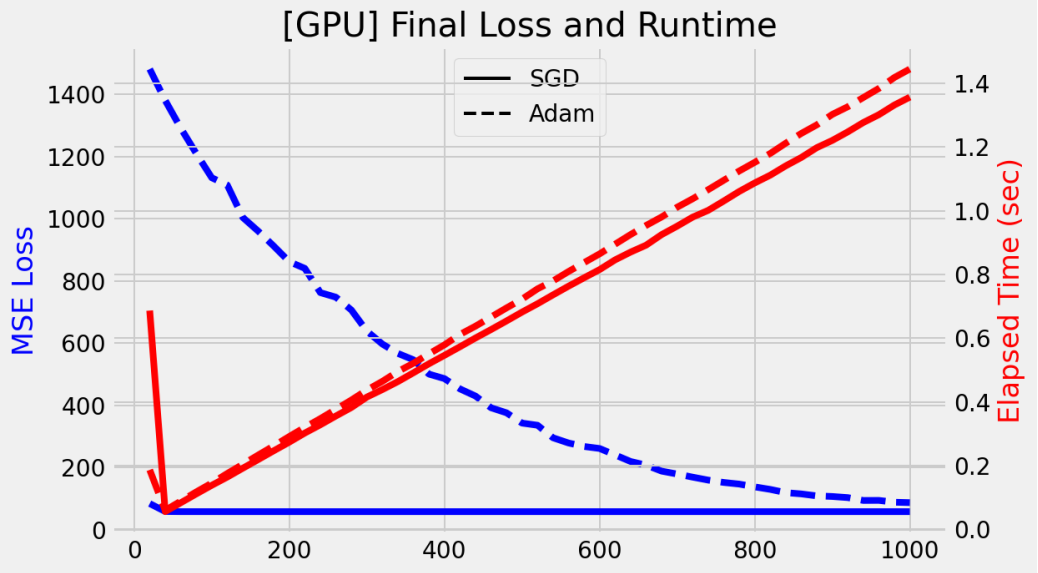

Figure 8: GPU Final Training loss and Runtime vs Epochs

Out of curiosity I tried to see if I could demonstrate a significant GPU performance increase. To address the small dataset problem I created a $1,000,000$ point 50 feature data dataset with a trivial noise linear model. Figure 9 shows the final runtime for training it with the CPU and GPU. It can be seen that my GPU did improve performance by roughly a factor of 4. Meaning in the above experiment, had the dataset been larger the runtimes would've been improved.

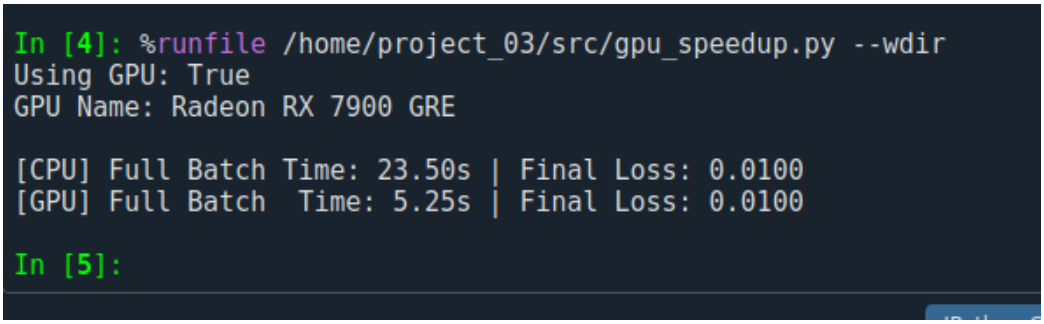

Figure 9: GPU Performance Increase