

MSML605 Assignment2 Report

Topic: - ClearML Implementation

Ayush Sinha LNU

UID: 121334060

1. Introduction:

This report is all about the process of training a Convolutional Neural Network (CNN) on the FashionMNIST dataset while utilizing ClearML for tracking purposes. The main aim was to get an accuracy of at least 90% on the testing data.

2. Model Parameters:

The following parameters were mentioned in the assignment to be used during training:

- Batch Size: 64
- Number of epochs: 100
- Learning Rate: 0.01
- SGD momentum: 0.5
- CUDA usage: Enabled (if available)
- Logging interval: 10 iterations

This is the screenshot of the model configuration from ClearML Dashboard:

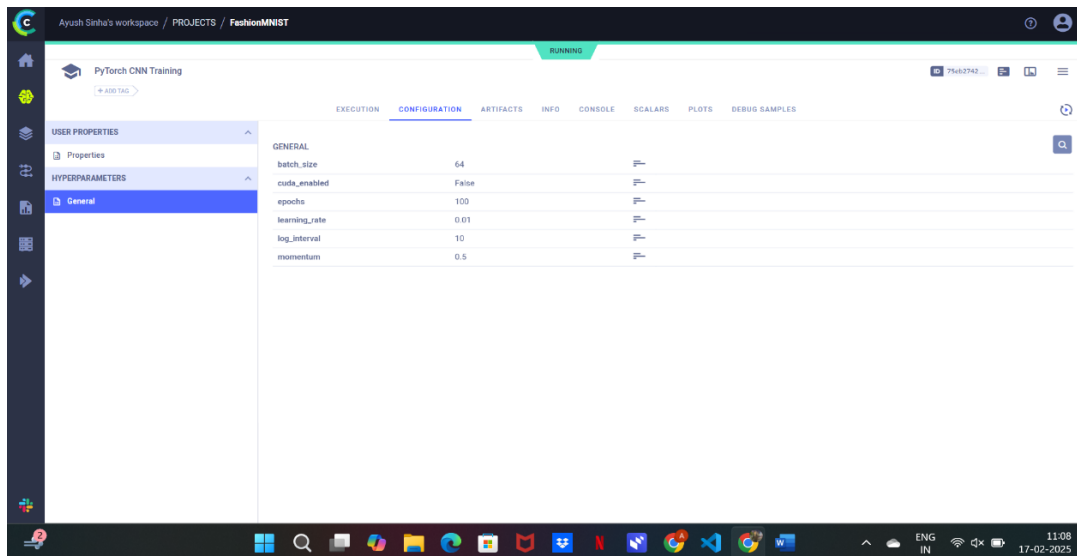


Fig 1. Configuration Screenshot

3. Training Summary:

The training process included the following: -

- Loading the FashionMNIST dataset and doing the preprocessing.
- Defining a CNN with two convolutional layers followed by two fully connected layers.
- Training done for 100 iterations/epochs and tracking was done in ClearML.
- Evaluating the model on the test data to get the accuracy.

Model Architecture:

- The Model consists of the two convolutional layers which are Conv1 and Conv2.
- Conv1 – 32 filters, kernel size 3x3 with ReLu activation.
- Conv2 – 64 filters, kernel size 3x3 with ReLu activation.
- MaxPooling was applied after each convolutional layer.

Fully Connected Layers:

- Flatten layer transforms the feature maps into a 1D array.
- FC1 is made up of 128 neurons and dropout of 0.25
- Output layer is of 10 neurons (one per class) with softmax activation.

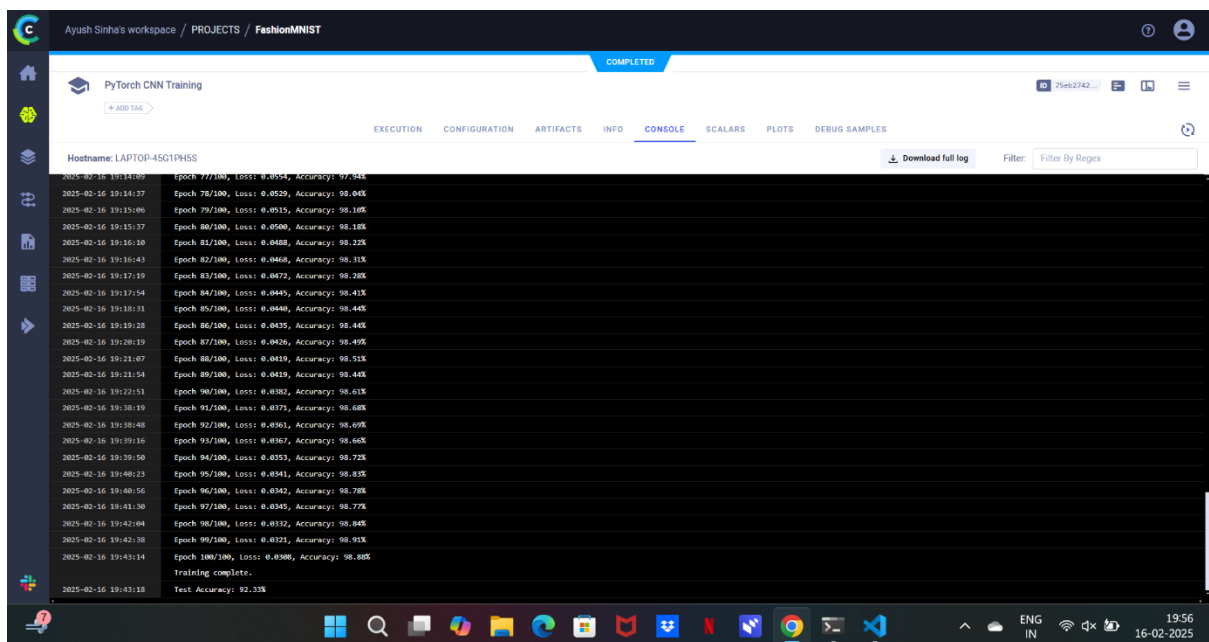
Training:

- The model was trained using CrossEntropyLoss.
- The SGD optimizer was used (learning rate = 0.01, momentum = 0.5).
- The ClearML log was used to keep track of the losses and accuracy.

The Final outputs were as follows:

- Final Training Loss: 0.0308
- Final Training Accuracy: 98.88%
- Final Test Accuracy: 92.33%

The screenshot of the console output has been provided below:



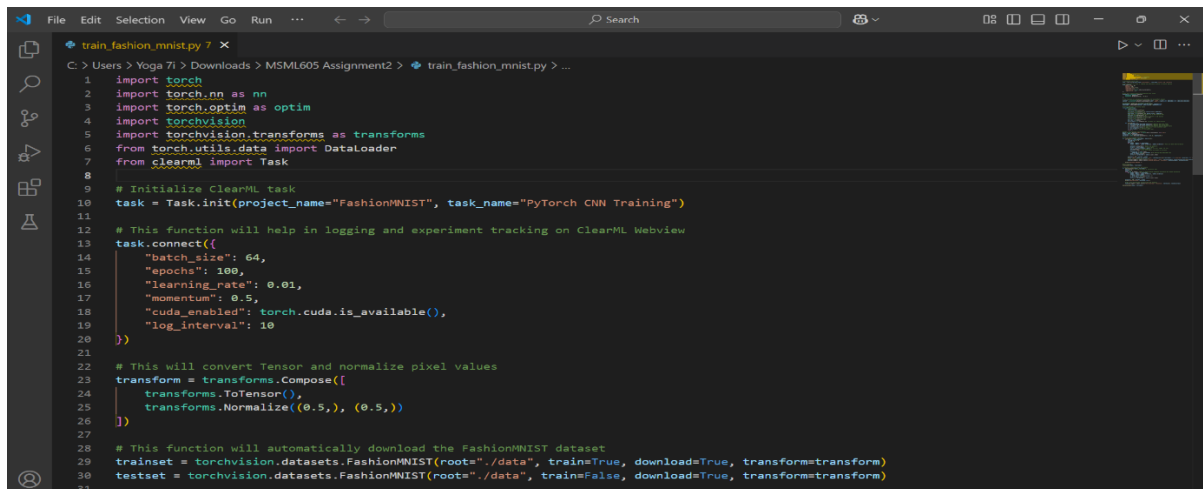
```
Hostname: LAPTOP-45G1PH5S
2025-02-16 19:14:09 Epoch 77/100, Loss: 0.0554, Accuracy: 97.84%
2025-02-16 19:14:37 Epoch 78/100, Loss: 0.0529, Accuracy: 98.04%
2025-02-16 19:15:06 Epoch 79/100, Loss: 0.0515, Accuracy: 98.16%
2025-02-16 19:15:37 Epoch 80/100, Loss: 0.0500, Accuracy: 98.18%
2025-02-16 19:16:10 Epoch 81/100, Loss: 0.0488, Accuracy: 98.22%
2025-02-16 19:16:43 Epoch 82/100, Loss: 0.0468, Accuracy: 98.31%
2025-02-16 19:17:19 Epoch 83/100, Loss: 0.0472, Accuracy: 98.28%
2025-02-16 19:17:54 Epoch 84/100, Loss: 0.0445, Accuracy: 98.41%
2025-02-16 19:18:31 Epoch 85/100, Loss: 0.0440, Accuracy: 98.44%
2025-02-16 19:19:28 Epoch 86/100, Loss: 0.0435, Accuracy: 98.44%
2025-02-16 19:20:19 Epoch 87/100, Loss: 0.0426, Accuracy: 98.49%
2025-02-16 19:21:07 Epoch 88/100, Loss: 0.0419, Accuracy: 98.51%
2025-02-16 19:21:54 Epoch 89/100, Loss: 0.0419, Accuracy: 98.44%
2025-02-16 19:22:51 Epoch 90/100, Loss: 0.0382, Accuracy: 98.61%
2025-02-16 19:30:19 Epoch 91/100, Loss: 0.0371, Accuracy: 98.68%
2025-02-16 19:38:48 Epoch 92/100, Loss: 0.0361, Accuracy: 98.69%
2025-02-16 19:39:16 Epoch 93/100, Loss: 0.0367, Accuracy: 98.66%
2025-02-16 19:39:50 Epoch 94/100, Loss: 0.0353, Accuracy: 98.72%
2025-02-16 19:40:23 Epoch 95/100, Loss: 0.0341, Accuracy: 98.81%
2025-02-16 19:40:56 Epoch 96/100, Loss: 0.0342, Accuracy: 98.78%
2025-02-16 19:41:30 Epoch 97/100, Loss: 0.0345, Accuracy: 98.77%
2025-02-16 19:42:04 Epoch 98/100, Loss: 0.0332, Accuracy: 98.84%
2025-02-16 19:42:38 Epoch 99/100, Loss: 0.0321, Accuracy: 98.91%
2025-02-16 19:43:14 Epoch 100/100, Loss: 0.0308, Accuracy: 98.88%
Training complete.
Test Accuracy: 92.33%
```

Fig 2. Console Outputs

4. Code Implementation:

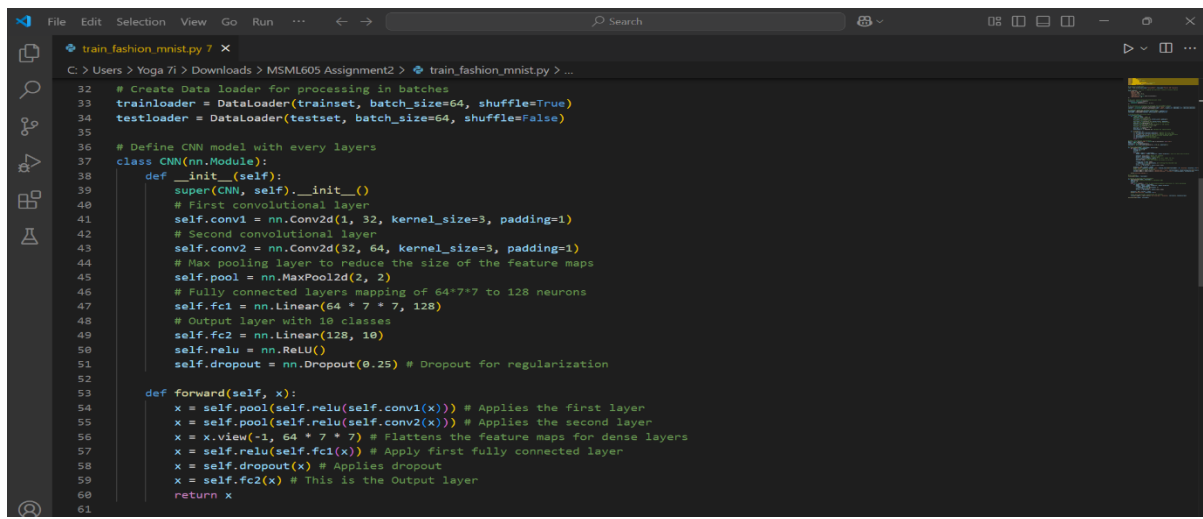
The assignment was implemented using Python on Visual Studio Code IDE. This code also integrates ClearML which is used to track the progress of the code in terms of various parameters.

Below is the screenshot of the implemented code



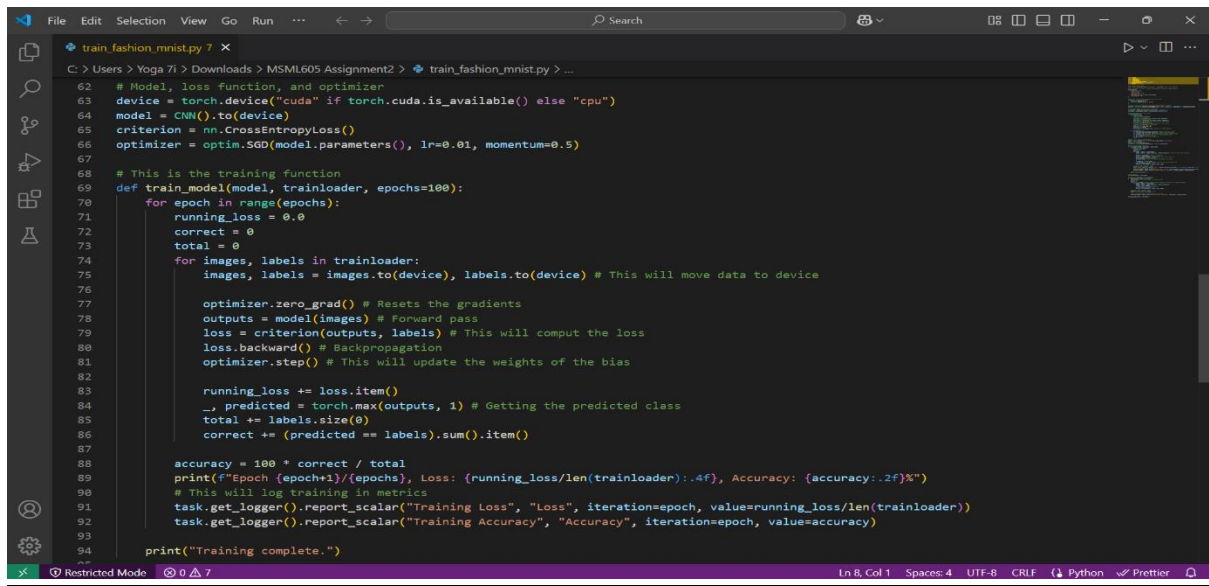
```
1 import torch
2 import torch.nn as nn
3 import torch.optim as optim
4 import torchvision
5 import torchvision.transforms as transforms
6 from torch.utils.data import DataLoader
7 from clearml import Task
8
9 # Initialize ClearML task
10 task = Task.init(project_name="FashionMNIST", task_name="PyTorch CNN Training")
11
12 # This function will help in logging and experiment tracking on ClearML Webview
13 task.connect({
14     "batch_size": 64,
15     "epochs": 100,
16     "learning_rate": 0.01,
17     "momentum": 0.5,
18     "cuda_enabled": torch.cuda.is_available(),
19     "log_interval": 10
20 })
21
22 # This will convert Tensor and normalize pixel values
23 transform = transforms.Compose([
24     transforms.ToTensor(),
25     transforms.Normalize((0.5,), (0.5,))
26 ])
27
28 # This function will automatically download the FashionMNIST dataset
29 trainset = torchvision.datasets.FashionMNIST(root="./data", train=True, download=True, transform=transform)
30 testset = torchvision.datasets.FashionMNIST(root="./data", train=False, download=True, transform=transform)
31
```

Fig 3.



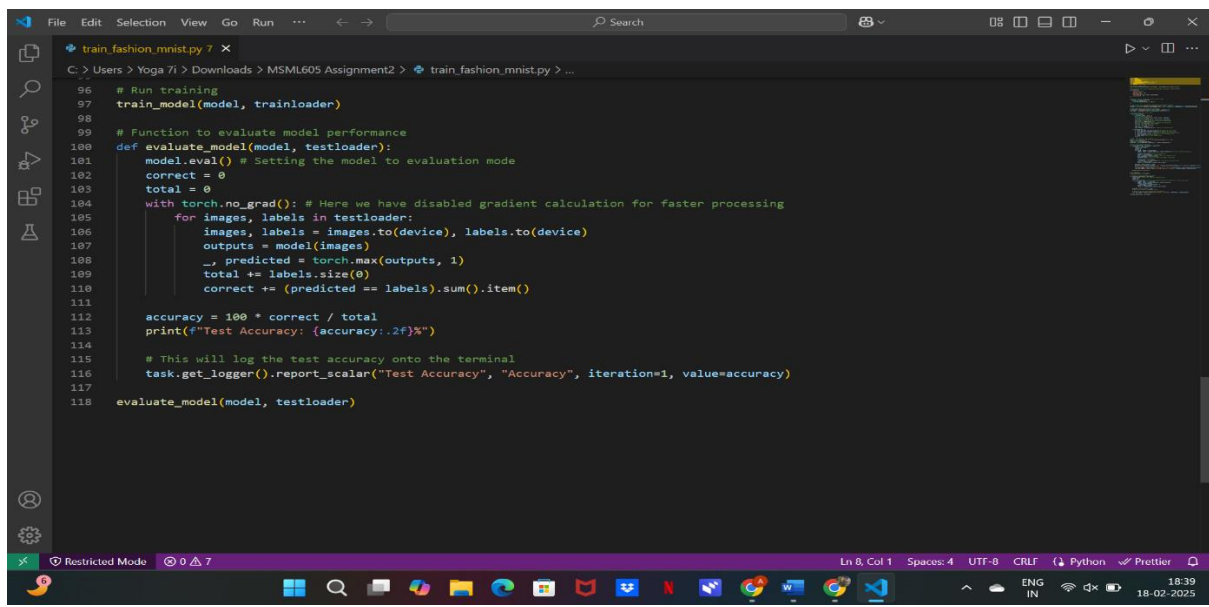
```
32 # Create Data loader for processing in batches
33 trainloader = DataLoader(trainset, batch_size=64, shuffle=True)
34 testloader = DataLoader(testset, batch_size=64, shuffle=False)
35
36 # Define CNN model with every layers
37 class CNN(nn.Module):
38     def __init__(self):
39         super(CNN, self).__init__()
40         # First convolutional layer
41         self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
42         # Second convolutional layer
43         self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
44         # Max pooling layer to reduce the size of the feature maps
45         self.pool = nn.MaxPool2d(2, 2)
46         # Fully connected layers mapping of 64*7*7 to 128 neurons
47         self.fc1 = nn.Linear(64 * 7 * 7, 128)
48         # Output layer with 10 classes
49         self.fc2 = nn.Linear(128, 10)
50         self.relu = nn.ReLU()
51         self.dropout = nn.Dropout(0.25) # Dropout for regularization
52
53     def forward(self, x):
54         x = self.pool(self.relu(self.conv1(x))) # Applies the first layer
55         x = self.pool(self.relu(self.conv2(x))) # Applies the second layer
56         x = x.view(-1, 64 * 7 * 7) # Flattens the feature maps for dense layers
57         x = self.relu(self.fc1(x)) # Apply first fully connected layer
58         x = self.dropout(x) # Applies dropout
59         x = self.fc2(x) # This is the Output layer
60         return x
61
```

Fig 4.



```
62 # Model, loss function, and optimizer
63 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
64 model = CNN().to(device)
65 criterion = nn.CrossEntropyLoss()
66 optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
67
68 # This is the training function
69 def train_model(model, trainloader, epochs=100):
70     for epoch in range(epochs):
71         running_loss = 0.0
72         correct = 0
73         total = 0
74         for images, labels in trainloader:
75             images, labels = images.to(device), labels.to(device) # This will move data to device
76
77             optimizer.zero_grad() # Resets the gradients
78             outputs = model(images) # Forward pass
79             loss = criterion(outputs, labels) # This will compute the loss
80             loss.backward() # Backpropagation
81             optimizer.step() # This will update the weights of the bias
82
83             running_loss += loss.item()
84             _, predicted = torch.max(outputs, 1) # Getting the predicted class
85             total += labels.size(0)
86             correct += (predicted == labels).sum().item()
87
88     accuracy = 100 * correct / total
89     print(f"Epoch {epoch+1}/{epochs}, Loss: {running_loss/len(trainloader):.4f}, Accuracy: {accuracy:.2f}%")
90     # This will log training in metrics
91     task.get_logger().report_scalar("Training Loss", "Loss", iteration=epoch, value=running_loss/len(trainloader))
92     task.get_logger().report_scalar("Training Accuracy", "Accuracy", iteration=epoch, value=accuracy)
93
94     print("Training complete.")
```

Fig 5.



```
96 # Run training
97 train_model(model, trainloader)
98
99 # Function to evaluate model performance
100 def evaluate_model(model, testloader):
101     model.eval() # Setting the model to evaluation mode
102     correct = 0
103     total = 0
104     with torch.no_grad(): # Here we have disabled gradient calculation for faster processing
105         for images, labels in testloader:
106             images, labels = images.to(device), labels.to(device)
107             outputs = model(images)
108             _, predicted = torch.max(outputs, 1)
109             total += labels.size(0)
110             correct += (predicted == labels).sum().item()
111
112     accuracy = 100 * correct / total
113     print(f"Test Accuracy: {accuracy:.2f}%")
114
115     # This will log the test accuracy onto the terminal
116     task.get_logger().report_scalar("Test Accuracy", "Accuracy", iteration=1, value=accuracy)
117
118     evaluate_model(model, testloader)
```

Fig 6.

The figure 3 to 6 (Fig 3-6) are the screenshots of the code implementation.

4. Training and Evaluation:

ClearML was used to track the following:

- Training Loss: Monitored over Iterations.

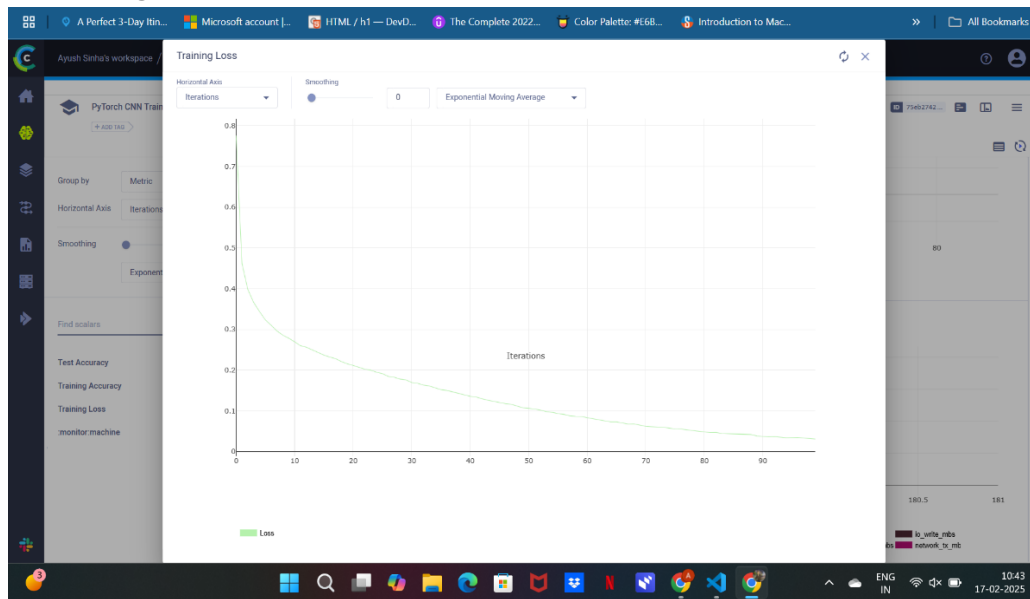


Fig 7. Training Loss

- Training Accuracy: Tracking improvements for every iteration.

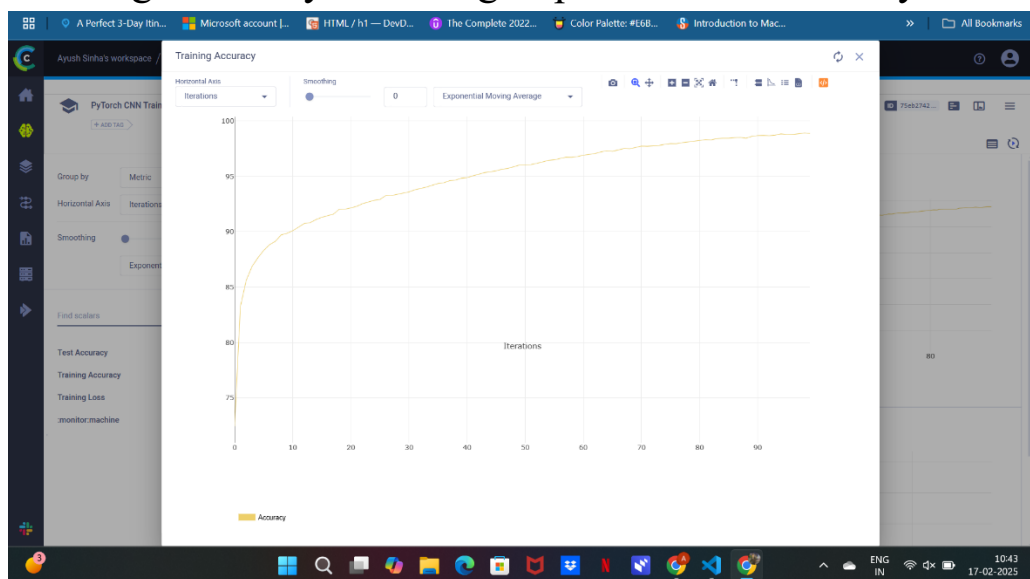


Fig 8. Training Accuracy

- Test Accuracy: Measured accuracy over testing data.

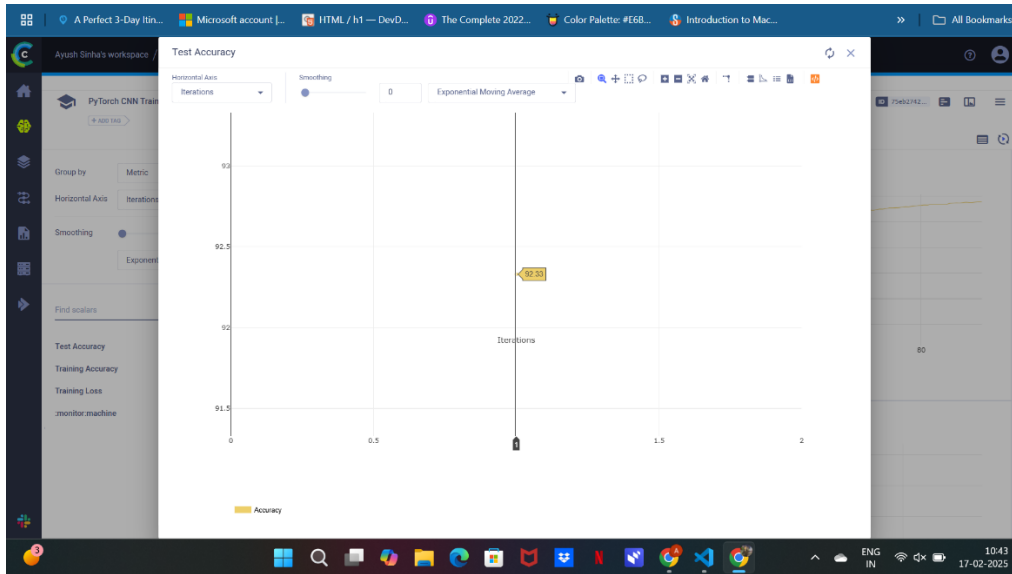


Fig 9. Test Accuracy

- Machine Utilization: Logged the CPU usage, IO read and Memory usage.

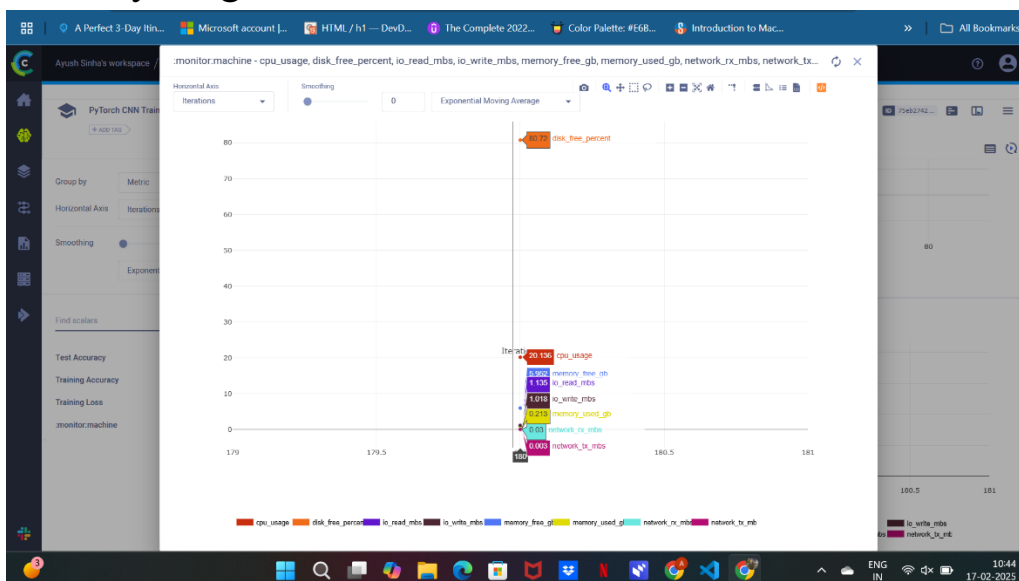


Fig 10. Machine Utilization

5. Conclusion:

The CNN model was able to achieve a test accuracy of 92.33%. ClearML proved to be a very useful tool to track and recognize the model performance in detailed way.