# 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:

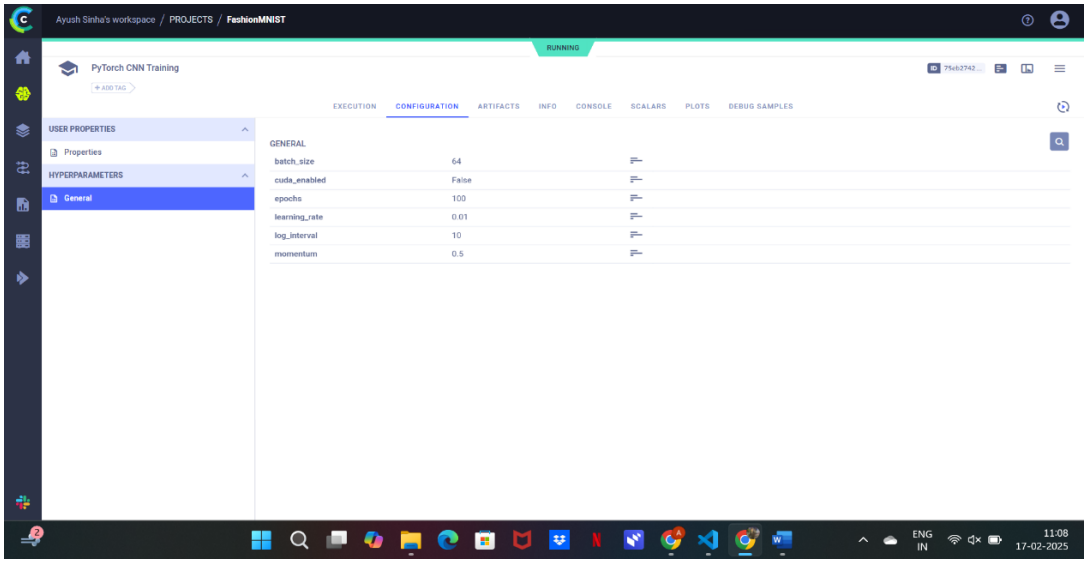



Fig 1. Configuration Screenshot

# 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:

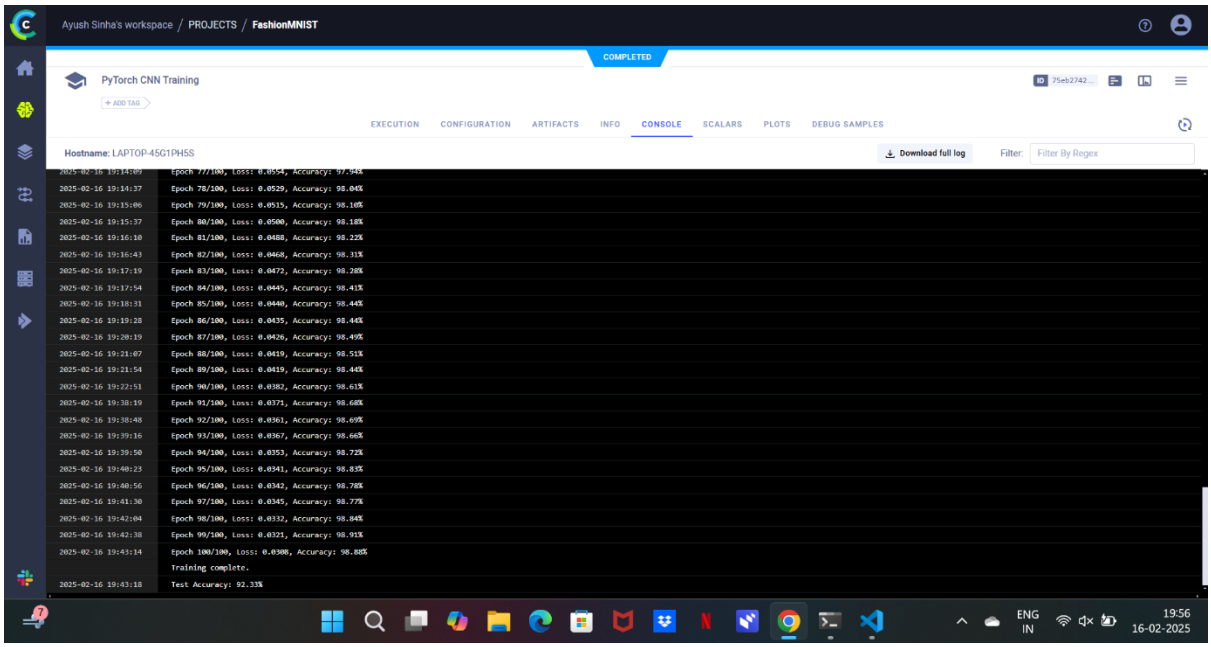



Fig 2. Console Outputs

# 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.

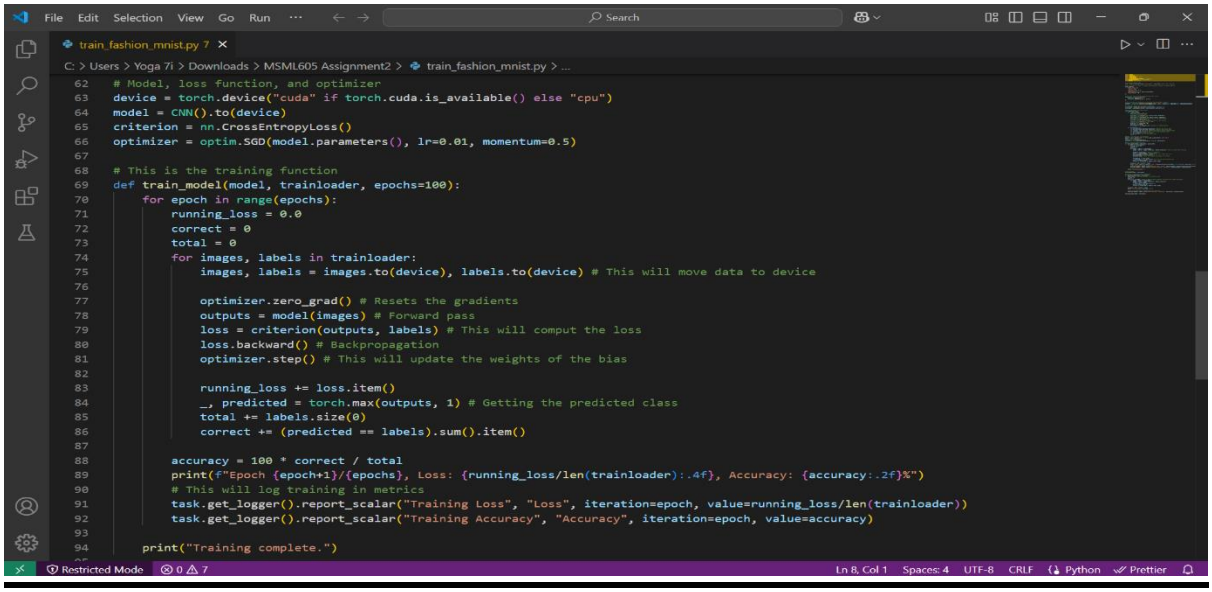
Below is the screenshot of the implemented code

```python
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from clearml import Task

# Initialize ClearML task
task = Task.init(project_name="FashionMNIST", task_name="PyTorch CNN Training")

# This function will help in logging and experiment tracking on ClearML Webview
task.connect({
    "batch_size": 64,
    "epochs": 100,
    "learning_rate": 0.01,
    "momentum": 0.5,
    "cuda_enabled": torch.cuda.is_available(),
    "log_interval": 10
})

# This will convert Tensor and normalize pixel values
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# This function will automatically download the FashionMNIST dataset
trainset = torchvision.datasets.FashionMNIST(root="./data", train=True, download=True, transform=transform)
testset = torchvision.datasets.FashionMNIST(root="./data", train=False, download=True, transform=transform)
```

Fig 3.

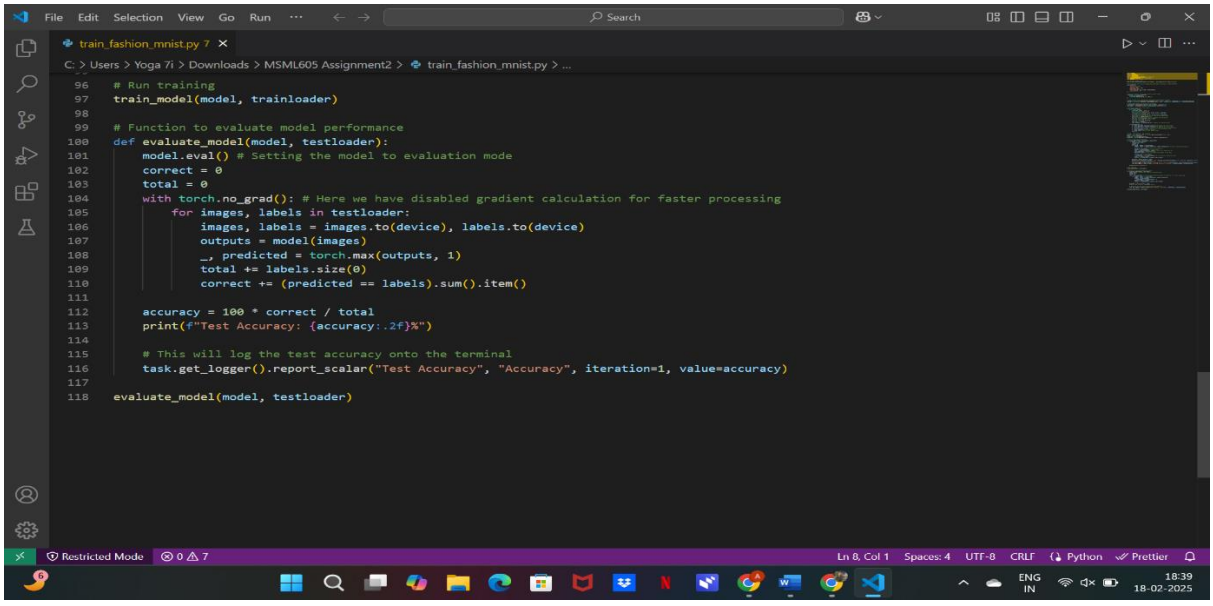
```python
# Create Data loader for processing in batches
trainloader = DataLoader(trainset, batch_size=64, shuffle=True)
testloader = DataLoader(testset, batch_size=64, shuffle=False)

# Define CNN model with every layers
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        # First convolutional layer
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        # Second convolutional layer
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        # Max pooling layer to reduce the size of the feature maps
        self.pool = nn.MaxPool2d(2, 2)
        # Fully connected layers mapping of 64*7*7 to 128 neurons
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        # Output layer with 10 classes
        self.fc2 = nn.Linear(128, 10)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.25) # Dropout for regularization

    def forward(self, x):
        x = self.pool(self.relu(self.conv1(x))) # Applies the first layer
        x = self.pool(self.relu(self.conv2(x))) # Applies the second layer
        x = x.view(-1, 64 * 7 * 7) # Flattens the feature maps for dense layers
        x = self.relu(self.fc1(x)) # Apply first fully connected layer
        x = self.dropout(x) # Applies dropout
        x = self.fc2(x) # This is the Output layer
        return x
```

Fig 4.



```python
# Model, loss function, and optimizer
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = CNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)

# This is the training function
def train_model(model, trainloader, epochs=100):
    for epoch in range(epochs):
        running_loss = 0.0
        correct = 0
        total = 0
        for images, labels in trainloader:
            images, labels = images.to(device), labels.to(device) # This will move data to device

            optimizer.zero_grad() # Resets the gradients
            outputs = model(images) # Forward pass
            loss = criterion(outputs, labels) # This will comput the loss
            loss.backward() # Backpropagation
            optimizer.step() # This will update the weights of the bias

            running_loss += loss.item()
            _, predicted = torch.max(outputs, 1) # Getting the predicted class
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        accuracy = 100 * correct / total
        print(f"Epoch {epoch+1}/{epochs}, Loss: {running_loss/len(trainloader):.4f}, Accuracy: {accuracy:.2f}%")
        # This will log training in metrics
        task.get_logger().report_scalar("Training Loss", "Loss", iteration=epoch, value=running_loss/len(trainloader))
        task.get_logger().report_scalar("Training Accuracy", "Accuracy", iteration=epoch, value=accuracy)

    print("Training complete.")
```

Fig 5.



```python
# Run training
train_model(model, trainloader)

# Function to evaluate model performance
def evaluate_model(model, testloader):
    model.eval() # Setting the model to evaluation mode
    correct = 0
    total = 0
    with torch.no_grad(): # Here we have disabled gradient calculation for faster processing
        for images, labels in testloader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    print(f"Test Accuracy: {accuracy:.2f}%")

    # This will log the test accuracy onto the terminal
    task.get_logger().report_scalar("Test Accuracy", "Accuracy", iteration=1, value=accuracy)

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.

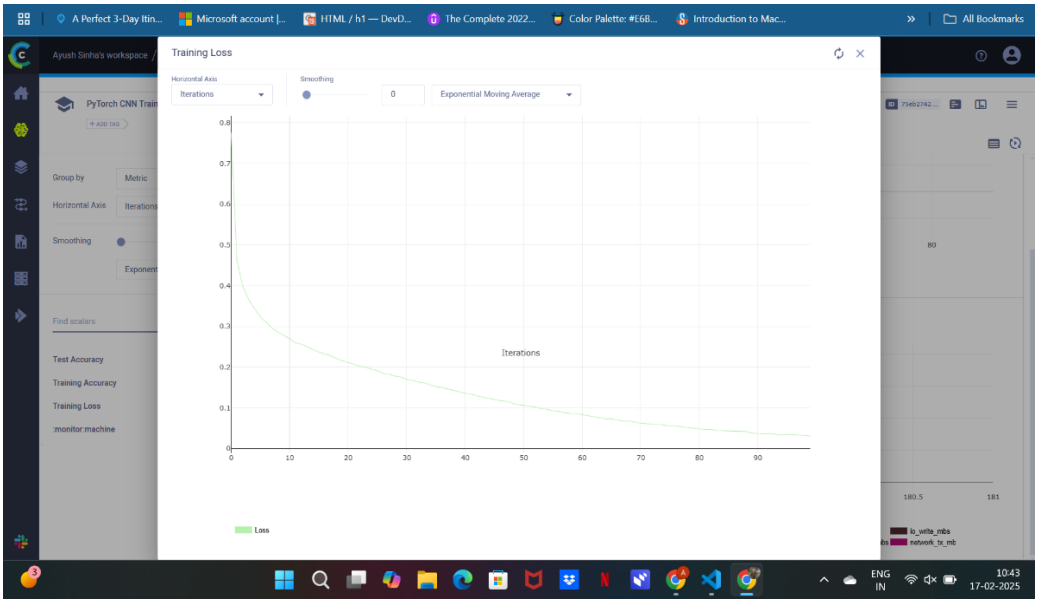



Fig 7. Training Loss

- Training Accuracy: Tracking improvements for every iteration.

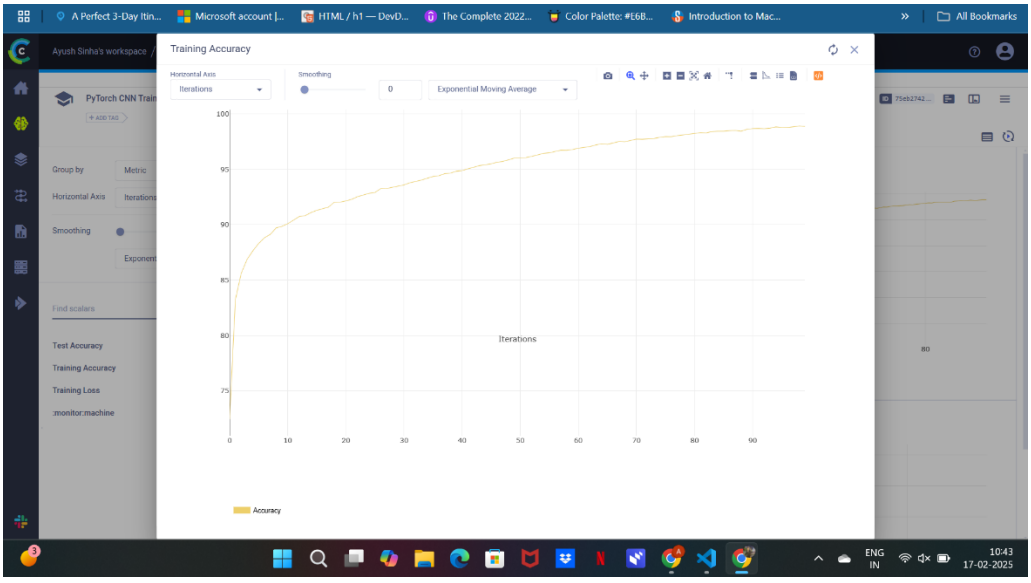



Fig 8. Training Accuracy

- Test Accuracy: Measured accuracy over testing data.

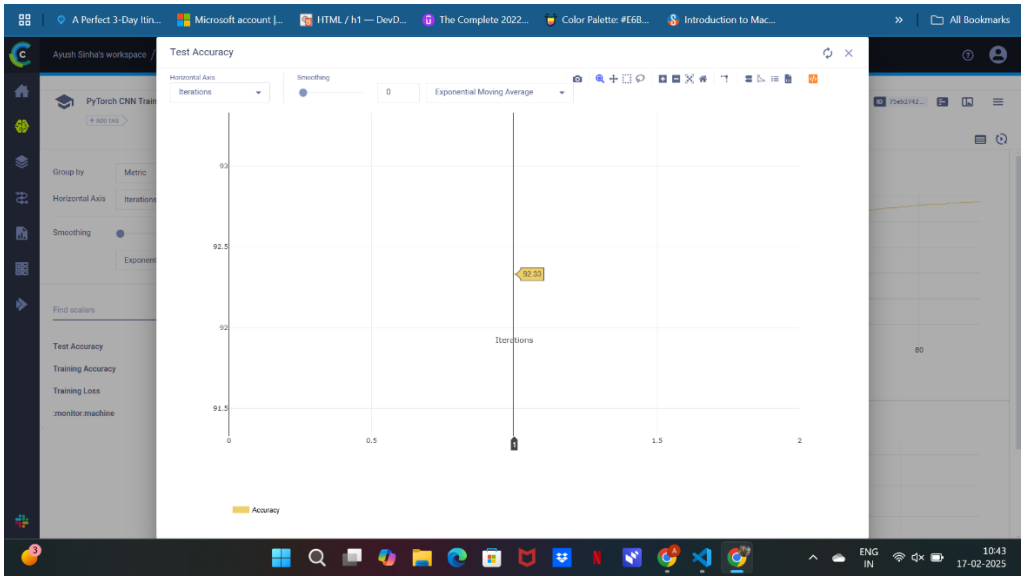



Fig 9. Test Accuracy

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

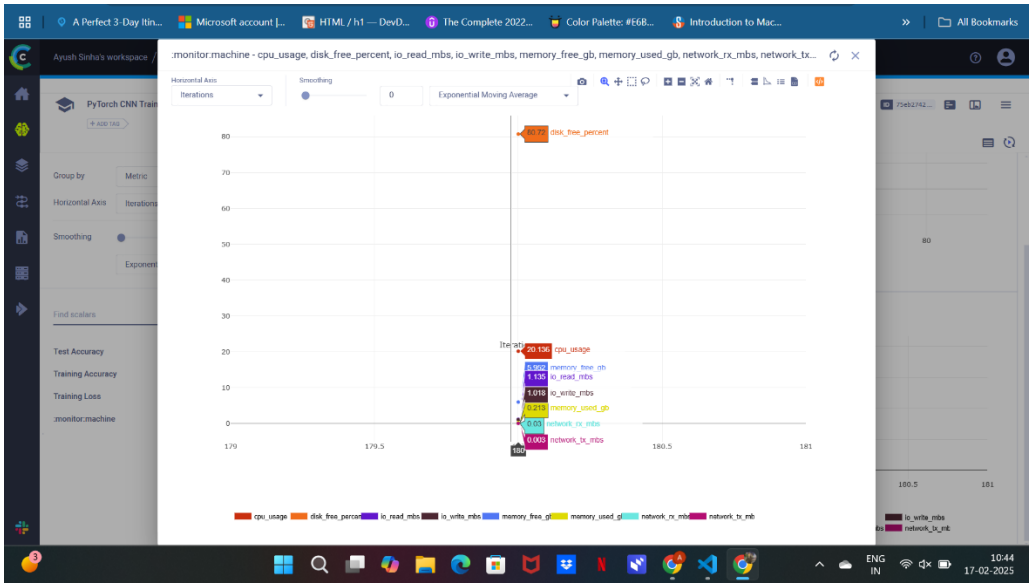



Fig 10. Machine Utilization

# 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.