- Imported the following libraries:
  - NumPy, torch (for Tensors),
  - torch.nn (Neural Network layers),
  - torch.optim (optimizers and improving model parameters),
  - torch.utils.data (for data handling),
  - torchmetrics (for measuring model accuracy),
  - torchvision.datasets (for importing popular vision datasets), and
  - torchvision.transforms (for image processing and data augmentation).
- Built a single callable using tensor.Compose([...]) by chaining several image transforms in order. Done for the **purpose of data augmentation and changing PIL images to tensors**.
- Applied tensor.Compose([...]) to both train and test subsets of the FashionMNIST dataset, a de-factor dataset that contains 10 categories of clothing.
- Identified several other variables before proceeding:
  - Classes = train_data.classes
  - Num_classes = len(classes)
  - num_input_channels = 1: As FashionMNIST images are greyscale.
  - Num_output_channels = 16: Model Choice, number of feature maps (i.e. outputs of convolutional operator between feature detector and original images) each convolutional layer will produce
  - Image_size = train_data[0][0].shape[1]: to extract **height and width** value of (any) training sample/image.
- Declared a PyTorch model class which inherits from **nn.Module** (the base class for all neural network models).
  - Within this class, I implemented the **constructor**, which is a special method in Python automatically called when a new object of the class is created. This constructor defines the architecture of the model by:
    - Receiving num_classes as an input argument, which determines how many output scores (logits) the model will produce (e.g., 10 for Fashion-MNIST).
    - Calling super().__init__(), which ensures that all internal machinery of nn.Module is properly initialized, including parameter tracking, registration of layers, and enabling save/load functionality.
    - Declaring and initializing all layers (convolutional, activation, pooling, flattening, and fully connected) that will be used during the forward pass.
- **Defined the forward pass** (forward method), which specifies how input data is transformed step by step through the network:
  - Passes input images through convolutional layers to extract hierarchical spatial features.
  - Applies activation functions (e.g., ReLU) to introduce non-linearity and enable the model to learn complex patterns and to avoid overfitting.
  - Uses pooling layers to downsample feature maps, reducing computational complexity while retaining essential information.
  - Flattens the high-level feature maps into a vector suitable for classification
  - Processes the flattened vector through fully connected layers to produce class logits corresponding to num_classes.
- **Engineered an efficient data handling pipeline**

    - Utilized DataLoader to batch and shuffle training samples, improving training throughout and model generalization.
    - Ensured compatibility between dataset structure and model input through proper preprocessing and dimensional alignment.
- **Implemented a robust training loop**
    - Defined CrossEntropyLoss as the objective function for multi-class classification tasks.
    - Applied the Adam optimizer with carefully chosen hyperparameters (e.g., learning rate = 0.001) for stable and fast convergence.
    - Automated the forward pass, loss computation, gradient backpropagation, and parameter updates across multiple epochs.
    - Integrated runtime loss tracking (running_loss) to monitor learning progress and detect under/overfitting trends.
- **Optimized for maintainability and experimentation**
    - Encapsulated the entire training procedure in a reusable train_model function to simplify future model iterations.
    - Facilitated model scaling by parameterizing class count (num_classes) and epoch control (num_epochs)
- **Built a performance evaluation pipeline for trained deep learning models**
    - Configured a DataLoader for the test dataset with controlled batching (batch size = 10) and sequential sampling (shuffle=False) to ensure consistent evaluation order.
    - Switched the model to evaluation mode (net.eval()) to deactivate dropout/batch norm updates and guarantee stable inference.
- **Integrated class-wise and overall performance metrics**
    - Implemented accuracy measurement using Accuracy(task='multiclass') to assess global classification correctness.
    - Added Precision and Recall metrics with class-wise granularity (average=None) to diagnose per-class strengths and weaknesses.
- **Automated batched inference and prediction aggregation**
    - Performed forward passes on each batch with input reshaping to match model requirements (features.reshape(-1, 1, image_size, image_size)).
    - Extracted class predictions using torch.argmax, extending results into a consolidated prediction list for later analysis.
- **Computed and reported final evaluation statistics**
    - Aggregated computed metrics via .compute() calls, converting them into Python-native types for reporting and further processing.
    - Produced summary outputs including overall accuracy and per-class precision/recall, enabling interpretable model performance benchmarking.