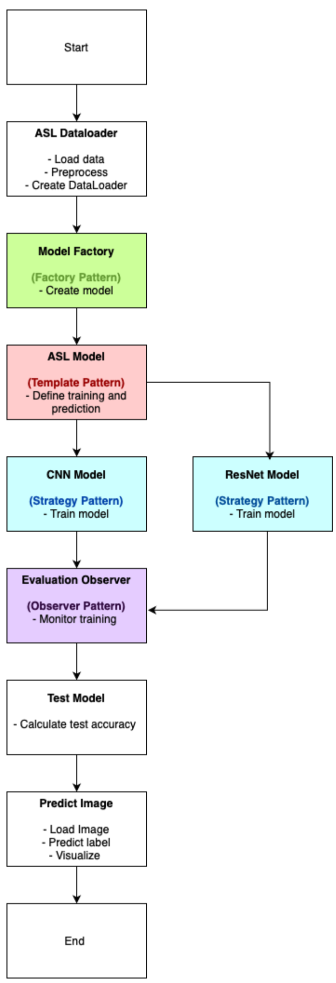
Application or use cases of various design patterns - Software design

A project by Maralgul Korpeyeva,
Franz Josef Diaz,
Sreeneha Samudrala Snighdha

Application Idea:

Sign language detection:
The objective is to develop an ML model that detects Sign language gestures from pictures and classifies what letter it translates to in ASL. Core functionalities of our application include Data preprocessing for the images, training Multiple ML models to classify the images, create a Pipeline for evaluation and testing of the model, Load and Visualize the data and results. You can find below the architecture of our application in detail.

Application Design Flowchart



Implementation:

Data Loader:

```
class ASLDataLoader():
    # DataLoader class for loading and preprocessing ASL dataset
    train_path = "/content/drive/MyDrive/asl/sign_mnist_train.csv"
    test_path = "/content/drive/MyDrive/asl/sign_mnist_test.csv"
    batch_size = 64
    def _init_(self, train_path, test_path, batch_size):
      # initializing paths and batch size
        self.train_path = train_path
        self.test_path = test_path
        self.batch_size = batch_size

    def load_data(self):
      # loading dataset from csv file
        df = pd.read_csv(self.train_path)
        df_test = pd.read_csv(self.test_path)
        return df, df_test

    def preprocess_data(self, df, df_test):
      # reshaping  normalising and splitting the data
        X = df.drop('label', axis=1).values
        X = X.reshape(-1, 1, 28, 28).astype('float32') / 255.0
        y = df['label'].values

        X_test = df_test.drop('label', axis=1).values
        X_test= X_test.reshape(-1, 1, 28, 28).astype('float32') / 255.0
        y_test = df_test['label'].values

        # splitting the data into training and validation sets
        X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)

        return (X_train, y_train), (X_val, y_val), (X_test, y_test)

    def create_dataloaders(self):
      # craeting dataloader onjects for training, validation and test set
        df, df_test = self.load_data()
        (X_train, y_train), (X_val, y_val), (X_test, y_test) = self.preprocess_data(df, df_test)

        train_data = TensorDataset(torch.from_numpy(X_train), torch.from_numpy(y_train))
        val_data = TensorDataset(torch.from_numpy(X_val), torch.from_numpy(y_val))
        test_data = TensorDataset(torch.from_numpy(X_test), torch.from_numpy(y_test))

        train_loader = DataLoader(train_data, batch_size=self.batch_size, shuffle=True)
        val_loader = DataLoader(val_data, batch_size=self.batch_size, shuffle=True)
        test_loader = DataLoader(test_data, batch_size=self.batch_size, shuffle=True)

        return train_loader, val_loader, test_loader
```

Data Visualizer:

```
# class to visualize the images with asl
class visualisation_asl():
  def load_image(self, image_path):
        transform = transforms.Compose([
            transforms.Grayscale(num_output_channels=1),
            transforms.Resize((28, 28)),
            transforms.ToTensor(),
            transforms.Normalize((0.5,), (0.5,))
        ])
        image = Image.open(image_path)
        image = transform(image) #transforming data
        image = image.unsqueeze(0)  # adding batch dimension (batch size = 1)
        image = image.to(device)  # ensuring  the image is on the correct device
        return image

  def plot_image(self, image, label):
        image = image.cpu() # Move the tensor to CPU before converting to NumPy
        image = np.squeeze(image.numpy())  # Convert tensor to NumPy array
        plt.figure(figsize=(1.5, 3))
        plt.imshow(image, cmap='gray')
        plt.title(f"Label: {label}")
        plt.axis('off')
        plt.show()

visualisation = visualisation_asl()
```

Template:

```python
# Template Pattern: Abstarct base class for asl model
class ASLModel(ABC, nn.Module):
    def __init__(self):
        super().__init__()

    @abstractmethod
    def forward(self, x):
      # abstract method for forward pass
        pass

    @abstractmethod
    def train_model(self, train_loader, criterion, optimizer, epochs):
      # abstract method for training model
        pass

    def predict(self, image):
      #method for making predictions
        self.eval()
        with torch.no_grad():
            return self.forward(image)
```

Factory:

```python
# Factory Pattern: Model Factory fro selecting model type : either CNN or ResNet
class ModelFactory:
    @staticmethod
    def get_model(model_type):
        if model_type == "CNN":
            return CNNModel()
        elif model_type == "ResNet":
            return ResNetModel()
        else:
            raise ValueError("Unknown model type")
```

Strategy:

Strategy1:

```python
# Strategy Pattern: Different model implementation - CNN
class CNNModel(ASLModel):
    def __init__(self):
        super().__init__()
        # defining cnn layers
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(64 * 7 * 7, 256)
        self.fc2 = nn.Linear(256, 25)
        self.pool = nn.MaxPool2d(2, 2)
        self.dropout = nn.Dropout(0.5)


    def forward(self, x):
      # defining forward pass
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 64 * 7 * 7)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x

    def train_model(self, train_loader,criterion, optimizer, num_epochs,device, observer=None):
      #training function
        train_losses, val_losses, val_accuracy = [], [],[]
        for epoch in range(num_epochs):
          running_loss = 0.0

          # train loop
          for images, labels in train_loader:
              images, labels = images.to(device), labels.to(device)
              # zero the parameter gradients
              optimizer.zero_grad()
              # forward pass
              outputs = model(images)
              # calculating the loss
              loss = criterion(outputs, labels)
              #backward pass and optimization
              loss.backward()
              optimizer.step()

              running_loss += loss.item()

          # calculating and storing the average training loss for the current epoch
          train_loss = running_loss / len(train_loader)
          train_losses.append(train_loss)

          # Validation loop
          val_running_loss = 0.0
          # disabling gradient computation during validation
          with torch.no_grad():
            # iterating through validation data
              total = 0
              correct = 0
              for images, labels in val_loader:
                  images, labels = images.to(device), labels.to(device)
                  outputs = model(images)
                  # calculating validation loss
                  _, predicted = torch.max(outputs.data, 1)
                  total += labels.size(0)
                  correct += (predicted == labels).sum().item()
                  val_running_loss += loss.item()

          # calculating and storing the average validation loss for the current epoch
          val_loss = val_running_loss / len(val_loader)
          val_accuracy = 100 * correct / total
          val_losses.append(val_loss)

          # implementing observer pattern for logging results
          if observer:
            observer.update(epoch, val_loss,val_accuracy)

        # implementing observer pattern for graph visualisation
        observer.train_val_graph(val_losses, train_losses)


    def predict(model, image):
         model.eval()
         with torch.no_grad():
          image = image.to(device)
          outputs = model(image)
          _ , predicted = torch.max(outputs, 1)
         return predicted.item()
```

Strategy 2:

```python
# Strategy Pattern: Different model implementation - Resnet
class ResNetModel(ASLModel):
    def __init__(self):
      #using pretrained resnet18 model
        super().__init__()
        self.model = models.resnet18(pretrained=True)
        # altering first layer of trained model to adjust to the data classes and grayscale input
        self.model.conv1 = nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3, bias=False)
        self.model.fc = nn.Linear(self.model.fc.in_features, 25)
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        x = self.model(x)
        return x

    def train_model(self, train_loader, criterion, optimizer, num_epochs, device, observer=None):
        train_losses, val_losses, val_accuracies = [], [], []

        for epoch in range(num_epochs):
            running_loss = 0.0

            # training loop
            self.train()
            for images, labels in train_loader:
                images, labels = images.to(device), labels.to(device)
                optimizer.zero_grad()
                outputs = self.forward(images)
                loss = criterion(outputs, labels)
                loss.backward()
                optimizer.step()
                running_loss += loss.item()

            train_loss = running_loss / len(train_loader)
            train_losses.append(train_loss)
```

```python
            # validation loop
            val_running_loss = 0.0
            correct, total = 0, 0
            self.eval()
            with torch.no_grad():
                for images, labels in val_loader:
                    images, labels = images.to(device), labels.to(device)
                    outputs = self.forward(images)
                    loss = criterion(outputs, labels)
                    _, predicted = torch.max(outputs.data, 1)
                    total += labels.size(0)
                    correct += (predicted == labels).sum().item()
                    val_running_loss += loss.item()

            val_loss = val_running_loss / len(val_loader)
            val_accuracy = 100 * correct / total
            val_losses.append(val_loss)
            val_accuracies.append(val_accuracy)

            # implementing observer pattern for logging results
            if observer:
                observer.update(epoch, val_loss, val_accuracy)

        # implementing observer pattern for graph visualisation
        observer.train_val_graph(val_losses, train_losses)

    def predict(model, image):
         model.eval()
         with torch.no_grad():
          image = image.to(device)
          outputs = model(image)
          _ , predicted = torch.max(outputs, 1)
         return predicted.item()
```

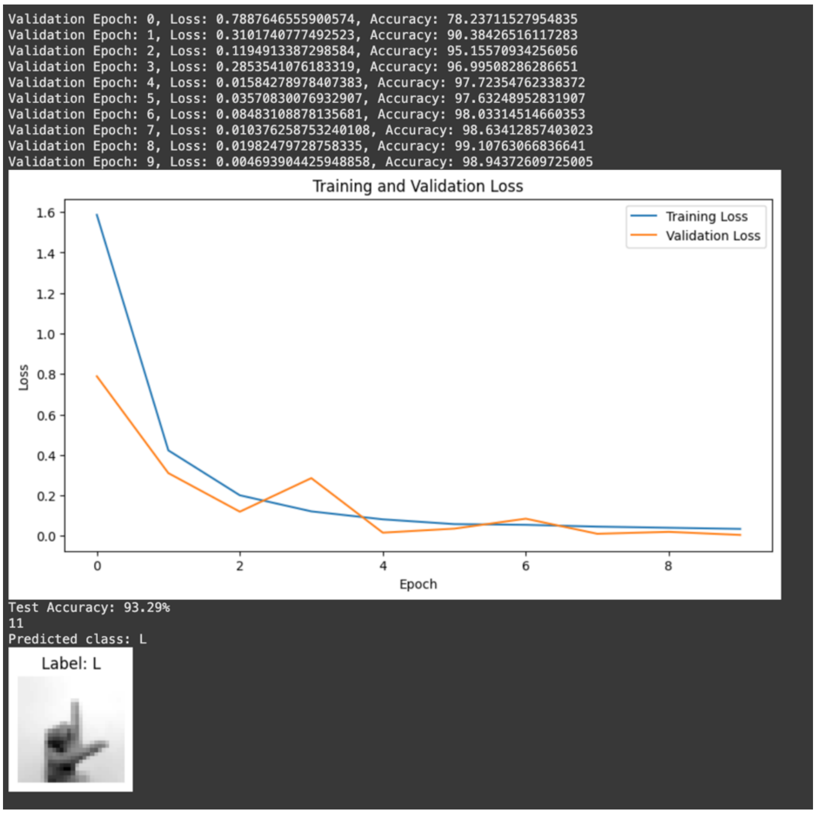
Observer:

```python
# Observer Pattern: Training observer to monitor model training and validation
class EvalObserver:
    def update(self, epoch, loss,accuracy):
      # printing validation loss and accuracy per epoch
        print(f"Validation Epoch: {epoch}, Loss: {loss}, Accuracy: {accuracy}")

    def train_val_graph(self,val_losses, train_losses):
      # plotting training and validation loss over epochs
      plt.figure(figsize=(10,5))
      plt.plot(train_losses, label='Training Loss')
      plt.plot(val_losses, label='Validation Loss')
      plt.xlabel('Epoch')
      plt.ylabel('Loss')
      plt.title('Training and Validation Loss')
      plt.legend()
      plt.show()


observer = EvalObserver()
```

Main function: (here the choice was “CNN”)

```python
# running the ASL model training and evaluation
if __name__ == "__main__":
    # initialising the asl data loader
    data_loader = ASLDataLoader()

    # creating DataLoaders for training, validation, and testing
    train_loader, val_loader, test_loader = data_loader.create_dataloaders()

    # initiating the model using the factory pattern
    model = ModelFactory.get_model("CNN")

    # defining optimizer and loss function
    optimizer = optim.Adam(model.parameters(), lr=0.001)
    criterion = nn.CrossEntropyLoss()
    model.to(device)

     # training  the model with specified parameters
    model.train_model(train_loader, criterion, optimizer, num_epochs=10,device=device, observer=observer)

     # evaluating the model on the test dataset
    test_model()

     # loading an example image for prediction
    image_path = '/content/drive/MyDrive/asl/sign_images/L.jpeg'
    letter = image_path.split("/")[-1].split(".")[0]

    # loading the image using visualization utilities
    image = visualisation.load_image(image_path)

    # predicting the class label of the image using the trained model
    prediction = model.predict(image)
    print(f'Predicted class: {alph_dict[prediction]}')

     # displaying the image along with its true label
    visualisation.plot_image(image, letter)
```

Results:



Link to the code: https://colab.research.google.com/drive/1FjCFQeeIDC-mCgkGsMYmr7BLdra3Y8zF?usp=sharing
Link to the data: https://www.kaggle.com/datasets/datamunge/sign-language-mnist

Reflection on Design pattens Implemented in our application

For our project, we chose to develop a machine learning recognition system because of its practical relevance and the opportunity it provided to apply various software design patterns in a real-world scenario.
We started by exploring different datasets on Kaggle (online platform for ML datasets), looking for a problem that was both technically challenging and engaging. After browsing through multiple datasets, we found this project aligning well with our goal of enhancing our understanding of machine learning workflows while implementing robust software design patterns. The use of design patterns helped us improve code reusability, manage complexity and ensure flexibility in our implementation.

The application makes use of three behavioral design patterns which are Template, Strategy, and Observer, along with one creational design pattern which is Factory.

Factory was chosen as the first design pattern to be implemented in the main function were based on user input, it dynamically calls the respective model for training for the strategy pattern.

The strategy pattern then allows different machine learning algorithms to be used interchangeably. In this specific application, the choice between the use of CNN and Resnet was implemented with the help of strategy pattern.

Template pattern then gave a defined structure for training pipeline which is then called by each of the model. With the uniformity offered across all the models, it gets easier to maintain the pipeline when the strategies are used interchangeably.

Lastly, the observer pattern allows observers to respond to changes in another object without having to modify the factory code.

All the design patterns help to build and maintain a modular code which is helpful when creating a machine learning application with multiple algorithms. It removes unnecessary lines of codes and makes sure that the code can be used interchangeably.

Conventional Machine Learning Pipeline	**Machine Learning Pipeline with implemented Design Patterns**
Each model has its own training and evaluation code which introduces unnecessary code duplication. A new model will mean a new training loop.	The **Template Pattern** gives a reusable block of code for training and evaluation. New model introduced will only need new blocks of specific methods.
Models are usually coded in different notebooks and are hardcoded.	The **Strategy Pattern** allows different models to be interchangeable depending on which model is to be called by the user.
Training configurations are usually not as organized in the codebase and models are manually instantiated with specific configurations.	The **Factory Pattern introduces** a main function which ensures that hyperparameters and configurations are managed centrally.
Evaluation and debugging are usually implemented in different parts of the code as and when necessary.	The **Observer Pattern** gives a structured pattern to track metrics and triggers updates when necessary.
Future enhancements with optimizers and preprocessing should be implemented in different codebases.	The combination of all the design patterns offers modularity which makes it easier to enhance the code without having to implement it in different codebases.

Challenges:
Some of the Challenges we faced while designing our application were choosing suitable combination of design patterns for our use case since, we were working on an ML based use case. Another challenge we faced was adding adapting our application design from a 3-pattern strategy to a 4-pattern model as suggested by our professors. Another hurdle we came across while incorporating the design pattern was

Reorganizing base code into the suitable category for example, into classes and functions and choosing the order ang grouping of the functions into respective strategies.
The challenges we faced with respect to the implementation of code for our application is trying to identify what display commands fall under the observer and how to correctly implement the observer pattern in the training loop. We also faced a confusion as to if all the visualization or results falls under the observer pattern, but we concluded that it doesn’t as it does not get altered by each iteration of the training process. Lastly, we faced issues with the prediction function because of the model and data not being on the same device (GPU/CPU).

Conclusion:

In conclusion, incorporating the design patterns in the process of creating our application helped us organize, restructure our code. It also helped us create a flow in methodology during implementation of our project while also providing clarity on various components that can be reused to avoid redundancy. We also learned about the different kind of design patterns that can be used to specific use cases and how they complement each other. On a whole, it helped us understand the importance of writing clean, readable and executable code that is very efficient in performing the task it is designed for.