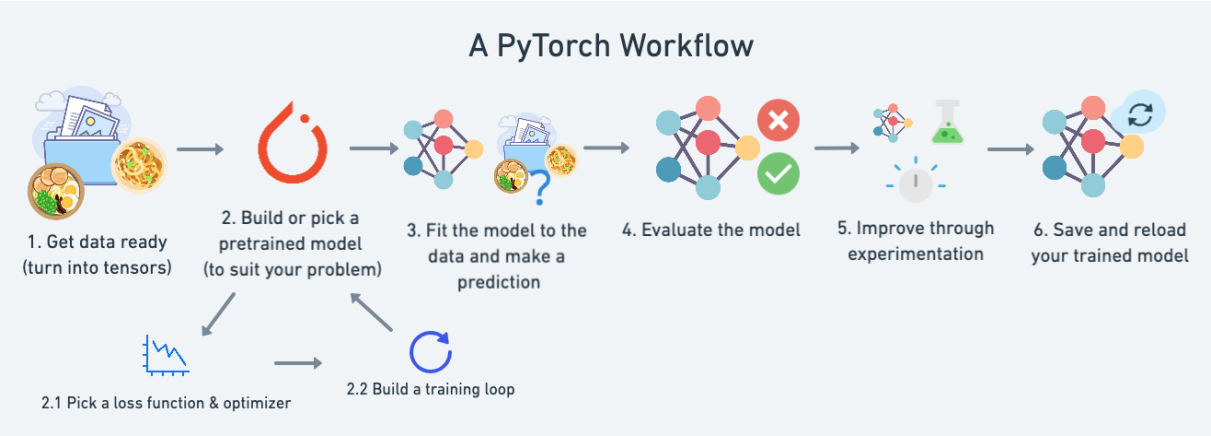
Practical class 2

Pytorch workflow

The blow diagram visually summarizes the steps involved in a Pytorch machine learning workflow, which include:



1. **Get Data Ready** – Load and preprocess your data, then convert it into tensors, which are the fundamental data structure in PyTorch.
2. **Build or Pick a Model** – Design a neural network architecture or select a pretrained model depending on your problem.
    - **Pick a Loss Function & Optimizer** – Choose an appropriate loss function (e.g., cross-entropy for classification) and an optimizer (e.g., SGD, Adam) to update model weights during training.
    - **Build a Training Loop** – Create a loop to process batches of data, perform forward and backward passes, calculate loss, and update weights.
3. **Fit the Model & Make Predictions** – Run the training loop to fit your model to the training data and make predictions.
4. **Evaluate the Model** – Test the model on separate validation or test data to measure performance metrics like accuracy.
5. **Improve through Experimentation** – Tune hyperparameters, modify the model architecture, or experiment with different optimizers or loss functions to improve performance.
6. **Save and Reload the Model** – Save the trained model for future use, enabling you to reload it without retraining.

Problem: Predicting `y` values from `x` using the equation y=w·x+by = w \cdot x + by=w·x+b where `w` (weight) and `b` (bias) are parameters that we want to learn.

1. **Import Libraries**

```
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
```

2. **Create Sample Data**

Let's create some synthetic data that follows a linear pattern with some noise.

```
# Generate synthetic data: y = 2 * x + 1 + noise
```

```
torch.manual_seed(0)
X = torch.linspace(0, 10, 100).reshape(-1, 1)  # Inputs (100 points)
Y = 2 * X + 1 + torch.randn(100, 1) * 2  # Outputs with some added noise
```

3. **Define the Model**
   In PyTorch, we define a linear regression model as a single layer with no activation.

```
class LinearRegressionModel(nn.Module):
    def __init__(self):
        super(LinearRegressionModel, self).__init__()
        self.linear = nn.Linear(1, 1)  # Single input and single output

    def forward(self, x):
        return self.linear(x)

model = LinearRegressionModel()
```

4. **Set the Loss Function and Optimizer**
   - Loss Function: Mean Squared Error (MSE) is commonly used for regression.
   - Optimizer: Stochastic Gradient Descent (SGD) or Adam are popular choices.

```
criterion = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

5. **Training Loop**
   Train the model by adjusting the parameters (`w` and `b`) to minimize the loss.

```
num_epochs = 1000
for epoch in range(num_epochs):
    # Forward pass: Compute prediction and loss
    predictions = model(X)
    loss = criterion(predictions, Y)

    # Backward pass: Compute gradients and update parameters
    optimizer.zero_grad()  # Zero the gradients
    loss.backward()       # Compute gradients
    optimizer.step()      # Update parameters

    # Print loss every 100 epochs
    if (epoch+1) % 100 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')
```

6. **Evaluate the Model**
   After training, we can visualize the fitted line against the data.

```
# Plot the data and the model's predictions
with torch.no_grad():
    predicted = model(X).detach()  # Get predictions
```

```
plt.scatter(X.numpy(), Y.numpy(), color='blue', label='Original Data')
plt.plot(X.numpy(), predicted.numpy(), color='red', label='Fitted Line')
plt.legend()
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```

7. **Save the Model (Optional)**

```
torch.save(model.state_dict(), 'linear_regression_model.pth')
```

Explanation

- **Data Generation**: We created a simple linear relationship with some noise: $y=2 \cdot x+1+\text{noise}$
- **Model Definition**: The model is a single linear layer, which is enough for basic linear regression.
- **Training**: The model is trained over 1000 epochs, minimizing the mean squared error between predictions and actual data points.
- **Evaluation**: After training, we plot the fitted line to see if it matches the data distribution