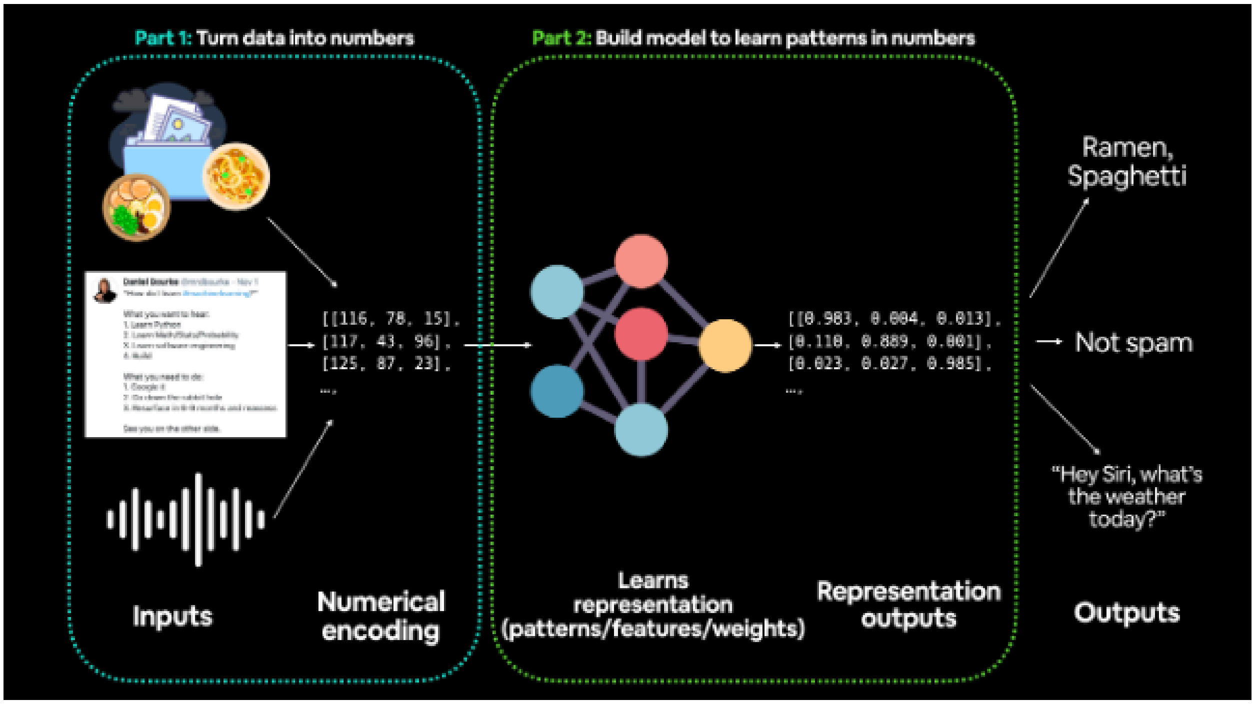
PyTorch Workflow Fundamentals

The essence of machine learning and deep learning is to take some data from the past, build an algorithm (like a neural network) to discover patterns in it and use the discovered patterns to predict the future.

Topic	Contents
1. Getting data ready	Data can be almost anything but to get started we're going to create a simple straight line
2. Building a model	Here we'll create a model to learn patterns in the data, we'll also choose a **loss function**, **optimizer** and build a **training loop**.
3. Fitting the model to data (training)	We've got data and a model, now let's let the model (try to) find patterns in the (**training**) data.
4. Making predictions and evaluating a model (inference)	Our model's found patterns in the data, let's compare its findings to the actual (**testing**) data.
5. Saving and loading a model	You may want to use your model elsewhere, or come back to it later, here we'll cover that.
6. Putting it all together	Let's take all of the above and combine it.

1. Data (preparing and loading)

I want to stress that "data" in machine learning can be almost anything you can imagine. A table of numbers (like a big Excel spreadsheet), images of any kind, videos (YouTube has lots of data!), audio files like songs or podcasts, protein structures, text and more.



Machine learning is a game of two parts:

1. Turn your data, whatever it is, into numbers (a representation).
2. Pick or build a model to learn the representation as best as possible.

Split data into training and test sets

1. *# Create train/test split*
2. *train_split* **=** *int(0.8 * len(X)) # 80% of data used for training set, 20% for testing*
3. *X_train, y_train* **=** *X[:train_split], y[:train_split]*
4. *X_test, y_test* **=** *X[train_split:], y[train_split:]*

Split	Purpose	Amount of total data	How often is it used?
Training set	The model learns from this data (like the course materials you study during the semester).	~60-80%	Always
Validation set	The model gets tuned on this data (like the practice exam you take before the final exam).	~10-20%	Often but not always
Testing set	The model gets evaluated on this data to test what it has learned (like the final exam you take at the end of the semester).	~10-20%	Always

2. Build model

```
# Create a Linear Regression model class
class LinearRegressionModel(nn.Module): # <- almost everything in PyTorch is a nn.Module (think of this as neural network lego blocks)
    def __init__(self):
        super().__init__()
        self.weights = nn.Parameter(torch.randn(1, # <- start with random weights (this will get adjusted as the model learns)
                                                dtype=torch.float), # <- PyTorch loves float32 by default
                                    requires_grad=True) # <- can we update this value with gradient descent?)

        self.bias = nn.Parameter(torch.randn(1, # <- start with random bias (this will get adjusted as the model learns)
                                             dtype=torch.float), # <- PyTorch loves float32 by default
                                 requires_grad=True) # <- can we update this value with gradient descent?))

    # Forward defines the computation in the model
    def forward(self, x: torch.Tensor) -> torch.Tensor: # <- "x" is the input data (e.g. training/testing features)
        return self.weights * x + self.bias # <- this is the linear regression formula (y = m*x + b)
```

PyTorch model building essentials

PyTorch has four (give or take) essential modules you can use to create almost any kind of neural network you can imagine.

They are `torch.nn`, `torch.optim`, `torch.utils.data.Dataset` and `torch.utils.data.DataLoader`.

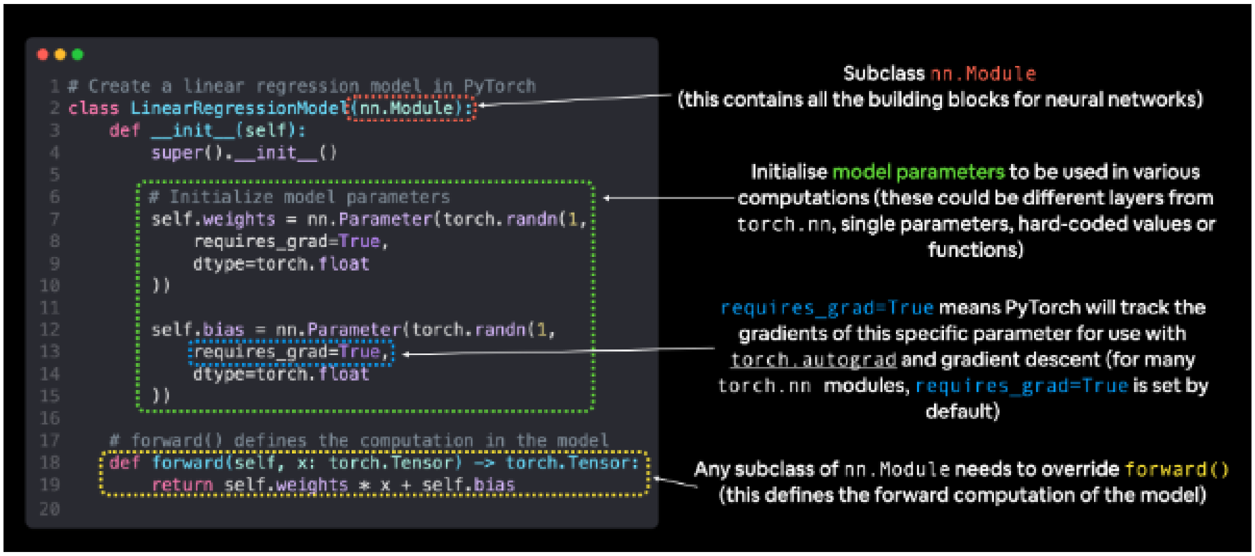
PyTorch module	What does it do?
`torch.nn`	Contains all of the building blocks for computational graphs (essentially a series of computations executed in a particular way).
`torch.nn.Parameter`	Stores tensors that can be used with `nn.Module`. If `requires_grad=True` gradients (used for updating model parameters via **gradient descent**) are calculated automatically, this is often referred to as "autograd".
`torch.nn.Module`	The base class for all neural network modules, all the building blocks for neural networks are subclasses. If you're building a neural network in PyTorch, your models should subclass `nn.Module`. Requires a `forward()` method be implemented.
`torch.optim`	Contains various optimization algorithms (these tell the model parameters stored in `nn.Parameter` how to best change to improve gradient descent and in turn reduce the loss).
`def forward()`	All `nn.Module` subclasses require a `forward()` method, this defines the computation that will take place on the data passed to the particular `nn.Module` (e.g. the linear regression formula above).

`nn.Module` contains the larger building blocks (layers)

`nn.Parameter` contains the smaller parameters like weights and biases (put these together to make `nn.Module`(s))

`forward()` tells the larger blocks how to make calculations on inputs (tensors full of data) within `nn.Module`(s)

`torch.optim` contains optimization methods on how to improve the parameters within `nn.Parameter` to better represent input data



Checking the contents of a PyTorch model

Check the nn.Parameter(s) within the nn.Module subclass we created

list(model_0.parameters())

We can also get the state (what the model contains) of the model using `.state_dict()`.

List named parameters

model_0.state_dict()

Making predictions using torch.inference_mode()

1. *# Make predictions with model*
2. ***with** torch.inference_mode():*
3. *y_preds **=** model_0(X_test)*
4.
5. *# Note: in older PyTorch code you might also see torch.no_grad()*
6. *# with torch.no_grad():*
7. *#   y_preds = model_0(X_test)*

`torch.inference_mode()` turns off a bunch of things (like gradient tracking, which is necessary for training but not for inference) to make **forward-passes** (data going through the `forward()` method) faster.

3. Train model

Creating a loss function and optimizer in PyTorch

Function	What does it do?	Where does it live in PyTorch?	Common values
Loss function	Measures how wrong your models predictions (e.g. `y_preds`) are compared to the truth labels (e.g. `y_test`). Lower the better.	PyTorch has plenty of built-in loss functions in `torch.nn`.	Mean absolute error (MAE) for regression problems (`torch.nn.L1Loss()`). Binary cross entropy for binary classification problems (`torch.nn.BCELoss()`).
Optimizer	Tells your model how to update its internal parameters to best lower the loss.	You can find various optimization function implementations in `torch.optim`.	Stochastic gradient descent (`torch.optim.SGD()`). Adam optimizer (`torch.optim.Adam()`).

```
# Create the loss function
loss_fn = nn.L1Loss() # MAE loss is same as L1Loss

# Create the optimizer
optimizer = torch.optim.SGD(params=model_0.parameters(), # parameters of target model to optimize
                            lr=0.01) # learning rate (how much the optimizer should change parameters at each step, higher=more (less stable), lower=less (might take a long time))
```

Creating an optimization loop in PyTorch

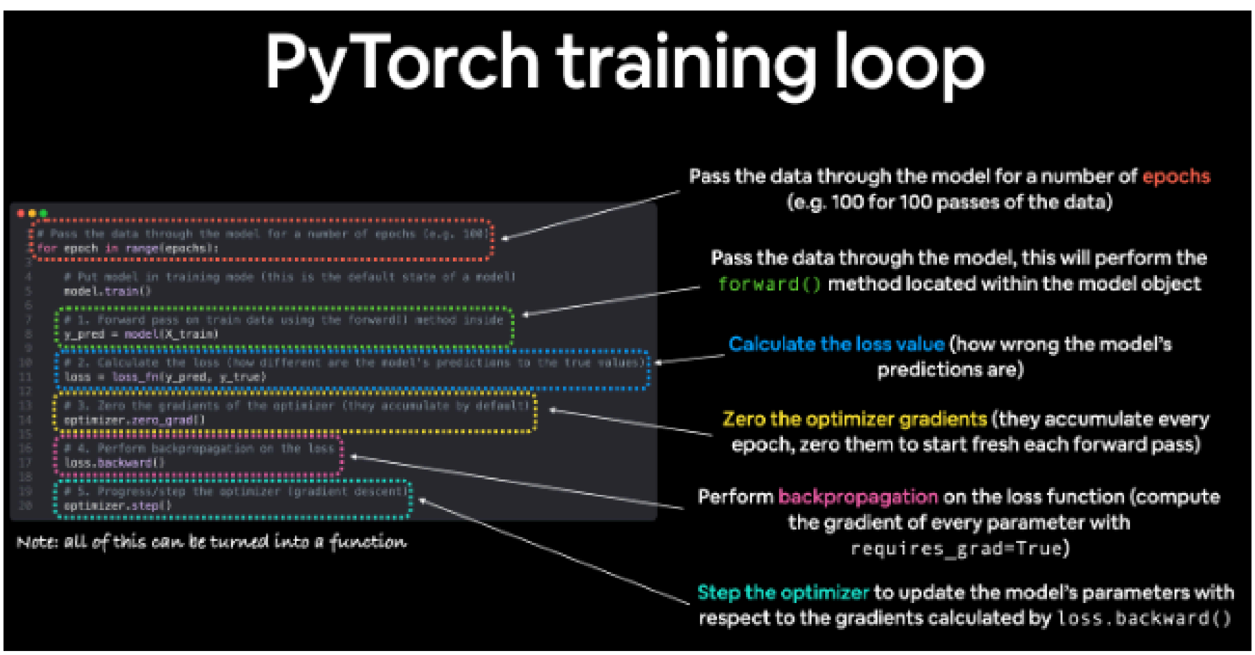
The training loop involves the model going through the training data and learning the relationships between the `features` and `labels`.

The testing loop involves going through the testing data and evaluating how good the patterns are that the model learned on the training data (the model never see's the testing data during training).

Each of these is called a "loop" because we want our model to look (loop through) at each sample in each dataset.

PyTorch training loop

Number	Step name	What does it do?	Code example
1	Forward pass	The model goes through all of the training data once, performing its `forward()` function calculations.	`model(x_train)`
2	Calculate the loss	The model's outputs (predictions) are compared to the ground truth and evaluated to see how wrong they are.	`loss = loss_fn(y_pred, y_train)`
3	Zero gradients	The optimizers gradients are set to zero (they are accumulated by default) so they can be recalculated for the specific training step.	`optimizer.zero_grad()`
4	Perform backpropagation on the loss	Computes the gradient of the loss with respect for every model parameter to be updated (each parameter with `requires_grad=True`). This is known as **backpropagation**, hence "backwards".	`loss.backward()`
5	Update the optimizer (**gradient descent**)	Update the parameters with `requires_grad=True` with respect to the loss gradients in order to improve them.	`optimizer.step()`



PyTorch testing loop

Number	Step name	What does it do?	Code example
1	Forward pass	The model goes through all of the training data once, performing its `forward()` function calculations.	`model(x_test)`
2	Calculate the loss	The model's outputs (predictions) are compared to the ground truth and evaluated to see how wrong they are.	`loss = loss_fn(y_pred, y_test)`
3	Calulate evaluation metrics (optional)	Alongisde the loss value you may want to calculate other evaluation metrics such as accuracy on the test set.	Custom functions

```
torch.manual_seed(42)

# Set the number of epochs (how many times the model will pass over the training data)
epochs = 100

# Create empty loss lists to track values
train_loss_values = []
test_loss_values = []
epoch_count = []

for epoch in range(epochs):
    ### Training

    # Put model in training mode (this is the default state of a model)
    model_0.train()

    # 1. Forward pass on train data using the forward() method inside
    y_pred = model_0(X_train)
    # print(y_pred)

    # 2. Calculate the loss (how different are our models predictions to the ground truth)
    loss = loss_fn(y_pred, y_train)

    # 3. Zero grad of the optimizer
    optimizer.zero_grad()
```

```
    # 4. Loss backwards
    loss.backward()

    # 5. Progress the optimizer
    optimizer.step()

    ### Testing

    # Put the model in evaluation mode
    model_0.eval()

    with torch.inference_mode():
      # 1. Forward pass on test data
      test_pred = model_0(X_test)

      # 2. Calculate loss on test data
      test_loss = loss_fn(test_pred, y_test.type(torch.float)) # predictions come in torch.float datatype, so comparisons need to be done with tensors of the same type

      # Print out what's happening
      if epoch % 10 == 0:
            epoch_count.append(epoch)
            train_loss_values.append(loss.detach().numpy())
            test_loss_values.append(test_loss.detach().numpy())
            print(f"Epoch: {epoch} | MAE Train Loss: {loss} | MAE Test Loss: {test_loss} ")
```

4. Making predictions with a trained PyTorch model (inference)

```
# 1. Set the model in evaluation mode
model_0.eval()

# 2. Setup the inference mode context manager
with torch.inference_mode():
  # 3. Make sure the calculations are done with the model and data on the same device
  # in our case, we haven't setup device-agnostic code yet so our data and model are
  # on the CPU by default.
  # model_0.to(device)
  # X_test = X_test.to(device)
  y_preds = model_0(X_test)
Y_preds
```

5. Saving and loading a PyTorch model

PyTorch method	What does it do?
`torch.save`	Saves a serialized object to disk using Python's `pickle` utility. Models, tensors and various other Python objects like dictionaries can be saved using `torch.save`.
`torch.load`	Uses `pickle`'s unpickling features to deserialize and load pickled Python object files (like models, tensors or dictionaries) into memory. You can also set which device to load the object to (CPU, GPU etc).
`torch.nn.Module.load_state_dict`	Loads a model's parameter dictionary (`model.state_dict()`) using a saved `state_dict()` object.

Saving a PyTorch model's state_dict()

The recommended way for saving and loading a model for inference (making predictions) is by saving and loading a model's `state_dict()`.

Let's see how we can do that in a few steps:

1. We'll create a directory for saving models to called `models` using Python's `pathlib` module.
2. We'll create a file path to save the model to.
3. We'll call `torch.save(obj, f)` where `obj` is the target model's `state_dict()` and `f` is the filename of where to save the model.

```
from pathlib import Path

# 1. Create models directory
MODEL_PATH = Path("models")
MODEL_PATH.mkdir(parents=True, exist_ok=True)

# 2. Create model save path
MODEL_NAME = "01_pytorch_workflow_model_0.pth"
MODEL_SAVE_PATH = MODEL_PATH / MODEL_NAME

# 3. Save the model state dict
print(f"Saving model to: {MODEL_SAVE_PATH}")
torch.save(obj=model_0.state_dict(), # only saving the state_dict() only saves the models learned parameters
```

f=MODEL_SAVE_PATH)

Loading a saved PyTorch model's state_dict()

We only saved the model's `state_dict()` which is a dictionary of learned parameters and not the *entire* model, we first have to load the `state_dict()` with `torch.load()` and then pass that `state_dict()` to a new instance of our model (which is a subclass of `nn.Module`).

 # Instantiate a new instance of our model (this will be instantiated with random weights)
loaded_model_0 = LinearRegressionModel()

 # Load the state_dict of our saved model (this will update the new instance of our model with trained weights)
loaded_model_0.load_state_dict(torch.load(f=MODEL_SAVE_PATH))

6. Build Model (nn.Linear)