# Introduction to PyTorch

## Basics

**Kirill MILINTSEVICH | 16.02.2024**

# About Myself

- 4th year PhD student at GREYC, team IMAGE
- Working with NLP and Mental Health
- Actively using PyTorch since 2019
- Cannot call myself a PyTorch expert, but hopefully good enough to teach you the fundamentals

# About the Course

- Seance 1 (16.02.2024) - PyTorch Basics (tensors, autograd, basic models and training loop, saving/loading models, GPU usage)
- Seance 2 (21.02.2024) - Optimisations (mixed precision training, gradient accumulation and checkpointing, model quantisation, etc.)
- Seance 3 (23.02.2024) - Small Project (building a network from scratch with a dataset of your choice or fine-tuning a 3B-parameter model on a Colab GPU)

# What is PyTorch?

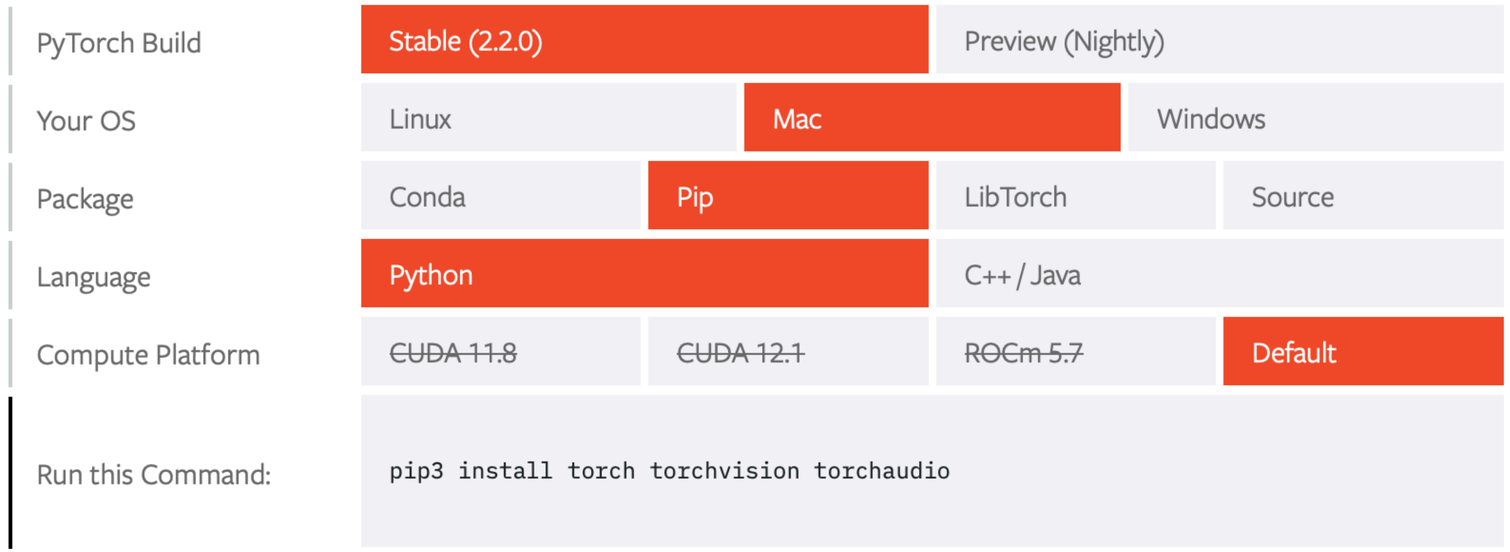
- PyTorch is an open source machine learning and deep learning framework.
- PyTorch is the most used deep learning framework on Papers With Code

# How to Install PyTorch?

## Locally

- Usually very easy, just follow the steps from pytorch.org
- If you have an NVIDIA GPU, you will have to install CUDA to use it

**NOTE:** Latest PyTorch requires Python 3.8 or later. For more details, see Python section below.



# How to Install PyTorch?

## On Google Colab

- Google Colab already has PyTorch pre-installed, so you can use it right away!

# Supported GPU Accelerations

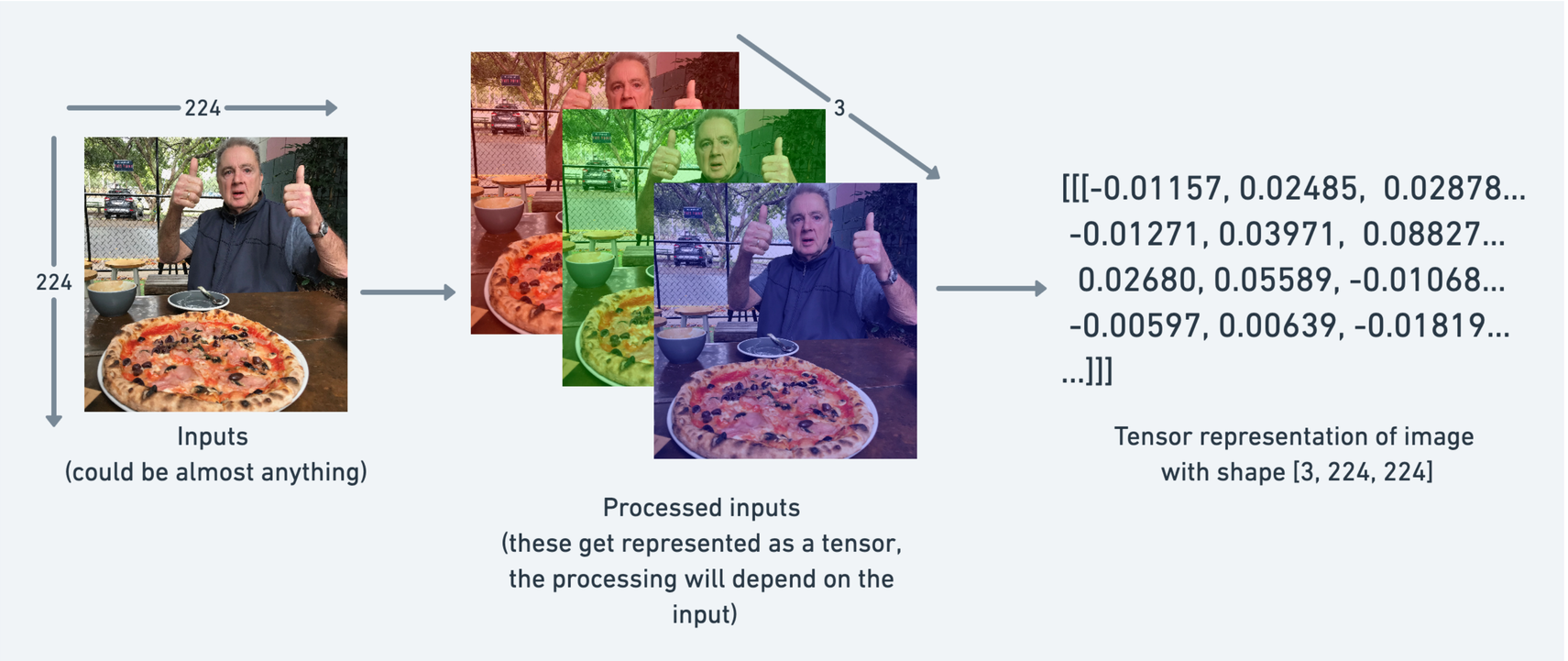
- CUDA (Nvidia)
- ROCm (AMD)
- MPS (Apple Silicon: M1, M2, M3)

# Places to Learn More About PyTorch

- PyTorch Documentation: https://pytorch.org/docs
- PyTorch Tutorials: https://pytorch.org/tutorials/
- Learn PyTorch: https://www.learnpytorch.io

# Tensors

- Tensors are the fundamental building block of machine learning
- Their job is to represent data in a numerical way




Source: https://www.learnpytorch.io

# Tensors

- Very similar to Numpy Arrays (some Tensors even share the data with Numpy)

```
[5]  my_tensor = torch.tensor([[1, 2], [3, 4]])
     print(my_tensor)
     print(my_tensor.size())
```

```
tensor([[1, 2],
        [3, 4]])
torch.Size([2, 2])
```

```
[6]  my_array = np.array([[1, 2], [3, 4]])
     print(my_array)
     print(my_array.shape)
```

```
[[1 2]
 [3 4]]
(2, 2)
```

# Tensors

- You can easily convert between PyTorch Tensors and Numpy Arrays

```
✓ 0s  [7]  torch.tensor(my_array)

      tensor([[1, 2],
              [3, 4]])

✓ 0s  [8]  my_tensor.numpy()

      array([[1, 2],
             [3, 4]])
```

# Tensor-Speak

| Name | What is it? | Number of dimensions | Lower or upper (usually/example) |
|---|---|---|---|
| **scalar** | a single number | 0 | Lower ( `a` ) |
| **vector** | a number with direction (e.g. wind speed with direction) but can also have many other numbers | 1 | Lower ( `y` ) |
| **matrix** | a 2-dimensional array of numbers | 2 | Upper ( `Q` ) |
| **tensor** | an n-dimensional array of numbers | can be any number, a 0-dimension tensor is a scalar, a 1-dimension tensor is a vector | Upper ( `X` ) |

Source: https://www.learnpytorch.io

# Autograd

- Every Tensor can have a gradient graph attached to it
- All Tensors **inside** the models have gradient enabled by default
- Used to calculate the gradients that are used to update the weights of the model during training

```
[15] x = torch.rand(3, 5, requires_grad=True)
     y = torch.rand(5, 1, requires_grad=True)

     print(x)
     print(y)
```

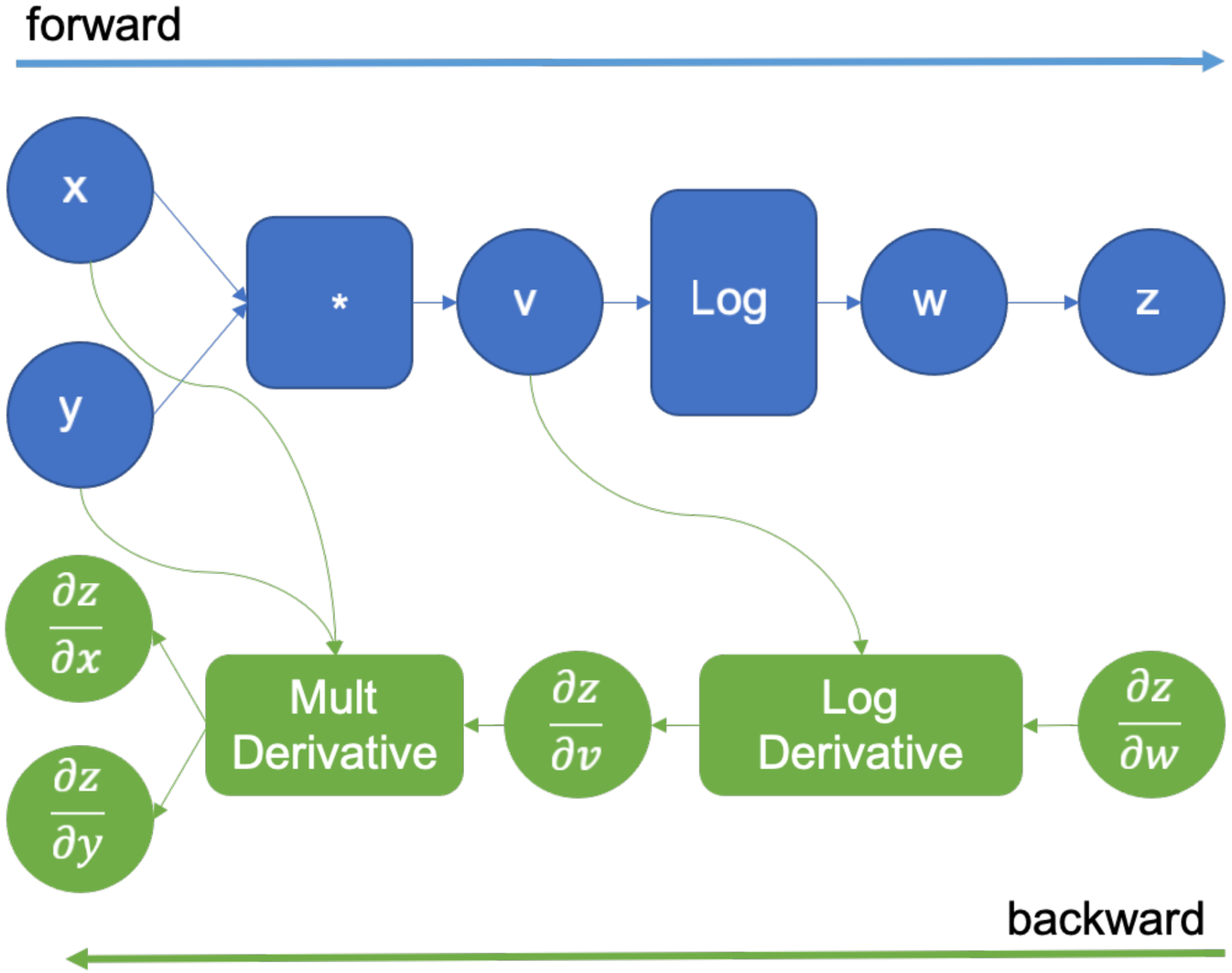
```
tensor([[0.6167, 0.6562, 0.8323, 0.2056, 0.4349],
        [0.2582, 0.2808, 0.2617, 0.4779, 0.8651],
        [0.2775, 0.1364, 0.8954, 0.6896, 0.6597]], requires_grad=True)
tensor([[0.5005],
        [0.2150],
        [0.6382],
        [0.0659],
        [0.6855]], requires_grad=True)
```

```
[16] z = x @ y

     print(z)
```

```
tensor([[1.2925],
        [0.9811],
        [1.2374]], grad_fn=<MmBackward0>)
```

# Autograd




Source: https://pytorch.org/blog/overview-of-pytorch-autograd-engine/

# Building Models

- PyTorch allows easy building of models, almost like stacking LEGO pieces together

```
[17] class TinyModel(torch.nn.Module):

        def __init__(self):
            super(TinyModel, self).__init__()

            self.linear1 = torch.nn.Linear(100, 200)
            self.activation = torch.nn.ReLU()
            self.linear2 = torch.nn.Linear(200, 10)
            self.softmax = torch.nn.Softmax()

        def forward(self, x):
            x = self.linear1(x)
            x = self.activation(x)
            x = self.linear2(x)
            x = self.softmax(x)
            return x

    tinymodel = TinyModel()
```

# Building Models

- First, we list all the building blocks

```
[17] class TinyModel(torch.nn.Module):

        def __init__(self):
            super(TinyModel, self).__init__()

            self.linear1 = torch.nn.Linear(100, 200)
            self.activation = torch.nn.ReLU()
            self.linear2 = torch.nn.Linear(200, 10)
            self.softmax = torch.nn.Softmax()

        def forward(self, x):
            x = self.linear1(x)
            x = self.activation(x)
            x = self.linear2(x)
            x = self.softmax(x)
            return x

    tinymodel = TinyModel()
```

# Building Models

- Then, we define the forward pass

```
[17] class TinyModel(torch.nn.Module):

        def __init__(self):
            super(TinyModel, self).__init__()

            self.linear1 = torch.nn.Linear(100, 200)
            self.activation = torch.nn.ReLU()
            self.linear2 = torch.nn.Linear(200, 10)
            self.softmax = torch.nn.Softmax()

        def forward(self, x):
            x = self.linear1(x)
            x = self.activation(x)
            x = self.linear2(x)
            x = self.softmax(x)
            return x

    tinymodel = TinyModel()
```

# Common Layer Types

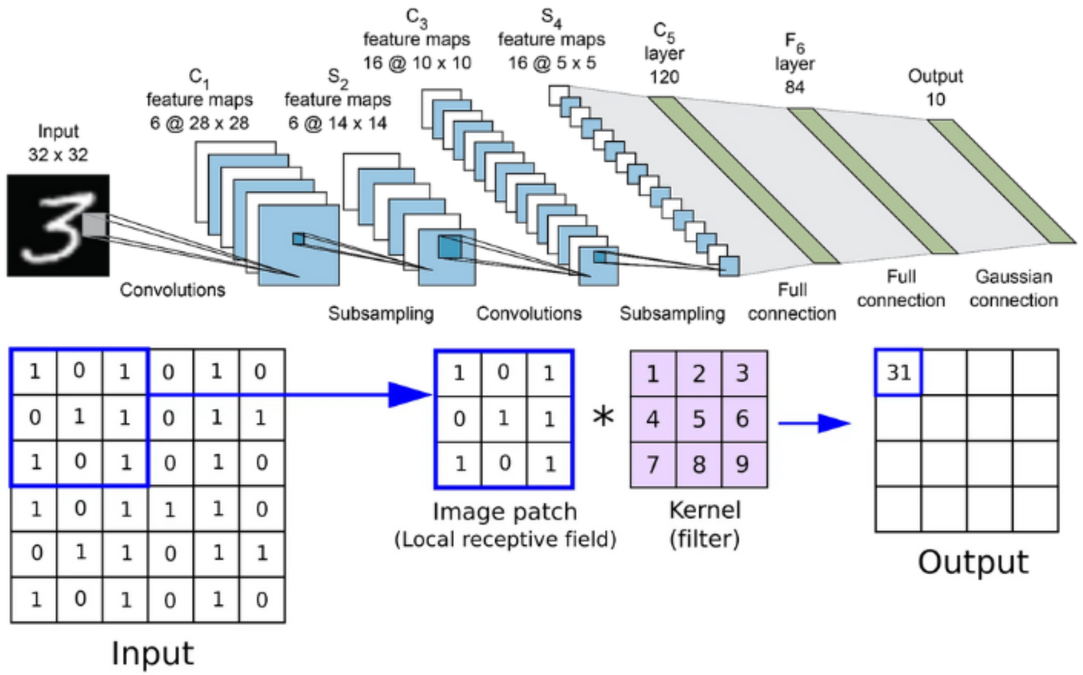
- Linear (or fully connected layer): your classic Ax+b layer

**Artificial Intelligence**

$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x} + b$$

# Common Layer Types

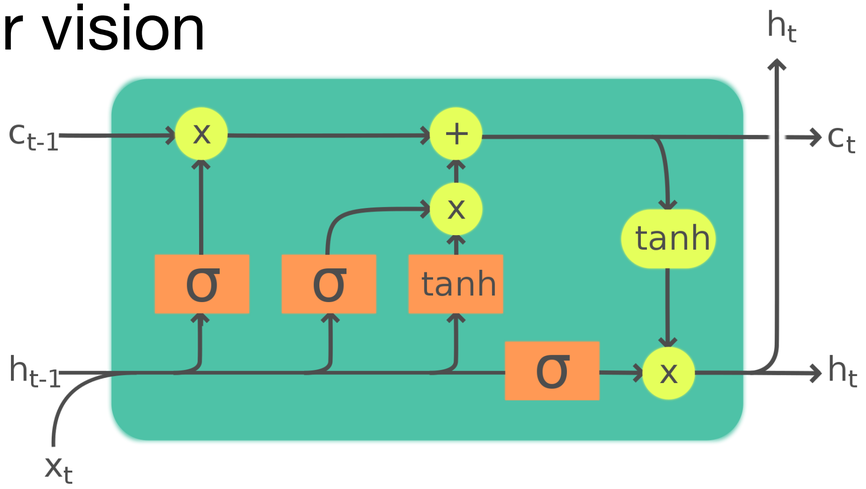
- Linear (or fully connected layer): your classic Ax+b layer
- Convolutional: commonly used in computer vision

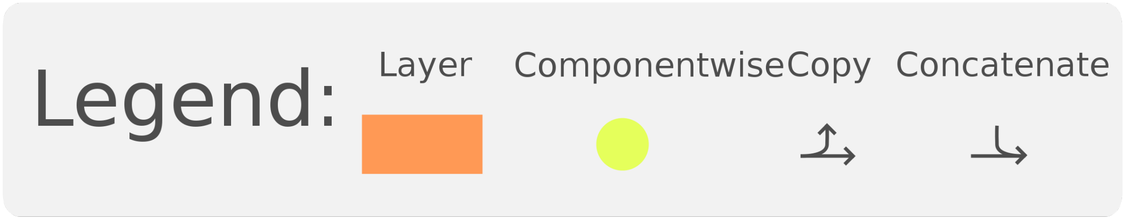



Source: https://www.superannotate.com/blog/guide-to-convolutional-neural-networks

# Common Layer Types

- Linear (or fully connected layer): your classic Ax+b layer
- Convolutional: commonly used in computer vision
- Recurrent (RNN): used for sequential data




Source: Wikipedia

# Common Layer Types

- Linear (or fully connected layer): your classic Ax+b layer
- Convolutional: commonly used in computer vision
- Recurrent (RNN): used for sequential data
- More layers can be found here: https://pytorch.org/docs/stable/nn.html

# Loss Function and Optimizer

| 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()`). |

Source: https://www.learnpytorch.io

# 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()` |

Source: https://www.learnpytorch.io

# PyTorch training loop

```python
# Pass the data through the model for a number of epochs (e.g. 100)
for epoch in range(epochs):

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

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

    # 2. Calculate the loss (how different are the model's predictions to the true values)
    loss = loss_fn(y_pred, y_true)

    # 3. Zero the gradients of the optimizer (they accumulate by default)
    optimizer.zero_grad()

    # 4. Perform backpropagation on the loss
    loss.backward()

    # 5. Progress/step the optimizer (gradient descent)
    optimizer.step()
```

*Note: all of this can be turned into a function*

Pass the data through the model for a number of epochs (e.g. 100 for 100 passes of the data)

Pass the data through the model, this will perform the `forward()` method located within the model object

Calculate the loss value (how wrong the model's predictions are)

Zero the optimizer gradients (they accumulate every epoch, zero them to start fresh each forward pass)

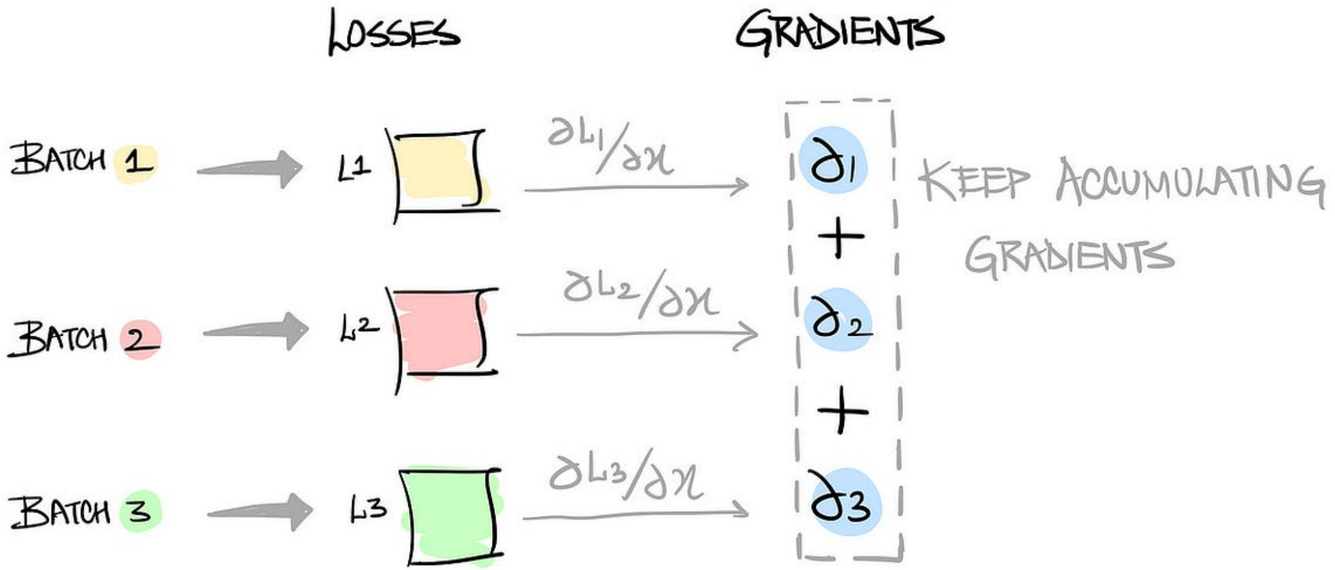
Perform backpropagation on the loss function (compute the gradient of every parameter with `requires_grad=True`)

Step the optimizer to update the model's parameters with respect to the gradients calculated by `loss.backward()`

Source: https://raw.githubusercontent.com/mrdbourke/pytorch-deep-learning/main/images/01-pytorch-training-loop-annotated.png

# Gradient Accumulation

- Don’t forget to zero the gradients after each train iteration, otherwise they will keep accumulating!




Source: https://medium.com/mlearning-ai/gradient-accumulation-307de7599e87

# Goal of Training

- Minimize the loss
- Make sure that the model generalises well (predicts well on an unknown data)

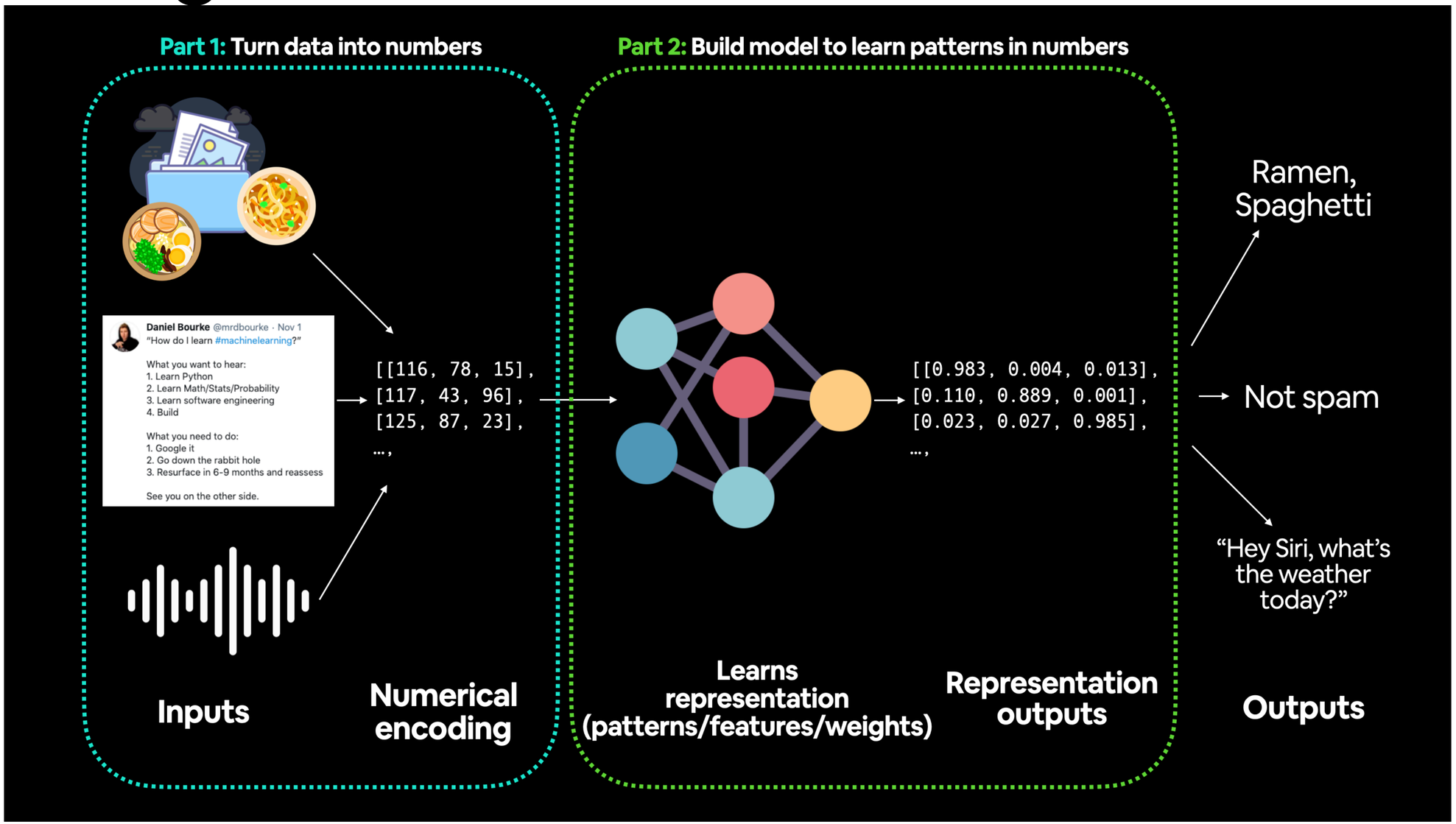
# Common Hyperparameters

- Learning rate (usually something small like 1e-3 or 1e-4, depends on the optimiser)
- Number of epochs
- Batch size (usually 16, 32, 64)

# Common Problems

- Overfitting and underfitting
- Overfitting is when the model “memorises” the training data and is incapable of generalising on the unseen data
- Undercutting is when the model is not powerful enough to learn from the data

# Handling the Data




Source: https://raw.githubusercontent.com/mrdbourke/pytorch-deep-learning/main/images/01-machine-learning-a-game-of-two-parts.png

# Train/Dev/Test Splits

| 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 |

# Data in PyTorch

- Dataset and DataLoader classes
- **Dataset** stores the samples and their corresponding labels, and **DataLoader** wraps an iterable around the Dataset to enable easy access to the samples.
- Task-specific libraries like torchaudio, torchvision, and torchtext have most popular datasets easily accessible

```
import torch
from torch.utils.data import Dataset
from torchvision import datasets
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt


training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)
```

# 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 |

Source: https://www.learnpytorch.io

# PyTorch testing loop

```python
# Setup empty lists to keep track of model progress
epoch_count = []
train_loss_values = []
test_loss_values = []

# Pass the data through the model for a number of epochs (e.g. 100) pochs):
for epoch in range(epochs):

    ### Training loop code here ###

    ### Testing starts ###

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

    # Turn on inference mode context manager
    with torch.inference_mode():
        # 1. Forward pass on test data
        test_pred = model(X_test)

        # 2. Caculate loss on test data
        test_loss = loss_fn(test_pred, y_test)

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

Note: all of this can be turned into a function

Create empty lists for storing useful values (helpful for tracking model progress)

Tell the model we want to evaluate rather than train (this turns off functionality used for training but not evaluation)

Turn on `torch.inference_mode()` context manager to disable functionality such as gradient tracking for inference (gradient tracking not needed for inference)

Pass the test data through the model (this will call the model's implemented `forward()` method)

Calculate the test loss value (how wrong the model's predictions are on the test dataset, lower is better)

Display information outputs for how the model is doing during training/testing every ~10 epochs (note: what gets printed out here can be adjusted for specific problems)

Source: https://raw.githubusercontent.com/mrdbourke/pytorch-deep-learning/main/images/01-pytorch-testing-loop-annotated.png

# Onto Practice!

- Introduction to PyTorch Tensors: https://pytorch.org/tutorials/beginner/introyt/tensors_deeper_tutorial.html
- The Fundamentals of Autograd: https://pytorch.org/tutorials/beginner/introyt/autogradyt_tutorial.html
- Building Models with PyTorch: https://pytorch.org/tutorials/beginner/introyt/modelsyt_tutorial.html
- Training with PyTorch: https://pytorch.org/tutorials/beginner/introyt/trainingyt.html