Reference: https://sebastianraschka.com/teaching/pytorch-1h/

This tutorial covers the following topics:

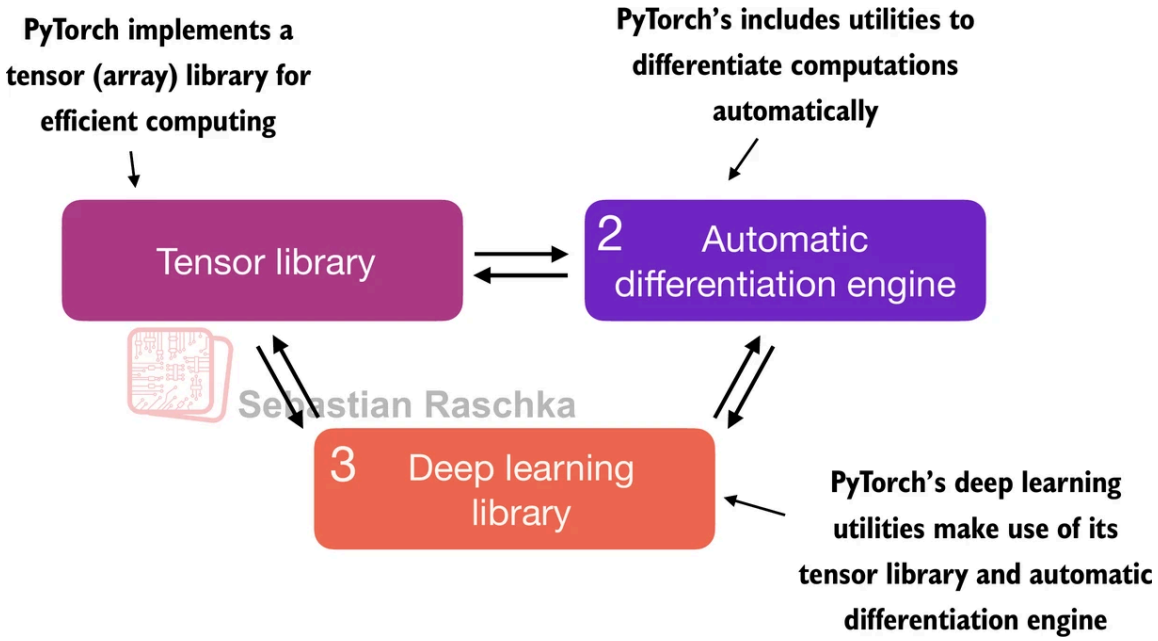
- An overview of the PyTorch deep learning library
- Tensors as a fundamental data structure for deep learning
- The mechanics of training deep neural networks

1. What is Pytorch

- Pytorch (https://pytorch.org/) is an open-source Python-based deep learning library.
- PyTorch hs been the most widely used deep learning library for research since 2019 by a wide margin.
- One of the reasons why PyTorch is so popular is its user-friendly interface and efficiency. Also, PyTorch provides high flexibility. Therefore, for many practitioners and researchers, PyTorch offers just the right balance between usability and features.

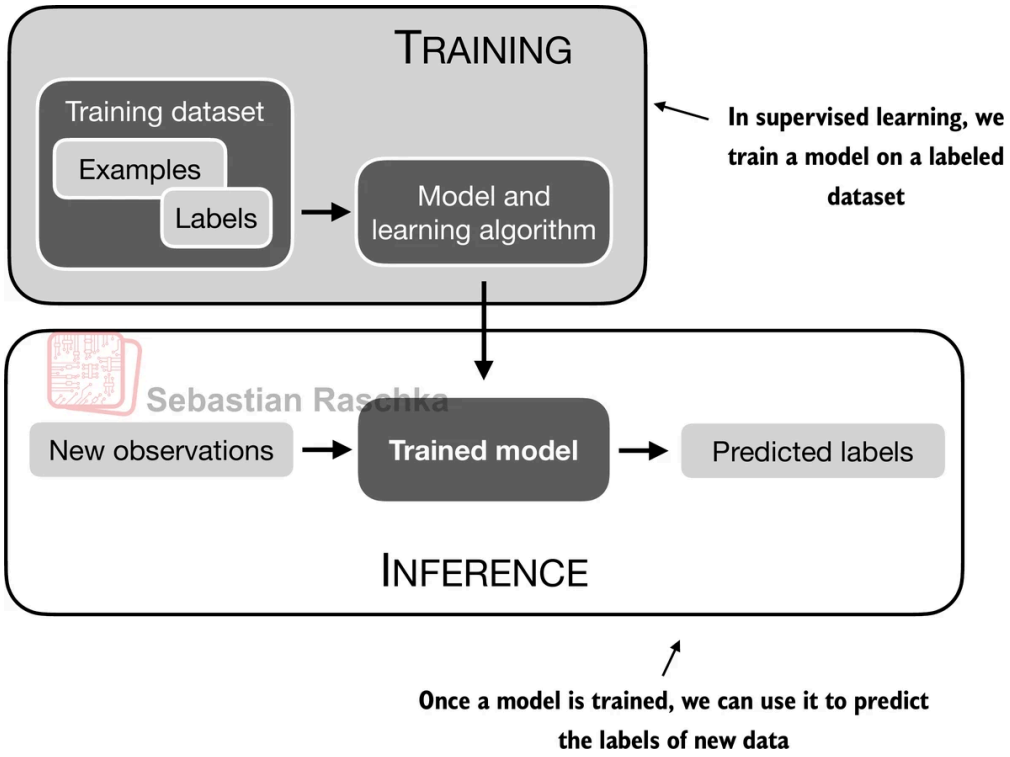
1.1 The three core components of PyTorch

- Firstly, PyTorch is a tensor library that extends the concept of array-oriented probramming library NumPy with the additional feature of accelerated computation on GPUs, thus providing a seamless switch between CPUs and GPUs.
- Secondly, PyTorch is an automatic differentiation engine, also known as autograd, which enables the automatic computation of gradients for tensor operations, simplifying backpropagation and model optimization.
- Finally, PyTorch is a deep learning library, meaning that it offers modular, flexible, and efficient building blocks (including pre-trained models, loss functions, and optimizers) for designing and training a wide range of deep learning models, catering to both researchers and developers.



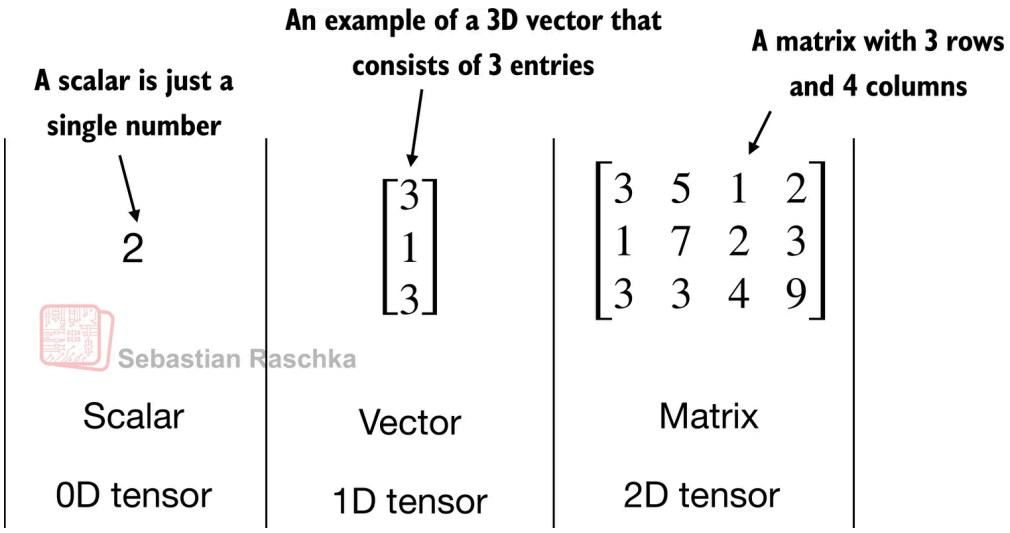
1.2 Defining deep learning

- AI > Machine Learning > Deep Learning
- The typical predictive modeling workflow (also referred to as supervised learning) in machine learning and deep learning is summarized in the figure below:



2. Understanding tensors

- Tensors represent a mathematical concept that generalizes vectors and matrices to potentially higher dimensions.
- Tensors are mathematical objects that can be characterized by their order (or rank), which provides the number of dimensions.
- For example, a scalar is a tensor of rank 0, a vector is a tensor of rank 1, and a matrix is a tensor of rank 2.



- From a computational perspective, tensors serve as data containers. For instance, they hold multi-dimensional data, where each dimension represents a different feature.
- Tensor libraries, such as PyTorch, can create, manipulate, and compute with these multi-dimensional arrays efficiently. In this context, a tensor library functions as an array library.
- PyTorch tensors are similar to NumPy arrays but have several additional features important for deep learning. For example, PyTorch adds an automatic differentiation engine, simplifying computing gradients. PyTorch tensors also support GPU computations to speed up deep neural network training.

2.1. Scalars, vectors, matrices, and tensors

- PyTorch tensors are data containers for array-like structures. A scalar is a 0-dimensional tensor (for instance, just a number), a vector is a 1-dimensional tensor, and a matrix is a 2-dimensional tensor. There is no specific term for higher-dimensional tensors, so we typically refer to a 3-dimensional tensor as just a 3D tensor, and so forth.

In [1]:
```
import torch

# create a 0D tensor (scalar) from a Python integer
tensor0d = torch.tensor(1)
```

```
# create a 1D tensor (vector) from a Python list
tensor1d = torch.tensor([1, 2, 3])

# create a 2D tensor from a nested Python list
tensor2d = torch.tensor([[1, 2], [3, 4]])

# create a 3D tensor from a nested Python list
tensor3d = torch.tensor([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
```

2.2. Tensor data types

- In the previous section, we created tensors from Python integers. In this case, PyTorch adopts the default 64-bit integer data type from Python. We can access the data type of a tensor via the `.dtype` attribute of a tensor:

In [2]:
```
tensor1d = torch.tensor([1, 2, 3])
print(tensor1d.dtype)
```

```
torch.int64
```

- If we create tensors from Python floats, PyTorch creates tensors with a 32-bit precision by default:

In [3]:
```
floatvec = torch.tensor([1.0, 2.0, 3.0])
print(floatvec.dtype)
```

```
torch.float32
```

- This choice is primarily due to the balance between precision and computational efficiency. A 32-bit floating point number offers sufficient precision for most deep learning tasks, while consuming less memory and computational resources than a 64-bit floating point number. Moreover, GPU architectures are optimized for 32-bit computations, and using this data type can significantly speed up model training and inference.

- It is possible to readily change the precision using a tensor's `.to` method. The following code demonstrates this by changing a 64-bit integer tensor into a 32-bit float tensor:

In [4]:
```
floatvec = tensor1d.to(torch.float32)
print(floatvec.dtype)
```

```
torch.float32
```

- For more information about different tensor data types available in PyTorch, I recommend checking the official documentation at

https://pytorch.org/docs/stable/tensors.html.

2.3. Common PyTorch tensor operations

- In this section, we will briefly describe the essentials that you may require to understand for doing almost any project.
- We already introduced the `torch.tensor()` function to create new tensors.

In [5]:
```
tensor2d = torch.tensor([[1, 2, 3],
                         [4, 5, 6]])
print(tensor2d)
```

```
tensor([[1, 2, 3],
        [4, 5, 6]])
```

- The `.shape` attribute allows us to access the shape of a tensor:

In [6]:
```
print(tensor2d.shape)
```

```
torch.Size([2, 3])
```

- As you can see above, `.shape` returns `[2, 3]`, which means that the tensor has 2 rows and 3 columns. To reshape the tensor into a 3 by 2 tensor, we can use the `.reshape` method:

In [7]:
```
tensor2d.reshape(3, 2)
```

Out[7]:
```
tensor([[1, 2],
        [3, 4],
        [5, 6]])
```

- However, note that the more common command for reshaping tensors in PyTorch is `.view()`:

In [8]:
```
tensor2d.view(3, 2)
```

Out[8]:
```
tensor([[1, 2],
        [3, 4],
        [5, 6]])
```

- Next, we can use `.T` to transpose a tensor, which means flipping it across its diagonal. Note that this is similar from reshaping a tensor as you can see based on the result below:

In [9]:
```
tensor2d.T
```

Out[9]:
```
tensor([[1, 4],
        [2, 5],
        [3, 6]])
```

- Lastly, the common way to multiply two matrices in PyTorch is the `.matmul` method:

In [10]:
```
tensor2d.matmul(tensor2d.T)
```

Out[10]:
```
tensor([[14, 32],
        [32, 77]])
```

However, we can also adopt the `@` operator, which accomplishes the same thing more compactly:

In [11]:
```
tensor2d @ tensor2d.T
```

Out[11]:
```
tensor([[14, 32],
        [32, 77]])
```

- For you who'd like to browse through all the different tensor operations available in PyTorch (hint: we won't need most of these), I recommend checking out the official documentation at https://pytorch.org/docs/stable/tensors.html.

3. Seeing models as computation graphs

- In the previous section, we covered one of the major three components of PyTorch, namely, its tensor library.
- Next in line is PyTorch's automatic differentiation engine, also known as autograd.
- PyTorch's autograd system provides functions to compute gradients in dynamic computational graphs automatically.
- First, let's define the concept of a computational graph.

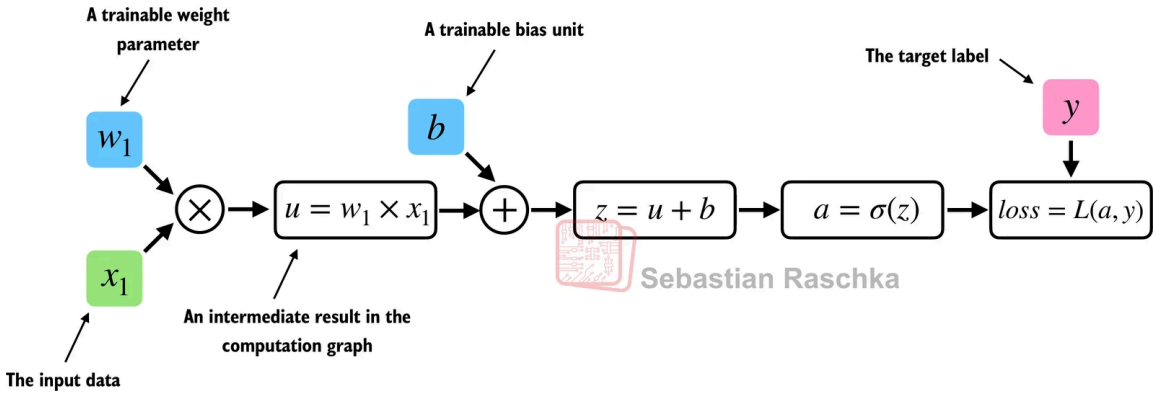
- A computational graph (or computation graph in short) is a directed graph that allows us to express and visualize mathematical expressions.
- In the context of deep learning, a computation graph lays out the sequence of calculations needed to compute the output of a neural networks.

- The following code implements the forward pass (prediction step) of a simple logistic regression classifier, which can be seen as a single-layer neural network, returning a score between 0 and 1 that is compared to the true class label (0 or 1) when computing the loss:

In [12]:
```
import torch
import torch.nn.functional as F

y = torch.tensor([1.0])  # true label
x1 = torch.tensor([1.1]) # input feature
w1 = torch.tensor([2.2]) # weight parameter
b = torch.tensor([0.0])  # bias unit

z = x1 * w1 + b          # net input
a = torch.sigmoid(z)     # activation & output

loss = F.binary_cross_entropy(a, y)
print(loss)
```

```
tensor(0.0852)
```

- If not all components in the code above make sense to you, don't worry. The point of this example is not to implement a logistic regression classifier but rather to illustrate how we can think of a sequence of computations as a computation graph:



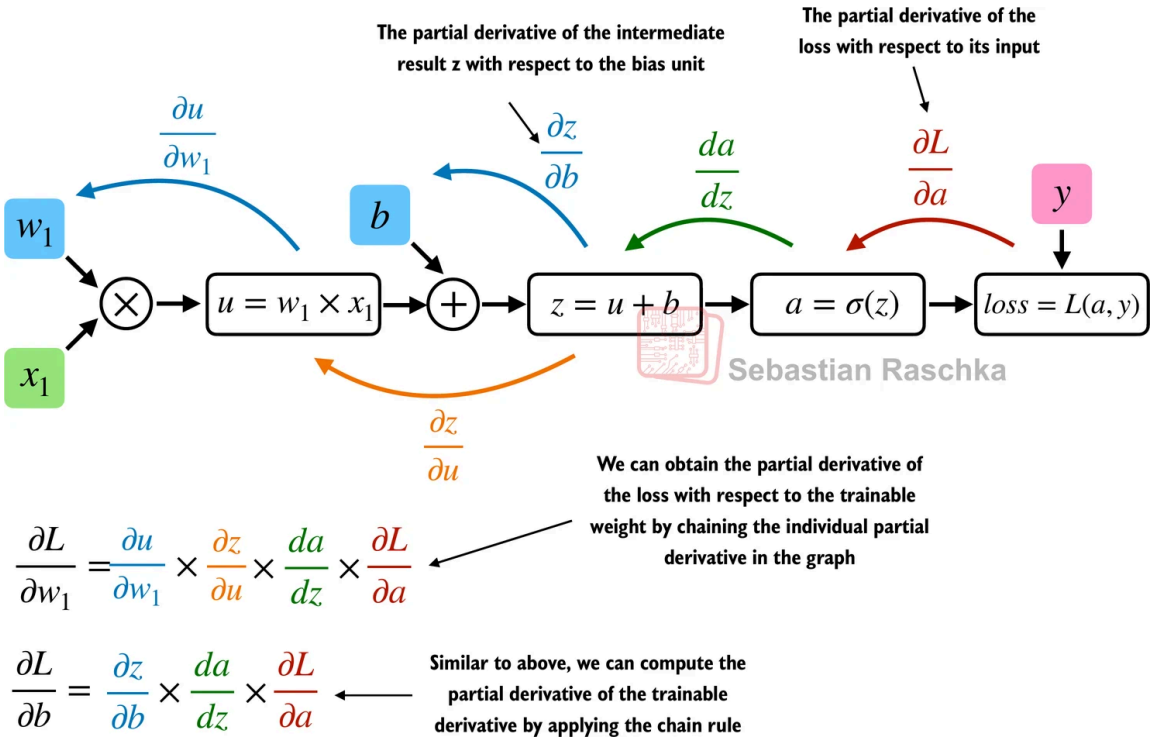
- A logistic regression forward pass as a computation graph. The input feature `x1` is multiplied to a model weight `w1` and passed through an activation function σ after adding the bias. The loss is computed by comparing the model output `a` with a given label `y`.

- In fact, PyTorch builds such a computation graph in the background, and we can use this to calculate gradients of a loss function with respect to the model parameters (here `w1` and `b`) to train the model.

4. Automatic differentiation made easy

- If we carry out computations in PyTorch, it will build such a graph internally by default if one of its terminal nodes has the `requires_grad` attribute set to `True`. This is useful if we want to compute gradients.

- Gradients are required when training neural networks via the popular backpropagation algorithm, which can be thought of as an implementation of the *chain rule* from calculus for neural networks.



- The most common way of computing the loss gradients in a computation graph involves applying the chain rule from right to left, which is also called reverse-model automatic differentiation or backpropagation. It means we start from the output layer (or the loss itself) and work backward through the network to the input layer. This is done to compute the gradient of the loss with respect to each parameter (weights and biases) in the network, which informs how we update these parameters during training.

- Now, how is this all related to the second component of the PyTorch library we mentioned earlier, the automatic differentiation (autograd) engine?
- By tracking every operation performed on tensors, PyTorch's autograd engine constructs a computational graph in the background.
- Then, calling the grad function, we can compute the gradient of the loss with respect to model parameter `w1` as follows:

In [13]:
```
import torch.nn.functional as F
from torch.autograd import grad

y = torch.tensor([1.0])
x1 = torch.tensor([1.1])
w1 = torch.tensor([2.2], requires_grad=True)
b = torch.tensor([0.0], requires_grad=True)

z = x1 * w1 + b
```

```
a = torch.sigmoid(z)

loss = F.binary_cross_entropy(a, y)

grad_L_w1 = grad(loss, w1, retain_graph=True)
grad_L_b = grad(loss, b, retain_graph=True)
```

- By default, PyTorch destroys the computation graph after calculating the gradients to free memory. However, since we are going to reuse this computation graph shortly, we set `retain_graph=True` so that it stays in memory.
- Let's show the resulting values of the loss with respect to the model's parameters:

In [14]:
```
print(grad_L_w1)
print(grad_L_b)
```

```
(tensor([-0.0898]),)
(tensor([-0.0817]),)
```

- Above, we have been using the grad function "manually," which can be useful for experimentation, debugging, and demonstrating concepts.
- But in practice, PyTorch provides even more high-level tools to automate this process. For instance, we can call `.backward` on the loss, and PyTorch will compute the gradients of all the leaf nodes in the graph, which will be stored via the tensors' `.grad` attributes:

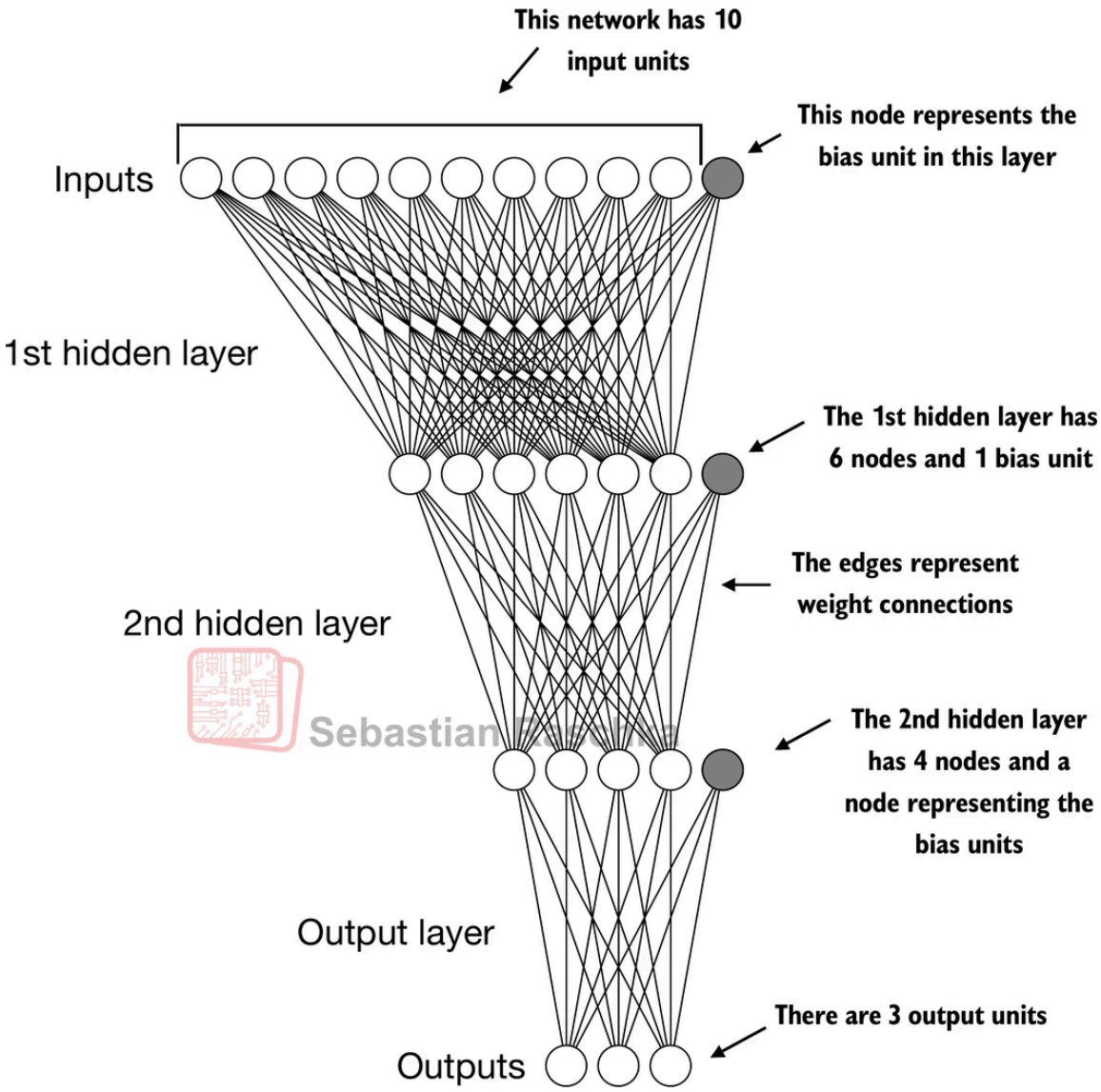
In [15]:
```
loss.backward()

print(w1.grad)
print(b.grad)
```

```
tensor([-0.0898])
tensor([-0.0817])
```

- PyTorch takes care of the calculus for us via the `.backward` method -- we usually don't need to compute any derivatives or gradients by hand when using PyTorch.

5. Implementing multilayer neural networks

- In the previous sections, we covered PyTorch's tensor and autograd components. This section focuses on PyTorch as a library for implementing deep neural networks.
- To provide a concrete example, we focus on a multilayer perceptron, which is a fully connected neural network, as illustrated in below:



- When implementing a neural network in PyTorch, we typically subclass the `torch.nn.Module` class to define our own custom network architecture. This `Module` base class provides a lot of functionality, making it easier to build and train models. For instance, it allows us to encapsulate layers and operations and keep track of the model's parameters.
- Within this subclass, we define the network layers in the `__init__` constructor and specify how they interact in the `forward` method. The `forward` method describes how the input data passes through the network and comes together as a computation graph.
- In contrast, the backward method, which we typically do not need to implement ourselves, is used during training to compute gradients of the loss function with respect to the model parameters.
- The following code implements a classic multilayer perceptron with two hidden layers to illustrate a typical usage of the `Module` class:

In [16]:
```
class NeuralNetwork(torch.nn.Module):
    def __init__(self, num_inputs, num_outputs):
        super().__init__()

        self.layers = torch.nn.Sequential(

            # 1st hidden layer
            torch.nn.Linear(num_inputs, 30),
            torch.nn.ReLU(),

            # 2nd hidden layer
            torch.nn.Linear(30, 20),
            torch.nn.ReLU(),

            # output layer
            torch.nn.Linear(20, num_outputs),
        )

    def forward(self, x):
        logits = self.layers(x)
        return logits
```

- We can then instantiate a new neural network object as follows:

In [17]:
```
model = NeuralNetwork(50, 3)
```

- But before using this new model object, it is often useful to call print on the model to see a summary of its structure:

In [18]:
```
print(model)
```

```
NeuralNetwork(
  (layers): Sequential(
    (0): Linear(in_features=50, out_features=30, bias=True)
    (1): ReLU()
    (2): Linear(in_features=30, out_features=20, bias=True)
    (3): ReLU()
    (4): Linear(in_features=20, out_features=3, bias=True)
  )
)
```

- Note that we used the `Sequential` class when we implemented the `NeuralNetwork` class. Using `Sequential` is not required, but it can make our life easier if we have a series of layers that we want to execute in a specific order, as is the case here.
- This way, after instantiating `self.layers = Sequential(...)` in the `__init__` constructor, we just have to call the `self.layers` instead of calling each layer individually in the `NeuralNetwork`'s forward method.
- Next, let's check the total number of trainable parameters of this model.

In [19]:
```
num_params = sum(
    p.numel() for p in model.parameters() if p.requires_grad
)
print("Total number of trainable model parameters:", num_params)
```

```
Total number of trainable model parameters: 2213
```

- Note that each parameter for which `requires_grad=True` counts as a trainable parameter and will be updated during training.
- In the case of our neural network model with the two hidden layers above, these trainable parameters are contained in the `torch.nn.Linear` layers. A linear layer multiplies the inputs with a weight matrix and adds a bias vector. This is sometimes also referred to as a *feedforward* or *fully connected* layer.
- Based on the `print(model)` call we executed above, we can see that the first Linear layer is at index position 0 in the layers attribute. We can access the corresponding weight parameter matrix as follows:

In [20]:
```
print(model.layers[0].weight)
```

```
Parameter containing:
tensor([[-0.0691,  0.0343, -0.0540,  ...,  0.0822, -0.0915, -0.0500],
        [-0.0509,  0.0484, -0.0885,  ..., -0.1298, -0.0206, -0.1410],
        [-0.0002,  0.1061,  0.0451,  ..., -0.0428,  0.1329, -0.0329],
        ...,
        [-0.0830,  0.1242,  0.0539,  ...,  0.0626,  0.1064,  0.1050],
        [ 0.0966, -0.0217, -0.1333,  ..., -0.1303,  0.0502, -0.0432],
        [ 0.0002, -0.0590,  0.1268,  ...,  0.0496,  0.0162,  0.1086]],
       requires_grad=True)
```

- Since this is a large matrix that is not shown in its entirety, let's use the `.shape` attribute to show its dimensions:

In [21]:
```
print(model.layers[0].weight.shape)
```

```
torch.Size([30, 50])
```

- Similarly, you could access the bias vector via `model.layers[0].bias`:

In [22]:
```
print(model.layers[0].bias)
```

```
Parameter containing:
tensor([ 0.1276,  0.1062, -0.0285, -0.1089,  0.0979,  0.0777,  0.1028, -0.0400,
        -0.0486,  0.1278,  0.1345,  0.0263, -0.0681,  0.0200,  0.0777, -0.0321,
         0.1294, -0.1294, -0.1332,  0.0828,  0.1337,  0.1033,  0.1089,  0.0091,
        -0.1002, -0.0727, -0.1138, -0.0250,  0.0469,  0.0323],
       requires_grad=True)
```

In [23]:
```
print(model.layers[0].bias.shape)
```

```
torch.Size([30])
```

- The weight matrix above is a 30x50 matrix, and we can see that the `requires_grad` is set to True, which means its entries are trainable – this is the default setting for weights and biases in `torch.nn.Linear`.
- Note that if you execute the code above on your computer, the numbers in the weight matrix will likely differ from those shown above. This is because the model weights are initialized with small random numbers, which are different each time we instantiate the network. In deep learning, initializing model weights with small random numbers is desired to break symmetry during training – otherwise, the nodes would be just performing the same operations and updates during backpropagation, which would not allow the network to learn complex mappings from inputs to outputs.
- However, while we want to keep using small random numbers as initial values for our layer weights, we can make the random number initialization reproducible by seeding PyTorch's random number generator via `manual_seed`:

In [24]:
```
torch.manual_seed(123)

model = NeuralNetwork(50, 3)
print(model.layers[0].weight)
```

```
Parameter containing:
tensor([[-0.0577,  0.0047, -0.0702,  ...,  0.0222,  0.1260,  0.0865],
        [ 0.0502,  0.0307,  0.0333,  ...,  0.0951,  0.1134, -0.0297],
        [ 0.1077, -0.1108,  0.0122,  ...,  0.0108, -0.1049, -0.1063],
        ...,
        [-0.0787,  0.1259,  0.0803,  ...,  0.1218,  0.1303, -0.1351],
        [ 0.1359,  0.0175, -0.0673,  ...,  0.0674,  0.0676,  0.1058],
        [ 0.0790,  0.1343, -0.0293,  ...,  0.0344, -0.0971, -0.0509]],
       requires_grad=True)
```

In [25]:
```
model2 = NeuralNetwork(50, 3)
print(model2.layers[0].weight)
```

```
Parameter containing:
tensor([[-0.0738, -0.0511,  0.0880,  ..., -0.0779,  0.1037,  0.0555],
        [ 0.0037,  0.0560, -0.0131,  ...,  0.0064, -0.0599, -0.0675],
        [ 0.1306,  0.0064, -0.0993,  ...,  0.0128, -0.0708, -0.1067],
        ...,
        [-0.0780, -0.0007,  0.0417,  ..., -0.0466, -0.0704,  0.1409],
        [ 0.0190, -0.0893,  0.1222,  ...,  0.0478,  0.0466,  0.0972],
        [ 0.1268,  0.0739, -0.0833,  ..., -0.0758,  0.0085,  0.1151]],
       requires_grad=True)
```

In [26]:
```
torch.manual_seed(123)

model3 = NeuralNetwork(50, 3)
print(model3.layers[0].weight)
```

```
Parameter containing:
tensor([[-0.0577,  0.0047, -0.0702,  ...,  0.0222,  0.1260,  0.0865],
        [ 0.0502,  0.0307,  0.0333,  ...,  0.0951,  0.1134, -0.0297],
        [ 0.1077, -0.1108,  0.0122,  ...,  0.0108, -0.1049, -0.1063],
        ...,
        [-0.0787,  0.1259,  0.0803,  ...,  0.1218,  0.1303, -0.1351],
        [ 0.1359,  0.0175, -0.0673,  ...,  0.0674,  0.0676,  0.1058],
        [ 0.0790,  0.1343, -0.0293,  ...,  0.0344, -0.0971, -0.0509]],
       requires_grad=True)
```

- Now, let's briefly see how the `NeuralNetwork` instance is used via the forward pass:

In [27]:
```
torch.manual_seed(123)

X = torch.rand((1, 50))
out = model(X)
print(out)
```

```
tensor([[-0.1262,  0.1080, -0.1792]], grad_fn=<AddmmBackward0>)
```

- In the code above, we generated a single random training example `X` as a toy input (note that our network expects 50-dimensional feature vectors) and fed it to the model, returning three scores. When we call `model(x)`, it will automatically execute the forward pass of the model.
- The forward pass refers to calculating output tensors from input tensors. This involves passing the input data through all the neural network layers, starting from the input layer, through hidden layers, and finally to the output layer.
- These three numbers returned above correspond to a score assigned to each of the three output nodes. Notice that the output tensor also includes a `grad_fn` value.
- Here, `grad_fn=<AddmmBackward0>` represents the last-used function to compute a variable in the computational graph. In particular, `grad_fn=<AddmmBackward0>` means that the tensor we are inspecting was created via a matrix multiplication and addition operation. PyTorch will use this information when it computes gradients during backpropagation. The `<AddmmBackward0>` part of `grad_fn=<AddmmBackward0>` specifies the operation that was performed. In this case, it is an `Addmm` operation. `Addmm` stands for matrix multiplication (`mm`) followed by an addition (`Add`).
- If we just want to use a network without training or backpropagation, for example, if we use it for prediction after training, constructing this computational graph for backpropagation can be wasteful as it performs unnecessary computations and consumes additional memory. So, when we use a model for inference (for instance, making predictions) rather than training, it is a best practice to use the `torch.no_grad()` context manager, as shown below. This tells PyTorch that it doesn't need to keep track of the gradients, which can result in significant savings in memory and computation.

In [28]:
```
with torch.no_grad():
    out = model(X)
```

```
print(out)
```

```
tensor([[-0.1262,  0.1080, -0.1792]])
```

- In PyTorch, it's common practice to code models such that they return the outputs of the last layer (`logits`) without passing them to a nonlinear activation function. That's because PyTorch's commonly used loss functions combine the softmax (or sigmoid for binary classification) operation with the negative log-likelihood loss in a single class. The reason for this is numerical efficiency and stability. So, if we want to compute class-membership probabilities for our predictions, we have to call the softmax function explicitly:

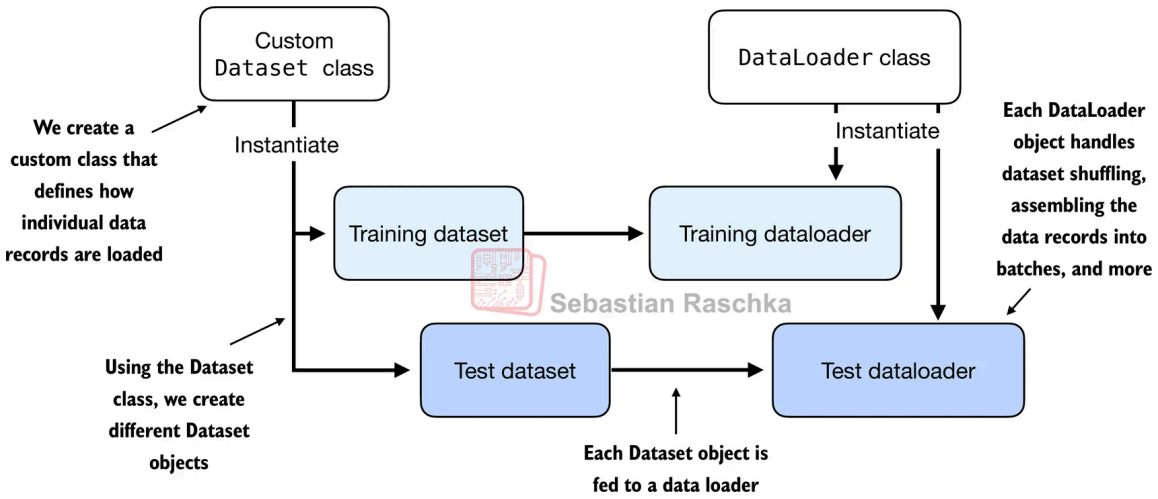
In [29]:
```
with torch.no_grad():
    out = torch.softmax(model(X), dim=1)
print(out)
```

```
tensor([[0.3113, 0.3934, 0.2952]])
```

- The values can now be interpreted as class-membership probabilities that sum up to 1. The values are roughly equal for this random input, which is expected for a randomly initialized model without training.

6. Setting up efficient data loaders

- In the previous section, we defined a custom neural network model. Before we can train this model, we have to briefly talk about creating efficient data loaders in PyTorch, which we will iterate over when training the model. The overall idea behind data loading in PyTorch is illustrated in the below:



- In this section, we will implement a custom `Dataset` class that we will use to create a training and a test dataset that we'll then use to create the data loaders.

- Let's start by creating a simple toy dataset of five training examples with two features each. Accompanying the training examples, we also create a tensor containing the corresponding class labels: three examples belong to class 0, and two examples belong to class 1. In addition, we also make a test set consisting of two entries. The code to create this dataset is shown below.

In [30]:
```
X_train = torch.tensor([
    [-1.2, 3.1],
    [-0.9, 2.9],
    [-0.5, 2.6],
    [2.3, -1.1],
    [2.7, -1.5]
])

y_train = torch.tensor([0, 0, 0, 1, 1])
```

In [31]:
```
X_test = torch.tensor([
    [-0.8, 2.8],
    [2.6, -1.6],
])

y_test = torch.tensor([0, 1])
```

- (**Class label numbering**) PyTorch requires that class labels start with label 0, and the largest class label value should not exceed the number of output nodes minus 1 (since Python index counting starts at 0. So, if we have class labels 0, 1, 2, 3, and 4, the neural network output layer should consist of 5 nodes).

- Next, we create a custom dataset class, `ToyDataset`, by subclassing from PyTorch's `Dataset` parent class, as shown below.

In [32]:
```
from torch.utils.data import Dataset

class ToyDataset(Dataset):
    def __init__(self, X, y):
        self.features = X
        self.labels = y

    def __getitem__(self, index):
        one_x = self.features[index]
        one_y = self.labels[index]
        return one_x, one_y

    def __len__(self):
        return self.labels.shape[0]

train_ds = ToyDataset(X_train, y_train)
test_ds = ToyDataset(X_test, y_test)
```

- This custom `ToyDataset` class's purpose is to use it to instantiate a PyTorch DataLoader. But before we get to this step, let's briefly go over the general structure of the ToyDataset code.
- In PyTorch, the three main components of a custom Dataset class are the `__init__` constructor, the `__getitem__` method, and the `__len__` method, as shown in code `ToyDataset` code above.
- In the `__init__` method, we set up attributes that we can access later in the `__getitem__` and `__len__` methods. This could be file paths, file objects, database connectors, and so on. Since we created a tensor dataset that sits in memory, we are simply assigning `X` and `y` to these attributes, which are placeholders for our tensor objects.
- In the `__getitem__` method, we define instructions for returning exactly one item from the dataset via an index. This means the features and the class label corresponding to a single training example or test instance. (The data loader will provide this index, which we will cover shortly.)
- Finally, the `__len__` method contains instructions for retrieving the length of the dataset. Here, we use the .shape attribute of a tensor to return the number of rows in the feature array. In the case of the training dataset, we have five rows, which we can double-check as follows:

In [33]:
```
len(train_ds)
```

Out[33]:
```
5
```

- Now that we defined a PyTorch `Dataset` class we can use for our toy dataset, we can use PyTorch's `DataLoader` class to sample from it, as shown in the code below:

In [34]:
```
from torch.utils.data import DataLoader

torch.manual_seed(123)

train_loader = DataLoader(
    dataset=train_ds,
    batch_size=2,
    shuffle=True,
    num_workers=0
)
```

In [35]:
```
test_loader = DataLoader(
    dataset=test_ds,
    batch_size=2,
    shuffle=False,
```

```
    num_workers=0
)
```

- After instantiating the training data loader, we can iterate over it as shown below:

In [36]:
```
for idx, (x, y) in enumerate(train_loader):
    print(f"Batch {idx+1}:", x, y)
```

```
Batch 1: tensor([[ 2.3000, -1.1000],
        [-0.9000,  2.9000]]) tensor([1, 0])
Batch 2: tensor([[-1.2000,  3.1000],
        [-0.5000,  2.6000]]) tensor([0, 0])
Batch 3: tensor([[ 2.7000, -1.5000]]) tensor([1])
```

- As we can see based on the output above, the `train_loader` iterates over the training dataset visiting each training example exactly once. This is known as a training epoch. Since we seeded the random number generator using `torch.manual_seed(123)` above, you should get the exact same shuffling order of training examples as shown above. However if you iterate over the dataset a second time, you will see that the shuffling order will change. This is desired to prevent deep neural networks getting caught in repetitive update cycles during training.

- Note that we specified a batch size of 2 above, but the 3rd batch only contains a single example. That's because we have five training examples, which is not evenly divisible by 2. In practice, having a substantially smaller batch as the last batch in a training epoch can disturb the convergence during training. To prevent this, it's recommended to set `drop_last=True`, which will drop the last batch in each epoch, as shown below:

In [37]:
```
train_loader = DataLoader(
    dataset=train_ds,
    batch_size=2,
    shuffle=True,
    num_workers=0,
    drop_last=True
)
```

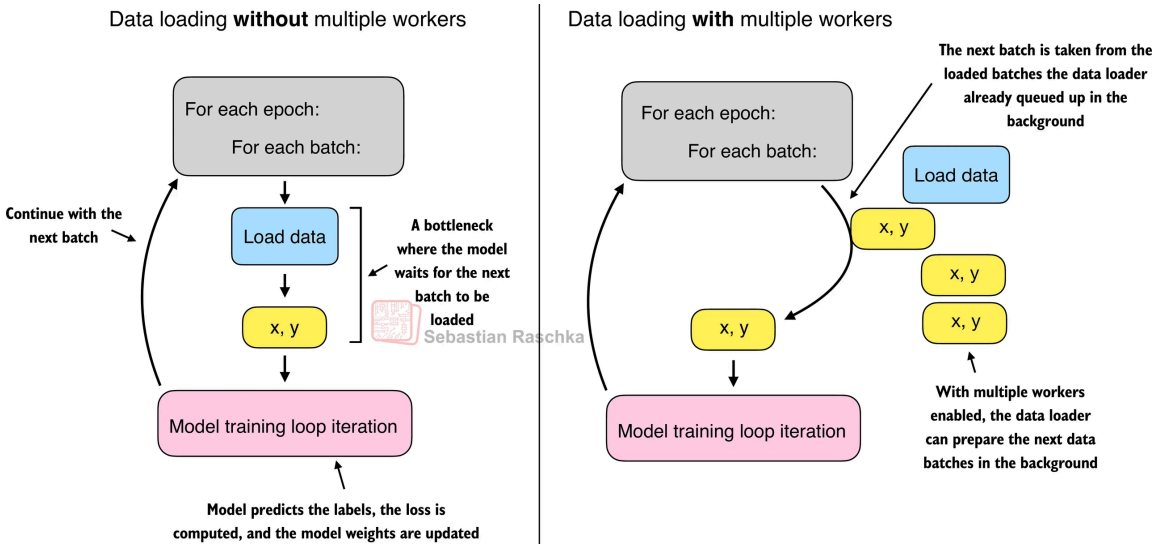
- Now, iterating over the training loader, we can see that the last batch is omitted:

In [38]:
```
for idx, (x, y) in enumerate(train_loader):
    print(f"Batch {idx+1}:", x, y)
```

```
Batch 1: tensor([[-1.2000,  3.1000],
        [-0.5000,  2.6000]]) tensor([0, 0])
Batch 2: tensor([[ 2.3000, -1.1000],
        [-0.9000,  2.9000]]) tensor([1, 0])
```

(Optional) the `num_workers` argument

- Lastly, let's discuss the setting `num_workers=0` in the `DataLoader`. This parameter in PyTorch's `DataLoader` function is crucial for parallelizing data loading and preprocessing. When `num_workers` is set to 0, the data loading will be done in the main process and not in separate worker processes. This might seem unproblematic, but it can lead to significant slowdowns during model training when we train larger networks on a GPU. This is because instead of focusing solely on the processing of the deep learning model, the CPU must also take time to load and preprocess the data. As a result, the GPU can sit idle while waiting for the CPU to finish these tasks. In contrast, when `num_workers` is set to a number greater than zero, multiple worker processes are launched to load data in parallel, freeing the main process to focus on training your model and better utilizing your system's resources, which is illustrated in the below:



- However, if we are working with very small datasets, setting `num_workers` to 1 or larger may not be necessary since the total training time takes only fractions of a second anyway. On the contrary, if you are working with tiny datasets or interactive environments such as Jupyter notebooks, increasing `num_workers` may not provide any noticeable speedup. They might, in fact, lead to some issues. One potential issue is the overhead of spinning up multiple worker processes, which could take longer than the actual data loading when your dataset is small.

- Furthermore, for Jupyter notebooks, setting `num_workers` to greater than 0 can sometimes lead to issues related to the sharing of resources between different processes, resulting in errors or notebook crashes. Therefore, it's essential to understand the trade-off and make a calculated decision on setting the `num_workers` parameter. When used correctly, it can be a beneficial tool but should be adapted to your specific dataset size and computational environment for optimal results.

- Setting `num_workers=4` usually leads to optimal performance on many real-world datasets, but optimal settings depend on your hardware and the code used for loading a training example defined in the `Dataset` class.

7. A typical training loop

- So far, we've discussed all the requirements for training neural networks: PyTorch's tensor library, autograd, the Module API, and efficient data loaders. Let's now combine all these things and train a neural network on the toy dataset from the previous section. The training code is shown in code below.

In [39]:
```
import torch.nn.functional as F

torch.manual_seed(123)
model = NeuralNetwork(num_inputs=2, num_outputs=2)
optimizer = torch.optim.SGD(model.parameters(), lr=0.5)

num_epochs = 3

for epoch in range(num_epochs):

    model.train()
    for batch_idx, (features, labels) in enumerate(train_loader):

        logits = model(features)

        loss = F.cross_entropy(logits, labels) # Loss function

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        ### LOGGING
        print(f"Epoch: {epoch+1:03d}/{num_epochs:03d}"
              f" | Batch {batch_idx:03d}/{len(train_loader):03d}"
              f" | Train/Val Loss: {loss:.2f}")

    model.eval()
    # Optional model evaluation
```

```
Epoch: 001/003 | Batch 000/002 | Train/Val Loss: 0.75
Epoch: 001/003 | Batch 001/002 | Train/Val Loss: 0.65
Epoch: 002/003 | Batch 000/002 | Train/Val Loss: 0.44
Epoch: 002/003 | Batch 001/002 | Train/Val Loss: 0.13
Epoch: 003/003 | Batch 000/002 | Train/Val Loss: 0.03
Epoch: 003/003 | Batch 001/002 | Train/Val Loss: 0.00
```

- As we can see, the loss reaches zero after 3 epochs, a sign that the model converged on the training set. However, before we evaluate the model's predictions, let's go over some of the details of the preceding code.
- First, note that we initialized a model with two inputs and two outputs. That's because the toy dataset from the previous section has two input features and two class labels to

predict. We used a stochastic gradient descent (`SGD`) optimizer with a learning rate (`lr`) of 0.5. The learning rate is a hyperparameter, meaning it's a tunable setting that we have to experiment with based on observing the loss. Ideally, we want to choose a learning rate such that the loss converges after a certain number of epochs – the number of epochs is another hyperparameter to choose.

- In practice, we often use a third dataset, a so-called validation dataset, to find the optimal hyperparameter settings. A validation dataset is similar to a test set. However, while we only want to use a test set precisely once to avoid biasing the evaluation, we usually use the validation set multiple times to tweak the model settings.

- We also introduced new settings called `model.train()` and `model.eval()`. As these names imply, these settings are used to put the model into a training and an evaluation mode. This is necessary for components that behave differently during training and inference, such as dropout or batch normalization layers. Since we don't have dropout or other components in our `NeuralNetwork` class that are affected by these settings, using `model.train()` and `model.eval()` is redundant in our code above. However, it's best practice to include them anyway to avoid unexpected behaviors when we change the model architecture or reuse the code to train a different model.

- As discussed earlier, we pass the logits directly into the `cross_entropy` loss function, which will apply the softmax function internally for efficiency and numerical stability reasons. Then, calling `loss.backward()` will calculate the gradients in the computation graph that PyTorch constructed in the background. The `optimizer.step()` method will use the gradients to update the model parameters to minimize the loss. In the case of the SGD optimizer, this means multiplying the gradients with the learning rate and adding the scaled negative gradient to the parameters.

- It is important to include an `optimizer.zero_grad()` call in each update round to reset the gradients to zero. Otherwise, the gradients will accumulate, which may be undesired.

- After we trained the model, we can use it to make predictions:

In [40]:
```
model.eval()

with torch.no_grad():
    outputs = model(X_train)

print(outputs)
```

```
tensor([[ 2.8569, -4.1618],
        [ 2.5382, -3.7548],
        [ 2.0944, -3.1820],
        [-1.4814,  1.4816],
        [-1.7176,  1.7342]])
```

- To obtain the class membership probabilities, we can then use PyTorch's softmax function, as follows:

In [41]:
```
torch.set_printoptions(sci_mode=False)
probas = torch.softmax(outputs, dim=1)
print(probas)
```

```
tensor([[    0.9991,     0.0009],
        [    0.9982,     0.0018],
        [    0.9949,     0.0051],
        [    0.0491,     0.9509],
        [    0.0307,     0.9693]])
```

- Let's consider the first row in the code output above. Here, the first value (column) means that the training example has a 99.91% probability of belonging to class 0 and a 0.09% probability of belonging to class 1. (The `set_printoptions` call is used here to make the outputs more legible.)

- We can convert these values into class labels predictions using PyTorch's `argmax` function, which returns the index position of the highest value in each row if we set `dim=1` (setting `dim=0` would return the highest value in each column, instead):

In [42]:
```
predictions = torch.argmax(probas, dim=1)
print(predictions)
```

```
tensor([0, 0, 0, 1, 1])
```

- Note that it is unnecessary to compute softmax probabilities to obtain the class labels. We could also apply the `argmax` function to the logits (`outputs`) directly:

In [43]:
```
predictions = torch.argmax(outputs, dim=1)
print(predictions)
```

```
tensor([0, 0, 0, 1, 1])
```

- Above, we computed the predicted labels for the training dataset. Since the training dataset is relatively small, we could compare it to the true training labels by eye and see that the model is 100% correct. We can double-check this using the `==` comparison operator:

In [44]:
```
predictions == y_train
```

Out[44]:
```
tensor([True, True, True, True, True])
```

- Using `torch.sum`, we can count the number of correct prediction as follows:

In [45]:
```
torch.sum(predictions == y_train)
```

Out[45]:
```
tensor(5)
```

- Since the dataset consists of 5 training examples, we have 5 out of 5 predictions that are correct, which equals 5/5 × 100% = 100% prediction accuracy.
- However, to generalize the computation of the prediction accuracy, let's implement a compute_accuracy function as shown in the following code.

In [46]:
```
def compute_accuracy(model, dataloader):

    model = model.eval()
    correct = 0.0
    total_examples = 0

    for idx, (features, labels) in enumerate(dataloader):

        with torch.no_grad():
            logits = model(features)

        predictions = torch.argmax(logits, dim=1)
        compare = labels == predictions
        correct += torch.sum(compare)
        total_examples += len(compare)

    return (correct / total_examples).item()
```

- Note that the following `compute_accuracy` function iterates over a data loader to compute the number and fraction of the correct predictions. This is because when we work with large datasets, we typically can only call the model on a small part of the dataset due to memory limitations. The `compute_accuracy` function above is a general method that scales to datasets of arbitrary size since, in each iteration, the dataset chunk that the model receives is the same size as the batch size seen during training.
- Notice that the internals of the `compute_accuracy` function are similar to what we used before when we converted the logits to the class labels.

We can then apply the function to the training as follows:

In [47]:
```
compute_accuracy(model, train_loader)
```

Out[47]:
```
1.0
```

In [48]:
```
compute_accuracy(model, test_loader)
```

Out[48]:
```
1.0
```

8. Saving and loading models

- In the previous section, we successfully trained a model. Let's now see how we can save a trained model to reuse it later.
- Here's the recommended way how we can save and load models in PyTorch:

In [49]:
```
torch.save(model.state_dict(), "model.pth")
```

- The model's `state_dict` is a Python dictionary object that maps each layer in the model to its trainable parameters (weights and biases). Note that `"model.pth"` is an arbitrary filename for the model file saved to disk. We can give it any name and file ending we like; however, `.pth` and `.pt` are the most common conventions.

Once we saved the model, we can restore it from disk as follows:

In [50]:
```
model = NeuralNetwork(2, 2) # needs to match the original model exactly
model.load_state_dict(torch.load("model.pth", weights_only=True))
```

Out[50]:
```
<All keys matched successfully>
```

- The `torch.load("model.pth")` function reads the file `"model.pth"` and reconstructs the Python dictionary object containing the model's parameters while `model.load_state_dict()` applies these parameters to the model, effectively restoring its learned state from when we saved it.
- Note that the line `model = NeuralNetwork(2, 2)` above is not strictly necessary if you execute this code in the same session where you saved a model. However, I included it here to illustrate that we need an instance of the model in memory to apply the saved parameters. Here, the `NeuralNetwork(2, 2)` architecture needs to match the original saved model exactly.

9. Optimizing training performance with GPUs

- In this last section of this tutorial, we will see how we can utilize GPUs, which will accelerate deep neural network training compared to regular CPUs. First, we will introduce the main concepts behind GPU computing in PyTorch. Then, we will train a model on a single GPU. (Optional) Finally, we'll then look at distributed training using multiple GPUs.

9.1 PyTorch computations on GPU devices

- As you will see, modifying the training loop from the previous section to optionally run on a GPU is relatively simple and only requires changing three lines of code.
- Before we make the modifications, it's crucial to understand the main concept behind GPU computations within PyTorch. First, we need to introduce the notion of devices. In PyTorch, a device is where computations occur, and data resides. The CPU and the GPU are examples of devices. A PyTorch tensor resides in a device, and its operations are executed on the same device.
- Let's see how this works in action. We can double-check that our runtime indeed supports GPU computing via the following code:

In [51]:
```
print(torch.cuda.is_available())
```

```
True
```

- Now, suppose we have two tensors that we can add as follows – this computation will be carried out on the CPU by default:

In [52]:
```
tensor_1 = torch.tensor([1., 2., 3.])
tensor_2 = torch.tensor([4., 5., 6.])

print(tensor_1 + tensor_2)
```

```
tensor([5., 7., 9.])
```

- We can now use the .to() method to transfer these tensors onto a GPU and perform the addition there:

In [53]:
```
tensor_1 = tensor_1.to("cuda")
tensor_2 = tensor_2.to("cuda")

print(tensor_1 + tensor_2)
```

```
tensor([5., 7., 9.], device='cuda:0')
```

- Notice that the resulting tensor now includes the device information, `device='cuda:0'`, which means that the tensors reside on the first GPU. If your machine hosts multiple GPUs, you have the option to specify which GPU you'd like to transfer the tensors to. You can do this by indicating the device ID in the transfer command. For instance, you can use `.to("cuda:0")`, `.to("cuda:1")`, and so on.

- However, it is important to note that all tensors must be on the same device. Otherwise, the computation will fail, as shown below, where one tensor resides on the CPU and the other on the GPU:

In [54]:
```
tensor_1 = tensor_1.to("cpu")
print(tensor_1 + tensor_2)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
/tmp/ipython-input-2079609735.py in <cell line: 0>()
      1 tensor_1 = tensor_1.to("cpu")
----> 2 print(tensor_1 + tensor_2)

RuntimeError: Expected all tensors to be on the same device, but found at least two
devices, cuda:0 and cpu!
```

- In this section, we learned that GPU computations on PyTorch are relatively straightforward. All we have to do is transfer the tensors onto the same GPU device, and PyTorch will handle the rest. Equipped with this information, we can now train the neural network from the previous section on a GPU.

9.2 Single-GPU training

- Now that we are familiar with transferring tensors to the GPU, we can modify the training loop from the previous section, *A typical training loop*, to run on a GPU. This requires only changing three lines of code, as shown in the code below.

In [55]:
```
torch.manual_seed(123)
model = NeuralNetwork(num_inputs=2, num_outputs=2)

# New: Define a device variable that defaults to a GPU.
device = torch.device("cuda")
# New: Transfer the model onto the GPU.
model.to(device)

optimizer = torch.optim.SGD(model.parameters(), lr=0.5)

num_epochs = 3

for epoch in range(num_epochs):
```

```
    model.train()
    for batch_idx, (features, labels) in enumerate(train_loader):

        # New: Transfer the data onto the GPU.
        features, labels = features.to(device), labels.to(device)
        logits = model(features)
        loss = F.cross_entropy(logits, labels) # Loss function

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        ### LOGGING
        print(f"Epoch: {epoch+1:03d}/{num_epochs:03d}"
              f" | Batch {batch_idx:03d}/{len(train_loader):03d}"
              f" | Train/Val Loss: {loss:.2f}")

    model.eval()
    # Optional model evaluation
```

```
Epoch: 001/003 | Batch 000/002 | Train/Val Loss: 0.75
Epoch: 001/003 | Batch 001/002 | Train/Val Loss: 0.65
Epoch: 002/003 | Batch 000/002 | Train/Val Loss: 0.44
Epoch: 002/003 | Batch 001/002 | Train/Val Loss: 0.13
Epoch: 003/003 | Batch 000/002 | Train/Val Loss: 0.03
Epoch: 003/003 | Batch 001/002 | Train/Val Loss: 0.00
```

- We can also use `.to("cuda")` instead of `device = torch.device("cuda")`. Transferring a tensor to `"cuda"` instead of `torch.device("cuda")` works as well and is shorter. We can also modify the statement to the following, which will make the same code executable on a CPU if a GPU is not available, which is usually considered best practice when sharing PyTorch code:

In [56]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

- In the case of the modified training loop above, we probably won't see a speed-up because of the memory transfer cost from CPU to GPU. However, we can expect a significant speed-up when training deep neural networks, especially large language models.

9.3 (Optional) Training with multiple GPUs

- For those interested in training with multiple GPUs, please refer to the original article https://sebastianraschka.com/teaching/pytorch-1h/#93-training-with-multiple-gpus

Summary

- PyTorch is an open-source library that consists of three core components: a tensor library, automatic differentiation functions, and deep learning utilities.
- PyTorch's tensor library is similar to array libraries like NumPy.
- In the context of PyTorch, tensors are array-like data structures to represent scalars, vectors, matrices, and higher-dimensional arrays.
- PyTorch tensors can be executed on the CPU, but one major advantage of PyTorch's tensor format is its GPU support to accelerate computations.
- The automatic differentiation (autograd) capabilities in PyTorch allow us to conveniently train neural networks using backpropagation without manually deriving gradients.
- The deep learning utilities in PyTorch provide building blocks for creating custom deep neural networks.
- PyTorch includes `Dataset` and `DataLoader` classes to set up efficient data loading pipelines.
- It's easiest to train models on a CPU or single GPU.
- (Optional) Using `DistributedDataParallel` is the simplest way in PyTorch to accelerate the training if multiple GPUs are available.