Supplementary code for the Build a Large Language Model From Scratch book by Sebastian Raschka

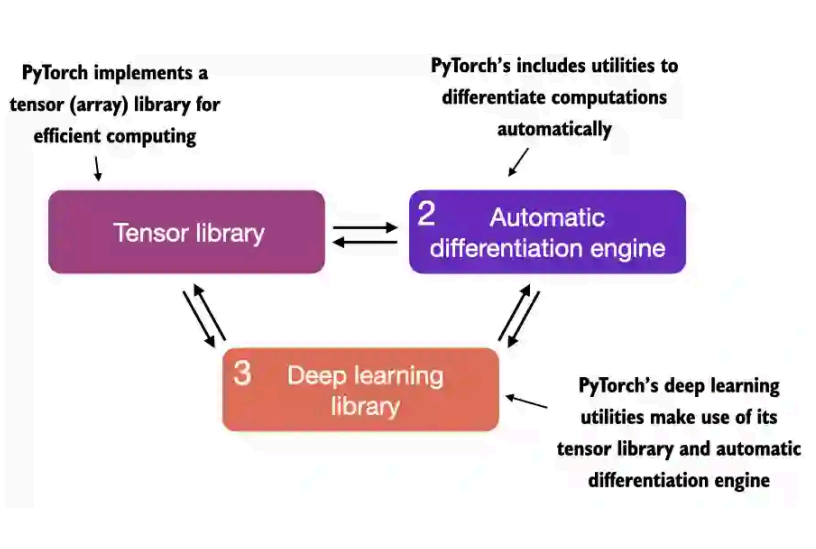
Code repository: https://github.com/rasbt/LLMs-from-scratch

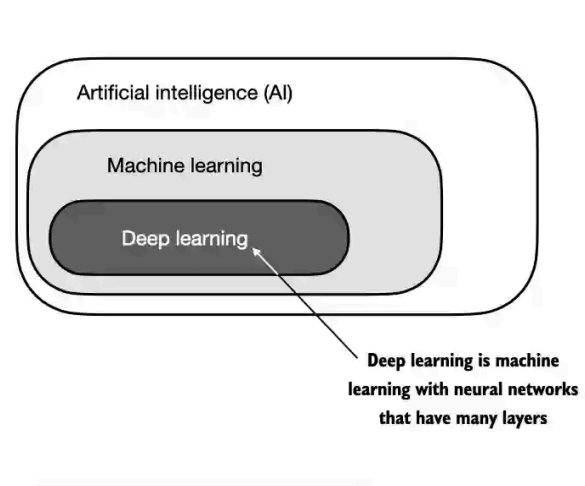
Appendix A: Introduction to PyTorch (Part 1)

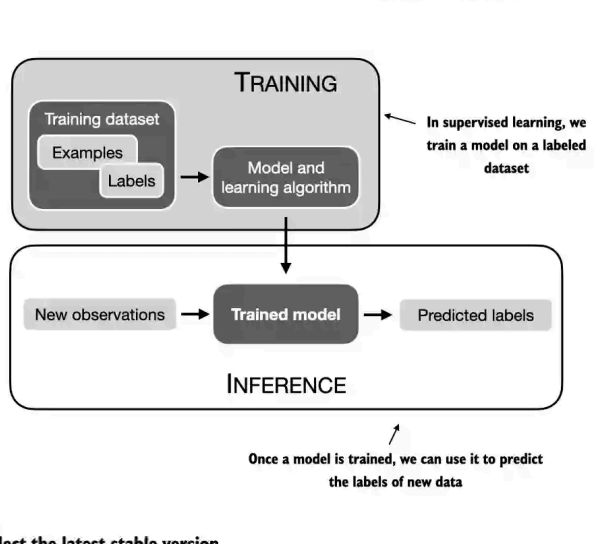
A.1 What is PyTorch

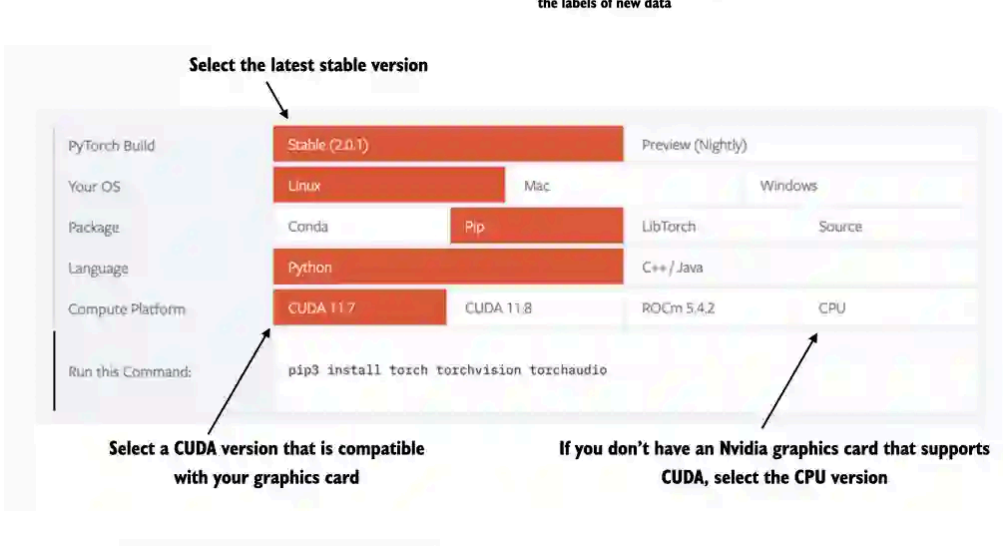
In [1]:
```python
import torch

print(torch.__version__)
```
```
2.4.0
```

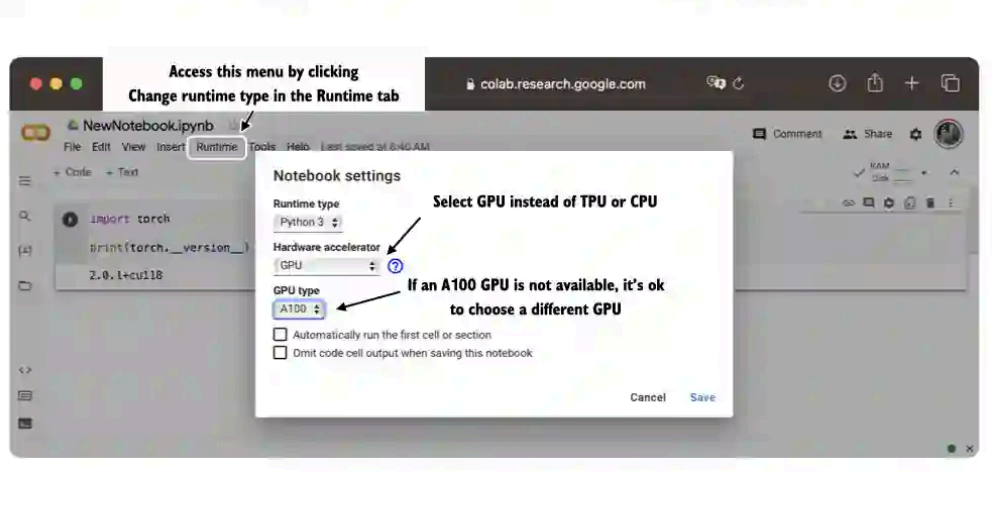
In [2]:
```python
print(torch.cuda.is_available())
```
```
False
```



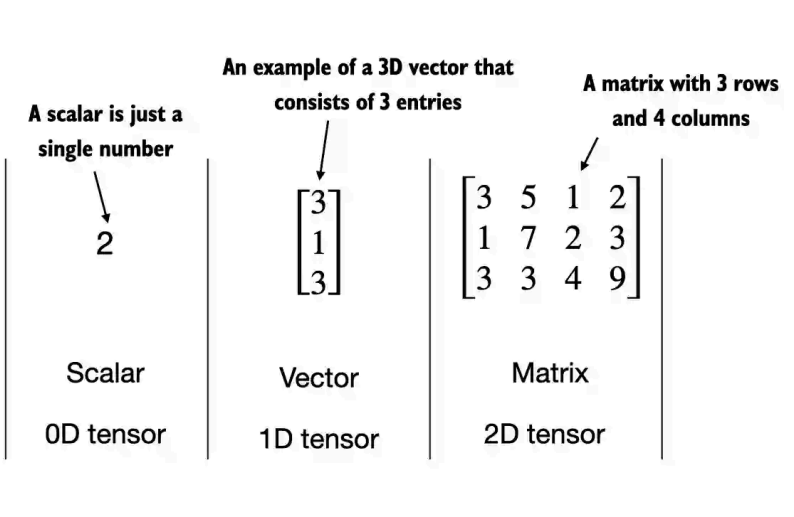








A.2 Understanding tensors



A.2.1 Scalars, vectors, matrices, and tensors

In [3]:
```python
import torch
import numpy as np

# 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_1 = torch.tensor([[[1, 2], [3, 4]],
                           [[5, 6], [7, 8]]])

# create a 3D tensor from NumPy array
ary3d = np.array([[[1, 2], [3, 4]],
                  [[5, 6], [7, 8]]])
tensor3d_2 = torch.tensor(ary3d)  # Copies NumPy array
tensor3d_3 = torch.from_numpy(ary3d)  # Shares memory with NumPy array
```

In [4]:
```python
ary3d[0, 0, 0] = 999
print(tensor3d_2) # remains unchanged
```
```
tensor([[[1, 2],
         [3, 4]],

        [[5, 6],
         [7, 8]]])
```

In [5]:
```python
print(tensor3d_3) # changes because of memory sharing
```
```
tensor([[[999,   2],
         [  3,   4]],

        [[  5,   6],
         [  7,   8]]])
```

A.2.2 Tensor data types

In [6]:
```python
tensor1d = torch.tensor([1, 2, 3])
print(tensor1d.dtype)
```
```
torch.int64
```

In [7]:
```python
floatvec = torch.tensor([1.0, 2.0, 3.0])
print(floatvec.dtype)
```
```
torch.float32
```

In [8]:
```python
floatvec = tensor1d.to(torch.float32)
print(floatvec.dtype)
```
```
torch.float32
```

A.2.3 Common PyTorch tensor operations

In [9]:
```python
tensor2d = torch.tensor([[1, 2, 3],
                         [4, 5, 6]])
tensor2d
```
Out[9]:
```
tensor([[1, 2, 3],
        [4, 5, 6]])
```

In [10]:
```python
tensor2d.shape
```
Out[10]:
```
torch.Size([2, 3])
```

In [11]:
```python
tensor2d.reshape(3, 2)
```
Out[11]:
```
tensor([[1, 2],
        [3, 4],
        [5, 6]])
```

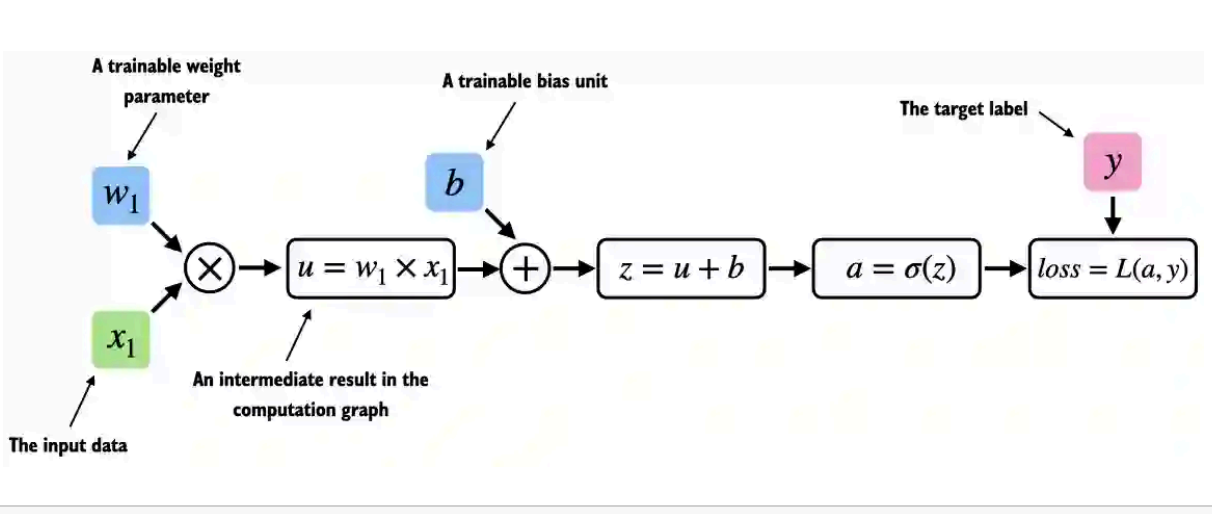
In [12]:
```python
tensor2d.view(3, 2)
```
Out[12]:
```
tensor([[1, 2],
        [3, 4],
        [5, 6]])
```

In [13]:
```python
tensor2d.T
```
Out[13]:
```
tensor([[1, 4],
        [2, 5],
        [3, 6]])
```

In [14]:
```python
tensor2d.matmul(tensor2d.T)
```
Out[14]:
```
tensor([[14, 32],
        [32, 77]])
```

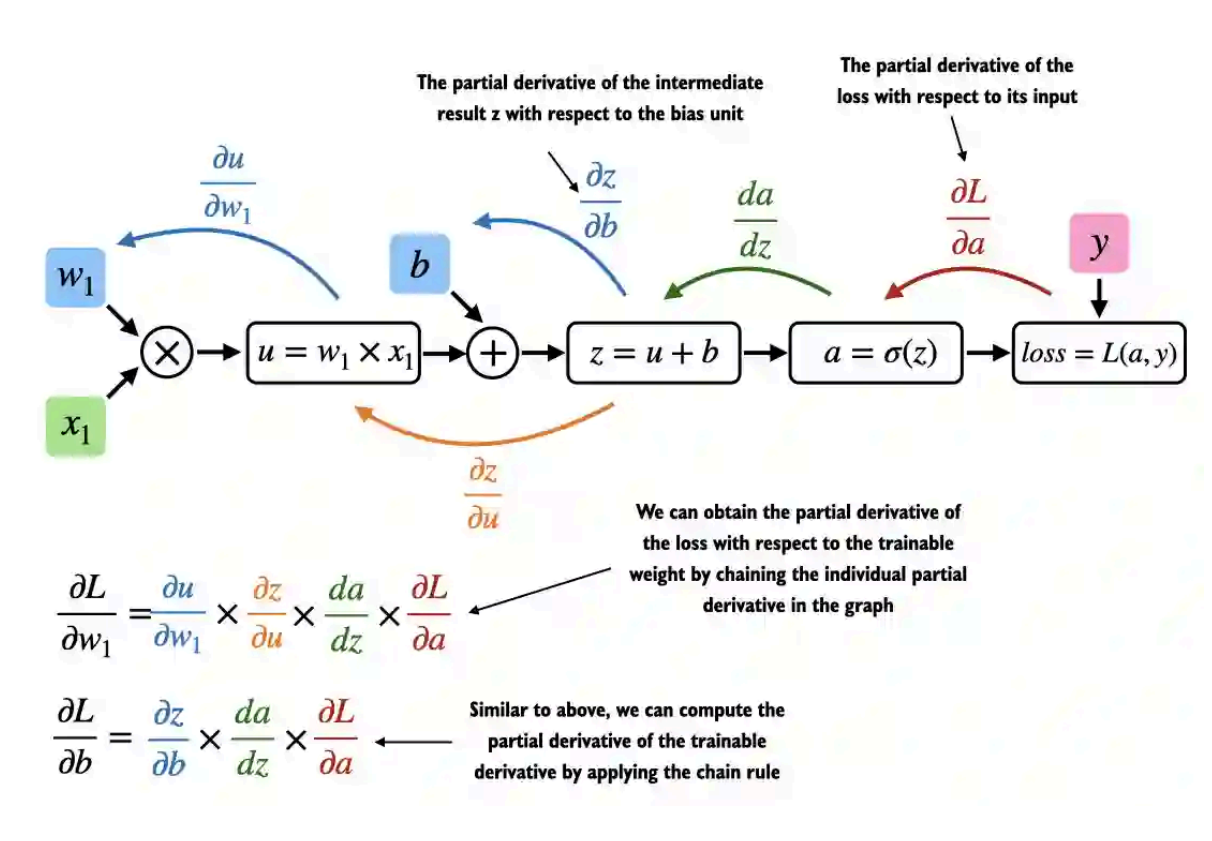
In [15]:
```python
tensor2d @ tensor2d.T
```
Out[15]:
```
tensor([[14, 32],
        [32, 77]])
```

A.3 Seeing models as computation graphs



In [16]:
```python
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)
```

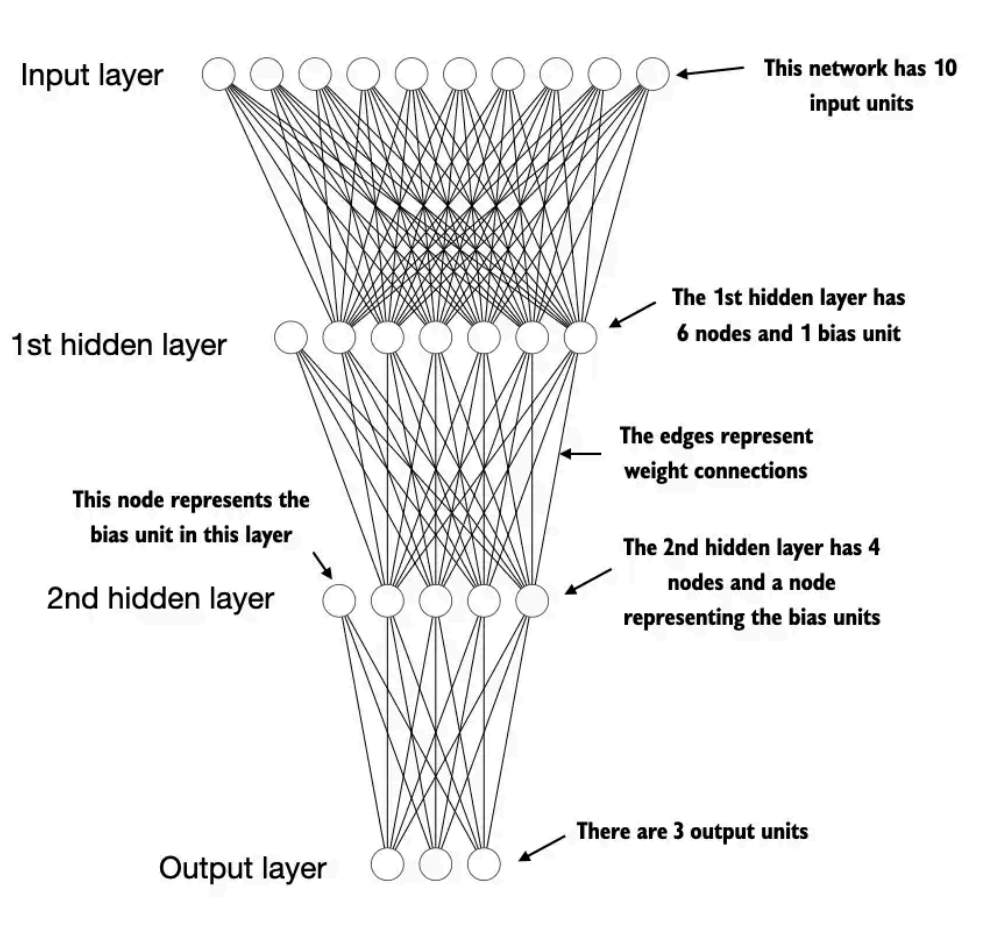
A.4 Automatic differentiation made easy



In [17]:
```python
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)

print(grad_L_w1)
print(grad_L_b)
```
```
(tensor([-0.0898]),)
(tensor([-0.0817]),)
```

In [18]:
```python
loss.backward()

print(w1.grad)
print(b.grad)
```
```
tensor([-0.0898])
tensor([-0.0817])
```

A.5 Implementing multilayer neural networks



Solution : Annotations Part 1

1. Function Class f_θ:

The **function class** f_θ refers to the neural network architecture that maps input data x to output logits. In this case, the function class is defined by the neural network model that consists of two hidden layers and an output layer. The network architecture is:

1. **Input layer**: The input data has num_inputs dimensions.
2. **1st hidden layer**: A fully connected (linear) layer with 30 neurons and a ReLU activation function.
3. **2nd hidden layer**: Another fully connected (linear) layer with 20 neurons and a ReLU activation function.
4. **Output layer**: A fully connected (linear) layer with num_outputs neurons, which outputs the logits for the final predictions.

Mathematically, the function class $f_\theta(x)$ can be represented as:

$$f_\theta(x) = W_3 \cdot \text{ReLU}(W_2 \cdot \text{ReLU}(W_1 \cdot x + b_1) + b_2) + b_3$$

where:

- W_1, b_1 are the weights and biases of the 1st hidden layer.
- W_2, b_2 are the weights and biases of the 2nd hidden layer.
- W_3, b_3 are the weights and biases of the output layer.
- ReLU is the ReLU activation function applied element-wise after the first two linear layers.

The function class describes how the input is transformed through the layers and activation functions to produce the output logits.

2. Parameter Space θ:

The **parameter space** θ refers to the set of all learnable parameters (weights and biases) in the neural network. In this model, θ includes:

- W_1 and b_1: Weights and biases of the 1st hidden layer (dimensions: [num_inputs, 30] for weights and [30] for biases).
- W_2 and b_2: Weights and biases of the 2nd hidden layer (dimensions: [30, 20] for weights and [20] for biases).
- W_3 and b_3: Weights and biases of the output layer (dimensions: [20, num_outputs] for weights and [num_outputs] for biases).

These parameters are learned during training by minimizing a loss function through optimization algorithms like stochastic gradient descent (SGD) or Adam. The parameter space θ is the collection of all these weights and biases that define the transformations applied by the neural network to the input data.

In [19]:
```python
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
```

In [20]:
```python
model = NeuralNetwork(50, 3)
```

In [21]:
```python
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)
  )
)
```

In [22]:
```python
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
```

In [23]:
```python
print(model.layers[0].weight)
```
```
Parameter containing:
tensor([[ 0.1182,  0.0606, -0.1292,  ..., -0.1126,  0.0735, -0.0597],
        [-0.0249,  0.0154, -0.0476,  ..., -0.1001, -0.1288,  0.1295],
        [ 0.0641,  0.0018, -0.0367,  ..., -0.0990, -0.0424, -0.0043],
        ...,
        [ 0.0618,  0.0867,  0.1361,  ..., -0.0254,  0.0399,  0.1006],
        [ 0.0842, -0.0512, -0.0960,  ..., -0.1091,  0.1242, -0.0428],
        [ 0.0518, -0.1390, -0.0923,  ..., -0.0954, -0.0668, -0.0037]],
       requires_grad=True)
```

In [24]:
```python
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)
```

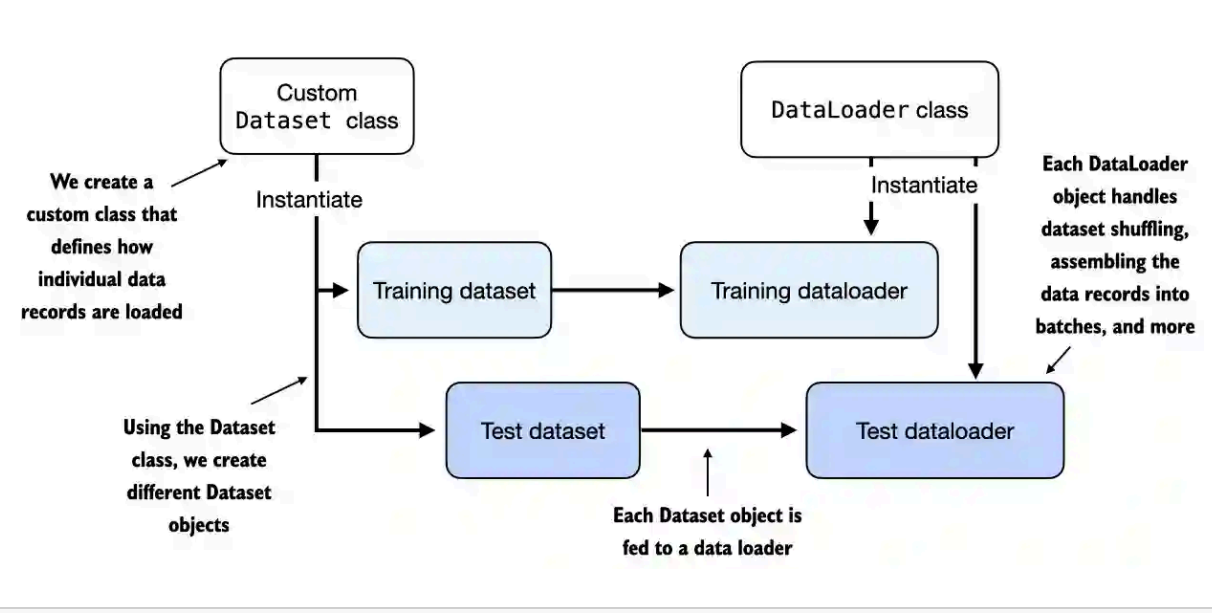
In [25]:
```python
print(model.layers[0].weight.shape)
```
```
torch.Size([30, 50])
```

In [26]:
```python
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 [27]:
```python
with torch.no_grad():
    out = model(X)
print(out)
```
```
tensor([[-0.1262,  0.1080, -0.1792]])
```

In [28]:
```python
with torch.no_grad():
    out = torch.softmax(model(X), dim=1)
print(out)
```
```
tensor([[0.3113, 0.3934, 0.2952]])
```

A.6 Setting up efficient data loaders



In [29]:
```python
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 [30]:
```python
X_test = torch.tensor([
    [-0.8, 2.8],
    [2.6, -1.6],
])

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

In [31]:
```python
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)
```

In [32]:
```python
len(train_ds)
```
Out[32]:
```
5
```

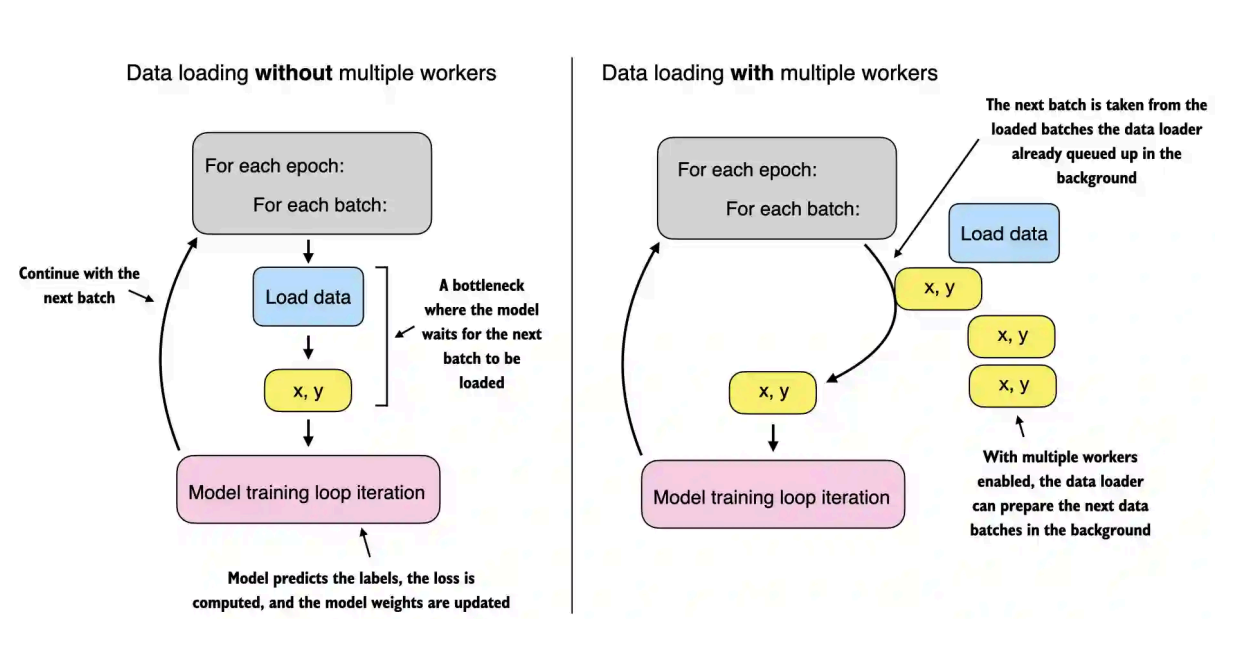
In [33]:
```python
from torch.utils.data import DataLoader

torch.manual_seed(123)

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

In [34]:
```python
test_ds = ToyDataset(X_test, y_test)

test_loader = DataLoader(
    dataset=test_ds,
    batch_size=2,
    shuffle=False,
    num_workers=0
)
```

In [35]:
```python
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])
```

In [36]:
```python
train_loader = DataLoader(
    dataset=train_ds,
    batch_size=2,
    shuffle=True,
    num_workers=0,
    drop_last=True
)
```

In [37]:
```python
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])
```



A.7 A typical training loop

Solution Annotations : Part 2

1. Loss Function l:

The **loss function** used in the code is **cross-entropy loss**, which is applied to the predicted logits and the true labels:

```python
loss = F.cross_entropy(logits, labels)
```

- **Cross-entropy loss** is used for classification tasks and measures the difference between the predicted class probabilities (after applying softmax) and the true class labels. It computes the negative log-likelihood of the true class based on the predicted probabilities.
- For a single data point (x_i, y_i), the cross-entropy loss $l(f_\theta(x_i), y_i)$ is given by:

$$l(f_\theta(x_i), y_i) = -\log\left(\frac{\exp(f_\theta(x_i)_{y_i})}{\sum_{k=1}^{K}\exp(f_\theta(x_i)_k)}\right)$$

  where:
  - $f_\theta(x_i)_k$ is the predicted logit for class k,
  - y_i is the true class label for input x_i,
  - K is the total number of classes (in this case, 2).

This loss function is computed for each data point and averaged over the mini-batch during training.

2. Empirical Risk L:

The **empirical risk** L represents the average loss over the entire training set. It is the quantity the model is trying to minimize during training. In the case of mini-batch training (as in this code), the empirical risk is computed as the average cross-entropy loss over the data points in the mini-batch.

Mathematically, if there are n training examples, the empirical risk $L(\theta)$ is:

$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n} l(f_\theta(x_i), y_i)$$

where $l(f_\theta(x_i), y_i)$ is the cross-entropy loss for each example (x_i, y_i).

In the code, the empirical risk is minimized using stochastic gradient descent (SGD) as the optimizer:

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

This step updates the model parameters θ to minimize the empirical risk.

3. Probability Distribution p_θ (Log-Likelihood View):

In the **log-likelihood view**, the cross-entropy loss can be interpreted as minimizing the negative log-likelihood of the true class under a predicted probability distribution.

The logits output by the model $f_\theta(x_i)$ are converted into class probabilities using the **softmax function**:

$$p_\theta(y_i|x_i) = \frac{\exp(f_\theta(x_i)_{y_i})}{\sum_{k=1}^{K}\exp(f_\theta(x_i)_k)}$$

where:

- $f_\theta(x_i)_k$ is the logit for class k,
- $p_\theta(y_i|x_i)$ is the predicted probability for the true class y_i.

The **log-likelihood** for a single data point is the log of the probability assigned to the correct class:

$$\log(p_\theta(y_i|x_i)) = \log\left(\frac{\exp(f_\theta(x_i)_{y_i})}{\sum_{k=1}^{K}\exp(f_\theta(x_i)_k)}\right)$$

The **negative log-likelihood** is what the cross-entropy loss computes, and by minimizing this loss, the model maximizes the likelihood of the correct class under the predicted probability distribution. The loss function and the empirical risk are thus directly related to the likelihood function of the true labels given the model's predictions.

In [38]:
```python
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
```

In [39]:
```python
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]])
```

In [40]:
```python
torch.set_printoptions(sci_mode=False)
probas = torch.softmax(outputs, dim=1)
print(probas)

predictions = torch.argmax(probas, dim=1)
print(predictions)
```
```
tensor([[    0.9991,     0.0009],
        [    0.9982,     0.0018],
        [    0.9949,     0.0051],
        [    0.0491,     0.9509],
        [    0.0307,     0.9693]])
tensor([0, 0, 0, 1, 1])
```

In [41]:
```python
predictions = torch.argmax(outputs, dim=1)
print(predictions)
```
```
tensor([0, 0, 0, 1, 1])
```

In [42]:
```python
predictions == y_train
```
Out[42]:
```
tensor([True, True, True, True, True])
```

In [43]:
```python
torch.sum(predictions == y_train)
```
Out[43]:
```
tensor(5)
```

In [44]:
```python
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()
```

In [45]:
```python
compute_accuracy(model, train_loader)
```
Out[45]:
```
1.0
```

In [46]:
```python
compute_accuracy(model, test_loader)
```
Out[46]:
```
1.0
```

A.8 Saving and loading models

In [47]:
```python
torch.save(model.state_dict(), "model.pth")
```

In [48]:
```python
model = NeuralNetwork(2, 2) # needs to match the original model exactly
model.load_state_dict(torch.load("model.pth", weights_only=True))
```
Out[48]:
```
<All keys matched successfully>
```

A.9 Optimizing training performance with GPUs

A.9.1 PyTorch computations on GPU devices

See [code-part2.ipynb](#)

A.9.2 Single-GPU training

See [code-part2.ipynb](#)

A.9.3 Training with multiple GPUs

See [DDP-script.py](#)