Data Science & Machine Learning Lab

PyTorch

Flavio Giobergia

Introduction to PyTorch

- PyTorch is an open-source Python library, mainly used for deep learning
  - And used by other libraries (HuggingFace, PyTorch Lightning, Ignite, TorchMetrics, Torchvision, …)
- **Agenda**
  - Tensors & automatic differentiation
  - Linear layers
  - Non-linear activation functions
  - Modules
  - Loss functions
  - Optimizer
  - Training loop

Tensors in PyTorch

- Tensors
  - core data structure in PyTorch
  - (similar to NumPy arrays)
  - They can be multi-dimensional arrays
  - Used as the fundamental building blocks for all computations.
- PyTorch Tensors can be easily converted from/to NumPy array
- Very similar interface (functions, broadcasting, indexing) to NumPy!

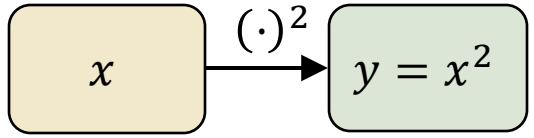
```python
import numpy as np
import torch

np_array = np.array([1, 2, 3, 4])

# Convert np array to PyTorch tensor
tensor = torch.from_numpy(np_array)
# or
tensor = torch.tensor(np_array)

# Convert back to a NumPy array
new_np_array = tensor.numpy()

print(type(np_array))     # <class 'numpy.ndarray'>
print(type(tensor))       # <class 'torch.Tensor'>
print(type(new_np_array)) # <class 'numpy.ndarray'>
```

Tensors vs NumPy arrays

- **Autograd**: Tensors come with automatic differentiation (autograd) which is essential for deep learning, whereas NumPy arrays do not have this capability
- **GPU Support**: PyTorch tensors can be moved to a GPU to accelerate computations, while NumPy arrays are CPU-bound
- **Additional Operations**: PyTorch allows additional operations on tensors for functionalities tailored for deep learning (activation functions, layers, quantization, mixed precision operations, sparse matrices

Automatic Differentiation with PyTorch

$$\frac{\partial y}{\partial x} = \frac{\partial x^2}{\partial x} = 2x$$



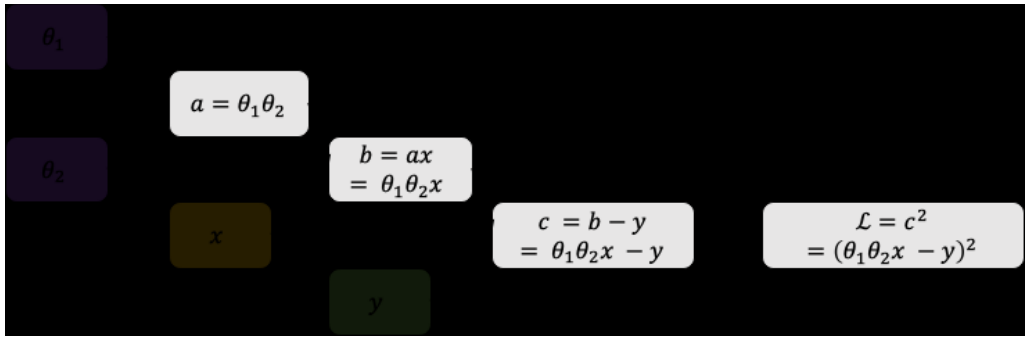
- You can enable automatic differentiation by setting `requires_grad=True` on a tensor
- The computation graph is built for all tensors that require keeping track of the gradient
- The .backward() method backpropagates gradients across the graph

```python
import torch

x = torch.tensor(3.0, requires_grad=True)
y = x ** 2

# This will backpropagate the gradients
# through the computation graph, resulting
# in thee computation of dy/dx
y.backward()

# x.grad contains the gradient of y with respect to x

# 6.0, since d(x^2)/dx = 2x, computed at x=3
print(x.grad)
```

- The .grad attribute contains the gradient computed for each tensor
- If multiple backward passes are executed, the gradient accumulates

Two-parameter model

```python
import torch

# Input and output scalars
x = torch.tensor(1.0) # example input
y = torch.tensor(2.0) # example output

# Model parameters (with requires_grad=True!)
theta1 = torch.tensor(3.0, requires_grad=True)
theta2 = torch.tensor(4.0, requires_grad=True)

# Model prediction: y_pred = x * theta1 * theta2
y_pred = x * theta1 * theta2

# Define the loss function: (y_pred - y)^2
loss = (y_pred - y) ** 2

# Perform backpropagation
loss.backward()

# Now theta1.grad and theta2.grad will contain the gradients
print(theta1.grad) # d Loss/d(theta1) (80.)
print(theta2.grad) # d Loss/d(theta2) (60.)
```

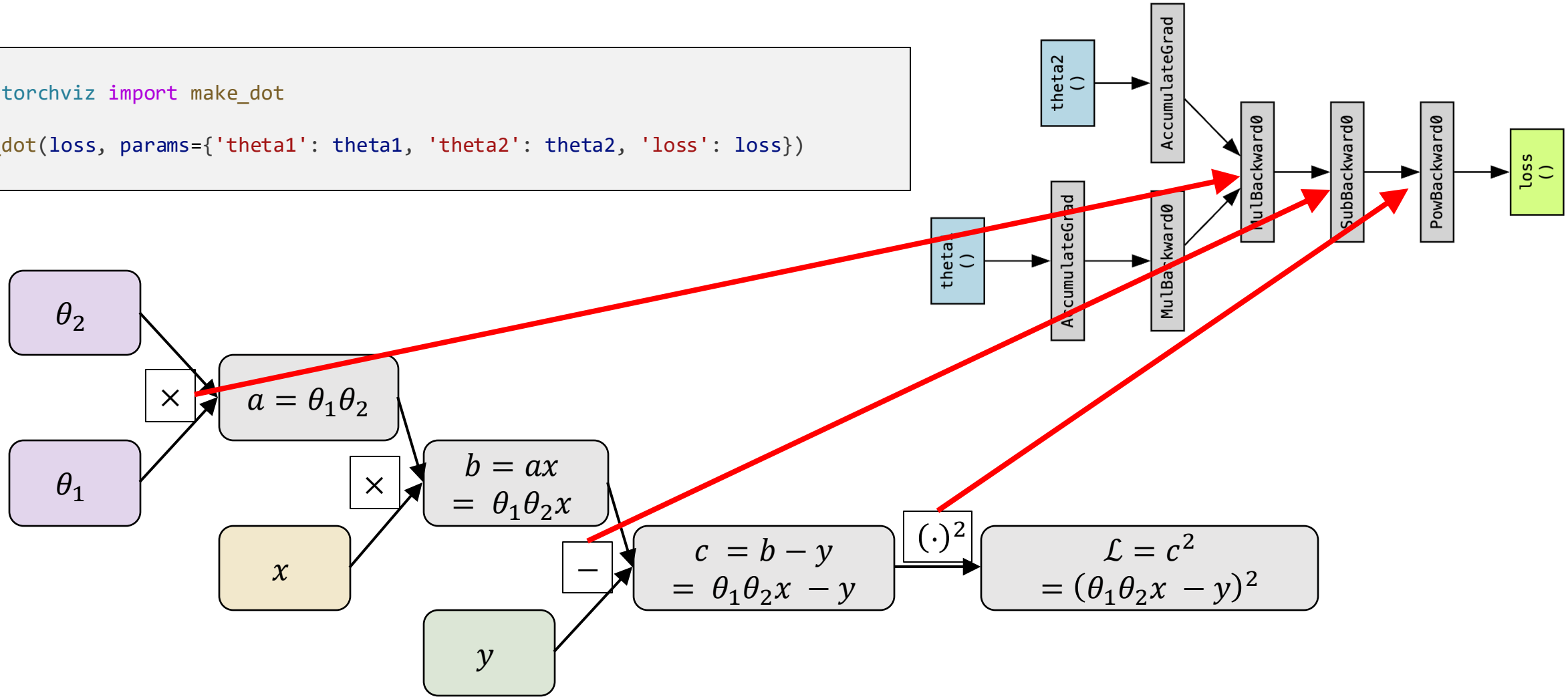


$$\frac{\partial \mathcal{L}}{\partial \theta_1} = 2(\theta_1\theta_2 x - y)x\theta_2$$

$$\frac{\partial \mathcal{L}}{\partial \theta_2} = 2(\theta_1\theta_2 x - y)x\theta_1$$

Visualizing computation graph

```
from torchviz import make_dot

make_dot(loss, params={'theta1': theta1, 'theta2': theta2, 'loss': loss})
```

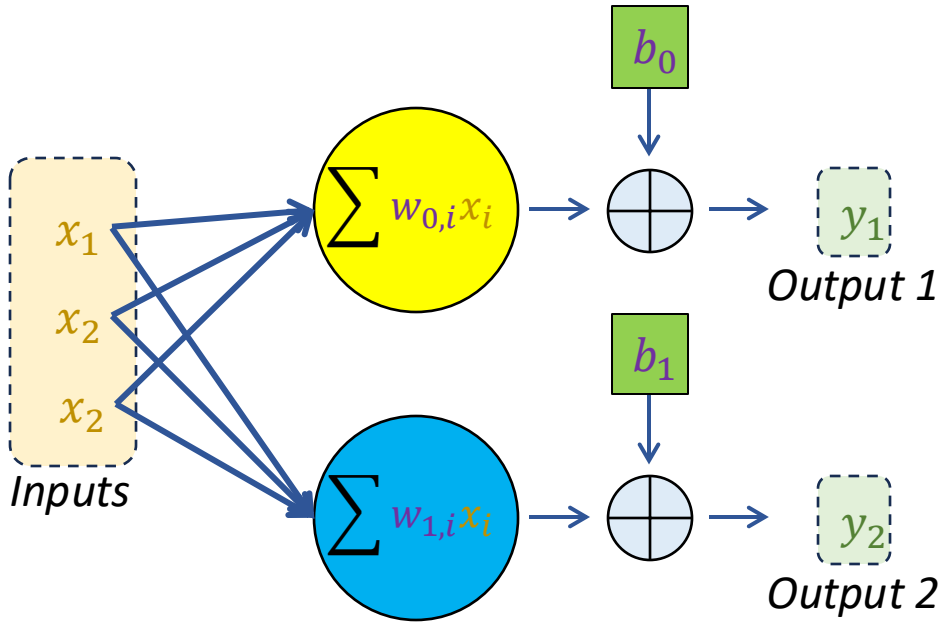


Linear Layers

- A linear layer (also called a fully connected layer) applies a linear transformation to the input data.
- A linear layer is defined by a weight matrix and an optional bias (bias=True, by default)
- Linear layers (like all Torch layers) require gradient computation by default

```python
import torch
import torch.nn as nn

# Create a linear layer that
# - takes in 3 input features, and
# - outputs 2 features
linear_layer = nn.Linear(in_features=3, out_features=2)

# The layer has a weight and a bias
print("Weight:", linear_layer.weight)
print("Bias:", linear_layer.bias)
```

```
Weight:Parameter containing:
tensor([[-0.4399, 0.1994, -0.5340],
        [-0.2535, 0.1024, -0.3496]], requires_grad=True)
Bias: Parameter containing:
tensor([0.5200, 0.2996], requires_grad=True)
```



Using a linear layer

```python
import torch
import torch.nn as nn

# Create a linear layer with 3 input features and 2 output features
linear_layer = nn.Linear(in_features=3, out_features=2)

# Create a 3-element input tensor
input_tensor = torch.tensor([[1.0, 2.0, 3.0]])

# Pass it through the linear layer
output_tensor = linear_layer(input_tensor)

# Print the output
print("Output:", output_tensor)
```

```
Output: tensor([[-1.1593, -0.8618]], grad_fn=<AddmmBackward0>)
```

```python
weight = linear_layer.weight
bias = linear_layer.bias

output_tensor = input_tensor @ weight.t() + bias
```

Activation functions

- Full list of available activation functions:
  - https://docs.pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity

`nn.ELU`	Applies the Exponential Linear Unit (ELU) function, element-wise.
`nn.Hardshrink`	Applies the Hard Shrinkage (Hardshrink) function element-wise.
`nn.Hardsigmoid`	Applies the Hardsigmoid function element-wise.
`nn.Hardtanh`	Applies the HardTanh function element-wise.
`nn.Hardswish`	Applies the Hardswish function, element-wise.
`nn.LeakyReLU`	Applies the LeakyReLU function element-wise.
`nn.LogSigmoid`	Applies the Logsigmoid function element-wise.
`nn.MultiheadAttention`	Allows the model to jointly attend to information from different representation subspaces.
`nn.PReLU`	Applies the element-wise PReLU function.
`nn.ReLU`	Applies the rectified linear unit function element-wise.
`nn.ReLU6`	Applies the ReLU6 function element-wise.
`nn.RReLU`	Applies the randomized leaky rectified linear unit function, element-wise.
`nn.SELU`	Applies the SELU function element-wise.
`nn.CELU`	Applies the CELU function element-wise.

`nn.GELU`	Applies the Gaussian Error Linear Units function.
`nn.Sigmoid`	Applies the Sigmoid function element-wise.
`nn.SiLU`	Applies the Sigmoid Linear Unit (SiLU) function, element-wise.
`nn.Mish`	Applies the Mish function, element-wise.
`nn.Softplus`	Applies the Softplus function element-wise.
`nn.Softshrink`	Applies the soft shrinkage function element-wise.
`nn.Softsign`	Applies the element-wise Softsign function.
`nn.Tanh`	Applies the Hyperbolic Tangent (Tanh) function element-wise.
`nn.Tanhshrink`	Applies the element-wise Tanhshrink function.
`nn.Threshold`	Thresholds each element of the input Tensor.
`nn.GLU`	Applies the gated linear unit function.

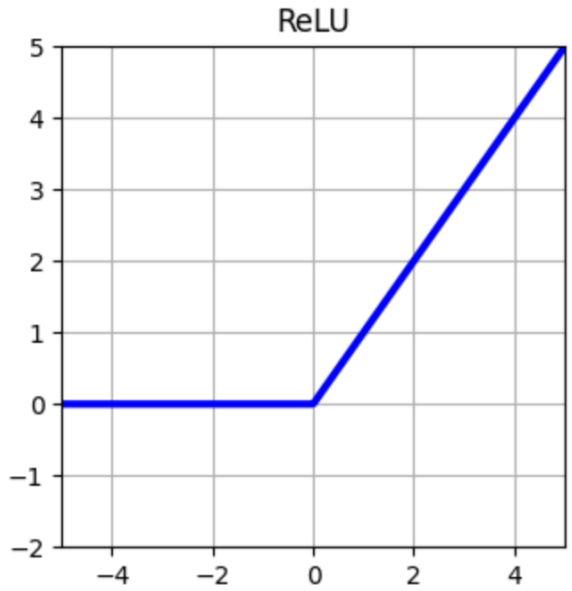
`nn.Softmin`	Applies the Softmin function to an n-dimensional input Tensor.
`nn.Softmax`	Applies the Softmax function to an n-dimensional input Tensor.
`nn.Softmax2d`	Applies SoftMax over features to each spatial location.
`nn.LogSoftmax`	Applies the $\log(\text{Softmax}(x))$ function to an n-dimensional input Tensor.
`nn.AdaptiveLogSoftmaxWithLoss`	Efficient softmax approximation.

ReLU Activation Function

- The ReLU (Rectified Linear Unit) activation function is defined as

$$f(x) = \max(0, x)$$

- (It sets all negative values to zero)




```python
import torch

# Example input vector
x = torch.tensor([-1.0, 0.0, 1.0, 2.0])

# Apply ReLU
relu_output = torch.relu(x)
print("ReLU output:", relu_output)
```

```
ReLU output: tensor([0., 0., 1., 2.])
```

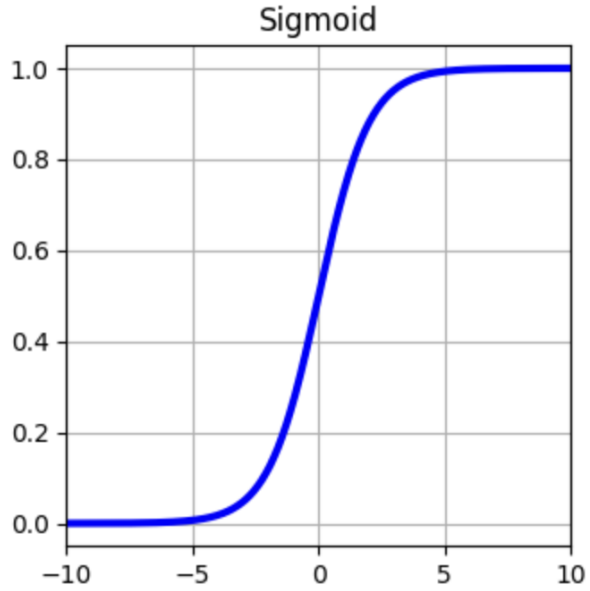
Sigmoid function

- The sigmoid activation function squashes input values into a range between 0 and 1

$$1/(1+\exp(-x))$$

```python
import torch

# Example input vector
x = torch.tensor([-3.0, -2.0, -1.0, 0.0, 1.0, 2.0, 3.0])

# Apply sigmoid
sigmoid_output = torch.sigmoid(x)
print("Sigmoid output:", sigmoid_output)
```

```
Sigmoid output: tensor([0.0474, 0.1192, 0.2689, 0.5000, 0.7311, 0.8808, 0.9526])
```



Softmax Activation Function

- The softmax activation function converts a list of logits (output of a neural network) into a valid probability distribution
  - (probabilities non-negative and that sum to 1)

```
import torch

# Example 2D input (two rows)
x = torch.tensor([
    [1.0, 2.0, 3.0],
    [2.0, 4.0, 6.0]
])

# Apply softmax to each row (dim=1 means across columns)
softmax_output = torch.softmax(x, dim=1)
print("Softmax output (row-wise):", softmax_output)
```

```
Softmax output (row-wise):
tensor([
    [0.0900, 0.2447, 0.6652],
    [0.0159, 0.1173, 0.8668]
])
```

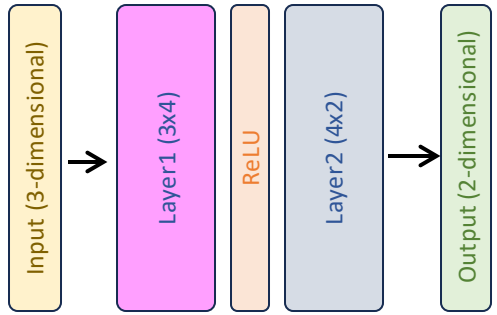
$$softmax(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

$$softmax([2,4,6])_1 = \frac{\exp(2)}{\exp(2) + \exp(4) + \exp(6)} = 0.0159$$

Modules

- In PyTorch, models are implemented as extensions of the `nn.Module` class
- `nn.Module` implements some overhead allowing them to be used with various PyTorch functionalities
- Two main methods need to be implemented:
- `__init__()`: the constructor, where the architecture of the network is defined
  - i.e., what components are contained in the network
  - Remember to always call the parent's `__init__()` at the beginning of the constructor
    - `super().__init__()`
- `forward()`: defines the forward pass through the layers
  - i.e., how the data "flows" through the network

2-layer fully-connected model




```python
import torch
import torch.nn as nn

class SimpleNeuralNet(nn.Module):

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

    # First linear layer (3 inputs to 4 outputs)
    self.layer1 = nn.Linear(3, 4)

    # Second linear layer (4 inputs to 2 outputs)
    self.layer2 = nn.Linear(4, 2)

  def forward(self, x):
    x = self.layer1(x) # Pass through first layer
    x = torch.relu(x)  # Apply ReLU
    x = self.layer2(x) # Pass through second layer
    return x
```

```python
# Create an instance
model = SimpleNeuralNet()

# two samples, each with three features
input_tensor = torch.tensor([
  [1.0, 2.0, 3.0],
  [2.0, 4.0, 6.0]
])

# Forward pass through the model
output = model(input_tensor)
print("Model output:", output)
```

```
Model output:
tensor([
  [ 0.3887, -0.0106],
  [ 0.3202, -0.1406]
], grad_fn=<AddmmBackward0>)
```

Loss functions

- A loss function measures how well the model's predictions match the actual targets
  - It is the function we aim to minimize during training
- PyTorch provides a wide variety of loss functions
  - Most common ones
    - `nn.MSELoss()` for regression
    - `nn.BCEWithLogitsLoss()` for binary classification
    - `nn.CrossEntropyLoss()` for multi-class classification

Note that `BCEWithLogitsLoss()` and `CrossEntropyLoss()` expect to receive the predictions as **logits**, and will apply the activation function (sigmoid and softmax, respectively).

This improves the numerical stability of the result (it avoids some unnecessary logarithms and exponentiations).

```
import torch.nn as nn

loss_fn = nn.MSELoss()

predictions = model(...)
targets = ...

loss = loss_fn(predictions, targets)
```

Optimizers

- Optimizers are algorithms that help adjust the model's parameters to minimize the loss function (e.g., Gradient Descent)
  - They perform the parameter updates during training
  - $\theta_{t+1} := \theta_t - \alpha \nabla_\theta \mathcal{L}(\theta_t)$
- In PyTorch, you create an optimizer by passing in the model's parameters
  - A common choice is the Adam optimizer [1]
  - Optimizers are included in the `torch.optim` module
  - The .step() method is used to take a "GD" step (i.e., to update the weights once)

```
import torch.optim as optim

# Create an Adam optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

[1] Kingma, Diederik P. "Adam: A method for stochastic optimization." *arXiv preprint arXiv:1412.6980* (2014).

Gradient descent step

A single gradient descent (GD) step involves 5 main sub-steps:

1. **Clear the gradients**
   - Before the backward pass, clear old gradients using `optimizer.zero_grad()`
2. **Forward pass**
   - Compute predictions by passing inputs through the model
3. **Compute the loss function**
   - Based on model's predictions and ground truth
4. **Backward pass**
   - Compute gradients with respect to the loss using `loss.backward()`
5. **Update parameters**
   - Adjust the model's weights using `optimizer.step()`

```
optimizer.zero_grad()      # Step 1: Clear gradients
output = model(x)          # Step 2: Forward pass
loss = loss_fn(output, y) # Step 3: Compute loss
loss.backward()            # Step 4: Backward pass
optimizer.step()           # Step 5: Update parameters
```

Training loop & SGD

- A single gradient descent update rarely converges to a good solution
- Multiple GD steps are needed to iteratively reduce the loss function
- One complete pass through the training set is called an *epoch*
- The "ideal" gradient is computed over the *entire* dataset
  - However, computing this full gradient is *computationally expensive*
  - Instead, we approximate the ideal gradient using a *subset* (*batch*) of the data
  - This yields a *noisier but much faster* estimate of the true gradient.
    - Stochastic Gradient Descent (SGD)
  - PyTorch has utilities to handle datasets (`torch.utils.data.Dataset`) and batches (`torch.utils.data.DataLoader`)

End-to-end exercise (Iris dataset)

1. **Load the data**
2. **Define a simple neural network**
    1. Input: 4 features (from Iris)
    2. Hidden layer: 4 → 2
    3. Non-linearity (ReLU)
    4. Output layer: 2 → 3 (logits)
    5. Softmax
3. **Train with a few steps of GD (with CrossEntropyLoss)**
4. **Check out the performance of the model**
5. **Plot the "latent" space (intermediate space, after 1st layer)**