PyTorch
- Building an NN model in PyTorch

Dr. Huiping Cao

Outline

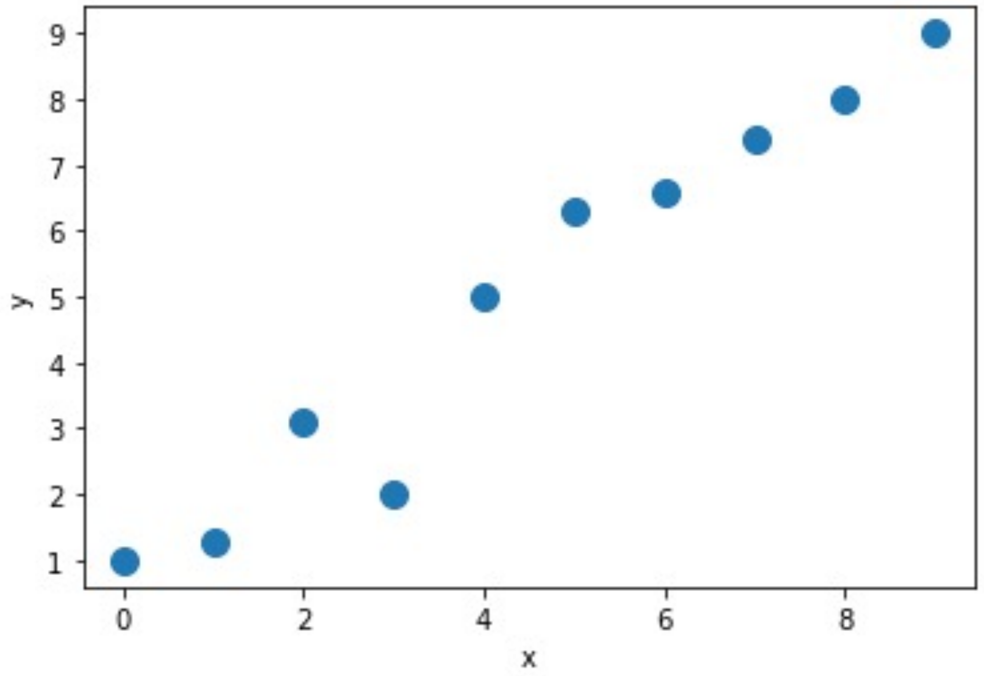
- Implement our first predictive model in PyTorch
- The **PyTorch neural network module torch.nn** is an elegantly designed module developed to help create and train NNs. It allows easy prototyping and the building of complex models in just a few lines of code.
- Simple example: linear regression model
  - Use only **basic PyTorch tensor** operations
- We will incrementally add features from **torch.nn** and **torch.optim**
- Examples of building an NN model using the **nn.Module** class
  - Most commonly used approach for building an NN in PyTorch is through nn.Module,

Building a linear regression model

- Step 1: create a toy dataset in NumPy and visualize it:

```
X_train = np.arange(10, dtype='float32').reshape((10, 1))
y_train = np.array([1.0, 1.3, 3.1, 2.0, 5.0,6.3, 6.6,7.4, 8.0,9.0], dtype='float32')
print(X_train.shape, y_train.shape)
plt.plot(X_train, y_train, 'o', markersize=10)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

```
Output:

(10, 1) (10,)
```



Building a linear regression model

- Step 2: standardize the features (mean centering and dividing by the standard deviation) and create a PyTorch Dataset for the training set and a corresponding DataLoader

```
from torch.utils.data import TensorDataset

#standardize features, create tensors
X_train_norm = (X_train - np.mean(X_train)) / np.std(X_train)
X_train_norm_tensor = torch.from_numpy(X_train_norm)
y_train_tensor = torch.from_numpy(y_train)

train_ds = TensorDataset(X_train_norm_tensor, y_train_tensor)

batch_size = 1
train_dl = DataLoader(train_ds, batch_size, shuffle=True)
```

Building a linear regression model

- Step 3: Define the models
  - (1) Define our **model** for linear regression as $z = w^T x + b$.
  - (2) Define the **loss function** that we want to minimize to find the optimal model weights. Here, we will choose the **mean squared error** (**MSE**) as our loss function.
  - (3) **Optimizer**: we will use **stochastic gradient descent (SGD) t**o learn the weight parameters of the model. To implement the SGD algorithm, we need to compute the gradients. Rather than manually computing the gradients, we will use **PyTorch's torch.autograd.backward** function.

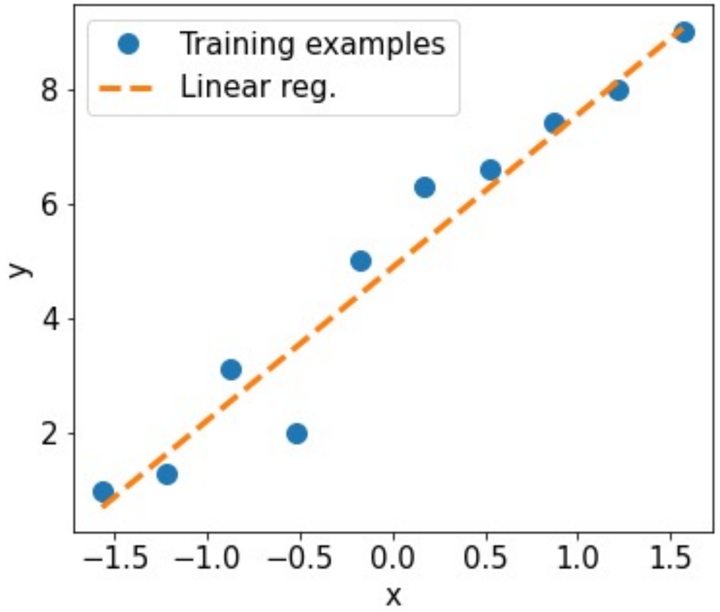
```
def model(xb):
        return xb @ weight + bias

def loss_fn(input, target):
        return (input-target).pow(2).mean()
```

Building a linear regression model

- Step 4: Set the **learning rate** and train the model for 200 epochs.
- Step 5: For the test data, we will create a NumPy array of values evenly spaced between 0 and 9. Since we trained our model with standardized features, we will also apply the **same standardization to the test data.**

Output:

Final Parameters:
2.669107387542725 4.879678249359131



Model training via the torch.nn and torch.optim modules

- Writing the loss function and gradient updates can be a **repeatable** task across different projects.
- The **torch.nn module** provides a set of **loss functions**
- The **torch.optim** supports most commonly used **optimization algorithms** that can be called to update the parameters based on the computed gradients.

OLD WAY

```
def model(xb):
        return xb @ weight + bias

def loss_fn(input, target):
        return (input-target).pow(2).mean()
```

PyTorch WAY

```
loss_fn = nn.MSELoss(reduction='mean’)
input_size = 1
output_size = 1
model = nn.Linear(input_size, output_size)
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

Build a linear regression model

OLD WAY

```
learning_rate = 0.001
num_epochs = 200
log_epochs = 100

for epoch in range(num_epochs):
    for x_batch, y_batch in train_dl:
        pred = model(x_batch)
        loss = loss_fn(pred, y_batch)
        loss.backward()
        with torch.no_grad():
            weight -= weight.grad * learning_rate
            bias -= bias.grad * learning_rate
            weight.grad.zero_()
            bias.grad.zero_()
        if epoch % log_epochs==0:
            print(f'Epoch {epoch} Loss {loss.item():.4f}')

print('Final Parameters:', weight.item(), bias.item())
```

PyTorch WAY

```
learning_rate = 0.001
num_epochs = 200
log_epochs = 100

for epoch in range(num_epochs):
    for x_batch, y_batch in train_dl:
        pred = model(x_batch)[:, 0] # 1. Generate predictions
        loss = loss_fn(pred, y_batch) # 2. Calculate loss
        loss.backward() # 3. Compute gradients

        optimizer.step() # 4. Update parameters using gradients
        optimizer.zero_grad() # 5. Reset the gradients to zero


        if epoch % log_epochs==0:
            print(f'Epoch {epoch} Loss {loss.item():.4f}')

print('Final Parameters:', model.weight.item(), model.bias.item())
```

OLD WAY Output:

```
Epoch 0 Loss 26.1645
Epoch 0 Loss 51.6394
Epoch 0 Loss 4.4119
Epoch 0 Loss 37.8745
Epoch 0 Loss 4.0603
Epoch 0 Loss 13.2212
Epoch 0 Loss 5.2580
Epoch 0 Loss 38.4253
Epoch 0 Loss 62.2189
Epoch 0 Loss 45.0782
Epoch 100 Loss 0.5433
Epoch 100 Loss 0.2602
Epoch 100 Loss 0.9621
Epoch 100 Loss 2.4702
Epoch 100 Loss 0.8288
Epoch 100 Loss 1.2402
Epoch 100 Loss 1.0365
Epoch 100 Loss 1.0889
Epoch 100 Loss 0.0021
Epoch 100 Loss 0.7653
Final Parameters: 2.6696107387542725
4.879678249359131
```

PyTorch WAY Output:

```
Epoch 0 Loss 0.1757
Epoch 0 Loss 8.0143
Epoch 0 Loss 3.5346
Epoch 0 Loss 96.7008
Epoch 0 Loss 0.7633
Epoch 0 Loss 47.8596
Epoch 0 Loss 41.6441
Epoch 0 Loss 73.6039
Epoch 0 Loss 60.8005
Epoch 0 Loss 24.6684
Epoch 100 Loss 2.6241
Epoch 100 Loss 1.2587
Epoch 100 Loss 0.6389
Epoch 100 Loss 1.2624
Epoch 100 Loss 0.0821
Epoch 100 Loss 1.2768
Epoch 100 Loss 0.0434
Epoch 100 Loss 1.2102
Epoch 100 Loss 1.2231
Epoch 100 Loss 0.8412
Final Parameters: 2.6496422290802
4.87706995010376
```

Building a multilayer perceptron

- Defining the model from scratch, even for such a simple case, is neither appealing nor good practice
- PyTorch instead provides already **defined layers through torch.nn** that can be readily used as the building blocks of an NN model.

- **Example**: use torch.nn layers to solve a classification task using the Iris flower dataset (identifying between three species of irises) and build a two-layer perceptron using the torch.nn module

Building a multilayer perceptron

- Using the **nn.Module class**, we can **stack a few layers** and build an NN.
  - The list of all the layers that are already available at https://pytorch.org/docs/stable/nn.html
- Each layer in an NN receives its inputs from the preceding layer

- Example: we are going to use the **Linear layer**, which is also known as a **fully connected layer** or **dense layer**,
  - It can be best represented by $f(w \times x + b)$, where x represents a tensor containing the input features, w and b are the weight matrix and the bias vector, and f is the activation function.

Building a multilayer perceptron

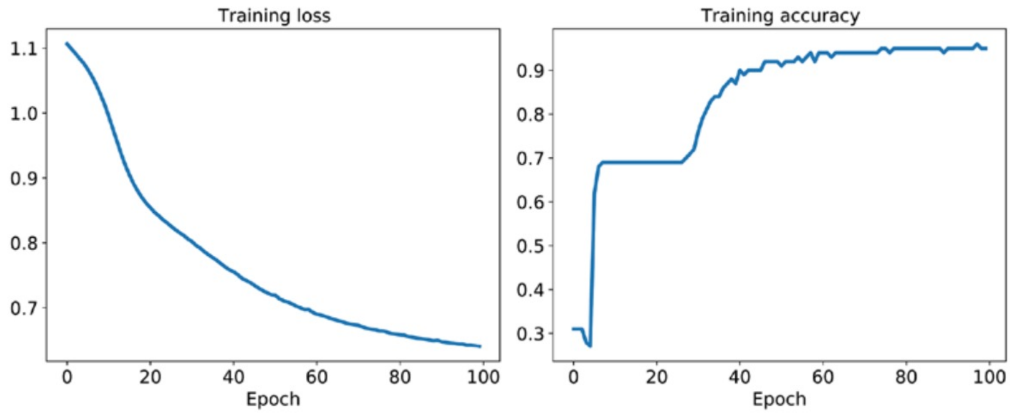
```
class Model(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.layer1 = nn.Linear(input_size, hidden_size)
        self.layer2 = nn.Linear(hidden_size, output_size)
    def forward(self, x):
        x = self.layer1(x)
        x = nn.Sigmoid()(x)
        x = self.layer2(x)
        x = nn.Softmax(dim=1)(x)
        return x

input_size = X_train_norm.shape[1]
hidden_size = 16
output_size = 3
model = Model(input_size, hidden_size, output_size)
learning_rate = 0.001
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

- We want to define a **model** with two hidden layers.
  - The first one receives an input of four features and projects them to 16 neurons.
  - The second layer receives the output of the previous layer (which has a size of *16*) and projects them to three output neurons, since we have three class labels.
- We used the **sigmoid** activation function for the first layer and **softmax** activation for the last (output) layer.
  - Softmax activation in the last layer is used to support multiclass classification since we have three class labels
- We specify the loss function as **cross-entropy loss** and the optimizer as **Adam.**
  - The Adam optimizer is a robust, gradient-based optimization method.

Building a multilayer perceptron

```
num_epochs = 100
loss_hist = [0] * num_epochs
accuracy_hist = [0] * num_epochs

for epoch in range(num_epochs):
    for x_batch, y_batch in train_dl:
        pred = model(x_batch)
        loss = loss_fn(pred, y_batch)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        loss_hist[epoch] += loss.item()*y_batch.size(0)
        is_correct = (torch.argmax(pred, dim=1) == y_batch).float()
        accuracy_hist[epoch] += is_correctmean()

    loss_hist[epoch] /= len(train_dl.dataset)
    accuracy_hist[epoch] /= len(train_dl.dataset)
```

- The loss_hist and accuracy_hist lists keep the training loss and the training accuracy after each epoch



Building a multilayer perceptron

- Applied the same standardization to the test data.
- Get the prediction accuracy.

```
X_test_norm = (X_test - np.mean(X_train)) / np.std(X_train)
X_test_norm_tensor = torch.from_numpy(X_test_norm).float()
y_test_tensor = torch.from_numpy(y_test)

pred_test = model(X_test_norm_tensor)
correct = (torch.argmax(pred_test, dim=1) == y_test_tensor).float()
accuracy = correct.mean()
print(f'Test Acc.: {accuracy:.4f}')
```

Output:
Test Acc1.: 0.9800

Saving and reloading the trained model

- Trained models can be saved on disk for future use
- **save(model)** will save both the model architecture and all the learned parameters.
- Reload the saved model
- Verify the model architecture by calling model_new.eval()
- Evaluate this new model that is reloaded on the test dataset

```
#save the model
path = 'iris_classifier.pt'
torch.save(model, path)

#reload the model
model_new1 = torch.load(path)
model_new1.eval()

#Evaluate this new model that is reloaded on the test dataset
#to verify that the results are the same as before:
pred_test1 = model_new1(X_test_norm_tensor)
correct1 = (torch.argmax(pred_test1, dim=1) ==
y_test_tensor).float()
accuracy1 = correct1.mean()
print(f'Test Acc2.: {accuracy1:.4f}')
```

Output:

Test Acc1.: 0.9800

Saving and reloading the trained model

- Save only the learned parameters, using **save(model.state_dict())**
- To reload the saved parameters, we first need to **construct the model** as we did before, then feed the loaded parameters to the model.
- Evaluate this new model that is reloaded on the test dataset

```
#save only the learned parameters
path_param = 'iris_classifier_state.pt'
torch.save(model.state_dict(), path_param)

#reload the model
model_new2 = Model(input_size, hidden_size, output_size)
model_new2.load_state_dict(torch.load(path_param))

#Evaluate this new model that is reloaded on the test dataset
#to verify that the results are the same as before:
pred_test2 = model_new2(X_test_norm_tensor)
correct2 = (torch.argmax(pred_test2, dim=1) ==
y_test_tensor).float()
accuracy2 = correct2.mean()
print(f'Test Acc2.: {accuracy2:.4f}')
```

Output:

Test Acc2.: 0.9800

Activation function in multilayer NNs

- We can use **any function** as an activation function in multilayer NNs as long as it is differentiable.
- Linear activation functions: such as in Adaline
- In practice, it would not be very useful to use linear activation functions for both hidden and output layers, since we want to **introduce nonlinearity** in a typical artificial NN to be able to tackle complex problems.
  - The sum of linear functions yields a linear function after all.

Non-linear activation functions

- **Sigmoid Logistic** (simplified as sigmoid): $\phi(z) = \frac{1}{1+e^{-z}}$

- *Interpretation*: the probability that a particular sample **x** belongs to the positive class.
- *Disadvantage*: it can be problematic if we **have highly negative** input
  - For such input, the output of the sigmoid function is close to **zero**.
  - For such output, the NN learns very slowly and it becomes more likely that it gets trapped in the local minima during training.
  - Thus, at hidden layers, the **hyperbolic tangent** function is more often used.

Sigmoid function

- For an input x, we calculate the net input (*z*): $z = \ w^T x$
- We use z to activate a logistic neuron with those particular feature values and weight coefficients, we get a value of 0.888.
- We can interpret as **an 88.8 percent probability** that this particular sample, *x*, belongs to the positive class.

Use sigmoid function in PyTorch

- The **torch.sigmoid**() function in PyTorch.
- Using torch.sigmoid(x) produces results that are equivalent **to torch. nn.Sigmoid()(x)**

```
z = np.arange(-5, 5, 0.005)

tensor_z = torch.from_numpy(z)
sigmoid1_tensor = torch.sigmoid(tensor_z)
sigmoid2_tensor = torch.nn.Sigmoid()(tensor_z)

print(sigmoid1_tensor)
print(sigmoid2_tensor)
```

Output:

```
tensor([0.0067, 0.0067, 0.0068, ..., 0.9932, 0.9932, 0.9933], dtype=torch.float64)
tensor([0.0067, 0.0067, 0.0068, ..., 0.9932, 0.9932, 0.9933], dtype=torch.float64)
```

Multiclass issue

- We can design the output layer consisting of multiple logistic activation units.
- An output layer consisting of multiple logistic activation units does **not produce meaningful, interpretable probability** values.
  - The resulting values cannot be interpreted as probabilities for a three-class problem. The reason for this is that **they do not sum to 1**.
- **One possible way to solve this**: predict the class label from the output units obtained earlier is to use the maximum value
  - y_class = np.argmax(Z, axis=0)
- Sometimes, we still need to compute **meaningful class probabilities for multiclass predictions**.

Softmax function

- The **softmax** function is a soft form of the argmax function.
  - Instead of giving a single class index, it provides the probability of each class.
- **softmax f**unction: compute meaningful class probabilities for multiclass predictions.
- The probability of a particular sample with net input *z* belonging to the *i*th class can be computed with a normalization term in the denominator.

$$p(y = i|z) = \phi(z) = \frac{e^{z_i}}{\sum_{j=1}^{M} e^{z_j}}$$

- The predicted class probabilities sum to 1.

Use softmax function in PyTorch

- When we build a multiclass classification model in PyTorch, we can use the **torch.softmax**() function to estimate the probabilities of each class membership for an input batch of examples
- We will **convert Z to a tensor** in the following code, with an additional dimension reserved for the batch size.

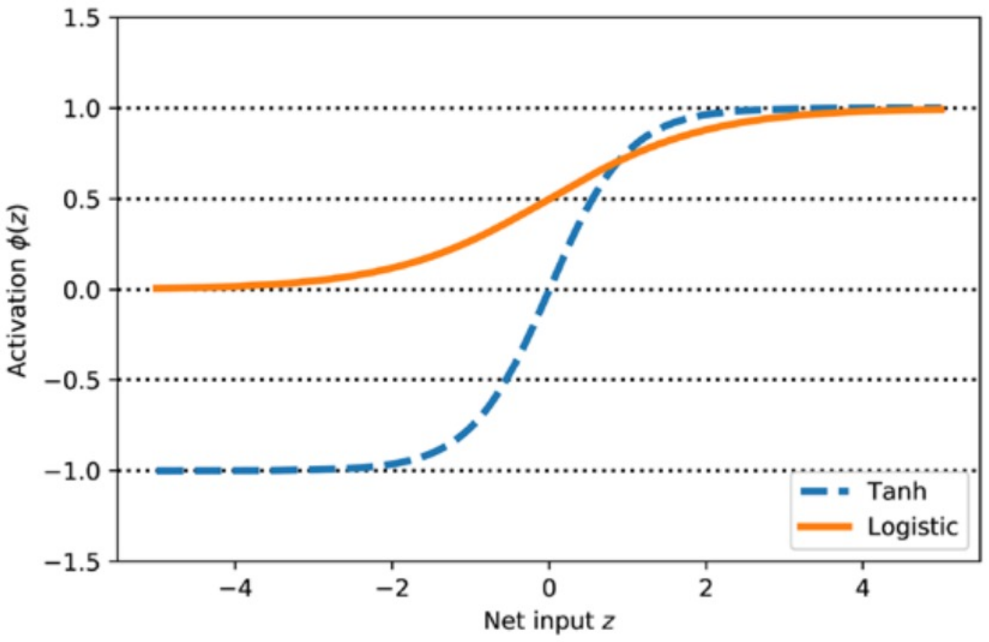
```
torch_y_probas = torch.softmax(torch.from_numpy(Z), dim=0)

print('Torch Probabilities:\n', torch_y_probas)
```

Hyperbolic Tangent (tanh)

- Another sigmoidal function that is often used in the **hidden** layers of artificial NNs is the **hyperbolic tangent** (commonly known as **tanh**), which can be interpreted as a rescaled version of the logistic function.
- $\phi_{logistic}(z) = \frac{1}{1+e^{-z}}$
- $\phi_{tanh}(z) = 2 * \phi_{logistic}(2z) - 1 = \frac{1-e^{-2z}}{1+e^{-2z}} = \frac{(1-e^{-2z})e^{z}}{(1+e^{-2z})e^{z}} = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$
- Hyperbolic Tangent (tanh): $\phi_{tanh}(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$
- *The* **advantage** of the **tanh** function over the sigmoid function is that (a) it has a broader output spectrum and ranges in the **open interval (-1,1)**, which can improve the convergence of the back propagation algorithm.

Tanh vs. sigmoid

- *The* **advantage** of the **tanh** function over the sigmoid function is that (a) it has a broader output spectrum and ranges in the **open interval (-1,1)**, which can improve the convergence of the back propagation algorithm.
- The **shapes** of the two sigmoidal curves look very **similar.**
- However, the tanh function has **double the output space** of the logistic function.



Use tanh function in PyTorch

- We can use NumPy's tanh function.
- When building an NN model, we can use **torch.tanh(x)** in PyTorch.

```
z = np.arange(-5, 5, 0.005)

tanh1 = np.tanh(z)
tanh1_tensor = torch.tanh(torch.from_numpy(z))

print(tanh1)
print(tanh1_tensor)
```

Output:

```
[-0.9999092 -0.99990829 -0.99990737 ... 0.99990644 0.99990737 0.99990829]
tensor([-0.9999, -0.9999, -0.9999, ..., 0.9999, 0.9999, 0.9999], dtype=torch.float64)
```

Vanishing gradient issue

- **Both the tanh and the sigmoid functions** have the problem **of vanishing gradients**, which means that the derivative of activations with respect to net input diminishes as z becomes large.
  - For example, for two net inputs $z_1 = 20$ and $z_2 = 25$, both *tanh*(z1) and *tanh*(z2) are close to 1.0. They show no change in the output. Thus, learning weights during the training phase becomes very slow because the gradient terms may be very close to zero.
- ReLU activation addresses this issue

ReLU (Rectified Linear Unit)

- ReLU (Rectified Linear Unit) $\phi(z) = \max(0, z) = \begin{cases} 0 & z < 0 \\ z & z > 0 \end{cases}$
- ReLU is still a nonlinear function.
- The derivative of ReLU for positive input values is always 1. Thus, it can solve the problem of vanishing gradients, making it suitable for DNNs.
- In DNNs, the ReLU activation function is more often used.

Use ReLU in PyTorch

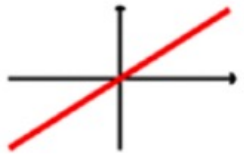
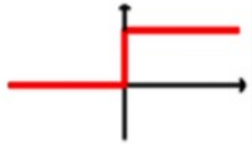
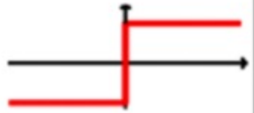
- We can apply the ReLU activation torch.relu() as follows.

```
z = np.arange(-5, 5, 0.005)
tensor_z = torch.from_numpy(z)
relu_tensor= torch.relu(tensor_z)
print(relu_tensor)
```

Output:
```
tensor([0.0000, 0.0000, 0.0000, ..., 4.9850, 4.9900, 4.9950], dtype=torch.float64)
```

Activation functions

- You can find the **list of all activation functions** available in the torch.nn module at https://pytorch.org/docs/stable/nn.functional.html#non-linear-activation-functions.

Activation function	Equation	Example	1D graph
Linear	$\sigma(z) = z$	Adaline, linear regression	
Unit step (Heaviside function)	$\sigma(z)=\begin{cases} 0 & z<0 \\ 0.5 & z=0 \\ 1 & z>0 \end{cases}$	Perceptron variant	
Sign (signum)	$\sigma(z)=\begin{cases} -1 & z<0 \\ 0 & z=0 \\ 1 & z>0 \end{cases}$	Perceptron variant	
Piece-wise linear	$\sigma(z)=\begin{cases} 0 & z\le -\frac{1}{2} \\ z+\frac{1}{2} & -\frac{1}{2}\le z\le \frac{1}{2} \\ 1 & z\ge \frac{1}{2} \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\sigma(z)=\dfrac{1}{1+e^{-z}}$	Logistic regression, multilayer NN	
Hyperbolic tangent (tanh)	$\sigma(z)=\dfrac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$	Multilayer NN, RNNs	
ReLU	$\sigma(z)=\begin{cases} 0 & z<0 \\ z & z>0 \end{cases}$	Multilayer NN, CNNs	

References

- Chapter 12: By Sebastian Raschka , Yuxi (Hayden) Liu , Vahid Mirjalili: Machine Learning with PyTorch and Scikit-Learn, Packt.