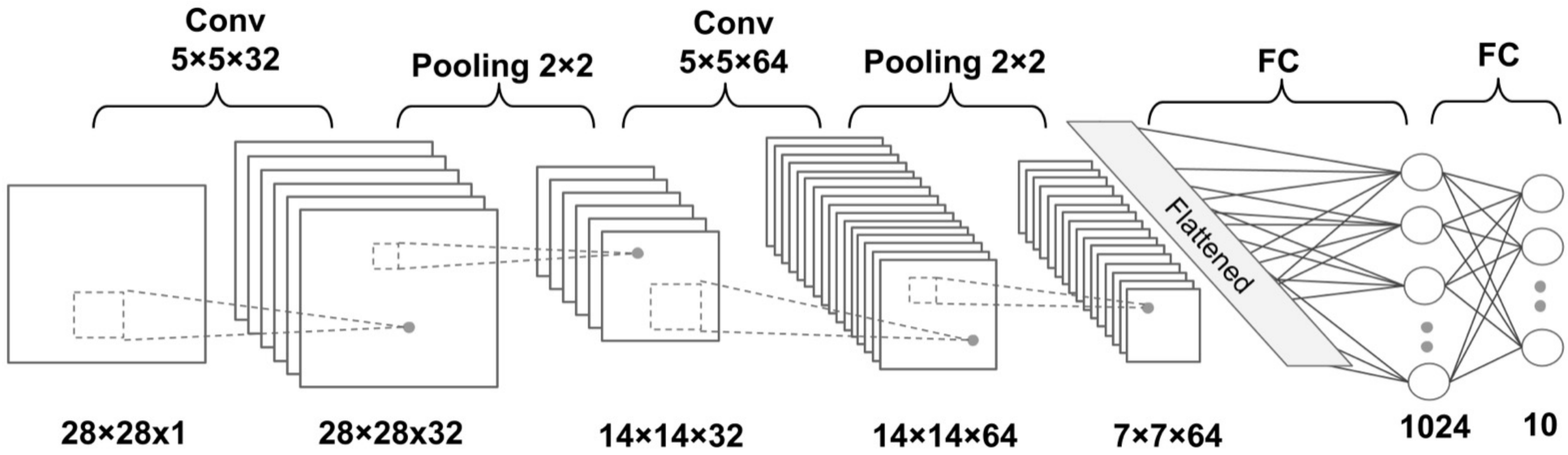
CNN implementation

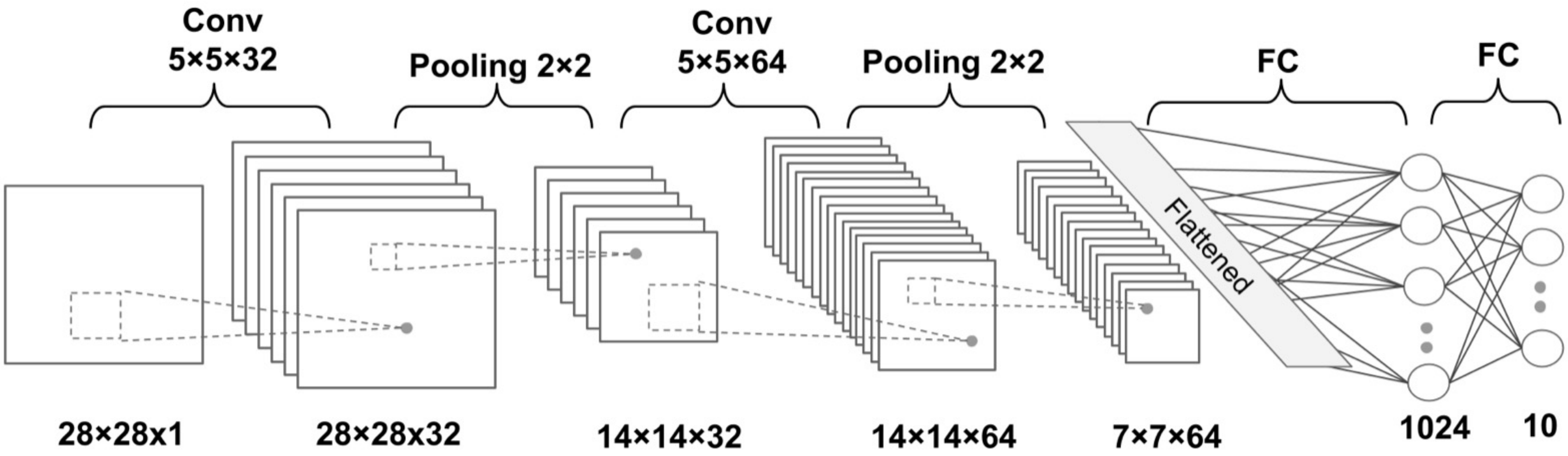
Dr. Huiping Cao

Multilayer CNN architecture



- Input: [*batchsize* × 28 × 28 × 1] Each image is 28 × 28 grayscale images. Thus, the number of input channels C_{in} is 1.
- Conv_1: [*batchsize* × 28 × 28 × 32]. Here, 32 is the number of output channels C_{out}. Kernel size is 5×5. padding = same.
- Pooling_1: [*batchsize* × 14 × 14 × 32]. Pooling size is 2 × 2. Thus, it reduces 28 to 14.
- **Batchsize**: The number of training samples in each mini-batch when splitting of the training data in each epoch for stochastic gradient descent. The gradient is computed for each mini-batch separately instead of the entire training data.

Multilayer CNN architecture



- Conv_2: [*batchsize*×14×14×64] Kernel size is 5×5. padding=same. Here, 64 is the number of output channels C_{out}.
- Pooling_2: [*batchsize* × 7 × 7 × 64] Pooling size is 2 × 2. Thus, it reduces 14 to 7.
- FC_1: [*batchsize* × 1024].
  FC_2 and softmax layer: [*batchsize* × 10].

Load and preprocess the data

- The **MNIST** dataset comes with a pre-specified training and test dataset partitioning.
- We want to create a validation split from the train partition.

Load and preprocess the data

- Three steps to load a dataset
  - Download the data
  - Get training, validation, and testing data

```
## Step 1: loading and preprocessing MNIST dataset
image_path = '/content/drive/MyDrive/ColabNotebooks/data/'
transform = transforms.Compose([transforms.ToTensor()])

mnist_dataset = torchvision.datasets.MNIST(root=image_path, train=True,transform=transform, download=False)

mnist_valid_dataset = Subset(mnist_dataset,torch.arange(10000))
mnist_train_dataset = Subset(mnist_dataset,torch.arange(10000, len(mnist_dataset)))
mnist_test_dataset = torchvision.datasets.MNIST(root=image_path, train=False,transform=transform, download=False)

print('number of items in mnist_dataset:', len(mnist_dataset))
print('number of items in mnist_train_dataset:', len(mnist_train_dataset))
print('number of items in mnist_valid_dataset:', len(mnist_valid_dataset))
print('number of items in mnist_test_dataset:', len(mnist_test_dataset))
```

```
number of items in mnist_dataset: 60000
number of items in mnist_train_dataset: 50000
number of items in mnist_valid_dataset: 10000
number of items in mnist_test_dataset: 10000
```

Construct data loader

- Construct the data loader with batches of 64 images for the training set and validation set, respectively

```
batch_size = 64
torch.manual_seed(1)

train_dl = DataLoader(mnist_train_dataset, batch_size, shuffle=True)
valid_dl = DataLoader(mnist_valid_dataset, batch_size, shuffle=False)
```

Implement a CNN using PyTorch

- We use the **torch.nn Sequential** class to stack different layers, such as convolution, pooling, and dropout, as well as the fully connected layers.
- The **torch.nn module provides classes** for each one
  - **nn.Conv2d** for a two-dimensional convolution layer
  - **nn.MaxPool2d** and **nn.AvgPool2d** for subsampling (max-pooling and average-pooling)
  - **nn.Dropout** for regularization using dropout.

Configuring CNN layers in PyTorch

- Input: when we read an image, the **default dimension for the channels** is the first dimension of the tensor array
  - **NCHW** format, where *N* stands for the number of images within the batch, *C* stands for channels, and *H* and *W* stand for height and width, respectively.
- For **Conv2d** class,
  - **Input** is in NCHW format
  - After the layer is constructed, it can be called by providing **a four-dimensional tensor**, with the first dimension reserved for a batch of examples; the second dimension corresponds to the channel; and the other two dimensions are the spatial dimensions.

Configuring CNN layers in PyTorch

- Need to specify different **parameters**
  - The number of output channels
  - Kernel size
    - The **kernel_size** argument determines the size of the window (or neighborhood) that will be used to compute the max or mean operations.
  - Stride
  - Padding
- **Dropout class** will construct the dropout layer for regularization, with the argument p that denotes the drop probability p_{drop}.
  - When calling this layer, its behavior can be controlled via model.train() and model.eval(), to specify whether this call will be made during training or during the inference.

Constructing a CNN in PyTorch

```
model = nn.Sequential()

model.add_module('conv1',\
            nn.Conv2d(in_channels=1,
            out_channels=32,kernel_size=5, padding=2))
model.add_module('relu1', nn.ReLU())

model.add_module('pool1', nn.MaxPool2d(kernel_size=2))

model.add_module('conv2',\
            nn.Conv2d(in_channels=32,
            out_channels=64,kernel_size=5, padding=2))
model.add_module('relu2', nn.ReLU())

model.add_module('pool2', nn.MaxPool2d(kernel_size=2))
```

- Add two convolution layers to the model.
- For each convolutional layer, we used a kernel of size 5×5 and padding=2.
  - Padding =2 is to get the same padding mode.

$$o = \left\lfloor \frac{n + 2p - m}{s} \right\rfloor + 1$$

- The max-pooling layers with pooling size 2×2 and stride of 2 will reduce the spatial dimensions by half.
  - Stride – the stride of the window. Default value is kernel_size.

```
TensorShape([16, 7, 7, 64])
```

Constructing a CNN in PyTorch

```
model = nn.Sequential()

model.add_module('conv1',\
        nn.Conv2d(in_channels=1,
        out_channels=32,kernel_size=5, padding=2))
model.add_module('relu1', nn.ReLU())

model.add_module('pool1', nn.MaxPool2d(kernel_size=2))

model.add_module('conv2',\
        nn.Conv2d(in_channels=32,
        out_channels=64,kernel_size=5, padding=2))
model.add_module('relu2', nn.ReLU())

model.add_module('pool2', nn.MaxPool2d(kernel_size=2))

x = torch.ones((4, 1, 28, 28))
model(x).shape
```

```
torch.Size([4, 64, 7, 7])
```

- PyTorch provides a convenient method to compute the size of the feature maps at this stage.
  - Providing the input shape as a tuple (4, 1, 28, 28) (4 images within the batch, 1 channel, and image size 28×28)
  - Output: a shape (4, 64, 7, 7), indicating feature maps with 64 channels and a spatial size of 7×7.

Constructing a CNN in PyTorch

```
model.add_module('flatten', nn.Flatten())

x = torch.ones((4, 1, 28, 28))
model(x).shape
```

```
torch.Size([4, 3136])
```

```
model.add_module('fc1', nn.Linear(3136, 1024))
model.add_module('relu3', nn.ReLU())
model.add_module('dropout', nn.Dropout(p=0.5))
model.add_module('fc2', nn.Linear(1024, 10))
```

- For a fully connected layer
  - The input to this layer must have rank 2, that is, shape [*batch-size* × *input_units*].
  - Flatten the output of the previous layers to meet this requirement for the fully connected layer.
- Add two fully connected layers with a dropout layer in between.
  - **The last fully connected layer**, named 'fc2', has 10 output units for the 10 class labels in the MNIST dataset
- No need to add a softmax activation function

Constructing a CNN in PyTorch

```
loss_fn = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

- Create the loss function
  - **softmax** function is already used internally inside PyTorch's **CrossEntropyLoss** implementation
- Create an optimizer for the model
  - The **Adam optimizer** is a robust, gradient-based optimization method suited to nonconvex optimization and machine learning problems.

Constructing a CNN in PyTorch – define training

```
def train(model, num_epochs, train_dl, valid_dl):
    loss_hist_train = [0] * num_epochs
    accuracy_hist_train = [0] * num_epochs
        loss_hist_valid = [0] * num_epochs
        accuracy_hist_valid = [0] * num_epochs

        for epoch in range(num_epochs):
            model.train()
            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()
                #calculate loss, accuracy
```

```
…
        #for each epoch
        model.eval()
        with torch.no_grad():
            for x_batch, y_batch in valid_dl:
                pred = model(x_batch)
                loss = loss_fn(pred, y_batch)
                #calculate loss, accuracy
```

- Using the designated settings for training **model.train()** and evaluation **model.eval()** will automatically set the mode for the dropout layer and rescale the hidden units appropriately so that we do not have to worry about that at all

Train the CNN model

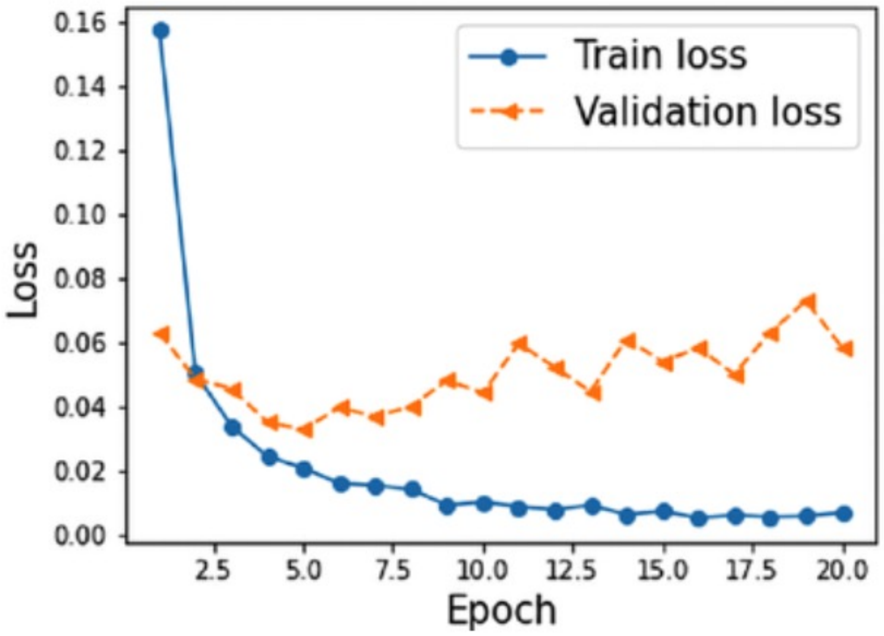
```
torch.manual_seed(1)
num_epochs = 20

hist = train(model, num_epochs, train_dl, valid_dl)
```

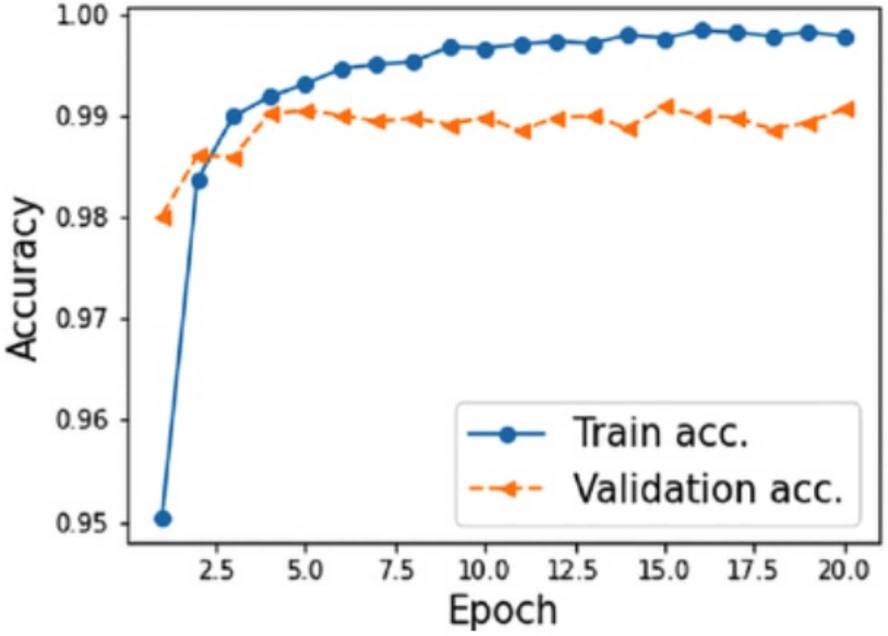
```
Epoch 1 accuracy: 0.9484 val_accuracy: 0.9799
Epoch 2 accuracy: 0.9836 val_accuracy: 0.9872
Epoch 3 accuracy: 0.9895 val_accuracy: 0.9862
Epoch 4 accuracy: 0.9916 val_accuracy: 0.9889
Epoch 5 accuracy: 0.9930 val_accuracy: 0.9879
Epoch 6 accuracy: 0.9945 val_accuracy: 0.9904
Epoch 7 accuracy: 0.9946 val_accuracy: 0.9886
Epoch 8 accuracy: 0.9964 val_accuracy: 0.9878
Epoch 9 accuracy: 0.9961 val_accuracy: 0.9902
…
Epoch 19 accuracy: 0.9981 val_accuracy: 0.9925
Epoch 20 accuracy: 0.9982 val_accuracy: 0.9906
```

- Train this CNN model and use the validation dataset that we created for monitoring the learning progress.

Plot the accuracy and loss

- Once the 20 epochs of training are finished, we can visualize the learning curves.





Evaluate the model using testing data

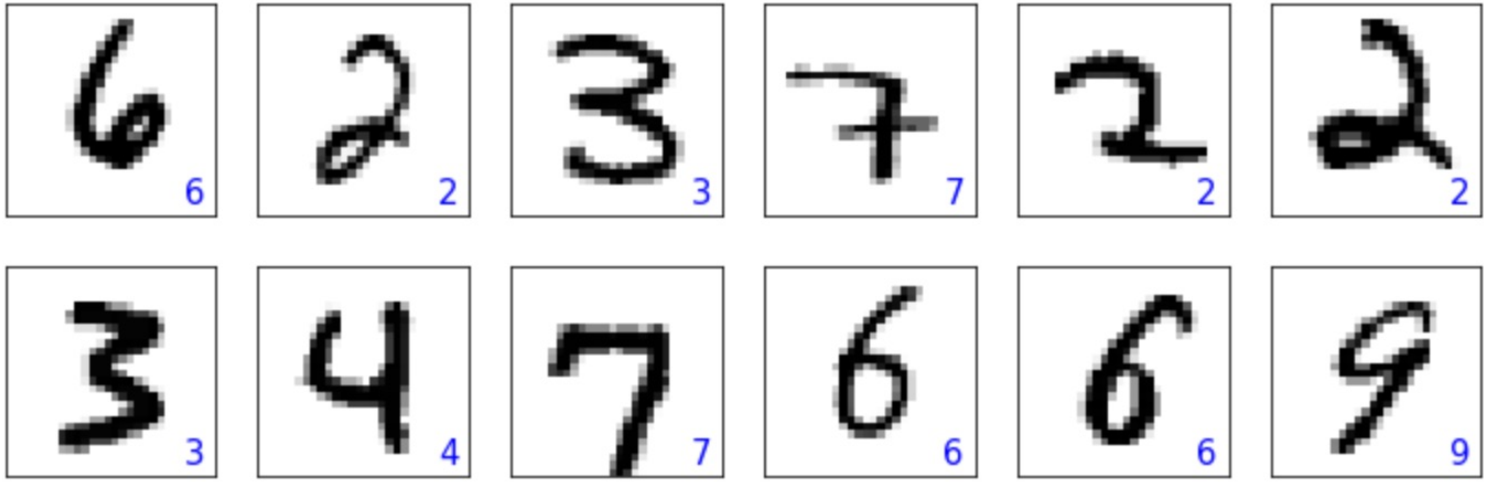
- Evaluate the trained model on the test dataset

```
pred = model(mnist_test_dataset.data.unsqueeze(1) / 255.)

is_correct = (torch.argmax(pred, dim=1) == mnist_test_dataset.targets).float()

print(f'Test accuracy: {is_correct.mean():.4f}')
```

```
Test accuracy: 0.9929
```

Predict the results

- Print the labels of 12 images (code see the textbook)



Adam Optimizer

- The adam optimizer is a robust, gradient-based optimization method suited to nonconvex optimization and machine learning problems.

- Details see the manuscript: *Adam: A Method for Stochastic Optimization*, Diederik P. Kingma and Jimmy Ma, 2014. https://arxiv.org/abs/1412.6980

One-hot encoding for categorical values

- Integer encoding: implies the order or categorical values
  - 0: ’Iris-setosa’
  - 1: ’Iris-versicolor’
  - 2: ’Iris-virginica’
- One-hot encoding: does not impose any order of categorical values
  - [1 0 0] - ’Iris-setosa’
  - [0 1 0] - ’Iris-versicolor’
  - [0 0 1] - ’Iris-virginica’
- Function: **tf.one_hot** (https://www.tensorflow.org/api_docs/python/tf/one_hot)

```
indices = [0, 1, 2]
depth = 3
tf.one_hot(indices, depth)
```

```
<tf.Tensor: shape=(3, 3), dtype=float32, numpy= array([[1., 0., 0.],
[0., 1., 0.], [0., 0., 1.]], dtype=float32)>
```

Loss function

- Activation functions: sigmoid, tanh, softmax, ReLU
  - Output layer: sigmoid (binary classification), softmax (multiclass classification)
  - When sigmoid/softmax is not utilized, the model will compute the **logits** (instead of the probability)
- **Loss function** purpose: minimize the error (difference between actual and predicted value) which is calculated by the loss function.
  - **Binary cross-entropy** is the loss function for a binary classification (with a single output unit)
  - **Categorical cross-entropy** is the loss function for multiclass classification.
  - **Mean Squared Error, L2 Loss** is the loss function for regression tasks. This loss is calculated by taking the mean of squared differences between actual(target) and predicted values.

Loss function – cross entropy

- **Entropy** is the number of bits required to transmit a randomly selected event from a probability distribution. A skewed distribution has a low entropy, whereas a distribution where events have equal probability has a larger entropy.
- **Cross-entropy** builds upon the idea of entropy from information theory and calculates the number of bits required to represent or transmit an average event from one distribution compared to another distribution.
- More details can be found from https://machinelearningmastery.com/cross-entropy-for-machine-learning/

Loss function

Loss function	Usage	Examples: Using probabilities `from_logits=False`	Examples: Using logits `from_logits=True`
`BinaryCrossentropy`	Binary classification	y_true: 1 y_pred: 0.69	y_true: 1 y_pred: 0.8
`CategoricalCrossentropy`	Multiclass classification	y_true: 0, 0, 1 y_pred: 0.30, 0.15, 0.55	y_true: 0, 0, 1 y_pred: 1.5, 0.8, 2.1
`Sparse CategoricalCrossentropy`	Multiclass classification	y_true: 2 y_pred: 0.30, 0.15, 0.55	y_true: 2 y_pred: 1.5, 0.8, 2.1

Another example

- See textbook/github repository

References

- Chapter 14: By Sebastian Raschka , Yuxi (Hayden) Liu , Vahid Mirjalili: Machine Learning with PyTorch and Scikit-Learn, Packt.