AdTrans project, 2023-1-PL01-KA220-HED-000158917

ADTRANS
ADVANCED DATA ANALYSIS FOR TRANSPORTATION AND MOBILITY ACADEMIC TEACHERS

Session 2

Optimization & Training of Neural Nets

Paraskevi Fasouli

Co-funded by the European Union

Overview

Types of Learning

Training Pipeline

Optimizer

Hyperparameters

Types of Learning

Supervised (inductive) learning	Training data includes desired outputs
Unsupervised learning	Training data does not include desired outputs
Semi-supervised learning	Training data includes a few desired outputs
Reinforcement learning	Rewards from sequence of actions

Supervised Learning

Input Data is **labeled**

Training set consists of {x, y} input pairs

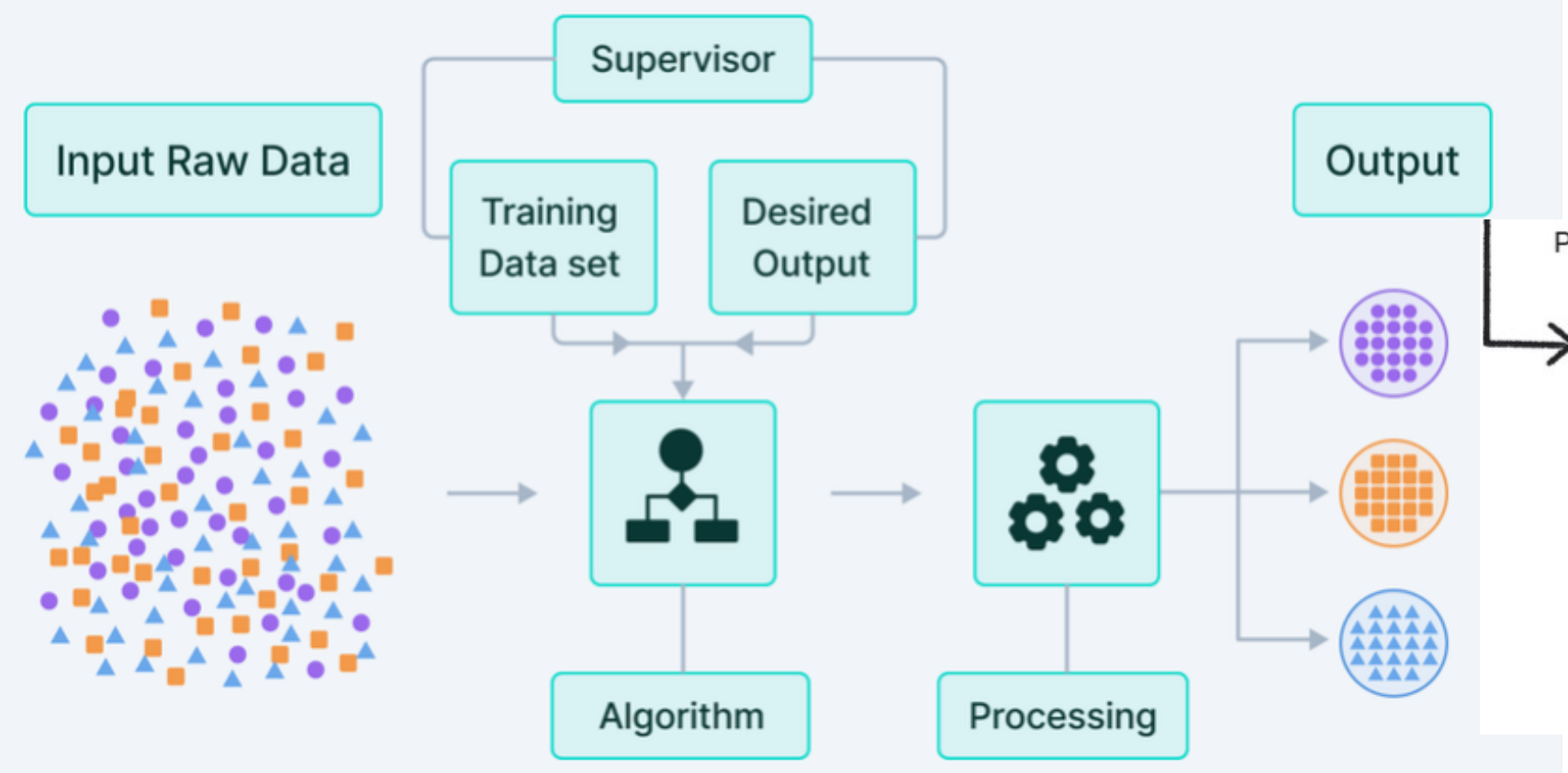
Known number of classes

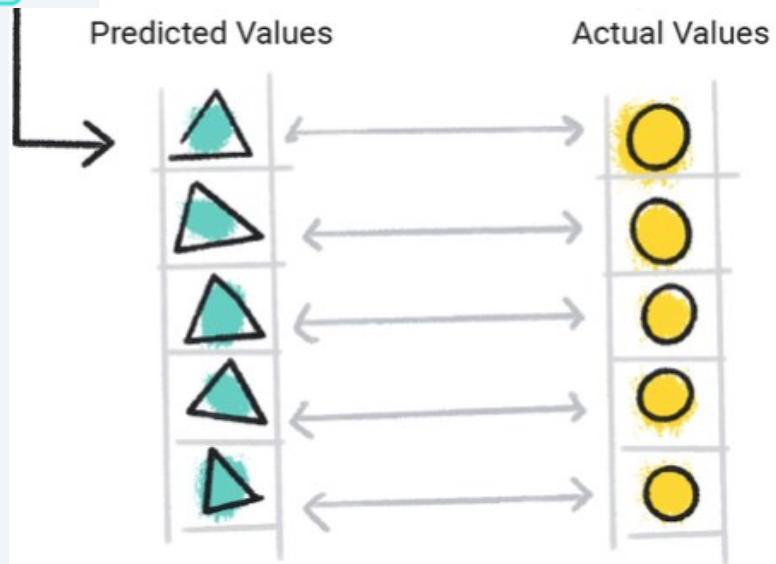
Has a feedback mechanism

Data is classified/predicted based on the training dataset

Used for data prediction

Supervised Learning





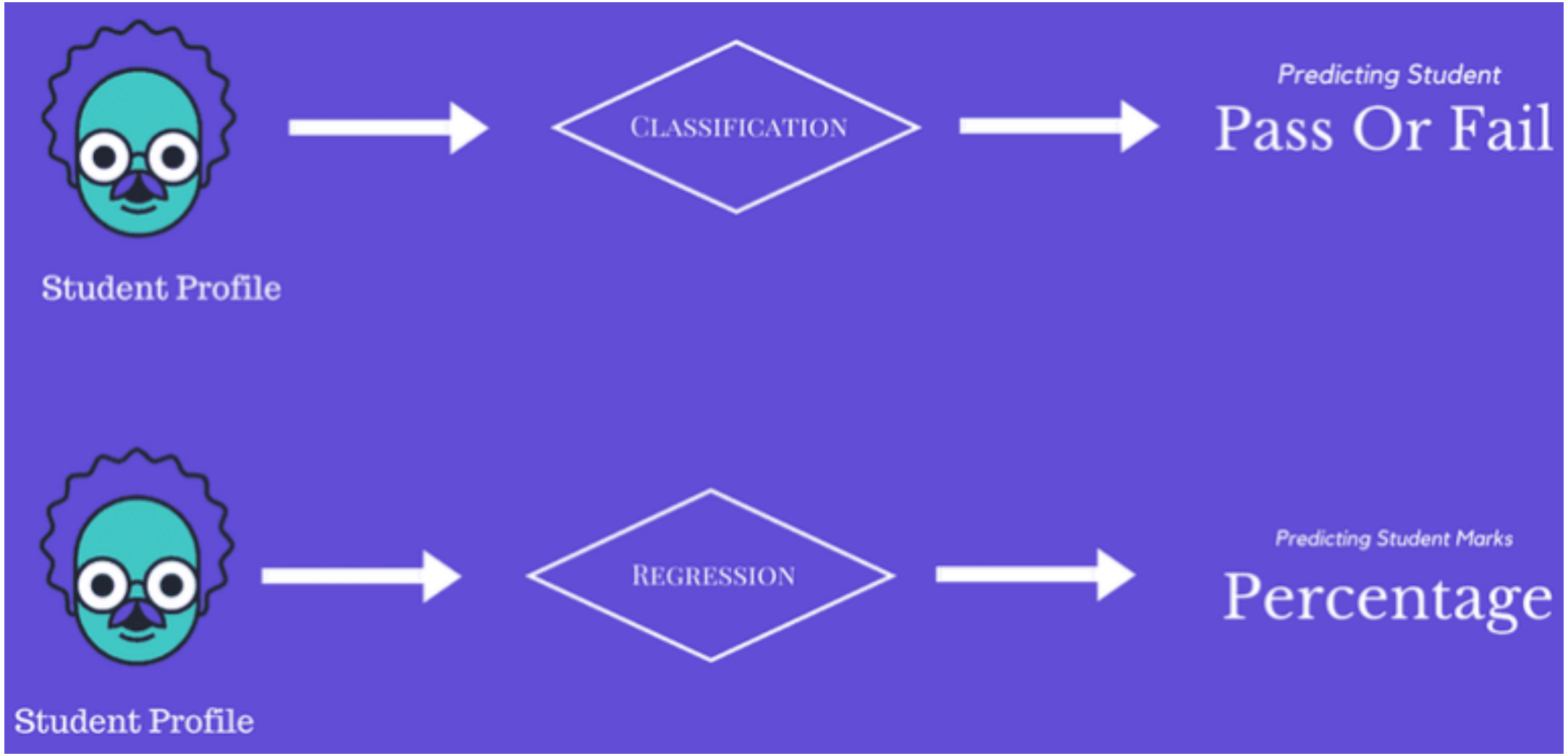
Supervised Learning

Forms of Learning

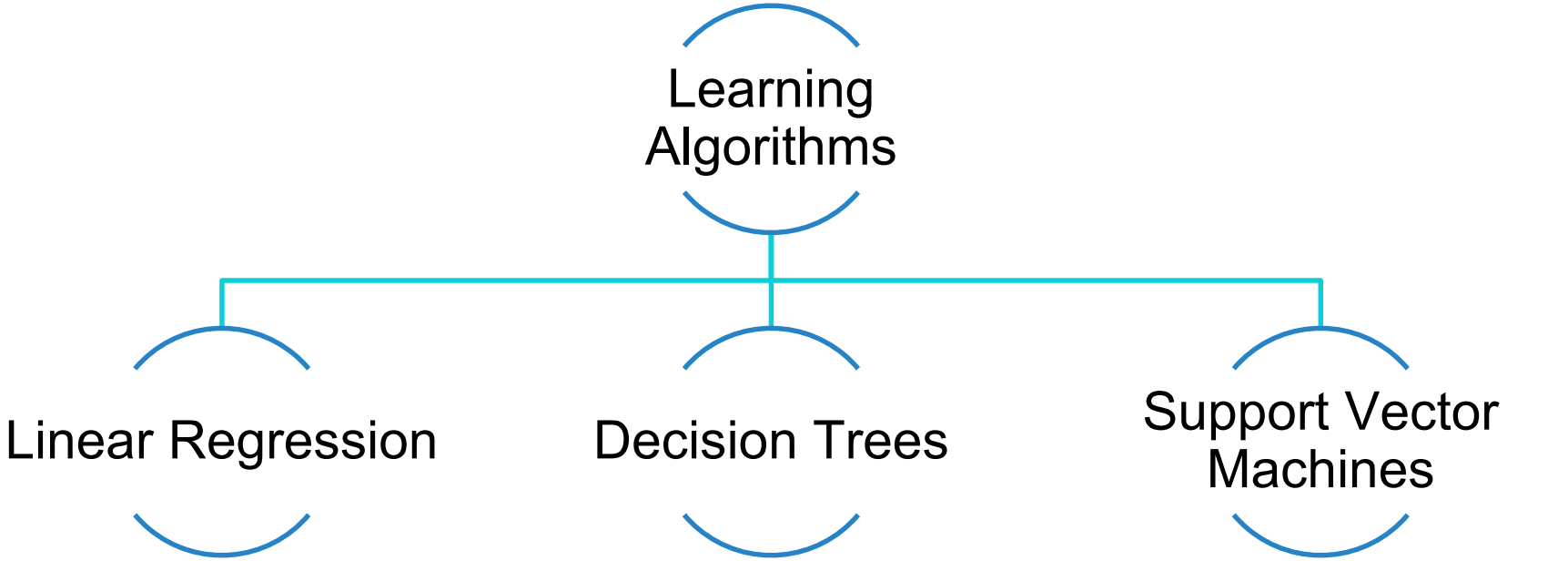
Regression – the training items are mapped to continuous values

Classification – the training items are mapped to distinct categories

Supervised Learning



Supervised Learning



- Sales Forecasting
- Risk Assessment

- Predictive Analytics
- Pricing

- Image classification
- Financial performance comparison

Unsupervised Learning

Input Data is **unlabeled**

Has no feedback mechanism

Unknown number of classes

Assigns Properties of given data to classify it

Used for data analysis

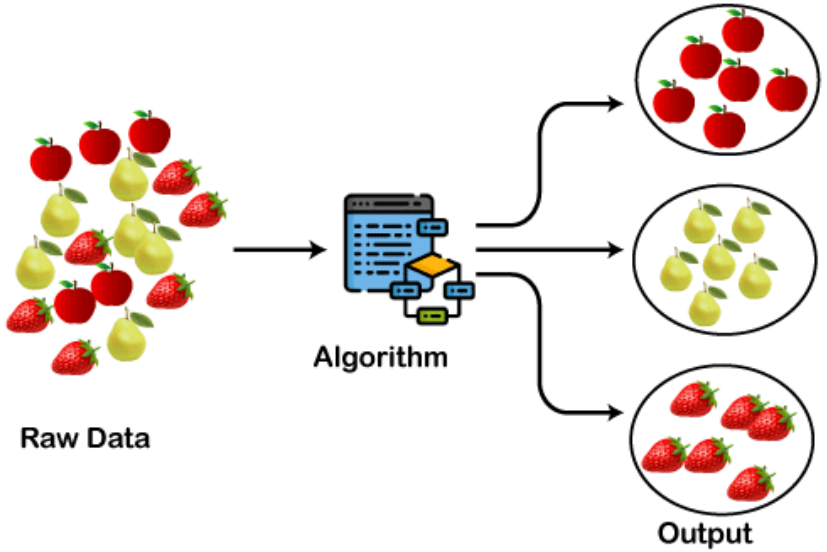
Solves Clustering and Association problems

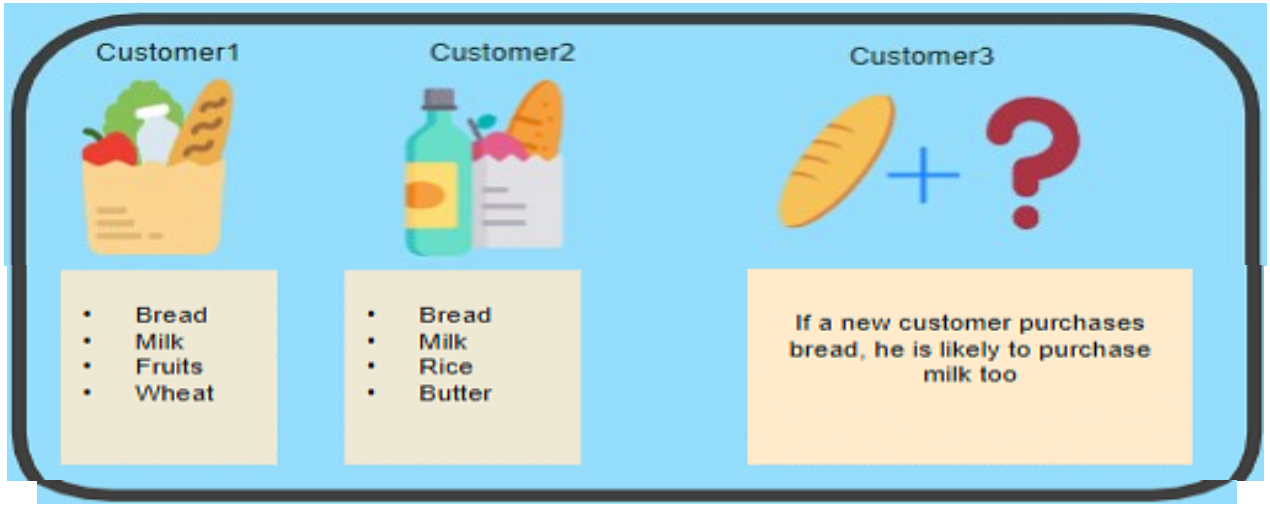
Unsupervised Learning

Forms of Learning

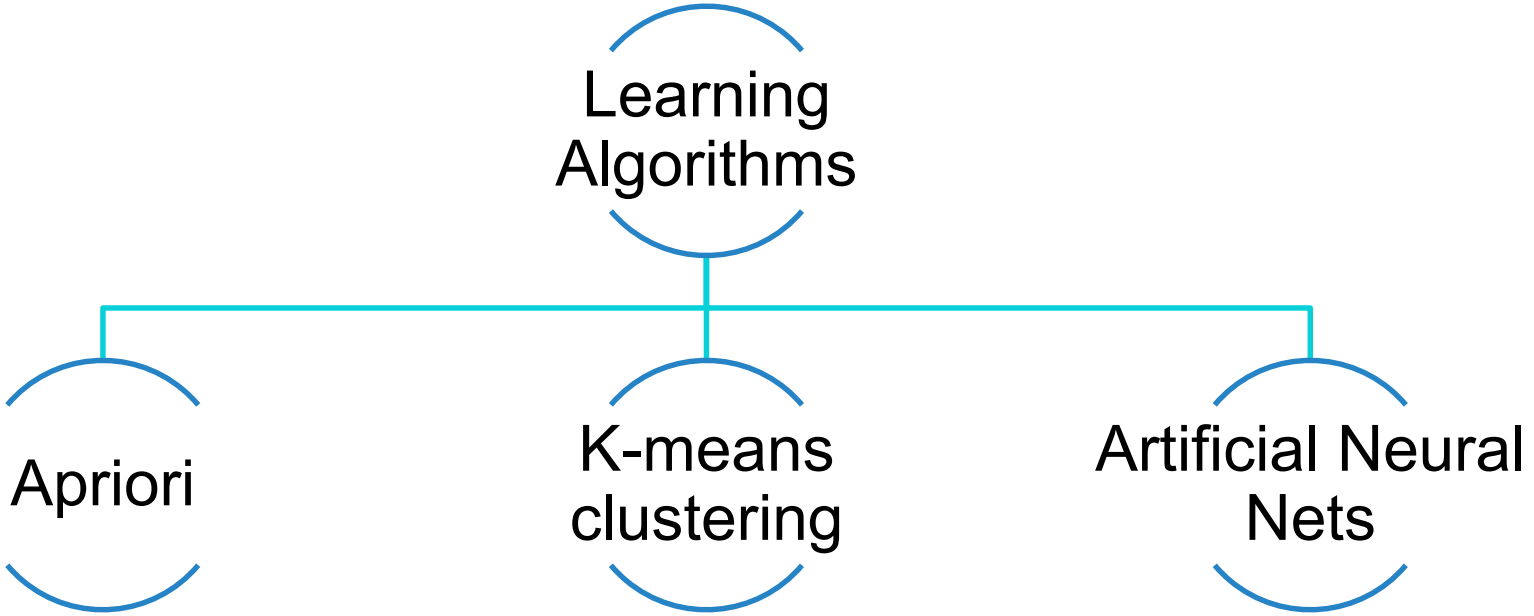
Clustering – the training items are grouped according to their observed similarities

Association – discovery of the rules that determine how or why certain items are connected





Unsupervised Learning



- Sales Functions
- Word Associations

- Customer Segmentation
- Insurance Fraud Detection

- Data Generation
- Data Mining
- Pattern Recognition

Semi-Supervised Learning

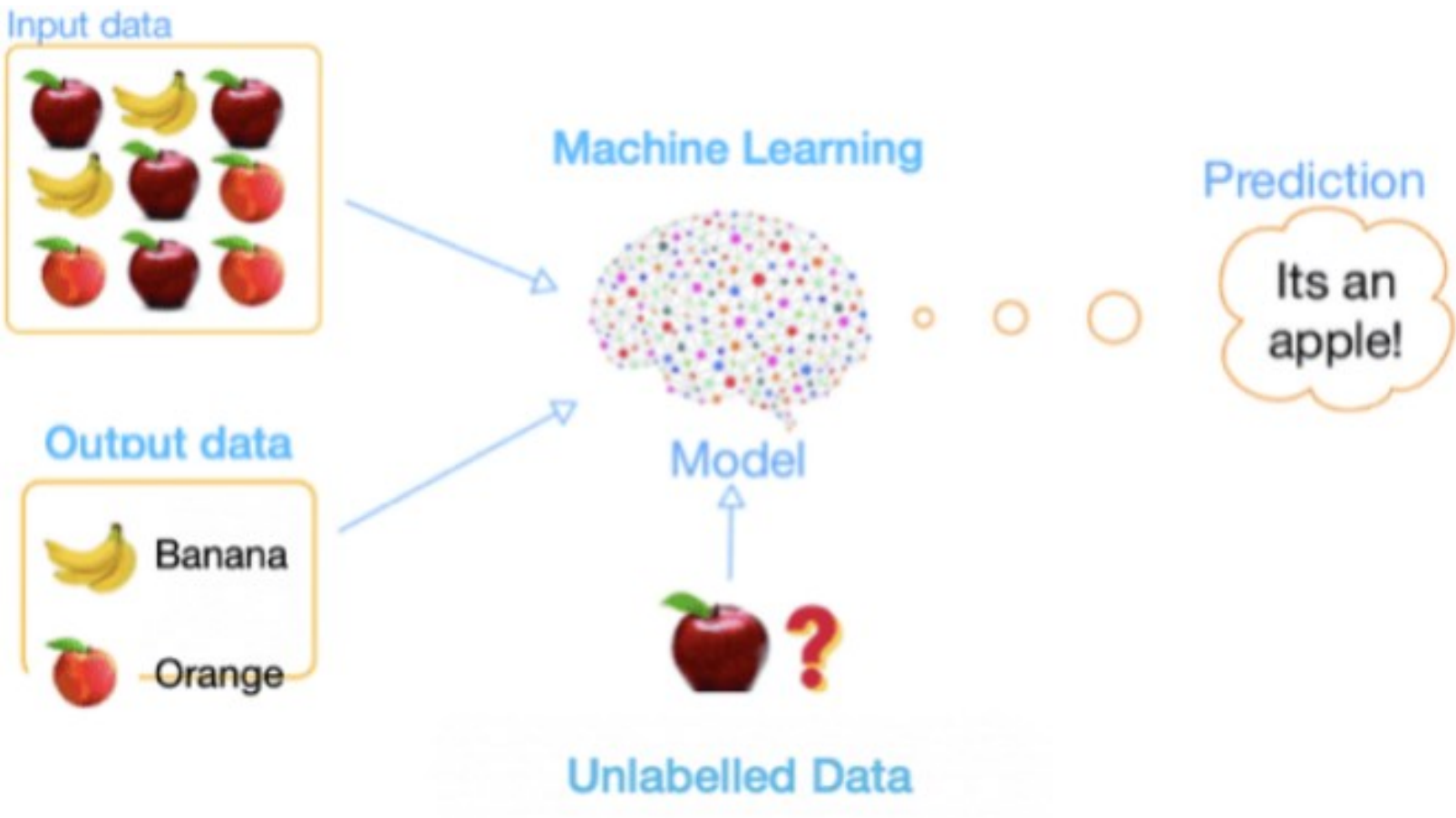
Input Data is **combination** of labeled & unlabeled

Hybrid technique

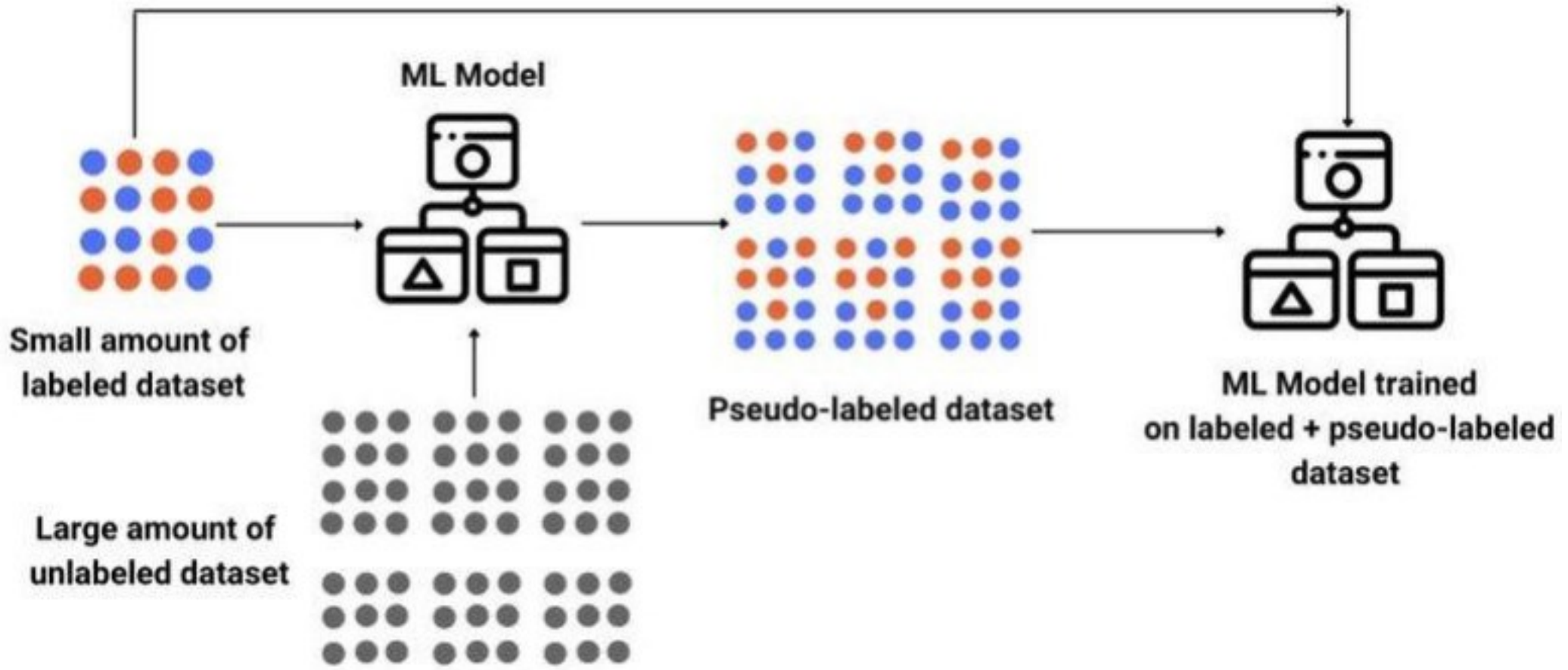
Prefect for large datasets

- For labeled points: the algorithm will use traditional supervision to update the model weights
- For unlabeled points: the algorithm minimizes the difference in predictions between other similar training examples

Semi-Supervised Learning

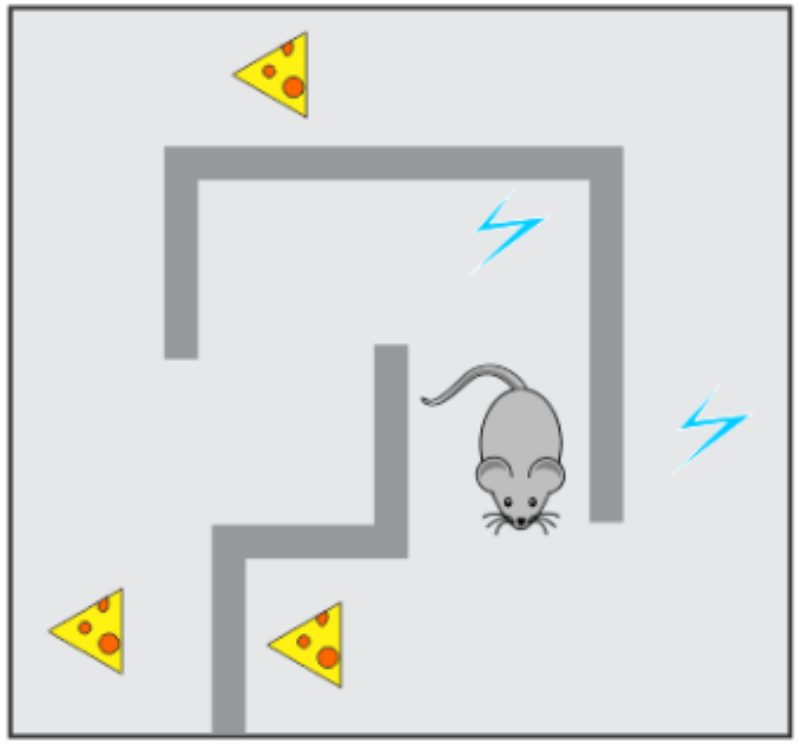


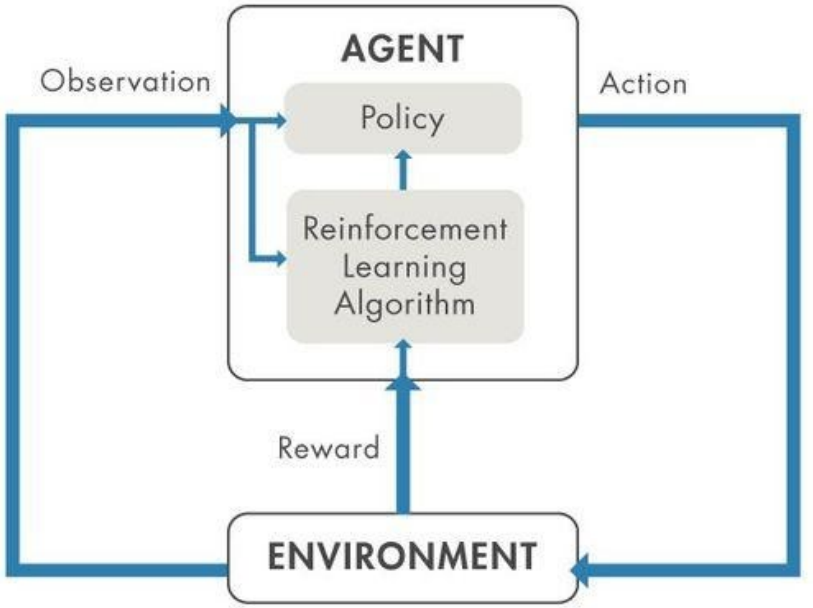
Semi-Supervised Learning



Reinforcement Learning

- Mouse => **Agent**
- Maze => **Environment**
- Mouse can move left, right, up & down => **Actions**
- Mouse wants the cheese without having electric shocks => **Rewards**
- Mouse can observe the environment => **Observations**





Train Your first NN

The training pipeline:

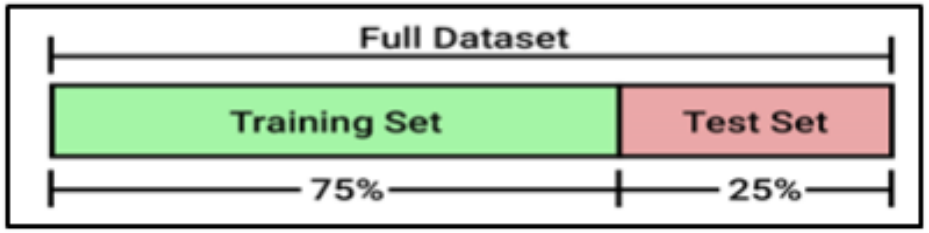
1. Define a problem (e.g. classification)
2. Define learning strategy (e.g. supervised learning)
3. Collect your dataset
4. Decide your model
5. Configure (Hyperparameters)
6. Train your model
7. Evaluate the performance
8. Save your model and deploy it

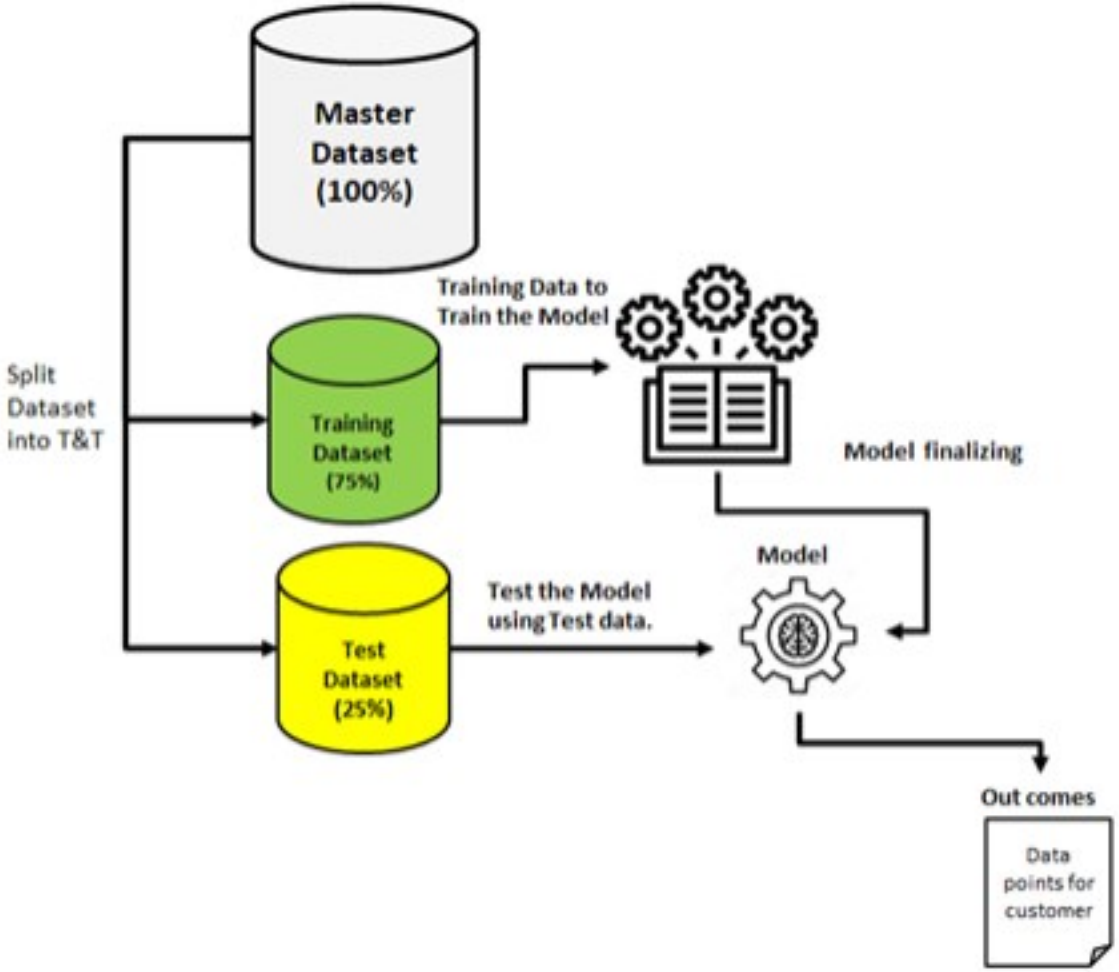
Done!

The Dataset

Preparation of dataset:

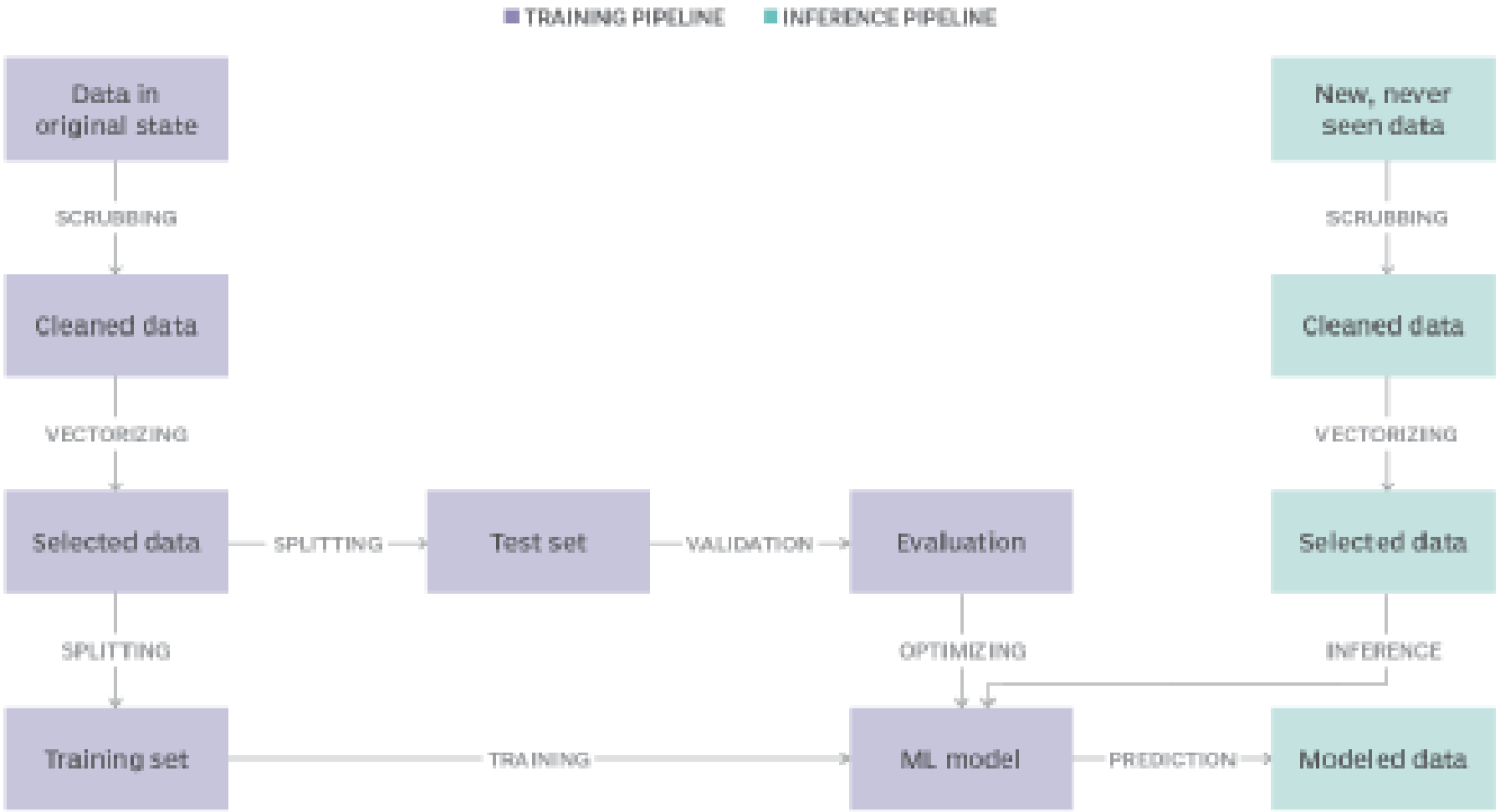
1. Data Pre-processing
2. Data Augmentation
3. Data Train-test split





Data pipelines for machine learning

Training pipelines and inference pipelines are both needed in order to continually train machine learning models.



The Dataset

In PyTorch, the **Dataset** and **DataLoader** classes encapsulate the process of pulling your data from storage and exposing it to your training loop in batches.

The **Dataset** is responsible for accessing and processing single instances of data.

The **DataLoader** pulls instances of data from the Dataset, collects them in batches, and returns them ready for your training loop.

We use **torchvision.transforms.Normalize()** to zero-center and normalize the distribution of content.

Then we download both training and validation data splits.

The Dataset

We will use the Fashion-MNIST dataset. It consists of images and annotation labels.
We will make supervised learning.

```
transform = transforms.Compose(
    [transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))])

# Create datasets for training & validation, download if necessary
training_set = torchvision.datasets.FashionMNIST('./data', train=True,
transform=transform, download=True)
validation_set = torchvision.datasets.FashionMNIST('./data', train=False,
transform=transform, download=True)

# Create data loaders for our datasets; shuffle for training, not for validation
training_loader = torch.utils.data.DataLoader(training_set, batch_size=4,
shuffle=True)
validation_loader = torch.utils.data.DataLoader(validation_set, batch_size=4,
shuffle=False)

# Class labels
classes = ('T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
        'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle Boot')
```

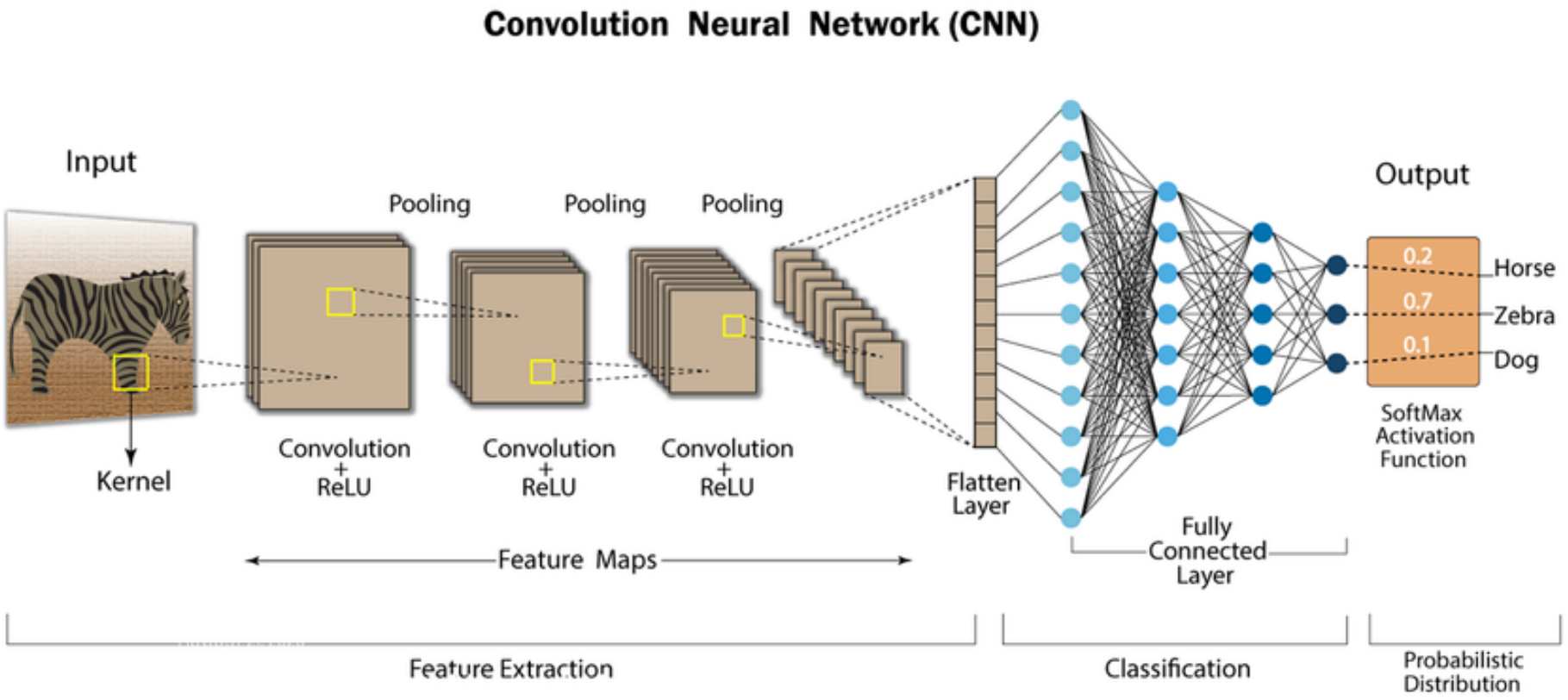
The Model

The model we'll use in this example is a variant of LeNet-5.
It belongs to **CNN** architecture as we want to classify images.

Convolution layers are used for feature extraction and fully connected layers for the classification of the images.
ReLU is the activation function (preferred in CNNs).

```
class GarmentClassifier(nn.Module):
    def __init__(self):
        super(GarmentClassifier, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 4 * 4, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 4 * 4)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

The Model


Convolution Neural Network (CNN)
Input
Kernel
Pooling
Pooling
Pooling
Convolution + ReLU
Convolution + ReLU
Convolution + ReLU
Feature Maps
Flatten Layer
Fully Connected Layer
Output
0.2
0.7
0.1
Horse
Zebra
Dog
SoftMax Activation Function
Feature Extraction
Classification
Probabilistic Distribution

The Training Loop

An ML model is trained by looping over data multiple times.
For each iteration, the weight values are adjusted with the backpropagation algorithm.

In each iteration the model makes a guess about the output, calculates the error in its guess (loss), collects the derivatives of the error with respect to its parameters and optimizes these parameters (using gradient descent).

We need to define a **Loss function** and an **Optimization** algorithm.

The Loss Function

The best loss function for problems like image classification, natural language processing, and recommender systems is cross-entropy.

Cross-entropy is a measure of the difference between the predicted probability distribution and the true probability distribution.

The goal is to minimize the cross-entropy loss, which is equivalent to maximizing the log-likelihood of the true probability distribution.

$$(x, y) = L = \{l_1, \dots, l_N\}^\top,$$

$$l_n = -\sum_{c=1}^{C} w_c \log \frac{\exp(x_{n,c})}{\sum_{i=1}^{C} \exp(x_{n,i})} y_{n,c}$$

where x is the input,
y is the target,
w is the weight,
C is the number of classes, and N spans the minibatch dimension

```
loss_fn = torch.nn.CrossEntropyLoss()
```

The Optimizer

Now that we have defined a loss function, we need to optimize the training so to minimize the loss function.

The most common optimization algorithm is **Gradient Descent**. The steps for forward propagation of this algorithm are:

1. Start from any random point
2. Determine which direction to go to reduce the loss (left or right) Gradient Descent
    1. Calculate the first order derivative (slope) of the loss function at this point
    2. Shift to the right if slope is negative or shift to the left if slope is positive by alpha (learning rate) times
3. Repeat

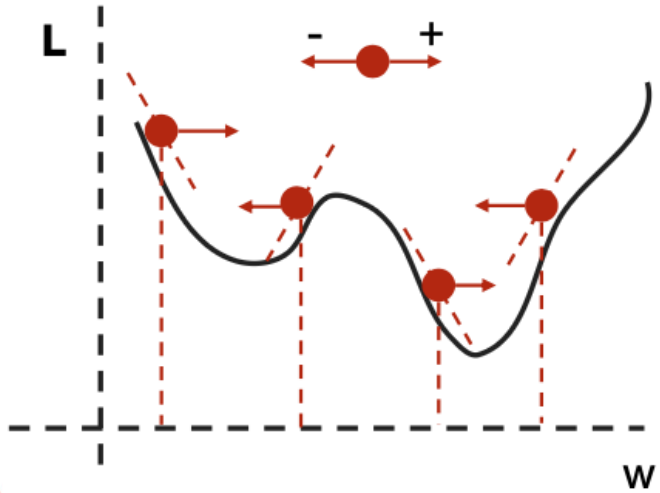
Gradient Descent

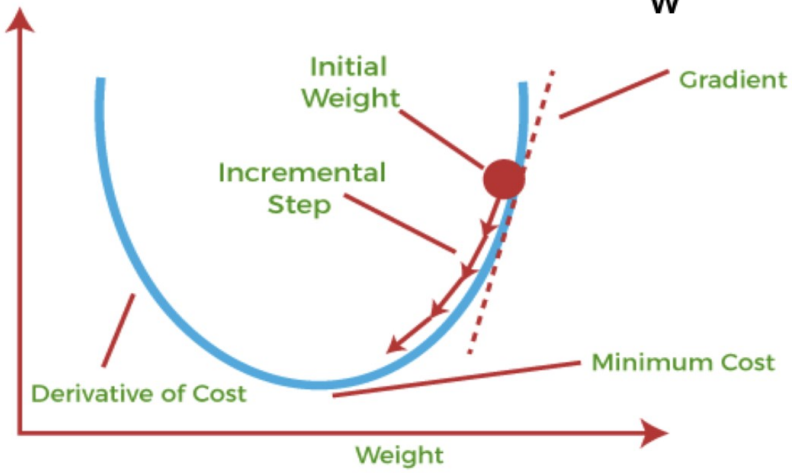
$$w^{(i+1)} = w^{(i)} - \lambda \frac{d\mathcal{L}}{dw}$$

This algorithm calculates the first order (slope) that finds a local minimum of a function.

- L (loss) is decreasing in the direction of the negative derivative.
- The learning rate (alpha) is controlled by the magnitude of λ.

The slope becomes steeper at the starting point, but whenever new parameters are generated, the steepness reduces and at the lowest point, it approaches the point of convergence (minimum loss).





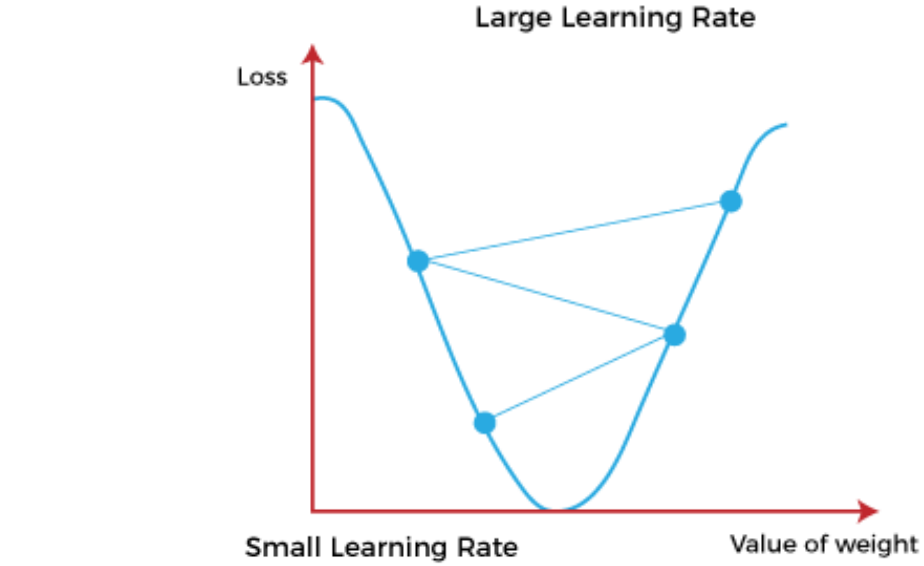
Learning Rate Parameter

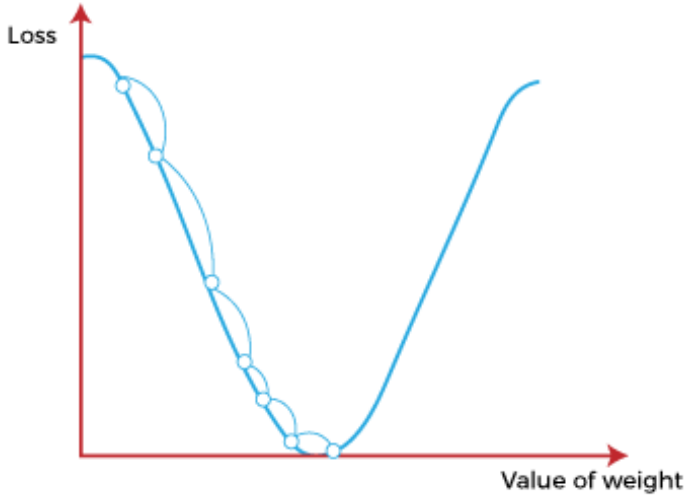
Defined as the **step size taken to reach the minimum** or lowest point.

Typically, a **small value** that is evaluated and updated based on the behavior of the loss function.

If the learning rate is high, it results in larger step but also leads to risks of overshooting the minimum.

If the learning rate is low, it results in smaller step sizes, which compromises overall efficiency but gives the advantage of more precision.



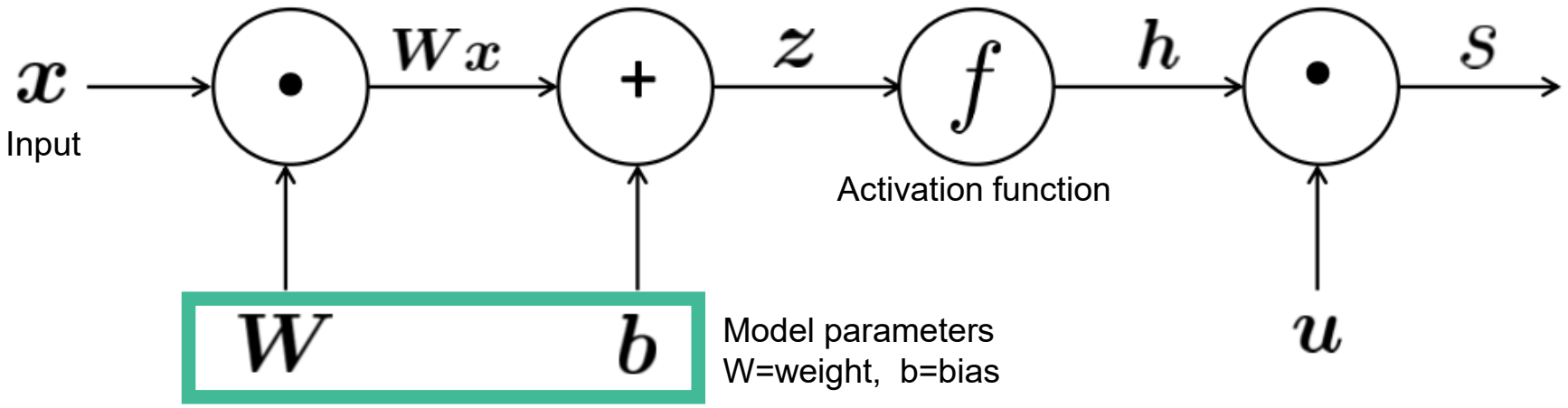


Computation Graphs

Forward Propagation

- Software represents our neural net equations as a graph
  - Source node: input
  - Interior nodes: operations
  - Edges pass along result of the operation

$$s = u^T h$$
$$h = f(z)$$
$$z = Wx + b$$
$$x \quad \text{(input)}$$

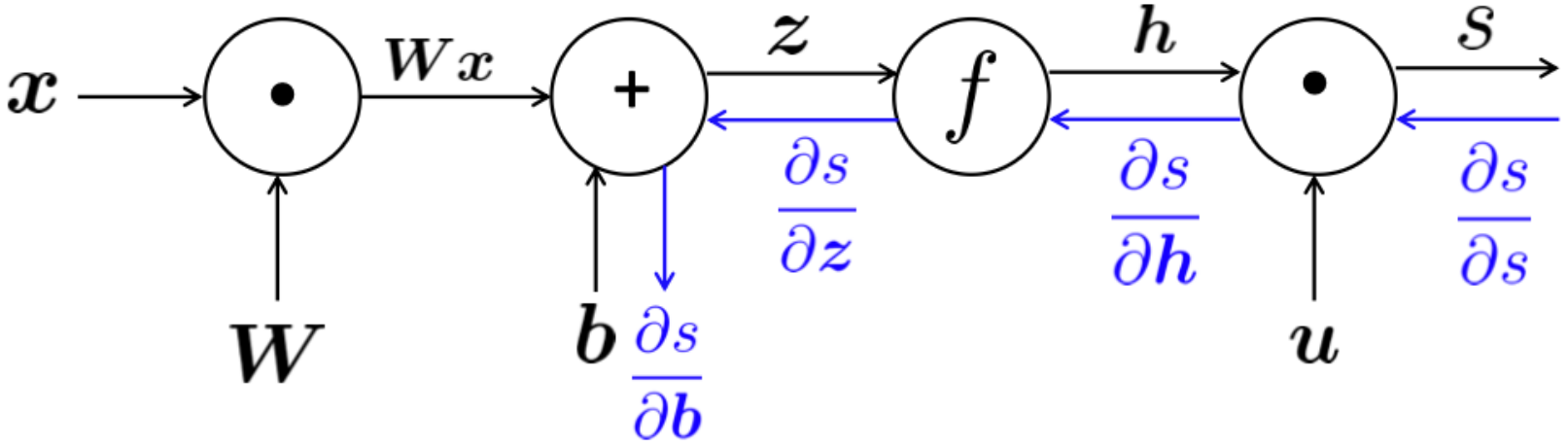


Computation Graphs

Back Propagation

- Then go backwards along edges
  - Pass along **gradients**

$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad \text{(input)}$$



The Optimizer

We use the Stochastic Gradient Descent optimizer. You can try other optimization algorithms such as Adam or Adagrad.

Hyperparameters to check:

- **Learning rate** determines the size of the steps the optimizer takes. How much to update model's parameters at each batch/epoch.
- **Momentum** nudges the optimizer in the direction of strongest gradient over multiple steps.

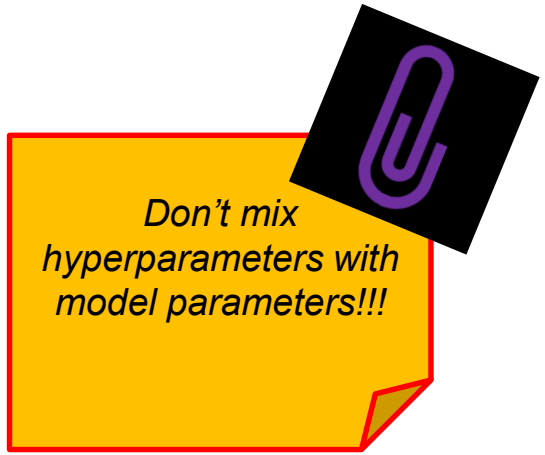
```
optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
```

Hyperparameters

Adjustable parameters that let you control the model optimization process. Different hyperparameter values can impact model training and convergence rates.

We define the following hyperparameters for training:

1. **Number of Epochs** - the number times to iterate over the dataset
2. **Batch Size** - the number of data samples propagated through the network before the parameters are updated
3. **Learning Rate** - how much to update model's parameters at each batch/epoch



Training Loop

```
def train_one_epoch(epoch_index, tb_writer):
    running_loss = 0.
    last_loss = 0.

    # Here, we use enumerate(training_loader) instead of
    # iter(training_loader) so that we can track the batch
    # index and do some intra-epoch reporting
    for i, data in enumerate(training_loader):
        # Every data instance is an input + label pair
        inputs, labels = data

        # Zero your gradients for every batch!
        optimizer.zero_grad()

        # Make predictions for this batch
        outputs = model(inputs)

        # Compute the loss and its gradients
        loss = loss_fn(outputs, labels)
        loss.backward()

        # Adjust learning weights
        optimizer.step()

        # Gather data and report
        running_loss += loss.item()
        if i % 1000 == 999:
            last_loss = running_loss / 1000 # loss per batch
            print('  batch {} loss: {}'.format(i + 1, last_loss))
            tb_x = epoch_index * len(training_loader) + i + 1
            tb_writer.add_scalar('Loss/train', last_loss, tb_x)
            running_loss = 0.

    return last_loss
```

- We define the number of epochs
- We define our training function
    1. We make a prediction
    2. We calculate the loss and gradients (forward pass)
    3. We backpropagate
    4. We adjust the weights
    5. We report the data

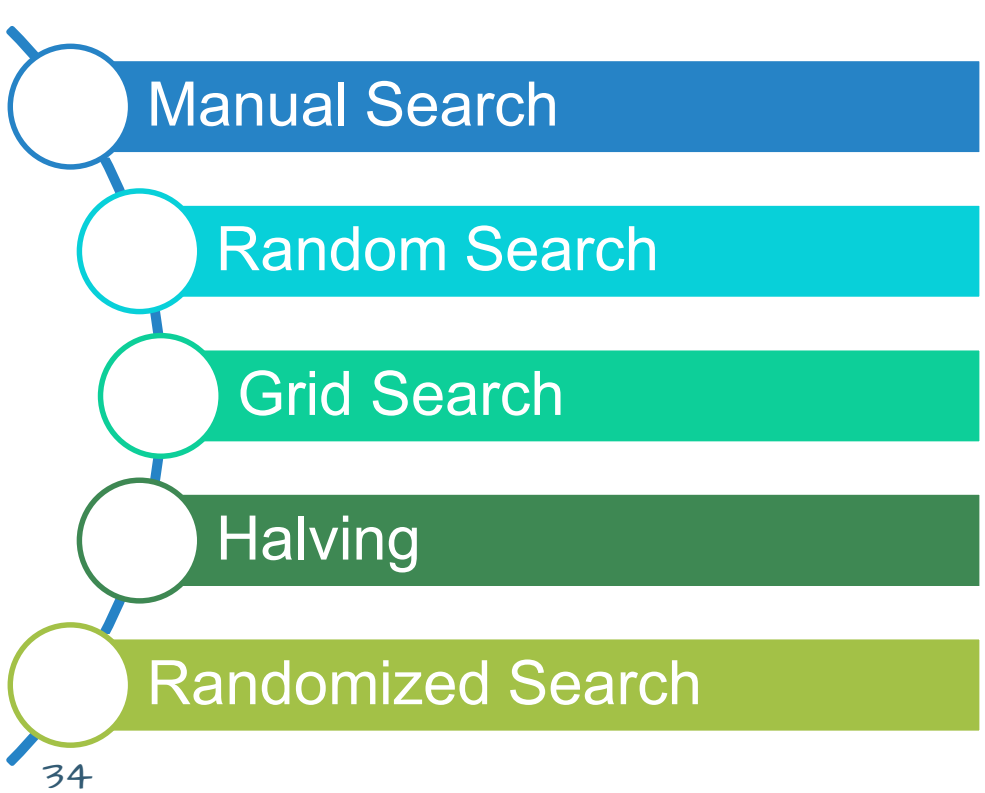
Training Loop

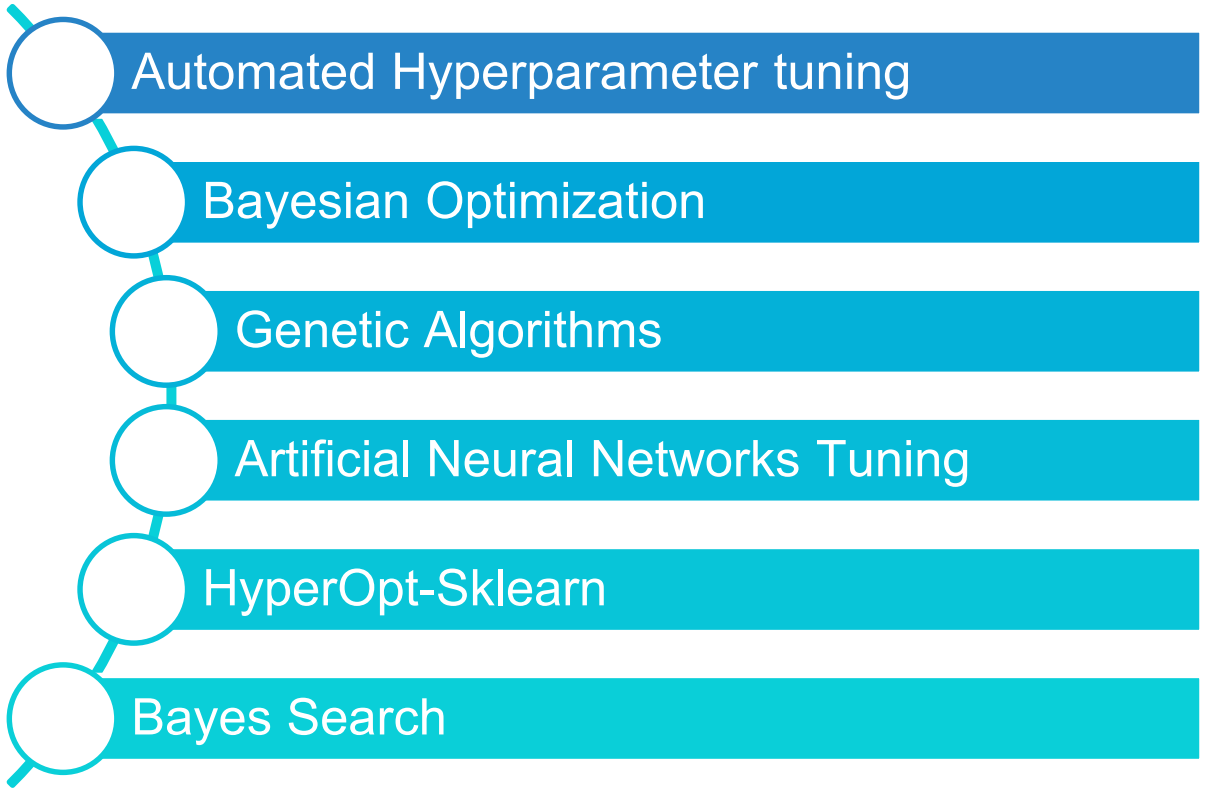
```
for epoch in range(EPOCHS):
    print('EPOCH {}:'.format(epoch_number + 1))

    # Make sure gradient tracking is on, and do a pass over the data
    model.train(True)
    avg_loss = train_one_epoch(epoch_number, writer)


    running_vloss = 0.0
    # Set the model to evaluation mode, disabling dropout and using population
    # statistics for batch normalization.
    model.eval()

    # Disable gradient computation and reduce memory consumption.
    with torch.no_grad():
        for i, vdata in enumerate(validation_loader):
            vinputs, vlabels = vdata
            voutputs = model(vinputs)
            vloss = loss_fn(voutputs, vlabels)
            running_vloss += vloss

    avg_vloss = running_vloss / (i + 1)
    print('LOSS train {} valid {}'.format(avg_loss, avg_vloss))
```

6. We run our training function
7. We evaluate its performance
8. We report the data
9. We repeat until the learning criteria have met

Hyperparameter Optimization Techniques

In the ML world, many Hyperparameter optimization techniques are available in case we want to automate the training tunning.





End of Session 2