

# Training Neural Networks in PyTorch

Sourav Karmakar

[souravkarmakar29@gmail.com](mailto:souravkarmakar29@gmail.com)

# Section-1: Autograd in Pytorch

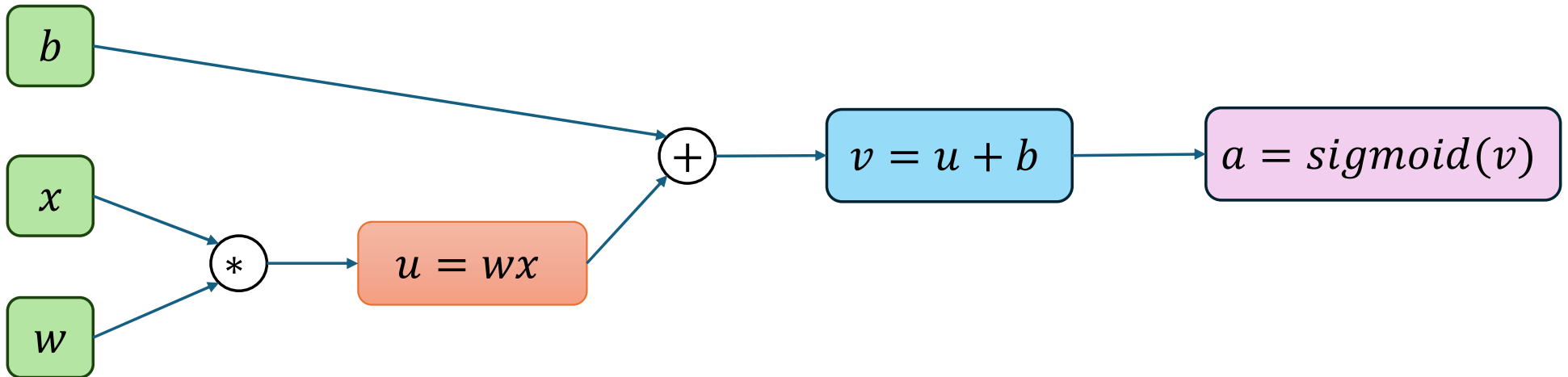
# Autograd in PyTorch

Let's consider the computation graph of following:

$$a(x, w, b) = \text{sigmoid}(\underbrace{w \cdot x}_u + b)$$

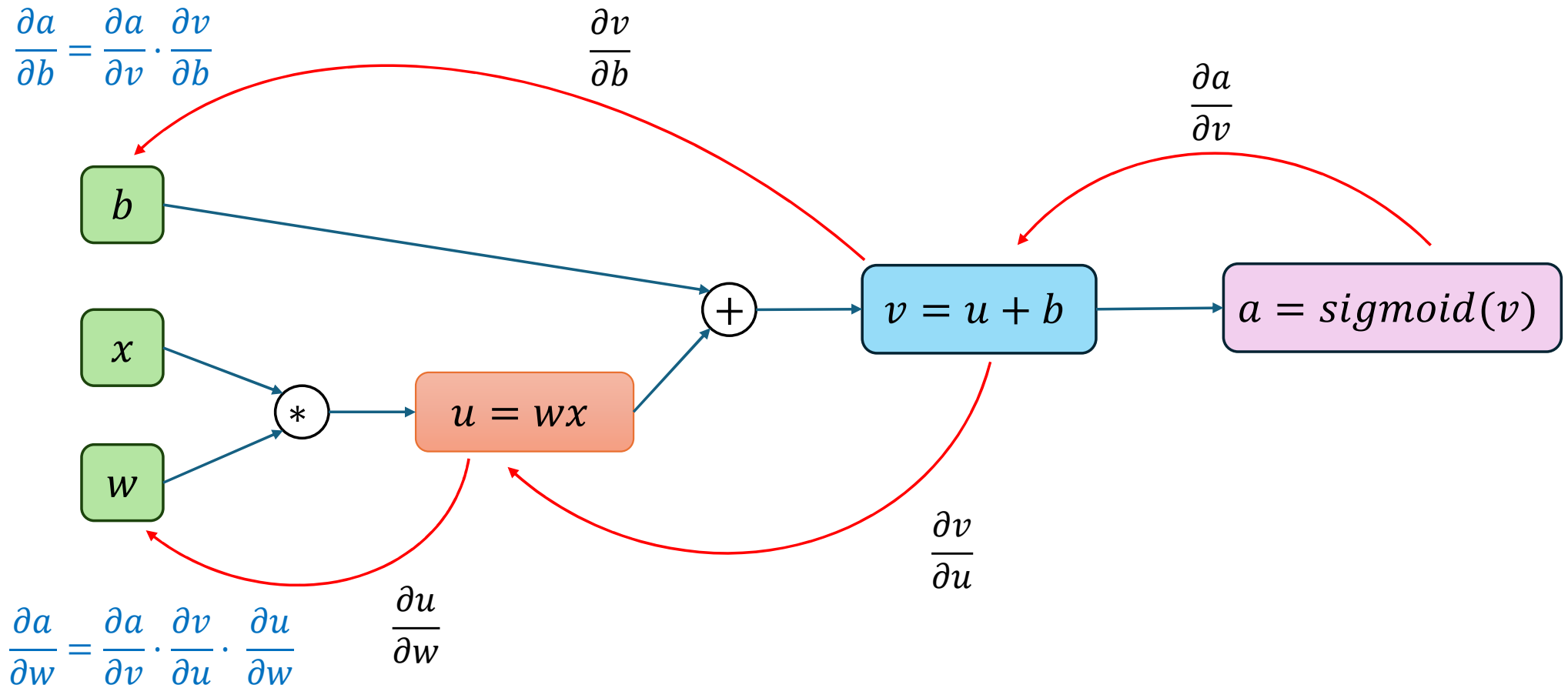
$v$

Where,  $\text{sigmoid}(z) = \frac{1}{1+e^{-z}}$

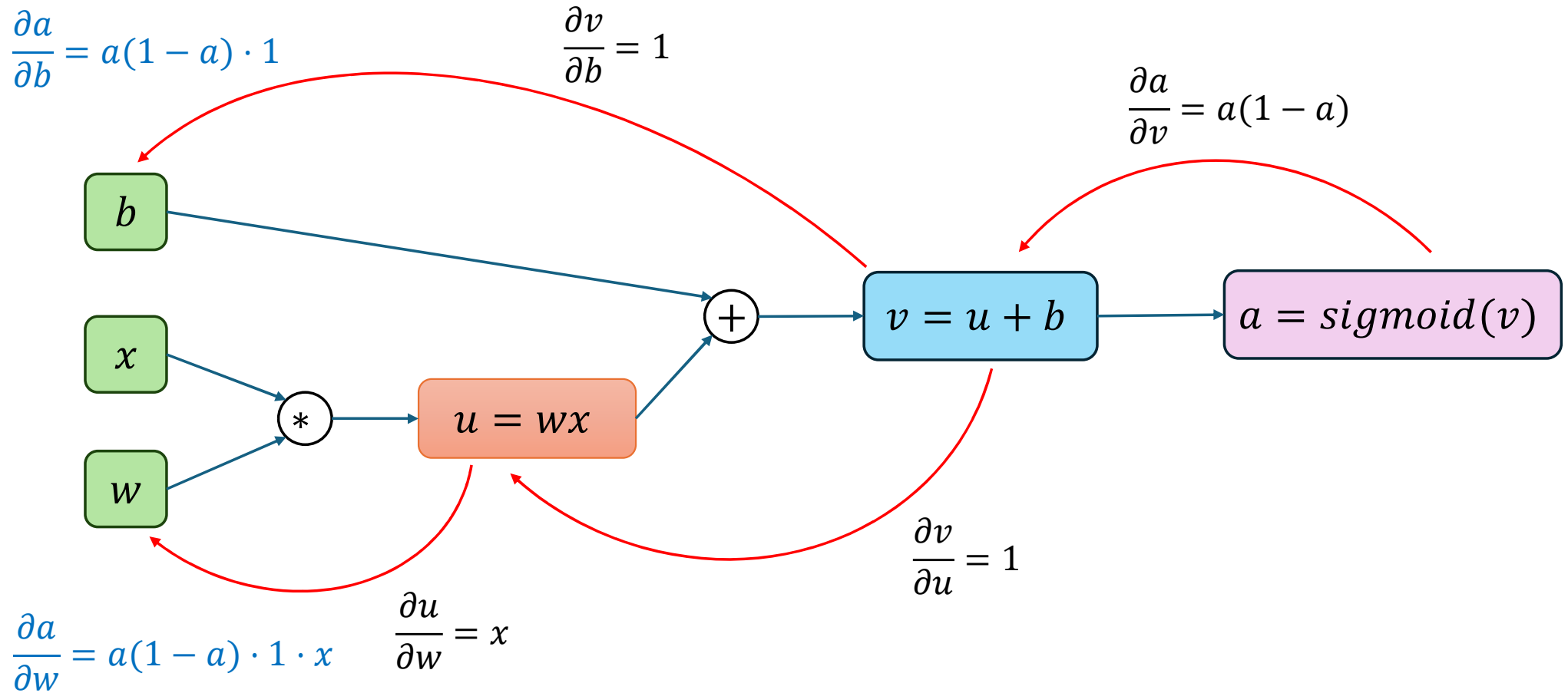


# Autograd in PyTorch

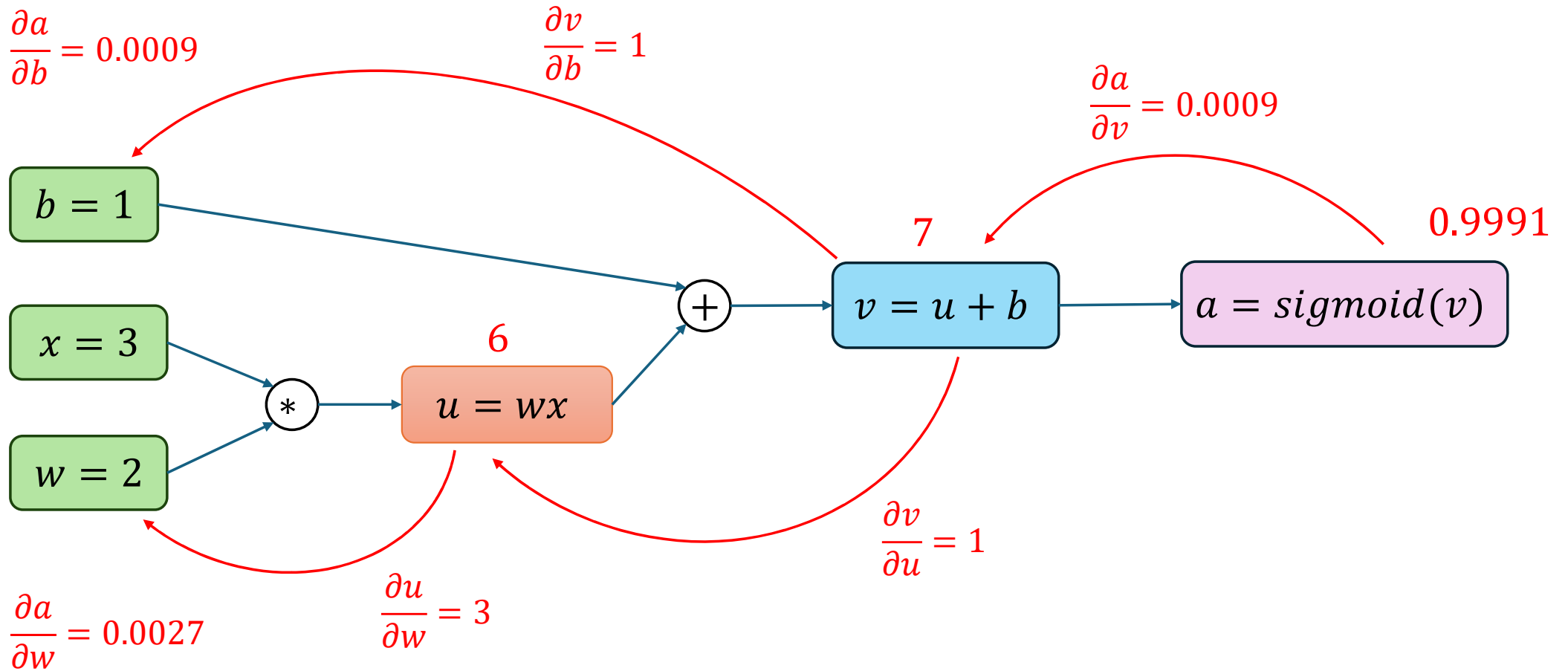
$$\frac{\partial a}{\partial b} = \frac{\partial a}{\partial v} \cdot \frac{\partial v}{\partial b}$$



# Autograd in PyTorch



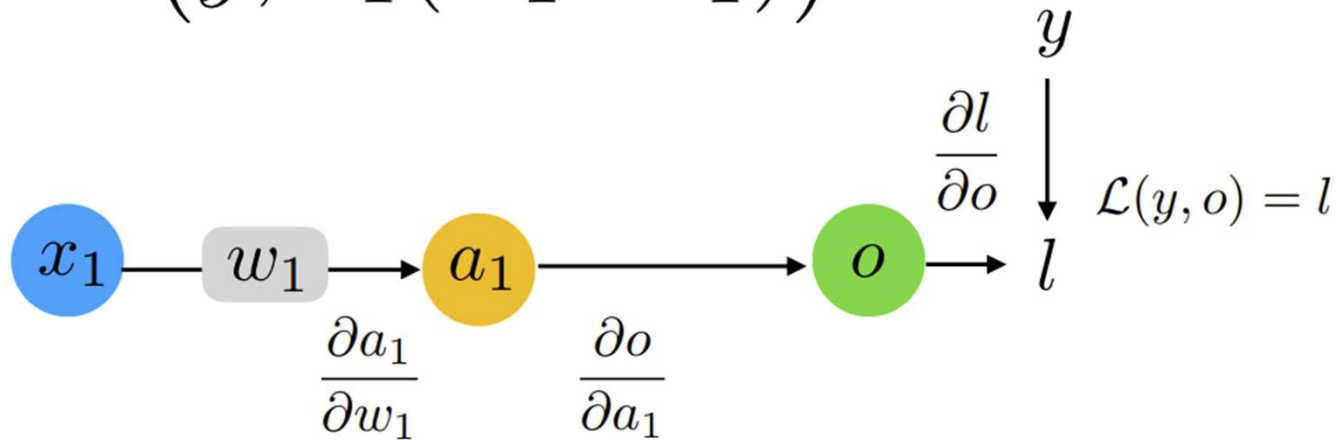
# Autograd in PyTorch



# Some More Computation Graphs

## Graphs with Single Path

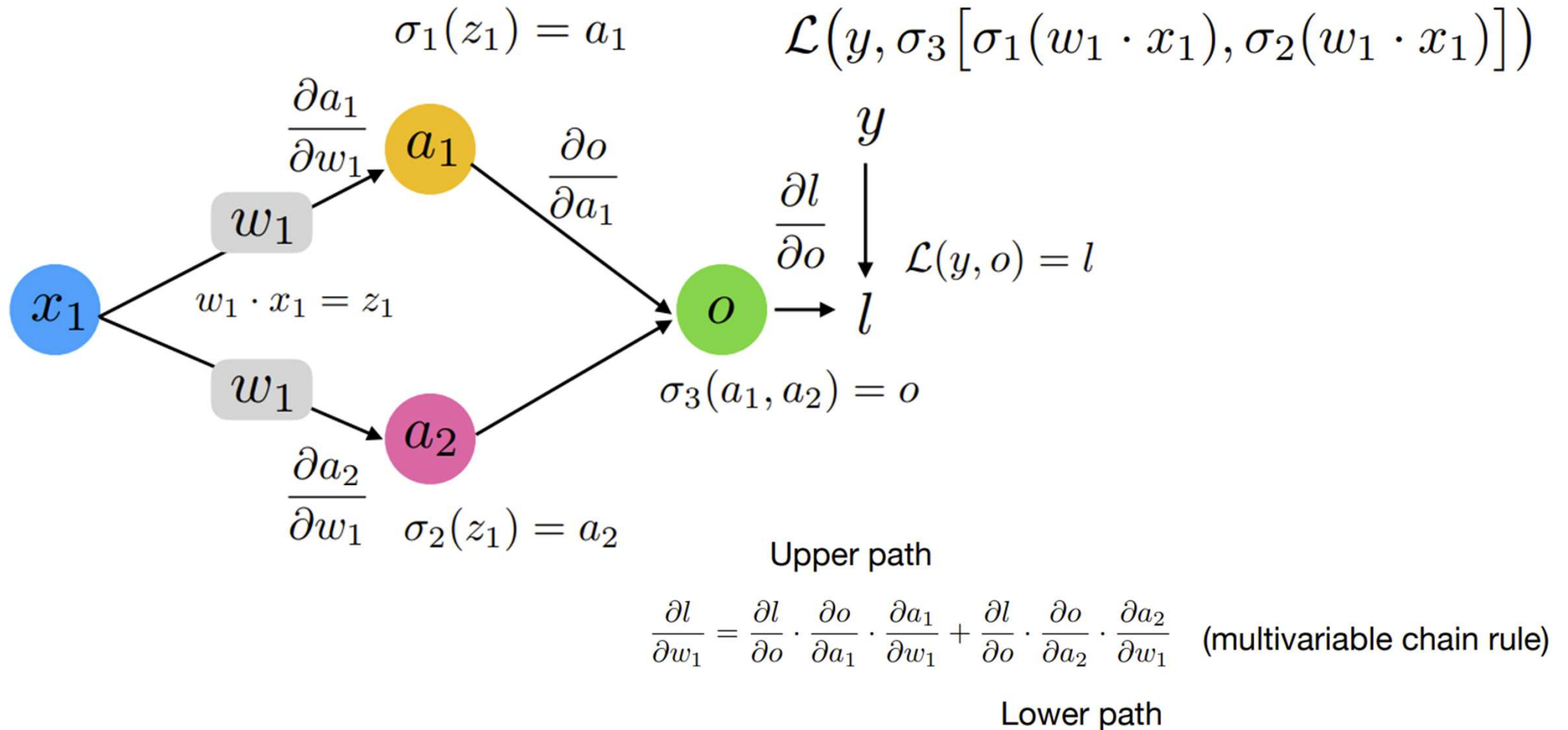
$$\mathcal{L}(y, \sigma_1(w_1 \cdot x_1))$$



$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1} \cdot \frac{\partial a_1}{\partial w_1} \quad (\text{univariate chain rule})$$

# Some More Computation Graphs

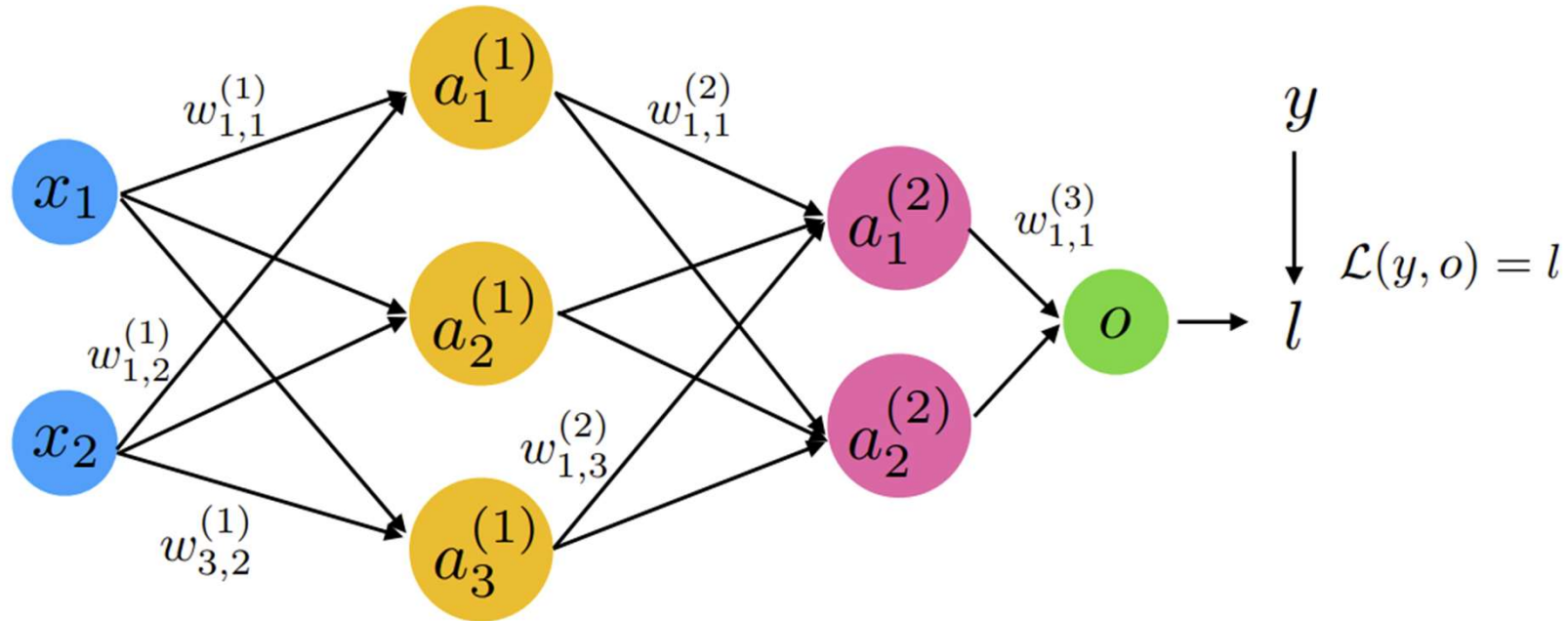
## Graphs with Weight Sharing





# Some More Computation Graphs

## Graphs with Fully Connected Layers (MLP)



$$\begin{aligned} \frac{\partial l}{\partial w_{1,1}^{(1)}} &= \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1^{(2)}} \cdot \frac{\partial a_1^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}} \\ &\quad + \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_2^{(2)}} \cdot \frac{\partial a_2^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}} \end{aligned}$$

## Section-2: Important Modules in PyTorch

# torch.nn module in PyTorch

## What is torch.nn?

- `torch.nn` = **Neural Network** building blocks in PyTorch.
- Provides layers, activations, loss functions, and utilities for deep learning.
- Designed to work seamlessly with **autograd** for gradient computation.
- Helps organize models into **modular**, reusable components.
- Central abstraction: `nn.Module` – base class for all neural network layers/ models.

## Key features of torch.nn

- **Predefined Layers:** `nn.Linear`, `nn.Conv2d`, `nn.LSTM`, etc.
- **Activation Functions:** `nn.ReLU`, `nn.Sigmoid`, `nn.Tanh`, etc.
- **Loss Functions:** `nn.CrossEntropyLoss`, `nn.BCELoss`, `nn.MSELoss`, etc.
- **Container Modules:** `nn.Sequential`, `nn.ModuleList`, `nn.ModuleDict`, etc. for structuring models.
- **Parameter Management:** automatically tracks model parameters (weights, biases) for optimization.
- **Integration with Optimizers:** works smoothly with `torch.optim` for training.

# torch.optim module in PyTorch

## What is torch.optim?

- `torch.optim` = **Optimization Algorithms** for training neural networks in PyTorch.
- Works with parameters tracked by `nn.Module`.
- Updates model weights based on computed gradients from **autograd**.
- Separates **model definition** (`torch.nn`) from **training logic** (`torch.optim`) maintaining **modular** components.
- Central abstraction: `torch.optim.Optimizer` class. All optimizers inherit from this base class.

## Key features of torch.optim

- **Popular Optimizers Available:**
  - Gradient descent (SGD).
  - Adaptive methods: Adam, RMSprop, Adagrad etc. (we will learn more about these adaptive methods)
- **Parameter Management:** Takes model parameters (`model.parameters()`) as input.
- **Hyper parameters:** learning rate, momentum, weight decay, etc.
- **Zeroing Gradients:** `optimizer.zero_grad()` prevents accumulation.
- **Updating model parameters:** `optimizer.step()` automatically updates model parameters.
- **Scheduler Support:** `torch.optim.lr_scheduler` for learning rate adjustment during training.

# torch.utils.data.Dataset and DataLoader

What are torch.utils.data.Dataset and DataLoader?

- **Dataset**

- Represents your data (images, text, tabular, etc.)
- Defines how to access a single sample and its label.
- Custom datasets are created by subclassing torch.utils.data.Dataset

- **DataLoader**

- Wraps a Dataset and provides batching, shuffling, and parallel loading.
- Iterates over data efficiently during training.

Together: Provide an easy and scalable way to feed data into neural networks.

## Key features

- **Dataset**

- `__len__` : returns dataset size
- `__getitem__` : fetches one data sample.
- **Flexible**: can load from csv, images, databases etc.

- **Dataloader**

- Batches data automatically by given `batch_size`.
- Shuffles data each epoch (`shuffle = True`)
- Supports parallelism (`num_workers`) for speed.
- Returns data as an iterator for training loops.

# Loading a csv dataset in PyTorch

```
1 import torch
2 from torch.utils.data import Dataset, DataLoader
3 import pandas as pd
4
5 # Custom Dataset for CSV
6 class CSVDataset(Dataset):
7     def __init__(self, csv_file):
8         # Load CSV into a Pandas DataFrame
9         self.data = pd.read_csv(csv_file)
10
11         # Separate features (X) and Labels (y)
12         self.X = self.data.iloc[:, :-1].values # all columns except last
13         self.y = self.data.iloc[:, -1].values # last column as label
14
15     def __len__(self):
16         return len(self.data) # number of rows
17
18     def __getitem__(self, idx):
19         # Convert to tensors
20         features = torch.tensor(self.X[idx], dtype=torch.float32)
21         label = torch.tensor(self.y[idx], dtype=torch.long)
22         return features, label
23
24
25 # Usage
26 dataset = CSVDataset("my_data.csv")
27 loader = DataLoader(dataset, batch_size=32, shuffle=True)
28
29 # Iterate through batches
30 for batch_X, batch_y in loader:
31     print(batch_X.shape, batch_y.shape)
```

Creating a custom dataset (CSVDataset) by inheriting from `torch.utils.data.Dataset`

Separate features (X) with labels (y). This can be done in different ways for different datasets.

Using the class to load the data. Then using `DataLoader` on top of it to provide batching, shuffling

# Functional vs Object Oriented APIs

## Object Oriented API (`nn.Module` based)

- Layers are defined as **objects** (e.g. `nn.Linear`, `nn.Conv2d` etc.)
- Object oriented representations are state-full. Parameters (weights and biases) are stored internally as `torch.nn.Parameter`.
- Automatically integrated with the model's parameter management (`model.parameters()`).
- Typically used when building standard models (MLPs, CNNs, RNNs etc.)
- Useful for automatic parameter registration, saving / loading the models.

## Functional API (`torch.nn.functional` based)

- Layers are defined as **functions**. (e.g. `F.linear`, `F.relu` etc.)
- Functional representations are state-less. Requires to explicitly pass weights and biases.
- Doesn't create persistent parameters – it just performs computation.
- Useful for building custom layers, experimenting.
- Functional APIs give low level controls, enabling the manipulation of the weights dynamically (e.g. weight sharing, custom initialization).

# Functional vs Object Oriented APIs

## Example of Object-Oriented API

```
1 import torch
2 import torch.nn as nn
3
4 # Define an input tensor (batch_size=2, input_dim=3)
5 x = torch.tensor([[1.0, 2.0, 3.0],
6                   [4.0, 5.0, 6.0]])
7
8 # Define a linear layer: input features=3, output features=2
9 linear_layer = nn.Linear(in_features=3, out_features=2)
10
11 # Forward pass (automatically uses internal weights & bias)
12 output = linear_layer(x)
```

- `nn.Linear` stores weights and bias as Parameters.
- You can get them by `linear_layer.weight` and `linear_layer.bias`
- When you call `linear_layer(x)`, it applies:
$$y = xW^T + b$$

## Example of Functional API

```
1 import torch
2 import torch.nn as nn
3 import torch.nn.functional as F
4
5 # Define an input tensor (batch_size=2, input_dim=3)
6 x = torch.tensor([[1.0, 2.0, 3.0],
7                   [4.0, 5.0, 6.0]])
8
9 # Explicitly register parameters
10 weights = nn.Parameter(torch.randn(out_features, in_features, dtype=torch.float),
11                             requires_grad=True)
12
13 bias = nn.Parameter(torch.randn(out_features, dtype=torch.float),
14                       requires_grad=True)
15
16 # Functional call: explicitly provide weight and bias
17 output_func = F.linear(x, weights, bias)
```

- `F.linear(x, weights, bias)` is just a function.
- Doesn't track parameter by itself.
- Good for custom forward definitions in `nn.Module`.



## Section-3: Different Loss Functions in PyTorch

# Different Loss Functions in PyTorch

Loss function usually takes two sets of inputs: predicted values (inputs) and target values.

Here, we will talk about few widely used loss functions defined in PyTorch.

We will see both functional and object-oriented forms of loss functions.

## Mean Square Error loss function:

- Object oriented form: **`nn.MSELoss()`**
- Functional form: **`nn.functional.mse_loss(inputs, targets)`**
- Used for regression (like: linear regression) or reconstruction (like: Auto-Encoders) type problems.
- The input and target needs to be torch tensors of same size.

```
1 import torch.nn as nn
2 import torch.nn.functional as F
3
4 # object oriented representation
5 loss_func = nn.MSELoss()
6
7 loss = loss_func(inputs, targets)
8
9 # functional representation
10
11 loss = F.mse_loss(inputs, targets)
```

→ `loss_func` defined as object of the class

# Different Loss Functions in PyTorch

## Binary Cross Entropy Loss:

- Object oriented form: **nn.BCELoss()**
- Functional form: **nn.functional.binary\_cross\_entropy(inputs, targets)**
- Used for binary classification problems.
- This takes logistic sigmoid values (i.e. probabilities) as inputs.
- The target needs to be encoded in 0 and 1.

```
1 import torch.nn as nn
2 import torch.nn.functional as F
3
4 # object oriented representation
5 loss_func = nn.BCELoss()
6
7 loss = loss_func(inputs, targets)
8
9 # functional representation
10
11 loss = F.binary_cross_entropy(inputs, targets)
```

loss\_func defined as object of the class

# Different Loss Functions in PyTorch

## Binary Cross Entropy With Logits Loss:

- Object oriented form: `nn.BCEWithLogitsLoss()`
- Functional form: `nn.functional.binary_cross_entropy_with_logits(inputs, targets)`
- Used for binary classification problems.
- This takes logits (before applying sigmoid) as inputs. It has built-in sigmoid layer that applies sigmoid internally.
- The target needs to be encoded in 0 and 1.

```
1 import torch.nn as nn
2 import torch.nn.functional as F
3
4 # object oriented representation
5 loss_func = nn.BCEWithLogitsLoss()
6
7 loss = loss_func(logits, targets)
8
9 # functional representation
10
11 loss = F.binary_cross_entropy_with_logits(logits, targets)
```

→ `loss_func` defined as object of the class

# Different Loss Functions in PyTorch

## Cross Entropy Loss:

- Object oriented form: **`nn.CrossEntropyLoss()`**
- Functional form: **`nn.functional.cross_entropy (inputs, targets)`**
- Used for multi-class classification problems.
- This takes logits (before applying softmax) as inputs. It has built-in softmax layer that applies softmax internally.
- The target needs to be label-encoded i.e. in  $0, 1, 2, \dots, C-1$ . [where  $C$  is the number of class]

```
1 import torch.nn as nn
2 import torch.nn.functional as F
3
4 # object oriented representation
5 loss_func = nn.CrossEntropyLoss()
6
7 loss = loss_func(logits, targets)
8
9 # functional representation
10
11 loss = F.cross_entropy(logits, targets)
```

→ `loss_func` defined as object of the class

# Different Loss Functions in PyTorch

## Negative Log Likelihood Loss:

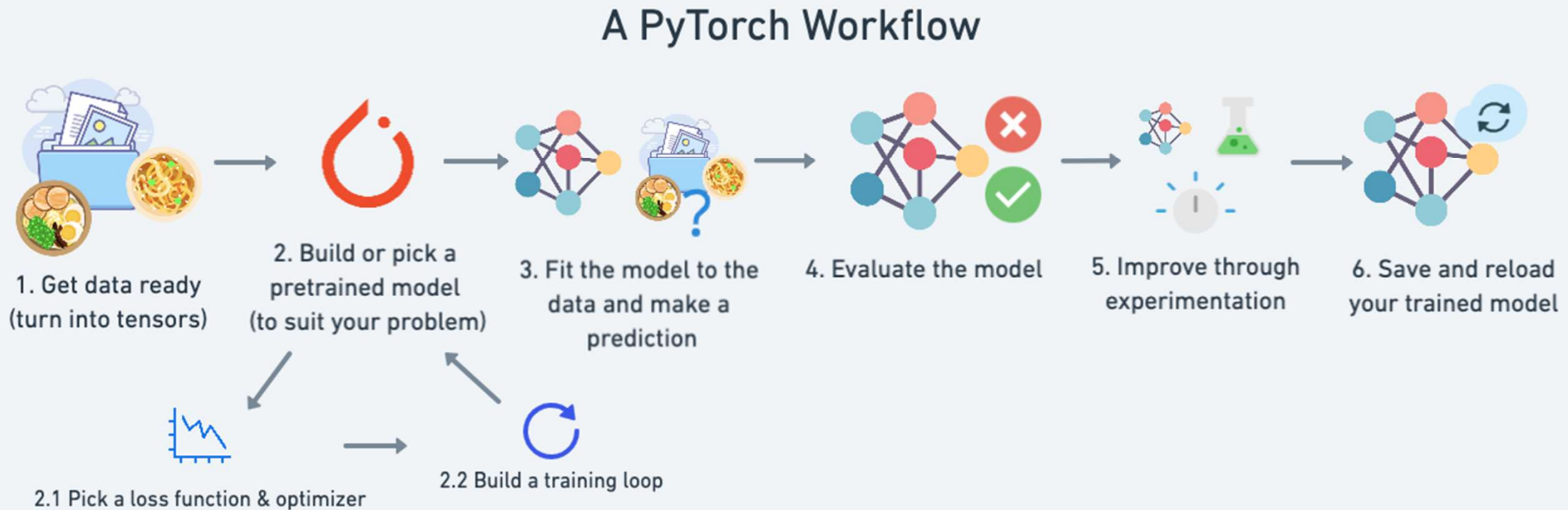
- Object oriented form: **nn.NLLLoss()**
- Functional form: **nn.functional.nll\_loss (inputs, targets)**
- Used for multi-class classification problems.
- This takes log-softmax (logarithm of softmax) as inputs.
- The target needs to be label-encoded i.e. in 0,1,2,...,C-1. [where C is the number of class]

```
1 import torch.nn as nn
2 import torch.nn.functional as F
3
4 inputs = torch.log_softmax(logits, dim=1)
5
6 # object oriented representation
7 loss_func = nn.NLLLoss()
8
9 loss = loss_func(inputs, targets)
10
11 # functional representation
12
13 loss = F.nll_loss(inputs, targets)
```

→ loss\_func defined as object of the class

# Section-4: Defining, Creating, Training, and Evaluating Neural Networks in PyTorch

# A PyTorch Workflow





# MLP in PyTorch: Step-1 (Definition)

```
1 import torch
2 import torch.nn as nn
3 import torch.nn.functional as F
4
5 class MultilayerPerceptron(nn.Module):
6
7     def __init__(self, input_size, hidden_size_1, hidden_size_2, num_classes):
8
9         super(MultilayerPerceptron, self).__init__()
10
11         # 1st hidden Layer
12         self.fc1 = nn.Linear(input_size, hidden_size_1)
13
14         # 2nd hidden Layer
15         self.fc2 = nn.Linear(hidden_size_1, hidden_size_2)
16
17         # Output Layer
18         self.linear_out = nn.Linear(hidden_size_2, num_classes)
19
20     def forward(self, x):
21
22         # passing through 1st hidden Layer
23         out = self.fc1(x)
24
25         # activation function of 1st hidden layer
26         out = F.relu(out)
27
28         # passing through 2nd hidden Layer
29         out = self.fc2(out)
30
31         # activation function of 2nd hidden layer
32         out = F.relu(out)
33
34         logits = self.linear_out(out)
35
36         # Softmax to get probabilities
37         probas = F.softmax(logits, dim=1)
38
39         return logits, probas
```

Functional API to call the activation functions.

**MultilayerPerceptron** class which inherits **torch.nn.Module**.

Initializing the super (parent) class. `super().__init__()` will also do.

Defining the architecture of the feedforward neural network (MLP).  
The layers are defined by **torch.nn.Linear**, which are also called fully connected layers or dense layers.

The **forward** method defines the forward pass of the neural network.  
When inheriting from **torch.nn.Module** we need to explicitly create the **forward** method. This is a fundamental requirement in pytorch's architecture.

Pre and post activation of first hidden layer.

Pre and post activation of second hidden layer.

Logits / pre-activation of final classification layer.

Converting final classification layer's output to probability using **softmax**. This is optional.

# MLP in PyTorch: Step-1 (Definition)

There is another neat way of creating the neural network using `torch.nn.Sequential` container module.

```
1 import torch
2 import torch.nn as nn
3 import torch.nn.functional as F
4
5 class MultiLayerPerceptron(nn.Module):
6
7     def __init__(self, input_size, hidden_size_1, hidden_size_2, num_classes):
8
9         super(MultiLayerPerceptron, self).__init__()
10
11         # All layers within Sequential container step by step
12         self.my_network = nn.Sequential(
13             nn.Linear(input_size, hidden_size_1),
14             nn.ReLU(),
15             nn.Linear(hidden_size_1, hidden_size_2),
16             nn.ReLU(),
17             nn.Linear(hidden_size_2, num_classes)
18         )
19
20     def forward(self, x):
21
22         # Forward pass through the network
23         logits = self.my_network(x)
24
25         # Softmax to get probabilities
26         probas = F.softmax(logits, dim=1)
27
28         return logits, probas
```

This code is more clear and more compact.

Note that how the activation functions are called as layers (object-oriented representation) in contrast to functional representation earlier.

But there are few caveats:

1. We can't use this, if you want to define your neural network with custom activation function and layers.
2. `forward` may be harder to debug if there are errors, as we can't simply add breakpoints or insert print statements within the `Sequential` block.

# MLP in PyTorch: Step-2 (Creation)

```
1 torch.manual_seed(random_seed) # for reproducibility
2
3 if torch.cuda.is_available():
4     device = "cuda" # Use NVIDIA GPU (if available)
5 elif torch.backends.mps.is_available():
6     device = "mps" # Use Apple Silicon GPU (if available)
7 else:
8     device = "cpu" # Default to CPU if no GPU is available
9
```

```
10 # Defining model
11 model = MultilayerPerceptron(input_size, hidden_size_1, hidden_size_2, num_classes)
```

Instantiate model (creates the model parameters)

```
13 # move the model parameters to CPU/GPU/Apple Silicon GPU
14 model = model.to(device)
```

Moving the model to the available device (GPU / CPU)

```
16 # Stochastic Gradient Descent (SGD) optimizer
17 learning_rate = 0.01
18 optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

Defining an optimization method

# MLP in PyTorch: Step-3 (Training)

```
1 train_data_loader = DataLoader(dataset=train_dataset, batch_size=32, shuffle=True)
2
3 num_epochs = 100
4 for epoch in range(num_epochs):
5     model.train() # set the model to training mode
6
7     for batch_idx, (features, labels) in enumerate(train_data_loader):
8         # move the features and labels to CPU/GPU/Apple Silicon GPU
9         features = features.to(device)
10        labels = labels.to(device)
11
12        # Forward pass: compute predicted outputs by passing inputs to the model
13        logits, probas = model(features)
14
15        # Calculate the loss
16        loss = F.cross_entropy(logits, labels)
17
18        # Zero the gradients before running the backward pass.
19        optimizer.zero_grad()
20
21        # Backward pass: compute gradient of the loss with respect to model parameters
22        loss.backward()
23
24        # Perform a single optimization step (parameter update)
25        optimizer.step()
```

→ Specifying a DataLoader for training dataset

→ Run for a specified number of epochs

→ Iterate over mini-batches in each epoch

→ Make sure your data is in the same hardware (device) as model.

→  $y = \text{model}(x)$  calls `__call__` and then `.forward()`.  
**Don't run `y = model.forward(x)` directly.**

→ Specify loss function based on the problem. Notice, how the `cross_entropy` loss is applied to `logits` and not on `probas`.

→ To prevent gradient accumulation, we need to zero them before each backward pass

→ These two steps calculates the loss gradients w.r.t. model parameters and then updates the parameter. i.e. these two steps together perform gradient descent operation.

# MLP in PyTorch: Step-4 (Evaluation)

## Training / Validation / Test Split

- Training data set is used for training the neural networks. It's not necessary to plot the training loss / accuracy during training but it can be useful.
- Validation set accuracy provides a rough estimate of the generalization performance and then can be used to tune the hyper-parameters of the model.
- Test set should only be used once to get an unbiased estimate of the generalization performance.
- The **train : validation : test** ratio depends on dataset size, but usually for the large dataset the 80:5:15 split is a good idea.
- Sometimes validation data is not present, only train and test datasets are available. Then usually it is a good idea to split the train dataset further (like 90:10 split), where the larger part will be used for training and smaller part for validation.
- In many cases, with the absence of validation dataset, people often skip the validation part entirely and evaluate the generalization performance of the model on the test dataset.

# MLP in PyTorch: Step-4 (Evaluation)

```
1 def compute_loss(net, data_loader):
2     net.eval() # evaluation mode
3     total_loss, total_samples = 0.0, 0
4     with torch.no_grad(): # no gradient tracking
5         for features, targets in data_loader:
6             features = features.to(device),
7             targets = targets.to(device)
8             logits = net(features)
9             loss = F.cross_entropy(logits, targets, reduction="sum") # sum, not mean
10            total_loss += loss.item()
11            total_samples += targets.size(0)
12    return total_loss / total_samples
13
14 def compute_accuracy(net, data_loader):
15     net.eval() # evaluation mode
16     correct_pred, num_examples = 0, 0
17     with torch.no_grad(): # no gradient tracking
18         for features, targets in data_loader:
19             features = features.to(device)
20             targets = targets.to(device)
21
22             logits = net(features)
23             predicted_labels = torch.argmax(logits, dim=1)
24
25             num_examples += targets.size(0)
26             correct_pred += (predicted_labels == targets).sum().item()
27
28    return correct_pred / num_examples * 100
```

By default, the cross-entropy loss computes the mean or average loss over entire mini-batch.

However, here we are trying to compute the sum of the loss for entire dataset and then compute average.

Hence, `cross_entropy` loss is used with reduction “sum”.



# MLP in PyTorch: Step-4 (Evaluation)

```
30 training_loss = []
31 validation_loss = []
32
33 for epoch in range(num_epochs):
34     model.train()
35     ### training codes ###
36
37     # training loss
38     loss_train = compute_loss(model, train_data_loader)
39     training_loss.append(loss_train)
40
41     # validation loss
42     loss_val = compute_loss(model, validation_data_loader)
43     validation_loss.append(loss_val)
44
45     # training accuracy for each epoch
46     training_accuracy = compute_accuracy(model, train_data_loader)
47
48     # validation accuracy for each epoch
49     validation_accuracy = compute_accuracy(model, validation_data_loader)
50
51 # compute test accuracy after completeing the training
52 test_accuracy = compute_accuracy(model, test_data_loader)
```

Keep track of training and validation loss and accuracy in each epoch / training loop

Test accuracy is computed only when training is complete

# Few Important Notes

## `model.train()` vs `model.eval()`

In PyTorch, **`model.train()`** and **`model.eval()`** are essential for managing the behavior of specific layers during the training and inference phases, primarily affecting layers like **Dropout** and **Batch Normalization**.

The **`model.train()`** method sets the model to training mode, but it doesn't perform the training. The **`model.eval()`** method switches the model to evaluation mode. In this mode, the behavior of the Dropout and Batch Normalization layer changes.

We will learn about dropout and batch normalization in details later.

## `torch.no_grad()` vs `torch.inference_mode()`

### Similarities:

- Both disables gradient tracking.
- Reduce memory usage and improve inference speed.
- Used during evaluation / inference, not training.

### Differences:

- **`torch.no_grad()`** temporarily disables autograd but still maintains some autograd metadata internally. However, **`torch.inference_mode()`** is the stronger version which completely disables autograd and related metadata.
- While **`torch.no_grad()`** is good for “one-off” inference or validation step, **`torch.inference_mode()`** is more optimized and faster, especially on large models.
- **`torch.inference_mode()`** was introduced in PyTorch 1.9 for production grade inferences.



***Thank You***