



# Natural Language Processing

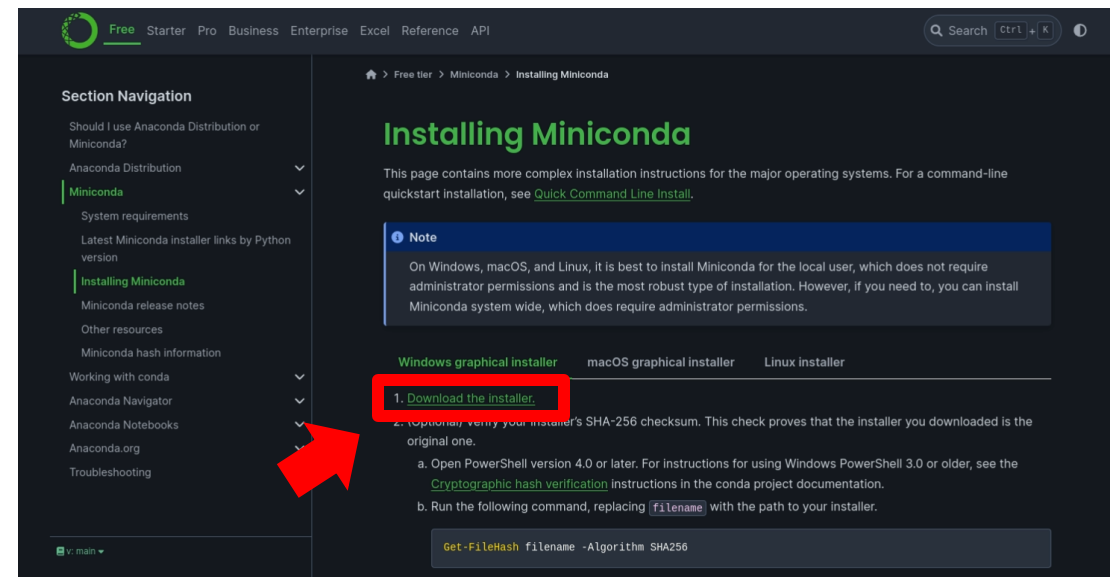
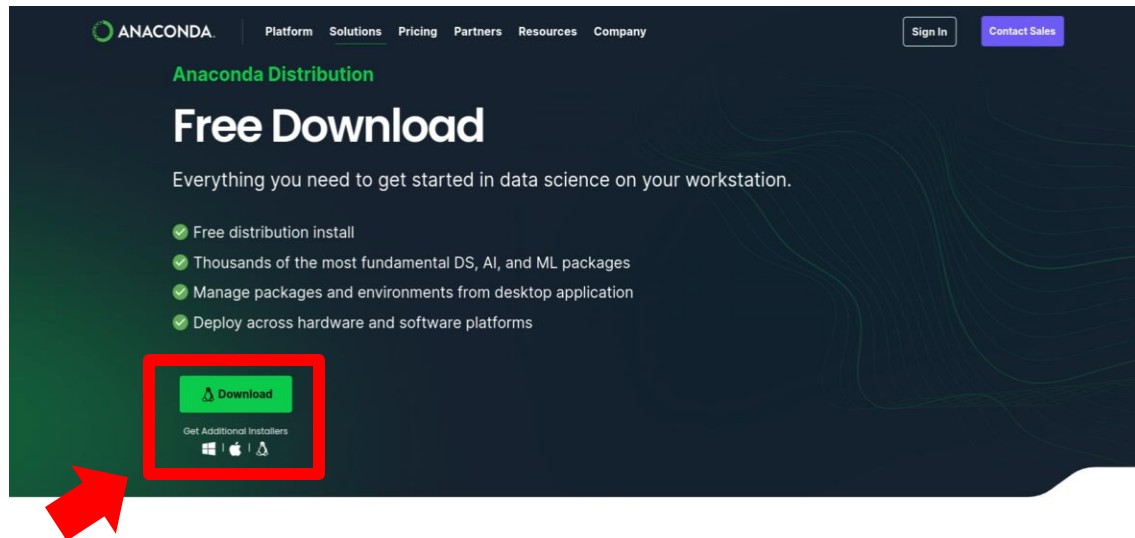
PyTorch tutorial



# Environment setup

Download anaconda (<https://www.anaconda.com/download>)

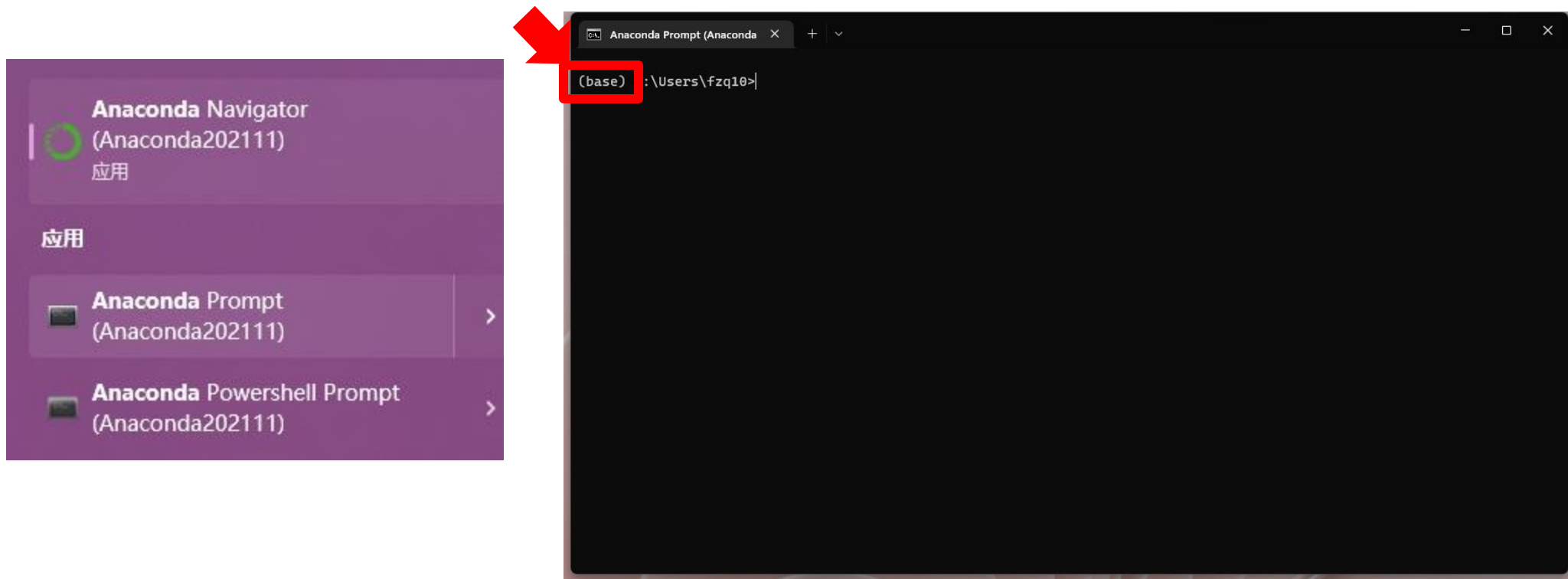
or miniconda (<https://docs.anaconda.com/free/miniconda/miniconda-install/>)



# Environment setup

Open Anaconda prompt

Current environment



# Basic Commands

---

After installing Anaconda, you can open the Anaconda Prompt and manage environments using commands. Alternatively, you can use the graphical interface, Anaconda Navigator, to perform operations directly.

1. Create python environment  
`conda create -n (name) python=3.X.X`
2. Activate python environment  
`conda activate (name)`
3. Exit environment  
`conda deactivate`
4. Check environments  
`conda env list`



# Install PyTorch

Install PyTorch: <https://pytorch.org/>

The screenshot shows the PyTorch website's installation guide. A red box highlights the 'Stable (2.2.1)' section, which includes the following options:

|           |            |          |        |
|-----------|------------|----------|--------|
| Linux     | Mac        | Windows  |        |
| Conda     | Pip        | LibTorch | Source |
| Python    | C++ / Java |          |        |
| CUDA 11.8 | CUDA 12.1  | ROCm 5.7 | CPU    |

Below the table, the command to install PyTorch is shown:

```
conda install pytorch torchvision torchaudio pytorch-cuda=11.8 -c pytorch -c nvidia
```

Two red arrows point from the 'Run this Command:' label to the command box. The right sidebar lists cloud providers: Amazon Web Services, Google Cloud Platform, and Microsoft Azure.

# Install PyTorch

You can also choose an earlier version, but the main consideration is that your computer's hardware must support the corresponding version of CUDA.

**NOTE:** Latest PyTorch requires Python 3.8 or later. For more details, see Python section below.

|                  |                |                   |          |        |
|------------------|----------------|-------------------|----------|--------|
| PyTorch Build    | Stable (2.2.1) | Preview (Nightly) |          |        |
| Your OS          | Linux          | Mac               | Windows  |        |
| Package          | Conda          | Pip               | LibTorch | Source |
| Language         | Python         | C++/Java          |          |        |
| Compute Platform | CUDA 11.8      | CUDA 12.1         | ROCm 5.7 | CPU    |

Run this Command:

```
conda install pytorch torchvision torchaudio pytorch-cuda=11.8 -c pytorch -c nvidia
```

[Previous versions of PyTorch >](#)

# Deep learning

---

Consider the polynomial relationship between  $y$  and  $x$  defined by the equation:

$$y = ax^2 + b$$

Here,  $a$  and  $b$  are the parameters we want to learn, and  $x$  is the input variable.

We are given four data points to train this model:

$$(-1, 1), (1, 2), (2, 3)$$

These data points represent pairs of  $x$  values and corresponding  $y$  values.

For example, when  $a=1$  and  $b=1$ , the predictions of  $-1, 1, 2$  are respectively  $2, 2, 5$ .  
We define the difference between predictions and ground truths as:

$$L = (\sum_i (y_i - y'_i)^2) / n$$





# Gradient descend

We can visualize how the difference (error) changes with respect to different values of the parameters  $a$  and  $b$  by plotting a surface, or plane.

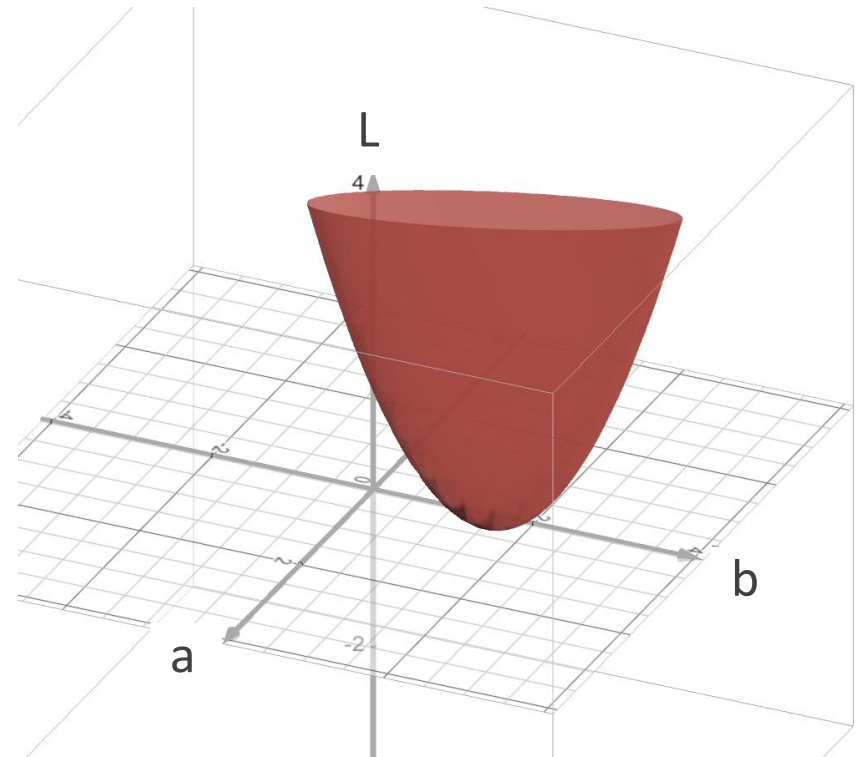
In deep learning, we optimize the parameters  $a$  and  $b$  using gradient descent.

We define:

$y = ax^2 + b$  --> model

$L = (\sum_i (y_i - y'_i)^2) / n$  --> Loss function

The tool used to update  $a$  and  $b$  using gradient descent. --> Optimizer





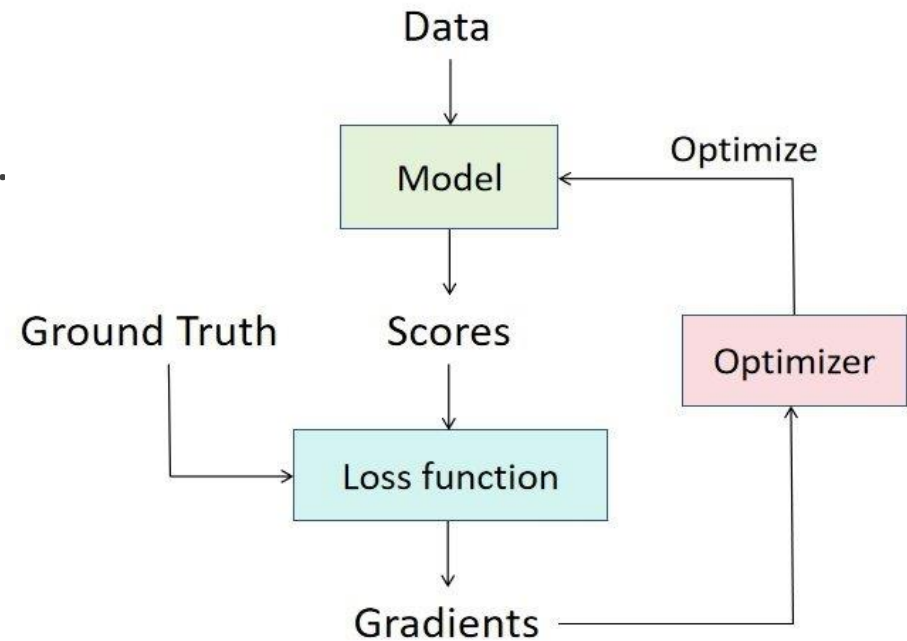
# Problem description

In a deep learning project, we need to:

1. Define a model
  - Input a batch of data, and compute the results.
2. Train the model
  - Record the derivation procedures.
  - Back propagate & Compute the gradients.
  - Optimize the parameters.
  - GPU acceleration.

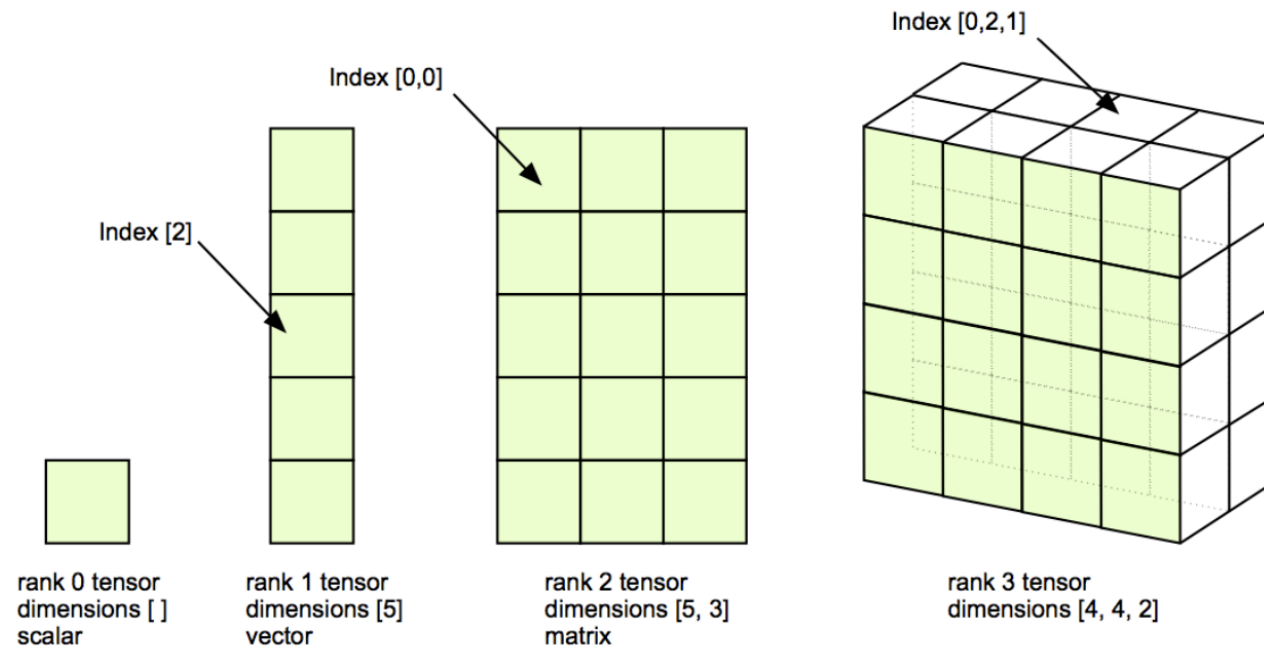
➡ A special data structure was invented.

Called "**Tensor**"(張量)



# The bricks of building models: Tensors

- Tensors are very similar to arrays and matrices.
- Tensors can run on GPUs or other hardware accelerators.



# Initialize a Tensor

---

There are many ways to initialize a PyTorch tensor:

From existed python list

```
T = torch.tensor([[1, 2], [3, 4]])
```

From numpy

```
T = torch.from_numpy(numpy.array([[1, 2], [3, 4]]))
```

Randomly initialized

Normal distributed: 

```
T = torch.randn(size=(2,2))
```

Uniform distributed: 

```
T = torch.rand(size=(2,2))
```



# Initialize a Tensor

---

Other methods

All of the elements are 1

```
T = torch.ones(size=(2,2))
```

All of the elements are 0

```
T = torch.zeros(size=(2,2))
```

Diagonalized matrix

```
T = torch.eye(n=2, m=3)
```



# Tensors

---

Initialize: `T = torch.tensor([[1, 2], [3, 4]])`

- Data (Return a copy of tensor)

`T.data => tensor([[1, 2], [3, 4]])`

- Data (Scalar only)

`t = torch.tensor(1.1), t.item() => 1.1`

- Shape (return a tuple)

`T.shape => torch.Size([2, 2])`

- Data type

`T.dtype => torch.int64`



# Tensors

---

- Device  
`T.device => device(type='cpu')`
- Require gradient  
`T.requires_grad => False`
- Gradient  
`T.grad => tensor([...])`



# An interesting insight of tensor.data

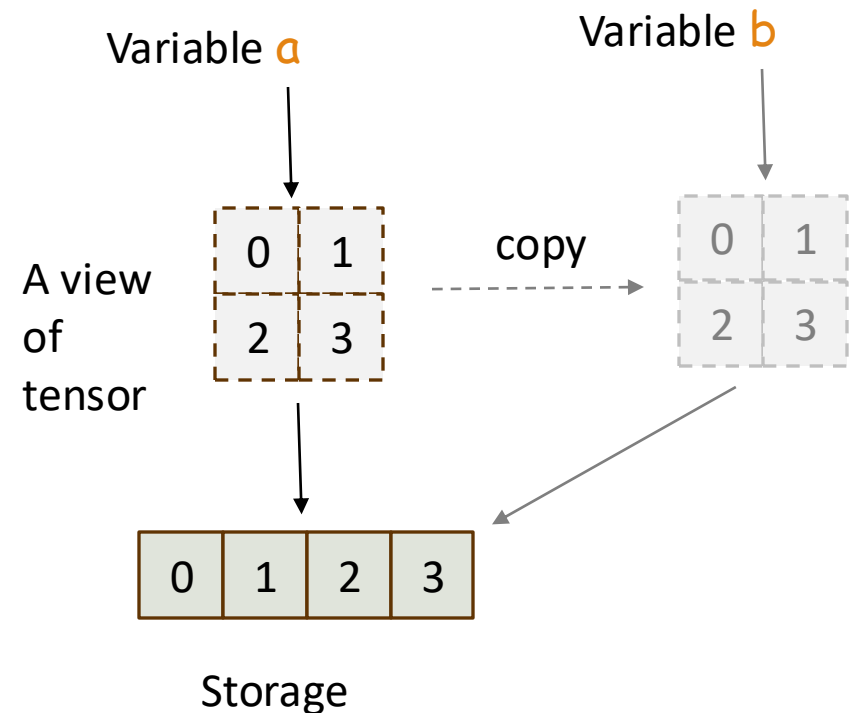
Consider the following code:

```
a = torch.arange(4).reshape(2, 2) # initialize
b = a.data
print(f"{id(a)}, {id(b)}") # print the memory locations
a[0, 1] = 100
print(a)
print(b)
```

The output is:

```
126169165861232, 126168491980688
tensor([[ 0, 100],
        [ 2,  3]])
tensor([[ 0, 100],
        [ 2,  3]])
```

Which means that `a.data` just creates a new view of tensor.





# Matrix operations

## Tensor & scalar

$A = \text{torch.tensor}([[1, 2], [3, 4]])$

$A+1 \Rightarrow \text{tensor}([[2, 3], [4, 5]])$

$A*2 \Rightarrow \text{tensor}([[2, 4], [6, 8]])$

| 符號                                  | 意義                                   |
|-------------------------------------|--------------------------------------|
| <code>torch.Tensor + scalar</code>  | 張量中的每個數值加上 <code>scalar</code>       |
| <code>torch.Tensor - scalar</code>  | 張量中的每個數值減去 <code>scalar</code>       |
| <code>torch.Tensor * scalar</code>  | 張量中的每個數值乘上 <code>scalar</code>       |
| <code>torch.Tensor / scalar</code>  | 張量中的每個數值除以 <code>scalar</code>       |
| <code>torch.Tensor // scalar</code> | 張量中的每個數值除以 <code>scalar</code> 所得之商  |
| <code>torch.Tensor % scalar</code>  | 張量中的每個數值除以 <code>scalar</code> 所得之餘數 |
| <code>torch.Tensor ** scalar</code> | 張量中的每個數值取 <code>scalar</code> 次方     |

The operation is applied equally to every elements in the tensor except matrix multiplication(`@`).



# Matrix operations

```
A = torch.tensor([[1, 2], [3, 4]])
```

```
B = torch.tensor([[0, 1], [2, 3]])
```

+, -

```
A+B => tensor([[1, 3], [5, 7]])
```

```
A-B => tensor([[1, 1], [1, 1]])
```

Matrix multiply

```
A@B => tensor([[4, 7], [8, 15]])
```

Element-wise multiply

```
A*B => tensor([[0, 2], [6, 12]])
```

| 符號     | 意義                              |
|--------|---------------------------------|
| A + B  | 張量 A 中的每個數值加上張量 B 中相同位置的數值      |
| A - B  | 張量 A 中的每個數值減去張量 B 中相同位置的數值      |
| A * B  | 張量 A 中的每個數值乘上張量 B 中相同位置的數值      |
| A / B  | 張量 A 中的每個數值除以張量 B 中相同位置的數值      |
| A // B | 張量 A 中的每個數值除以張量 B 中相同位置的數值所得之商  |
| A % B  | 張量 A 中的每個數值除以張量 B 中相同位置的數值所得之餘數 |
| A ** B | 張量 A 中的每個數值取張量 B 中相同位置的數值之次方    |



# Matrix operations

---

Concatenation (equivalent to `torch.concat`)

```
torch.cat((A,B), dim=0) => tensor([[1, 2], [3, 4], [0, 1], [2, 3]])
```

```
torch.cat((A,B), dim=1) => tensor([[1, 2, 0, 1], [3, 4, 2, 3]])
```

Split

```
T = torch.randn((5,6,7))
```

```
out = torch.split(T, 2, dim=0)
```



A tuple of tensors and their size are:

```
torch.Size([2, 6, 7]), torch.Size([2, 6, 7]), torch.Size([1, 6, 7])
```

Squeeze, Unsqueeze

```
A.unsqueeze(dim=-1) => tensor([[[1], [2]], [[3], [4]]])
```

```
A.squeeze(dim=-1) => tensor([[1, 2], [3, 4]])
```

Transpose

```
A.transpose(dim0=0, dim1=1) => tensor([[1, 3], [2, 4]])
```



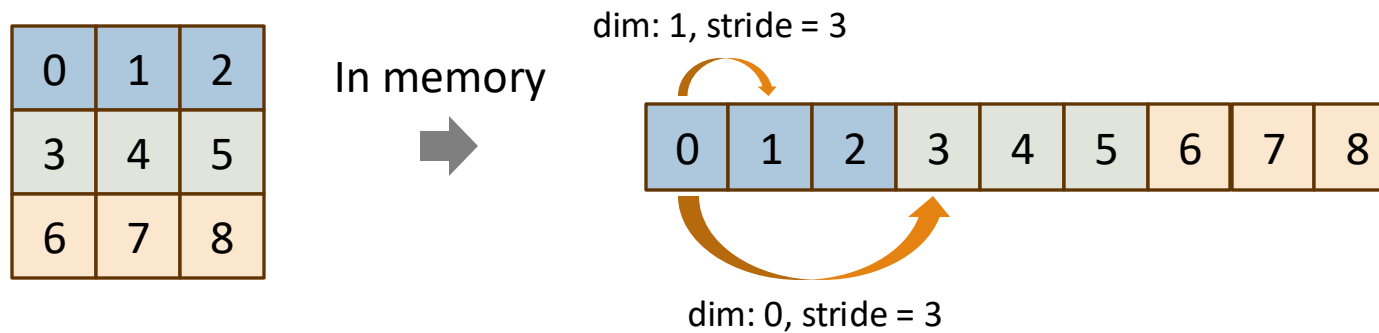
# Matrix operations

---

Stride: defines the step (in RAM) size used to access the next token along a given dimension.

```
A = torch.arange(9).reshape(3, 3)
```

```
A.stride() => (3, 1)
```

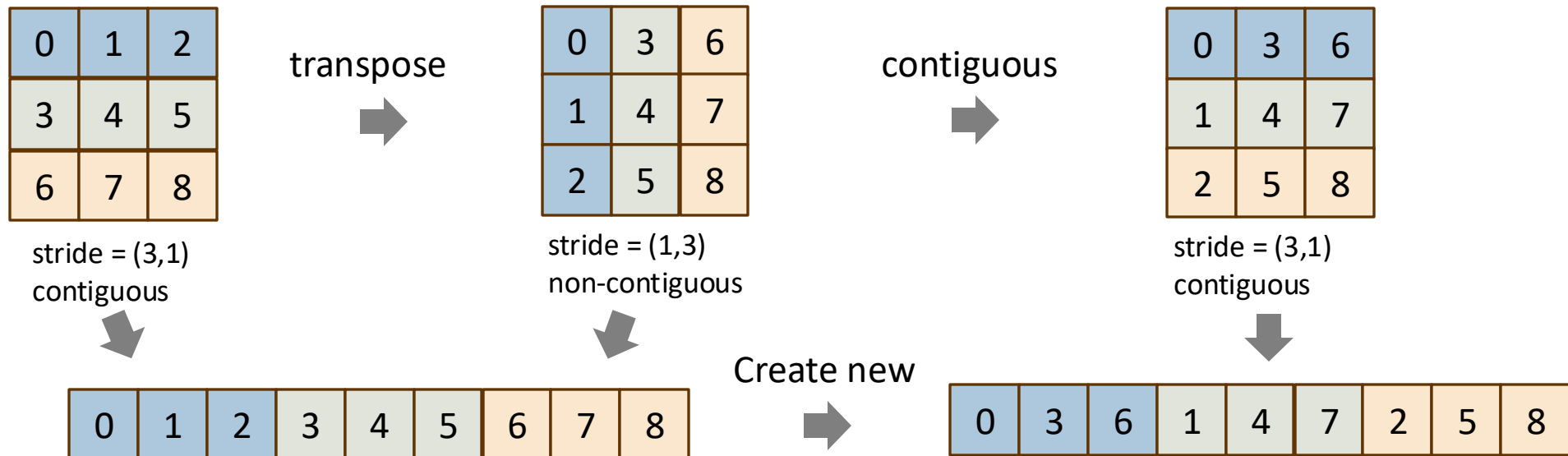


# Matrix operations

contiguous:

`A.transpose(0, 1)`

`A = A.contiguous()`



# Matrix operations

---

## Reshape

```
A = torch.tensor([[1, 2], [3, 4]])
```

```
A.view(4, 1) => tensor([[1], [2], [3], [4]])
```

```
A.view(1, 4) => tensor([[1, 2, 3, 4]])
```

```
A.reshape(4, 1) => tensor([[1], [2], [3], [4]])
```

```
A.reshape(1, 4) => tensor([[1, 2, 3, 4]])
```

# Matrix operations

---

The difference between **view** and **reshape**.

The **view** function does not modify the tensor's data in memory or create a new tensor; it simply reshapes the existing tensor by changing how the data is interpreted.

- It cannot be applied to non-contiguous tensors.

The **reshape** function can reshape non-contiguous tensors.

- For contiguous tensors, it is equivalent to **view**.
- For non-contiguous tensors, it will be a copy and the result is equivalent to **contiguous+view**.





# Matrix operations

---

Note: All PyTorch computations are performed in a matrix operation mode.

e.g. Randomly initialize a tensor(100X100) and normalize it:

Sequentially: 0.127 s

Matrix Operation: 0.000088s

```
def sequential(mat : torch.tensor):  
    t = time.time()  
    average = torch.mean(mat)  
    standard = torch.std(mat)  
    output = torch.zeros([mat.shape])  
    for i in range(mat.shape[0]):  
        for j in range(mat.shape[1]):  
            output[i, j] = (mat[i, j]-average)/standard  
    return output, time.time() - t  
  
def parallel(mat : torch.tensor):  
    t = time.time()  
    average = torch.mean(mat)  
    standard = torch.std(mat)  
    output = (mat - average)/standard  
    return output, time.time() - t
```

# Construct a Model

A Model in PyTorch is basically a `torch.nn.Module` object

The model is defined as:

```
class myModel(torch.nn.Module):  
    def __init__(self):  
        super().__init__()  
        self.layers = ...  
  
    def forward(self, input):  
        ...  
        return ...
```

Obtain the output logits

```
model = myModel()  
logits = model(input)
```

`torch.nn.Module.__call__`

Two member functions that must be defined  
when constructing deep learning models



# Construct a Model

---

## Components in a model

| Module                                   | Function                                  |
|--|---|
| <code>torch.nn.Linear</code>             | Linear projection layer (fully connected) |
| <code>torch.nn.Conv2d(1d/3d)</code>      | 1/2/3d convolution layer                  |
| <code>torch.nn.RNN</code>                | Recurrent Neural Network layer            |
| <code>torch.nn.LSTM</code>               | Long Short-term memory network layer      |
| <code>torch.nn.MaxPool2d(1d/3d)</code>   | Max pooling layer                         |
| <code>torch.nn.Embedding</code>          | Embedding layer                           |
| <code>torch.nn.BatchNorm2d(1d/3d)</code> | Batch normalization layer                 |
| <code>torch.nn.Sequential</code>         | Sequentially combined moduls              |
| <code>torch.nn.ModuleList</code>         | List of modules                           |
| <code>torch.nn.ModuleDict</code>         | Dictionary of modules                     |



# Member functions in a Model

---

Most commonly used member functions.

| Module                                | Function   |
|---------------------------------------|--|
| <code>model.forward()</code>          | Forward pass   |
| <code>model.state_dict()</code>       | Return an OrderedDict contains parameter names and tensors |
| <code>model.parameters()</code>       | Return an iterator for parameter tensors                   |
| <code>model.named_parameters()</code> | Return an iterator for parameter names and tensors         |
| <code>model.train()</code>            | Training mode  |
| <code>model.eval()</code>             | Evaluation mode  |



# Model Components

---

`torch.nn.ModuleList:`

```
class myModel(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.linear1 = torch.nn.Linear(128, 128)
        self.linear2 = torch.nn.Linear(128, 128)
        self.linear3 = torch.nn.Linear(128, 128)
    def forward(self, input):
        output1 = self.linear1(input)
        output2 = self.linear2(input)
        output3 = self.linear3(input)
        return output1, output2, output3
```

Rewrite



```
class myModel(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.linear_list = torch.nn.ModuleList([
            torch.nn.Linear(128, 128) for _ in range(3)
        ])
    def forward(self, input):
        outputs = ()
        for i in range(3):
            outputs += (self.linear_list[i](input), )
        return outputs
```

# Model Components

`torch.nn.Sequential:`

Source code

```
def forward(self, input):  
    for module in self:  
        input = module(input)  
    return input
```

```
class myModel(torch.nn.Module):
```

```
    def __init__(self):
```

```
        super().__init__()
```

```
        self.linear1 = torch.nn.Linear(128, 128)
```

```
        self.act = torch.nn.ReLU()
```

```
        self.linear2 = torch.nn.Linear(128, 2)
```

Rewrite



```
    def forward(self, input):
```

```
        x = self.linear1(input)
```

```
        x = self.act(x)
```

```
        output = self.linear2(x)
```

```
        return output
```

```
class myModel(torch.nn.Module):
```

```
    def __init__(self):
```

```
        super().__init__()
```

```
        self.layers = torch.nn.Sequential(
```

```
            torch.nn.Linear(128, 128),
```

```
            torch.nn.ReLU(),
```

```
            torch.nn.Linear(128, 2),
```

```
        )
```

```
    def forward(self, input):
```

```
        output = self.layers(input)
```

```
        return output
```



# Use Modules in Modules

---

We recommend using `torch.nn.Module` objects within a `torch.nn.Module` instead of basic Python data structures when parameters are involved.

- This ensures that the parameters are properly **registered**, **tracked**, and **optimized** during training.



# Use Modules in Modules

e.g. `model.state_dict()` (return a state dictionary that contains all parameter names and tensors) source code:

```
def state_dict(self, *args, destination=None, prefix='', keep_vars=False):
    ...
    if destination is None:
        destination = OrderedDict()
        destination._metadata = OrderedDict()
    ...
    for name, module in self._modules.items():
        if module is not None:
            module.state_dict(destination=destination, prefix=prefix + name + '.', keep_vars=keep_vars)
    ...
    return destination
```

In most of the functions in a Module object (`train()`, `eval()`, `parameters()` ... ), the model recursively applies operations to all sub-modules. (just like a tree structure)

As a result, it is better to use `ModuleList`, which is a `torch.nn.Module` object.



# Use Modules in Modules

## ModuleList version:

```
class myModel(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.linear_list = torch.nn.ModuleList([
            torch.nn.Linear(128, 128) for _ in range(3)
        ])
    def forward(self, input):
        outputs = ()
        for i in range(3):
            outputs += (self.linear_list[i](input), )
        return outputs
```



print(model.\_modules) :

```
OrderedDict([('linear_list',
  ModuleList(
    (0-2): 3 x Linear(in_features=128, out_features=128, bias=True)
  )
)])
```

## List version:

```
class myModel(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.linear_list = [
            torch.nn.Linear(128, 128) for _ in range(3)
        ]
    def forward(self, input):
        outputs = ()
        for i in range(3):
            outputs += (self.linear_list[i](input), )
        return outputs
```



print(model.\_modules) :

```
OrderedDict()
```



# train() and Eval()

---

The member functions `model.train()` and `model.eval()` are for mode transfer. (recursively)

## Training mode:

- The training mode in PyTorch activates certain layers and behaviors that are only needed during training such as **Dropout and Batch Normalization**.
- In this mode, the model adjusts its parameters through backpropagation and updates the gradients during training.

## Evaluation mode:

- The evaluation mode is used for evaluation and inference, where **layers like Dropout are disabled and Batch Normalization uses fixed statistics instead of updating them**.
- This ensures that the model behaves consistently and produces deterministic outputs during testing or inference without modifying its internal state.



# Training

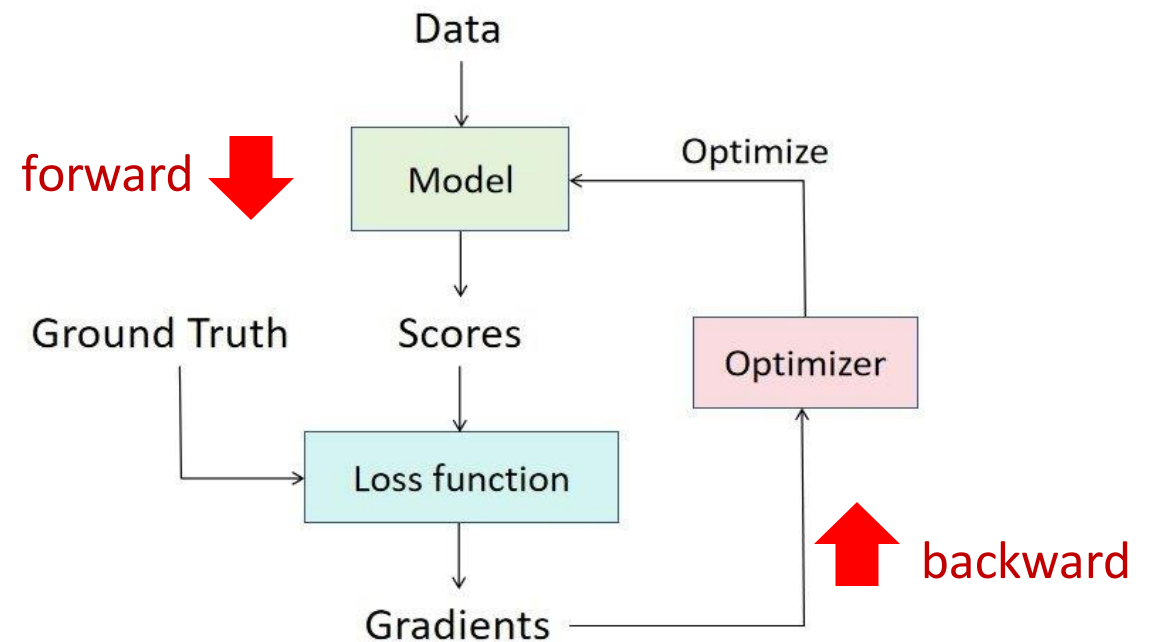
---

Forward: Obtain output logits

`model.forward()`

Backward: Back-propagation

`loss.backward()`



# Backward

---

The backwardfunction in PyTorch is used to perform **backpropagation**.

**Gradient Calculation:** PyTorch computes the gradient of the target tensor with respect to all the tensors that have `requires_grad=True`, using the chain rule of calculus.

**Gradient Storage:** The computed gradients are stored in the `.grad` attribute of the tensors.

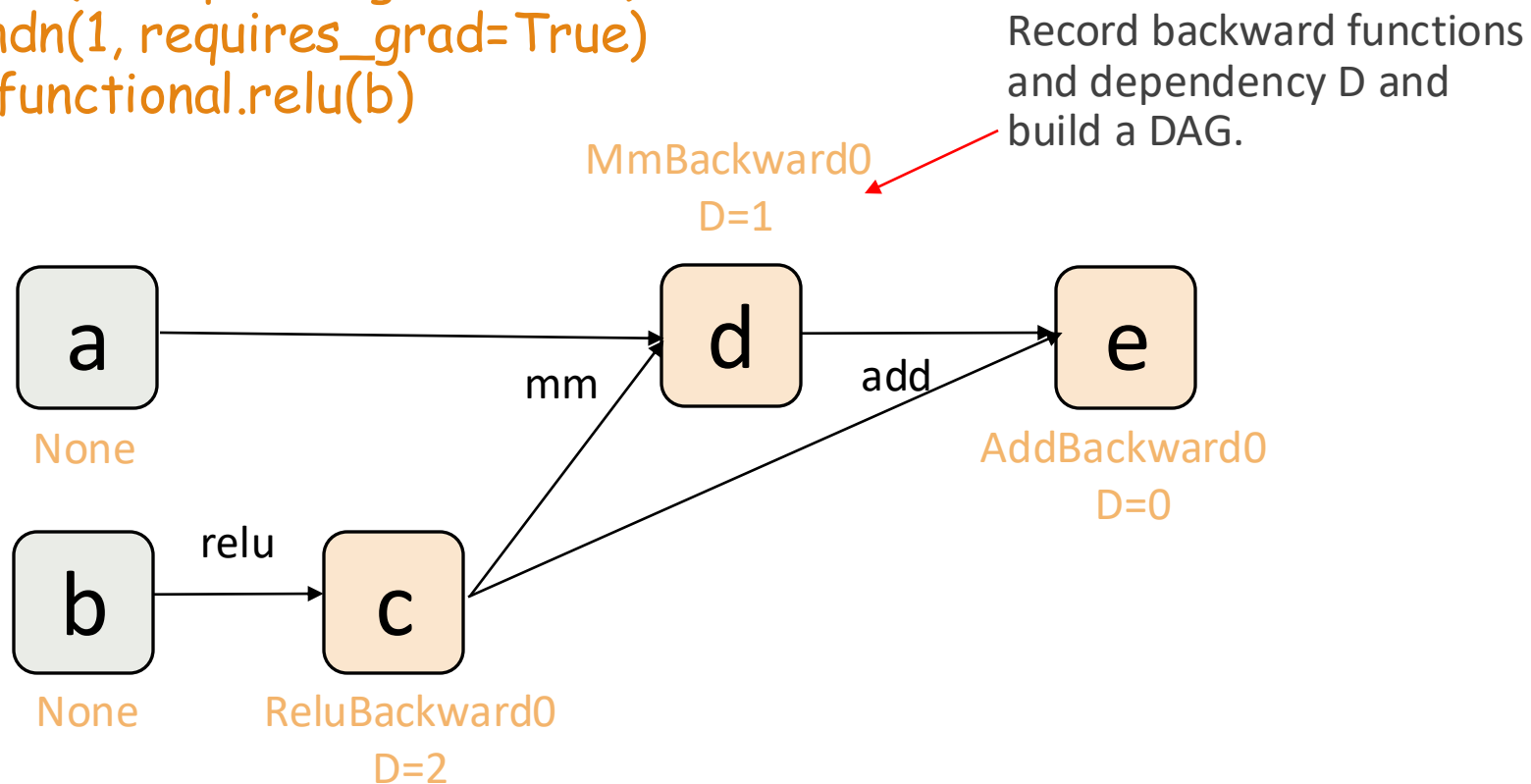
**Automatic Differentiation:** PyTorch builds a computation graph dynamically during the forward pass, and when `backward()` is called, it traverses this graph in reverse order, applying the chain rule to calculate gradients step-by-step from the output back to the inputs.



# A simple example

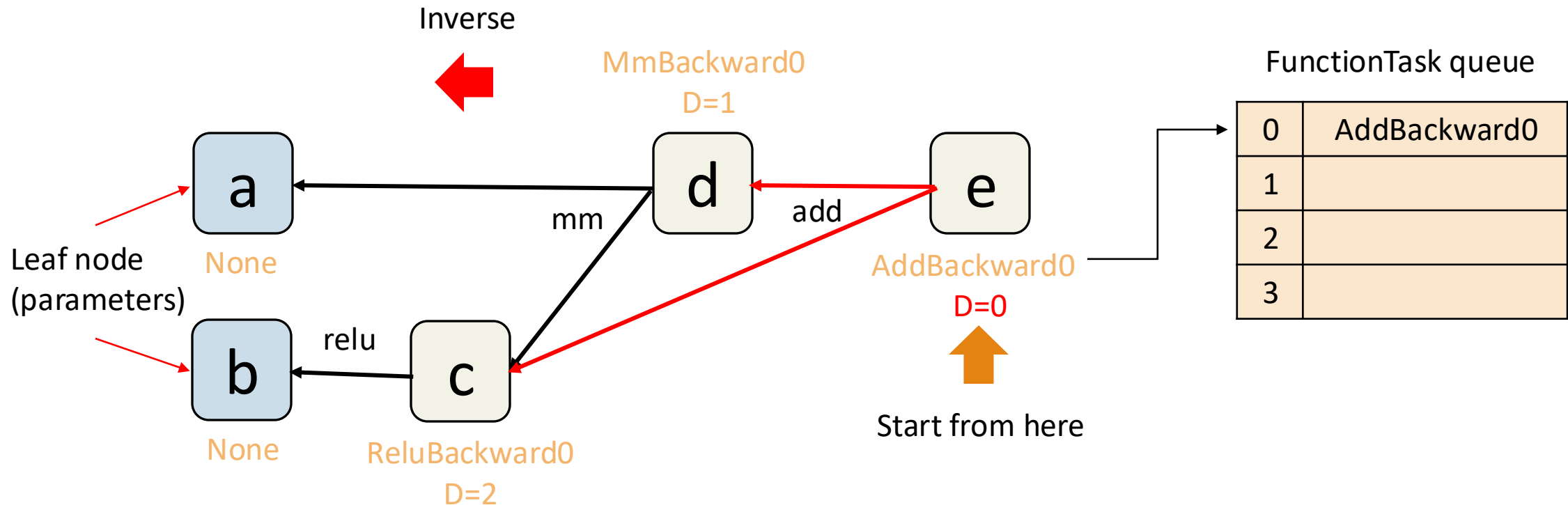
Sample code (forward) :

```
a = torch.randn(1, requires_grad=True)
b = torch.randn(1, requires_grad=True)
c = torch.nn.functional.relu(b)
d = a @ c
e = c + d
```



# A simple example

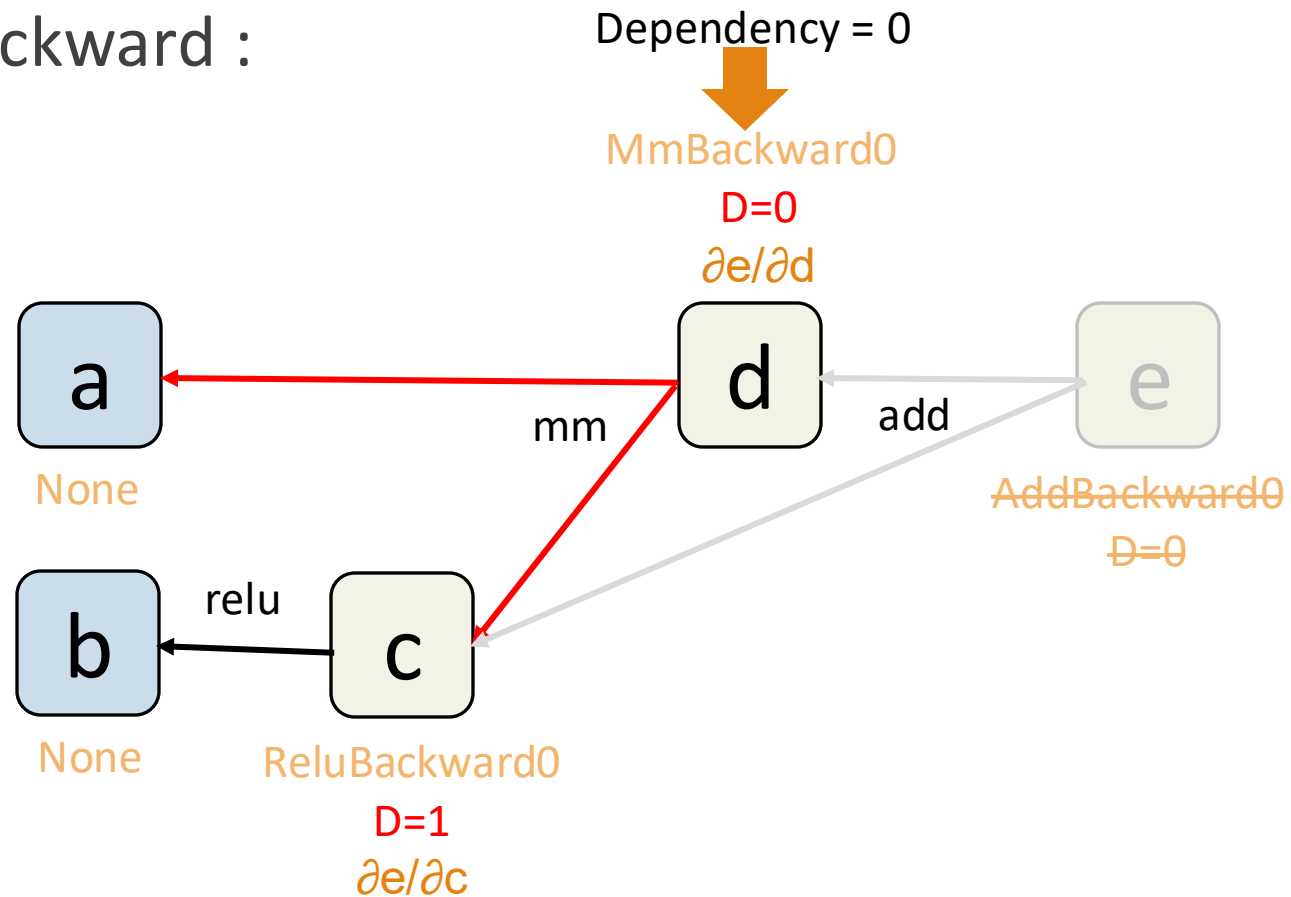
Backward :





# A simple example

Backward :



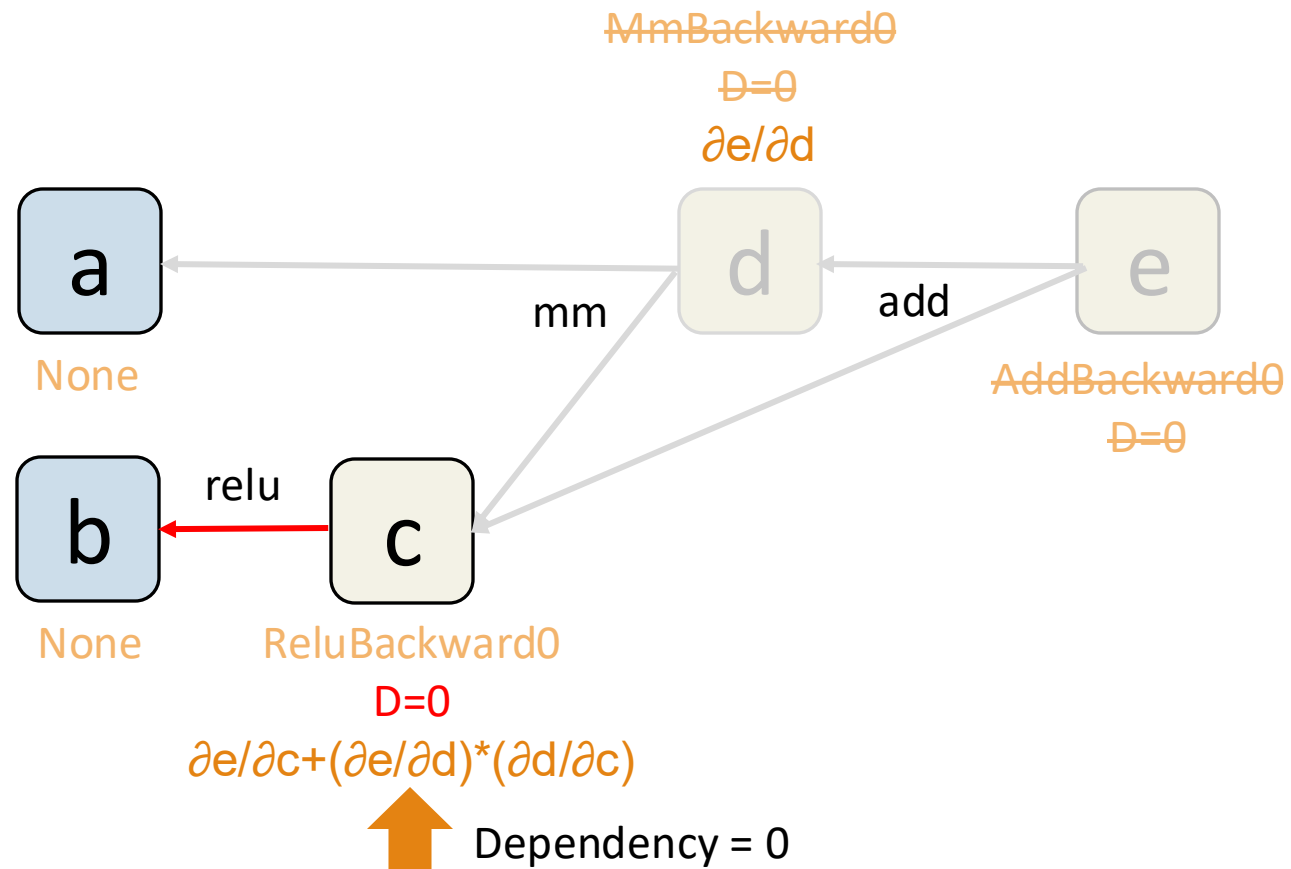
FunctionTask queue

|   |             |
|---|-------------|
| 0 | MmBackward0 |
| 1 |             |
| 2 |             |
| 3 |             |

Pop AddBackward0 and execute.  
Push MmBackward0

# A simple example

Backward :



FunctionTask queue

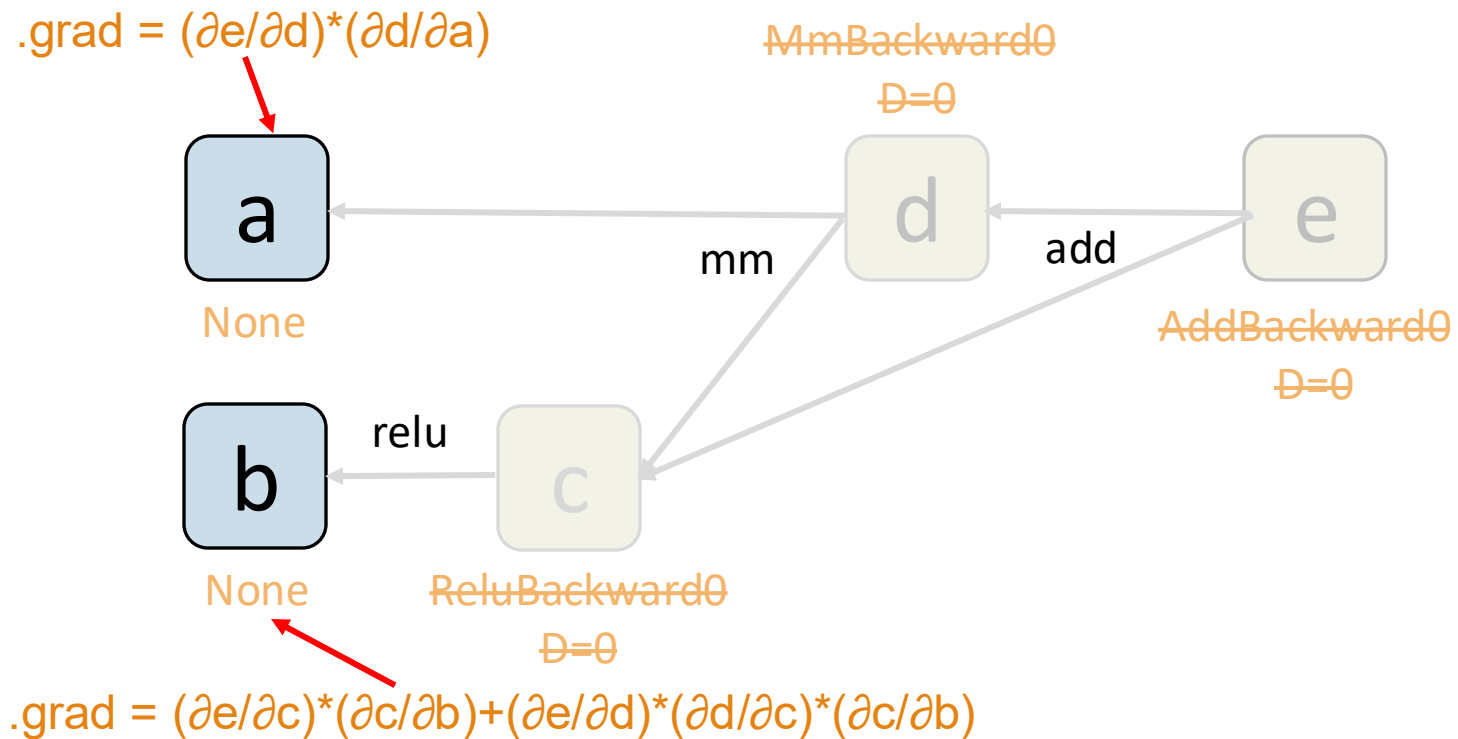
|   |               |
|---|---------------|
| 0 | ReluBackward0 |
| 1 |               |
| 2 |               |
| 3 |               |

Pop MmBackward0 and execute.  
Push ReluBackward0



# A simple example

Backward :



FunctionTask queue

|   |  |
|---|--|
| 0 |  |
| 1 |  |
| 2 |  |
| 3 |  |

Pop ReluBackward0

# Run without gradient

---

## Three ways to avoid gradient

### `tensor.requires_grad = False`

- When you set the `requires_grad` attribute of a tensor to `False`, even if it participates in computations, gradients will not be tracked for it.

### `with torch.no_grad():`

- This is a context manager that temporarily disables gradient tracking for all operations within its scope.

### `tensor.detach()`

- The `detach()` method **returns a new tensor** that shares the same data as the original tensor but is detached from the computation graph.

New view, Unchanged continuity



# Train a Model

---

Step1: Prepare the dataset

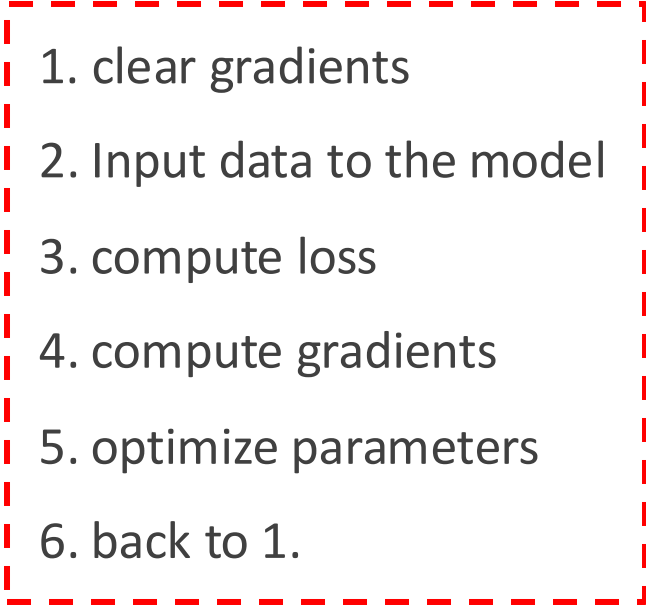
Step2: Construct the model

Step3: Define Optimizer

Step4: Define loss function

Step5: Train the model

Step6: Evaluate the model

- 
1. clear gradients
  2. Input data to the model
  3. compute loss
  4. compute gradients
  5. optimize parameters
  6. back to 1.

# Loss functions

---

A loss function is a critical component that **measures how well or poorly a model's predictions match the actual (true) values.**

In simple terms, it quantifies the "error" or "difference" between the predicted output and the true labels. The primary role of a loss function is to provide feedback to the model during the training process, helping the model improve its predictions over time.



# Loss functions

---

Some loss functions in torch.nn

| Loss functions                         | Usage                       |
|--|-----------------------------|
| <code>torch.nn.CrossEntropyLoss</code> | Classification tasks        |
| <code>torch.nn.MSELoss</code>          | Regression tasks            |
| <code>torch.nn.KLDivLoss</code>        | Knowledge Distillation      |
| <code>torch.nn.BCELoss</code>          | Binary classification tasks |

Some other loss functions

| Loss functions   | Usage   |
|------------------|---|
| Contrastive Loss | Contrastive learning  |
| Entropy          | A special term in loss function to improve model's confidence |



# Optimizers

---

Some optimizers (most commonly used) in torch.nn

| Optimizer                      | Description   |
|--------------------------------|---|
| <code>torch.optim.SGD</code>   | Stochastic gradient descend                                 |
| <code>torch.optim.Adam</code>  | Adam optimizer  |
| <code>torch.optim.AdamW</code> | A variant of the Adam optimizer on weight decay formulation |



# Training

---

1. clear gradients

`optimizer.zero_grad()`

2. Input data to the model

`output = model(**batch)`

3. compute loss

`loss = loss_fn(output, ground_truth)`

4. compute gradients

`loss.backward()`

5. optimize parameters

`optimizer.step()`

6. back to 1.

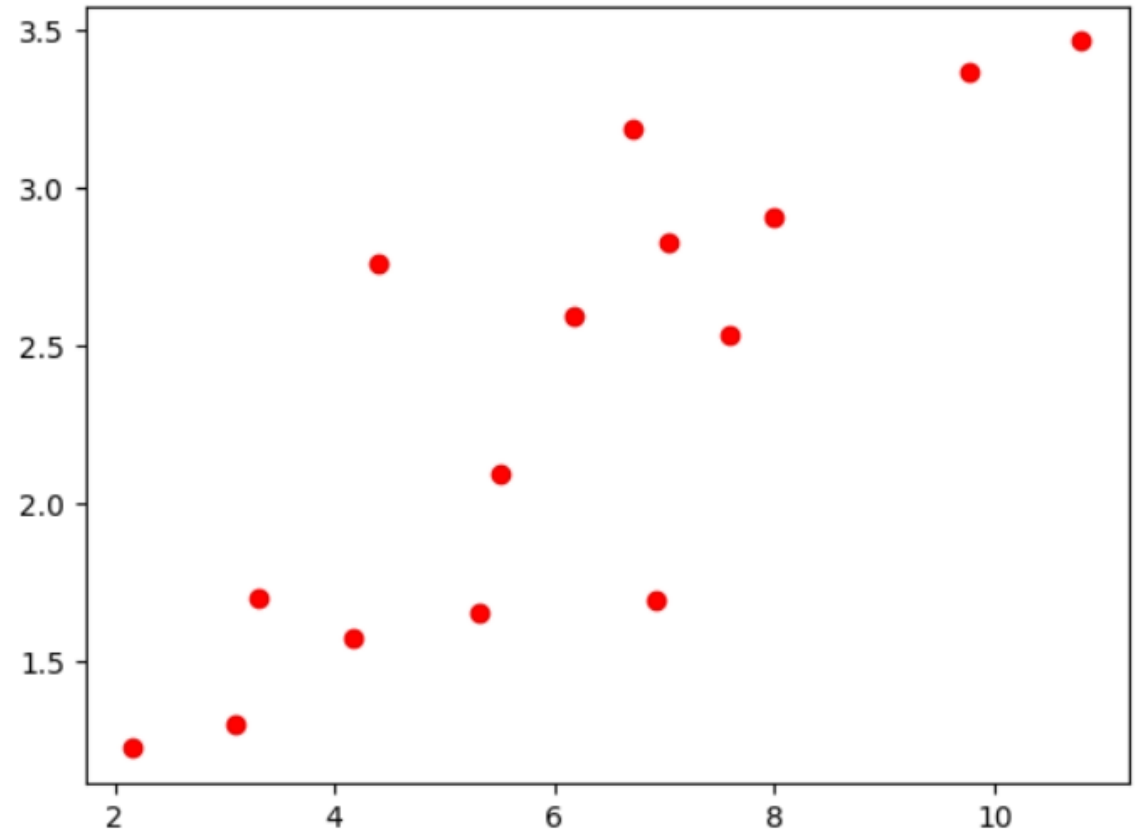


# An Example of Linear Regression

---

Data distribution:

Use a line to represent these data



# An Example of Linear Regression

```
# Toy dataset
x_train = torch.tensor([[3.3], [4.4], [5.5], [6.71], [6.93], [4.168],
                        [9.779], [6.182], [7.59], [2.167], [7.042],
                        [10.791], [5.313], [7.997], [3.1]], dtype=torch.float32)

y_train = torch.tensor([[1.7], [2.76], [2.09], [3.19], [1.694], [1.573],
                        [3.366], [2.596], [2.53], [1.221], [2.827],
                        [3.465], [1.65], [2.904], [1.3]], dtype=torch.float32)

# Linear regression model
model = nn.Linear(input_size, output_size)

# Loss and optimizer
criterion = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

# Train the model
for epoch in range(num_epochs):

    # Forward pass
    outputs = model(x_train)
    loss = criterion(outputs, y_train)

    # Backward and optimize
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (epoch+1) % 20 == 0:
        print ('Epoch [{}/{}], Loss: {:.4f}'.format(epoch+1, num_epochs, loss.item()))
```

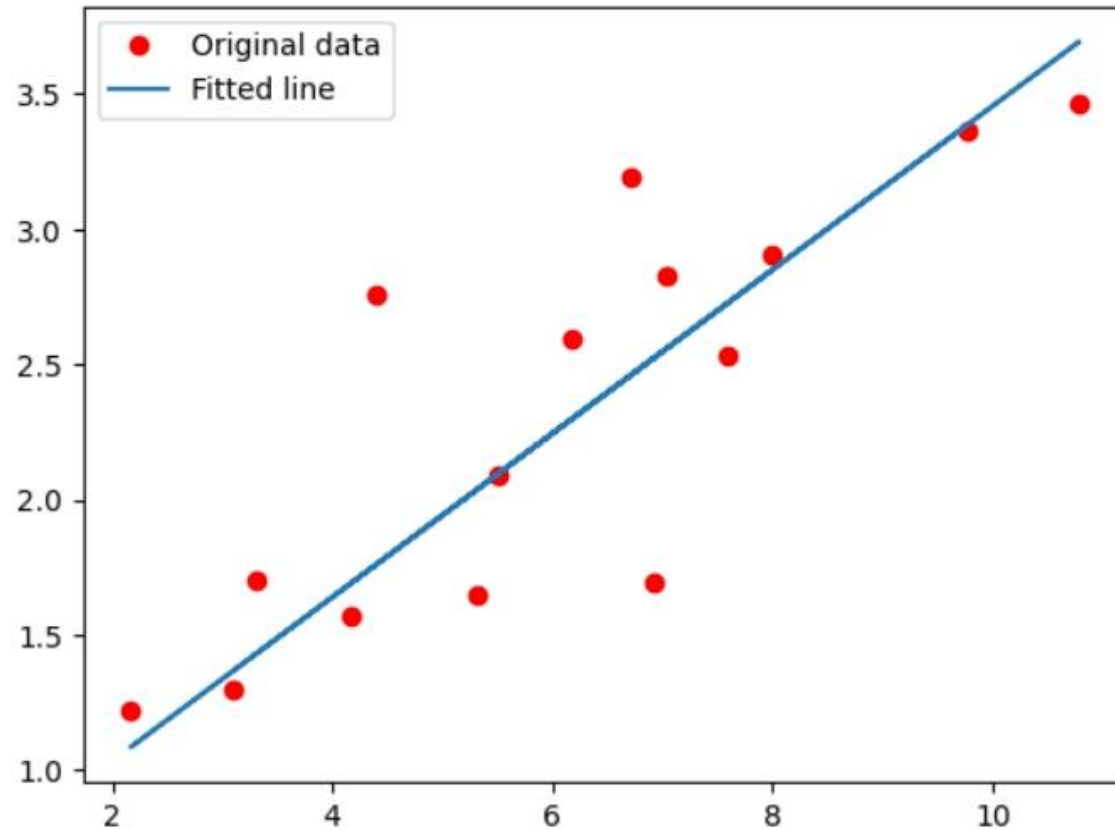
initialize  
data tensor  
model  
loss & optimizer  
load data  
1. Input data to the model  
2. compute loss  
3. clear gradients  
4. compute gradients  
5. optimize parameters



# An Example of Linear Regression

## Outputs:

```
Epoch [5/60], Loss: 11.2489  
Epoch [10/60], Loss: 4.6657  
Epoch [15/60], Loss: 1.9987  
Epoch [20/60], Loss: 0.9182  
Epoch [25/60], Loss: 0.4805  
Epoch [30/60], Loss: 0.3031  
Epoch [35/60], Loss: 0.2313  
Epoch [40/60], Loss: 0.2021  
Epoch [45/60], Loss: 0.1903  
Epoch [50/60], Loss: 0.1855  
Epoch [55/60], Loss: 0.1835  
Epoch [60/60], Loss: 0.1827
```



# Check the parameters & gradients

---

## step 1

```
optimizer.zero_grad()  
print_grads(model)
```

weight: Parameter containing:  
tensor([[0.4165]], requires\_grad=True)  
weight grad: None

bias: Parameter containing:  
tensor([0.4819], requires\_grad=True)  
bias grad: None

## step 2

```
loss.backward()  
print_grads(model)
```

weight: Parameter containing:  
tensor([[0.4165]], requires\_grad=True)  
weight grad: tensor([[10.0239]])

bias: Parameter containing:  
tensor([0.4819], requires\_grad=True)  
bias grad: tensor([1.3666])



# Check the gradients

## step 2

```
loss.backward()  
print_grads(model)
```

weight: Parameter containing:  
tensor([[0.4165]], requires\_grad=True)  
weight grad: tensor([[10.0239]])

bias: Parameter containing:  
tensor([0.4819], requires\_grad=True)  
bias grad: tensor([1.3666])

## step 3

```
optimizer.step()  
print_grads(model)
```

weight: Parameter containing:  
tensor([[0.4065]], requires\_grad=True)  
weight grad: tensor([[10.0239]])

bias: Parameter containing:  
tensor([0.4805], requires\_grad=True)  
bias grad: tensor([1.3666])

## step 4

```
optimizer.zero_grad()  
print_grads(model)
```

weight: Parameter containing:  
tensor([[0.4065]], requires\_grad=True)  
weight grad: None

bias: Parameter containing:  
tensor([0.4805], requires\_grad=True)  
bias grad: None



# Deep learning in NLP tasks

---

I mentioned regression tasks on a polynomial function with a few data samples.

However, in NLP tasks, the model size and dataset size **can reach hundreds of gigabytes or even terabytes**, making it impractical to plot a hyper-curve for loss progression over the entire dataset.

As a result, we need to sample batches of data that can approximate the distribution of the original dataset.

# Dataset & DataLoader

---

The Dataset object should be defined as:

```
From torch.utils.data import DataLoader, Dataset
```

```
Class myDataset(Dataset):  
    def __init__(self, split):  
        Super().__init__()  
        self.data = ...  
  
    def __getitem__(self, index):  
        return self.data[index]  
  
    def __len__(self):  
        return len(self.data)
```

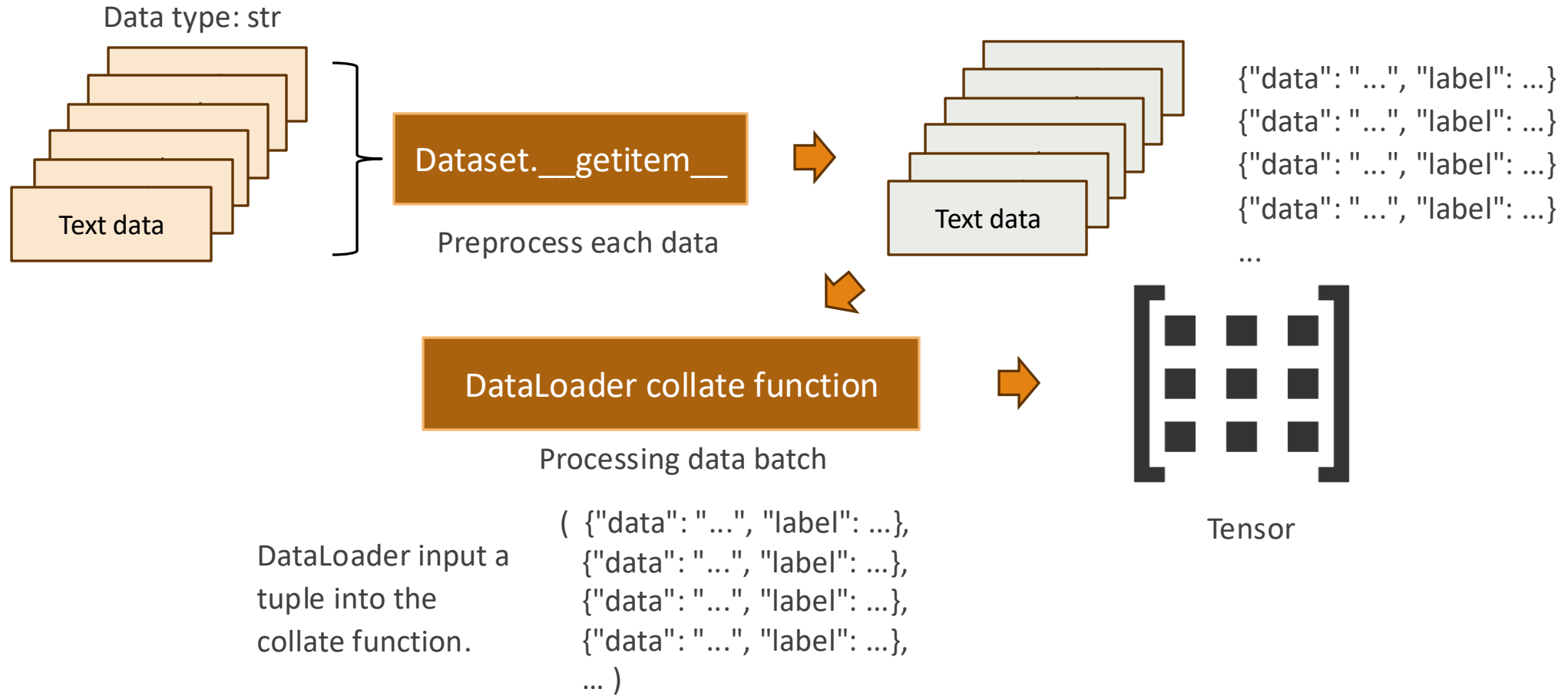
The Dataloader:

```
def get_batch(sample):  
    ...  
  
dl = DataLoader(myDataset(split=..., ), batch_size=...,  
                collate_fn=get_batch, shuffle=...)  
  
for batch in dl:  
    ...
```



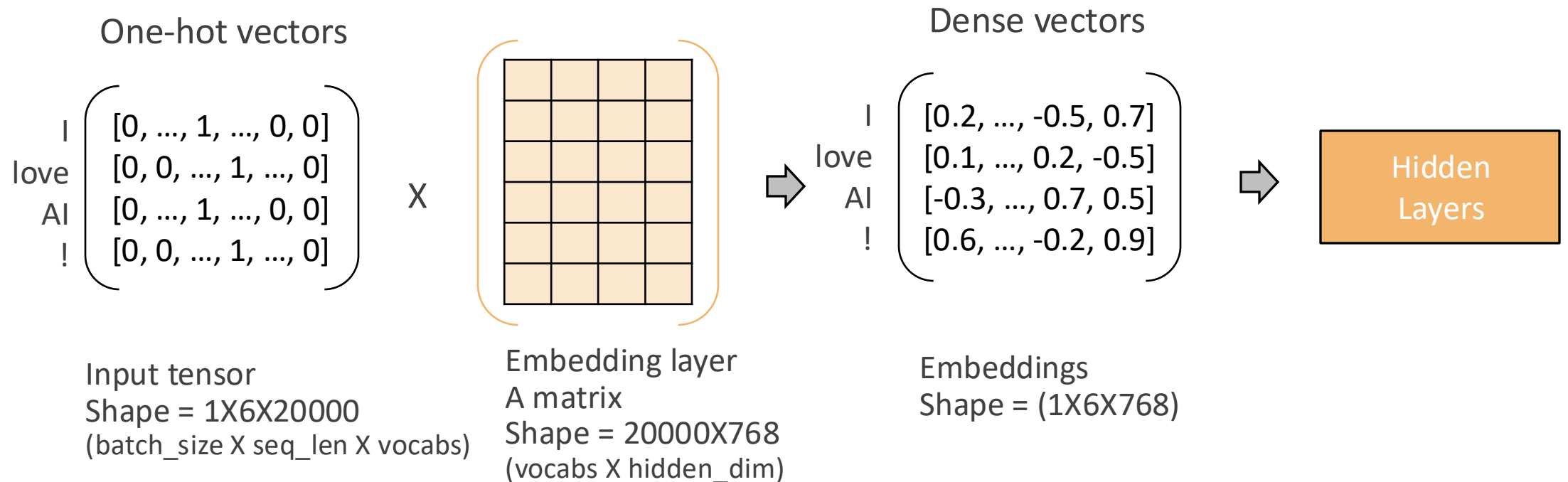


# Data Flow

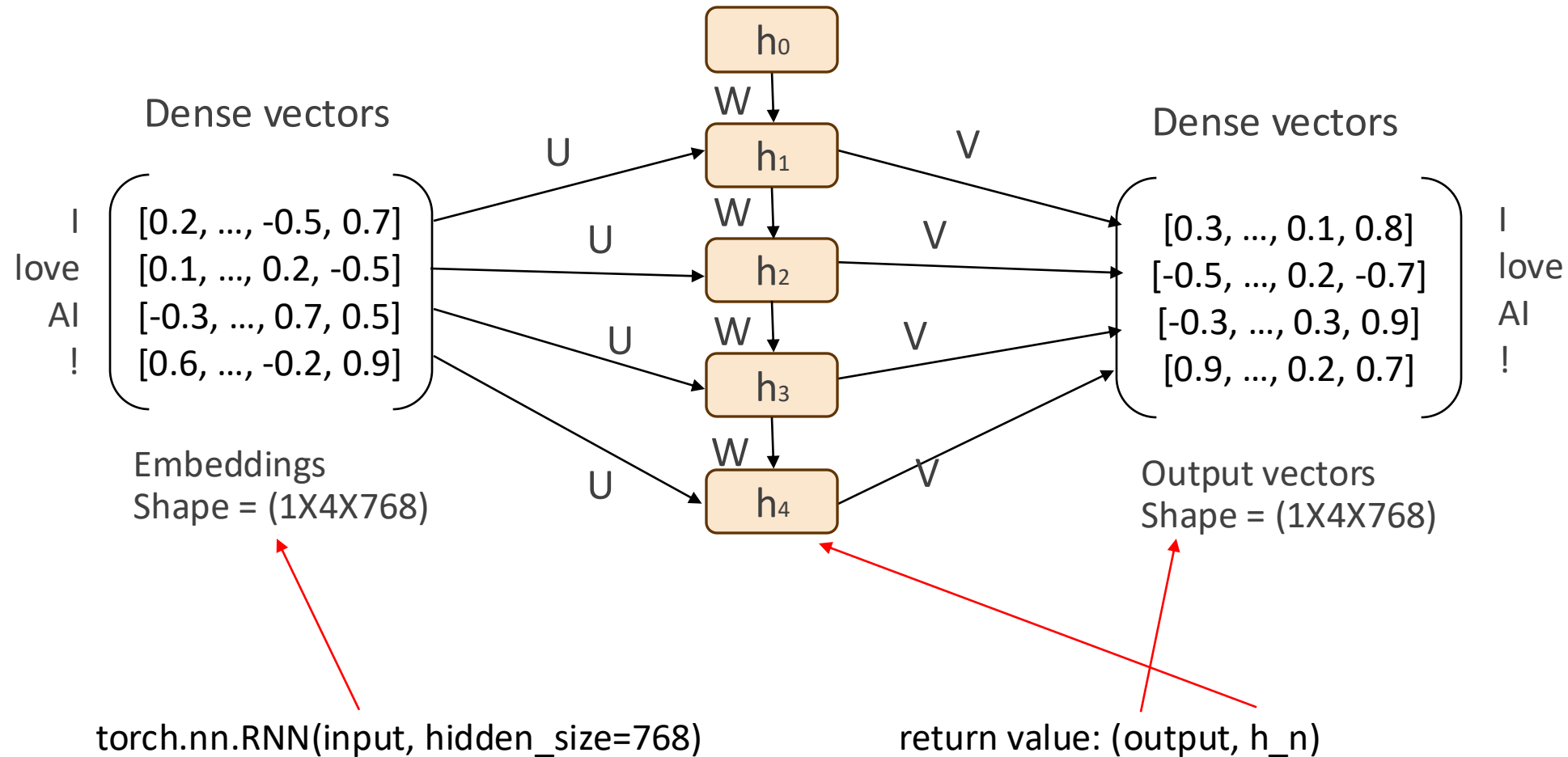


# Data Flow in RNN model

We assume that batch\_size = 1, hidden\_dim=768 and there are 20000 words in the dictionary



# Data Flow in RNN model

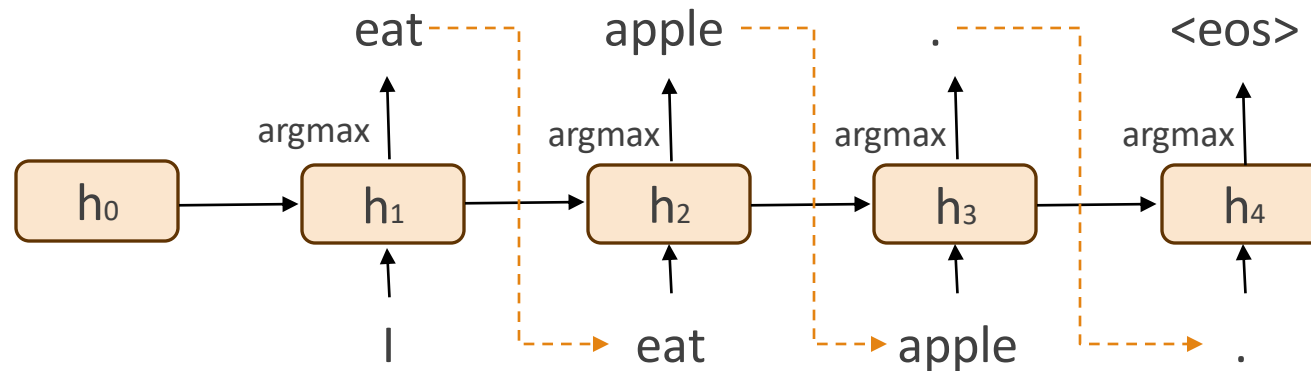


# Generative learning VS Teacher forcing

For example.

Our ground truth is: I love AI !

In generative training:



We optimize:

- $P(\text{love} \mid \text{I})$
- $P(\text{AI} \mid \text{I eat})$
- $P(! \mid \text{I eat apple})$
- $P(\text{<eos>} \mid \text{I eat apple !})$

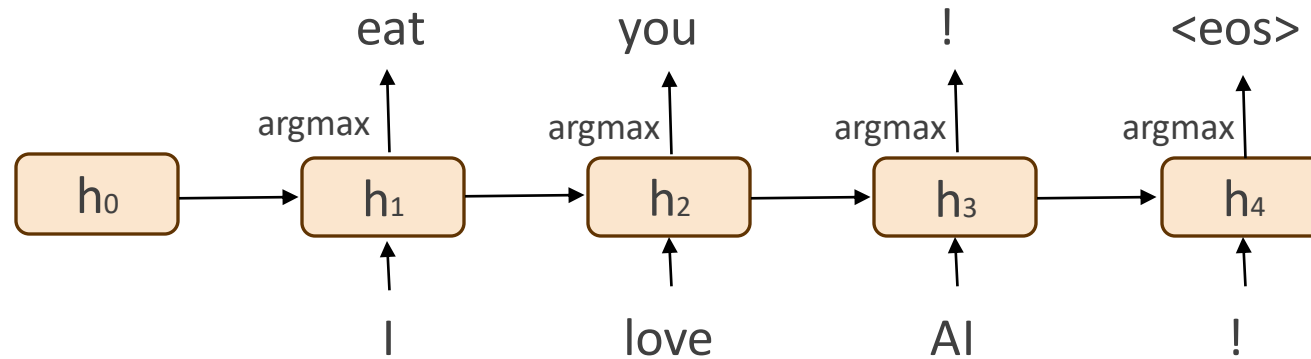
If the model makes a mistake early on, that error will be used in later training, which makes training unstable.



# Generative learning VS Teacher forcing

Our ground truth is still: I love AI !

In teacher forcing:



We optimize:

- $P(\text{love} \mid \text{I})$
- $P(\text{AI} \mid \text{I love})$
- $P(! \mid \text{I love AI})$
- $P(\text{<eos>} \mid \text{I love AI !})$

Ground truths

Nothing weird here.

Teacher forcing is more stable in training

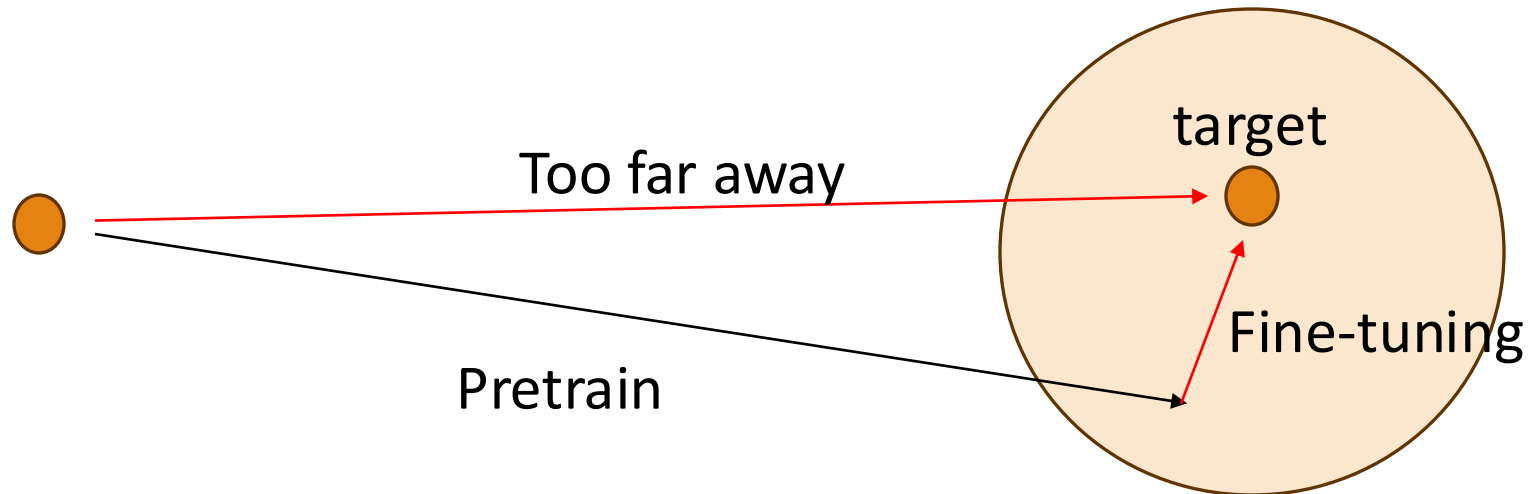
# Transformer-based models

---

Transformer-based models are more powerful and parallelable than RNN-based models.

Language models (LM) need pretraining to learn the underlying structure, patterns, and relationships in a language.

[Huggingface](#) is an excellent platform for accessing and downloading pretrained model weights and datasets.



# Load BERT from Huggingface

---

Import transformers package

```
import transformers as T
```

Model name

A local path to  
store the model

```
tokenizer = T.AutoTokenizer.from_pretrained("google-bert/bert-base-uncased", cache_dir="./cache/")
```



Tokenizer is used to transform tokens to ids, and it can also pack the text batch into tensors:

```
data = tokenizer.batch_encode_plus(text, padding=True, truncation=True, return_tensors="pt")
```

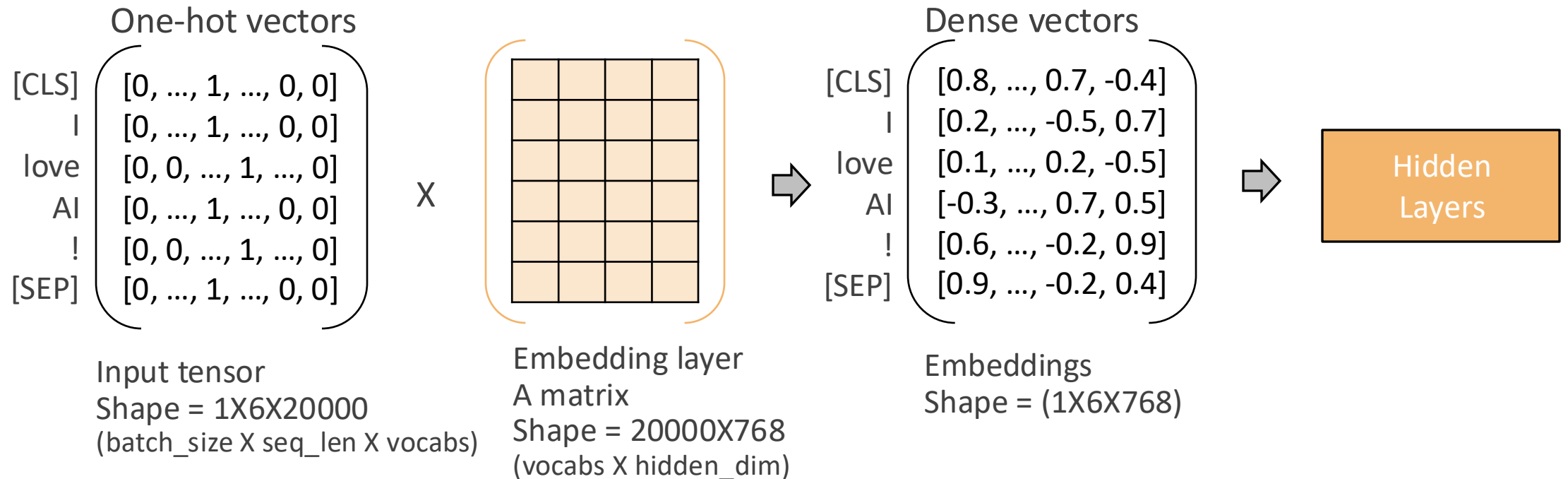
batch\_encode\_plus returns a "BatchEncoding" object which can be regarded as dict.  
(located on CPU ram)

The BERT model can be loaded by:

```
model = T.AutoModel.from_pretrained("google-bert/bert-base-uncased", cache_dir="./cache/")
```

# Data Flow in Transformer Auto-encoder

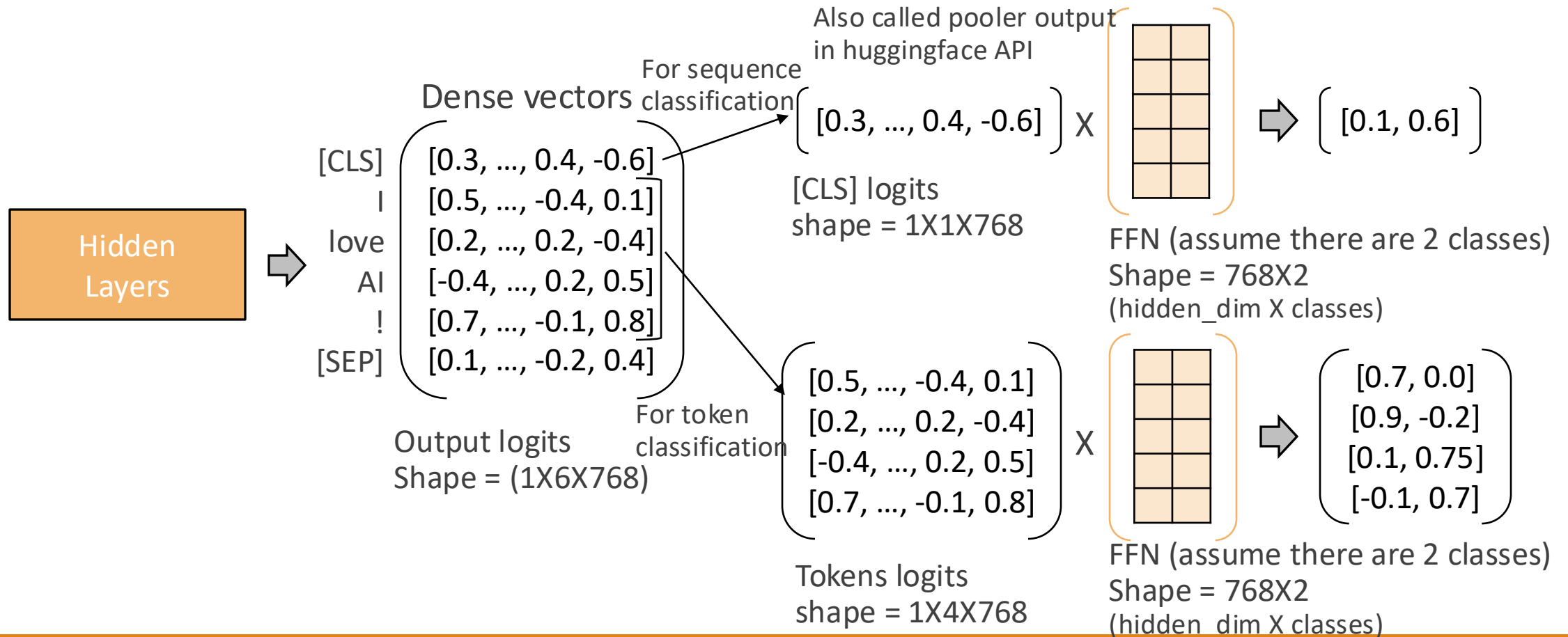
We assume that `batch_size = 1`, `hidden_dim=768` and there are 20000 words in the dictionary





# Data Flow in Transformer Auto-encoder

We assume that batch\_size = 1, hidden\_dim=768 and there are 20000 words in the dictionary



# Thank you for listening

---

大海為什麼是藍色的

因為海里的魚在吐泡泡，噗嚕噗嚕噗嚕

BlueBlueBlue

