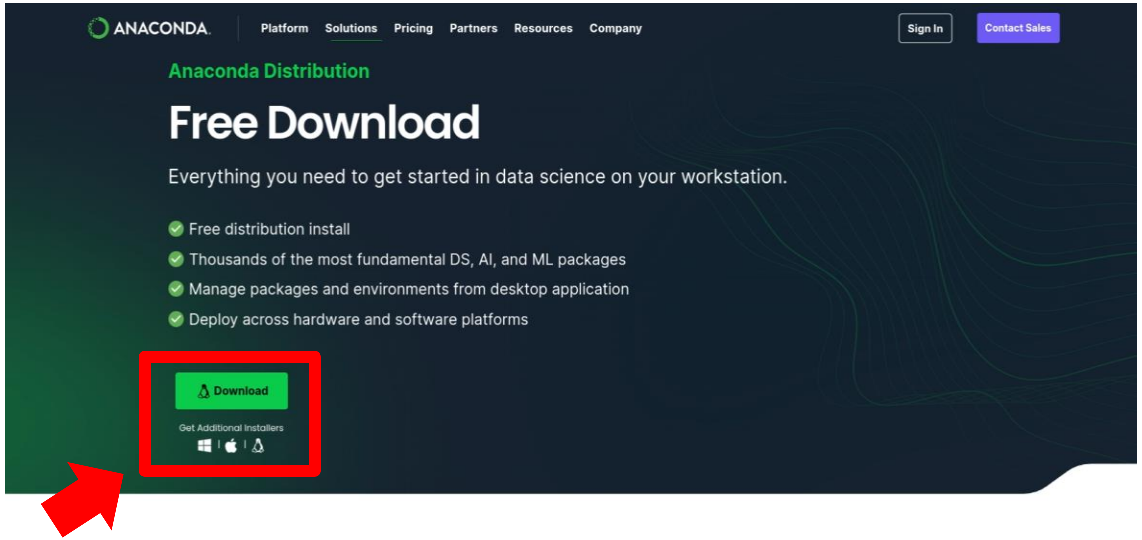
Generative Artificial Intelligence

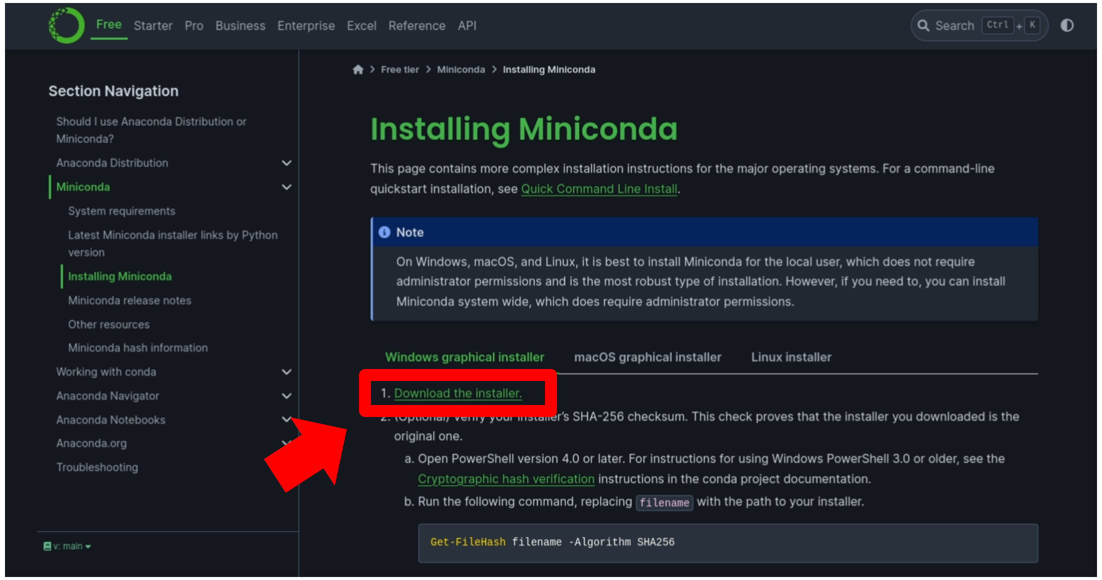
Pytorch tutorial

Environment setup

下載 anaconda (https://www.anaconda.com/download)

或者 miniconda (https://docs.anaconda.com/free/miniconda/miniconda-install/)

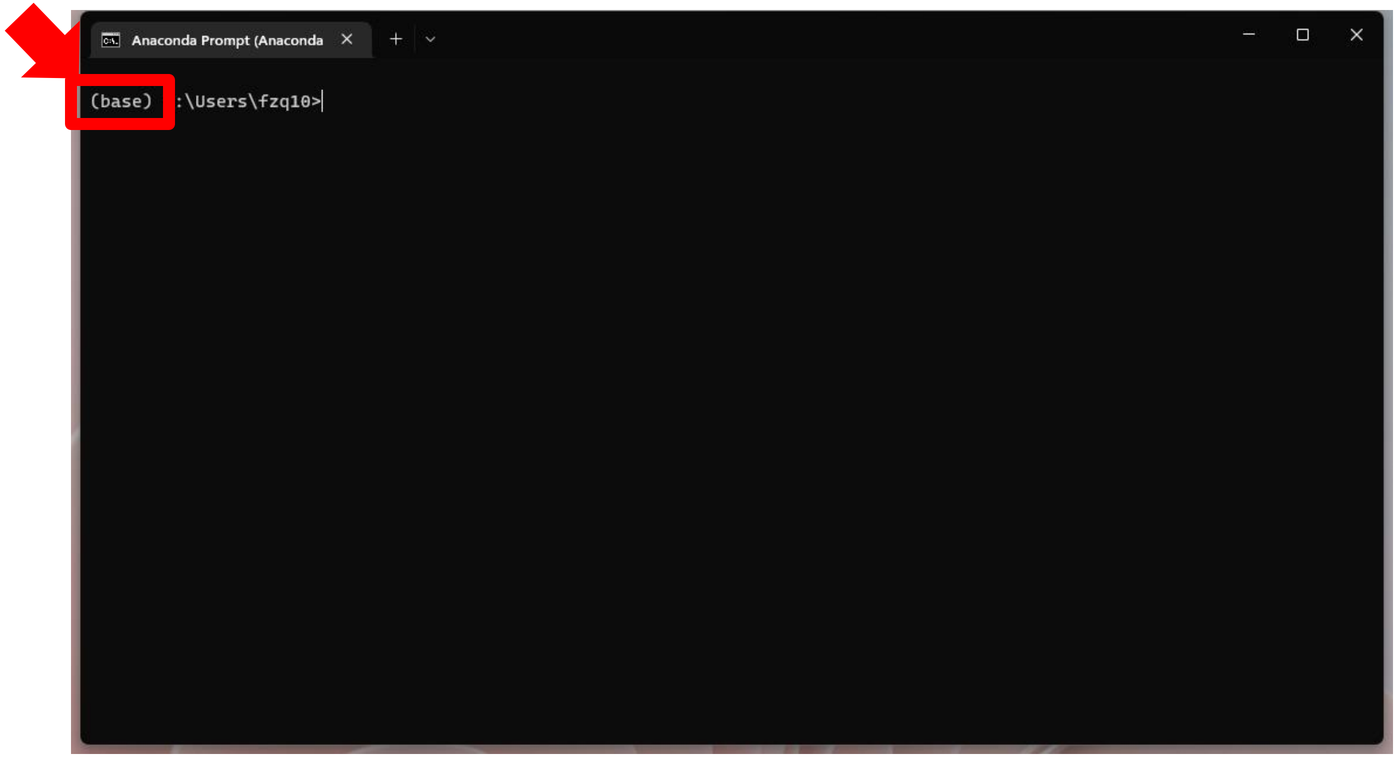




Environment setup

開啟Anaconda prompt

當前環境



Basic Commands

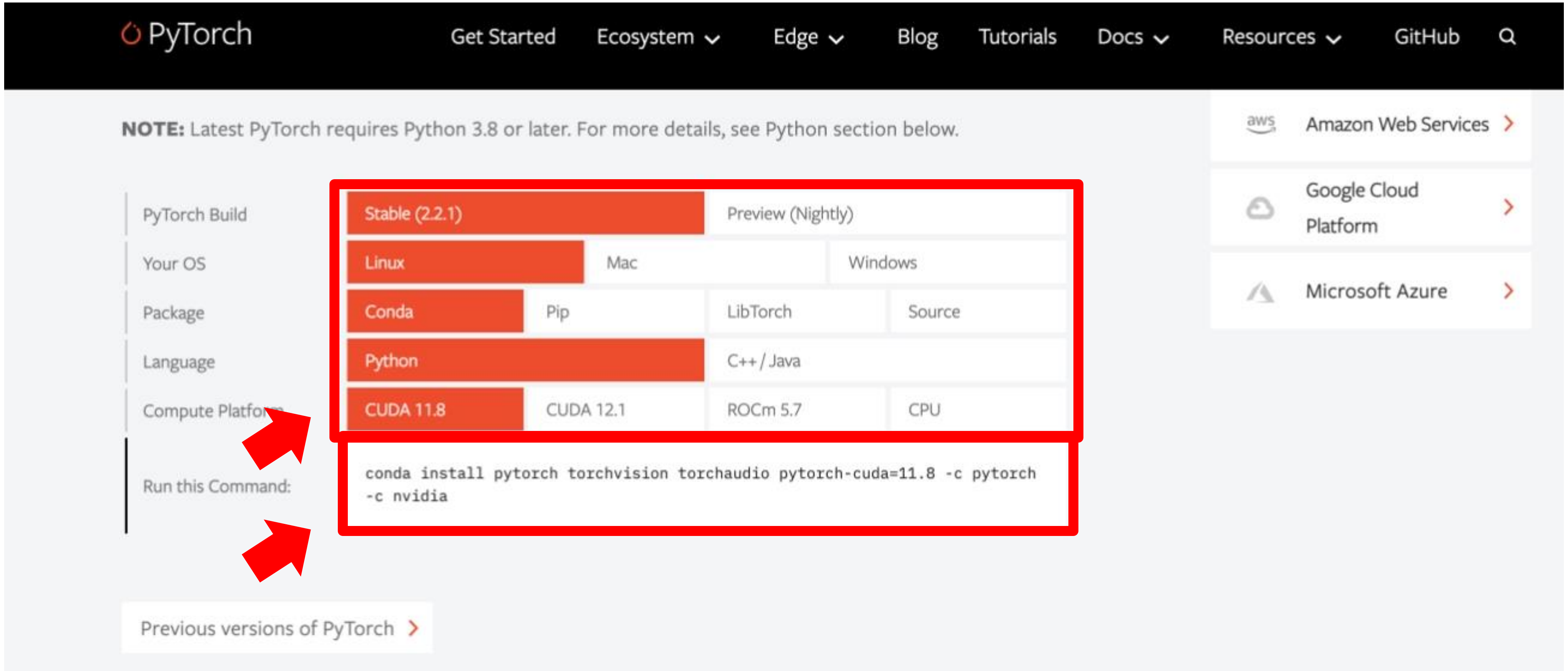
安裝完anaconda以後，開啟anaconda prompt，可以通過commands操作環境。

也可以直接通過圖形化介面navigator操作

1. 創建新的python環境：
   conda create –n (name) python=3.X.X
2. 激活python環境
   conda activate (name)
3. 退出環境
   conda deactivate
4. 查看所有環境
   conda env list

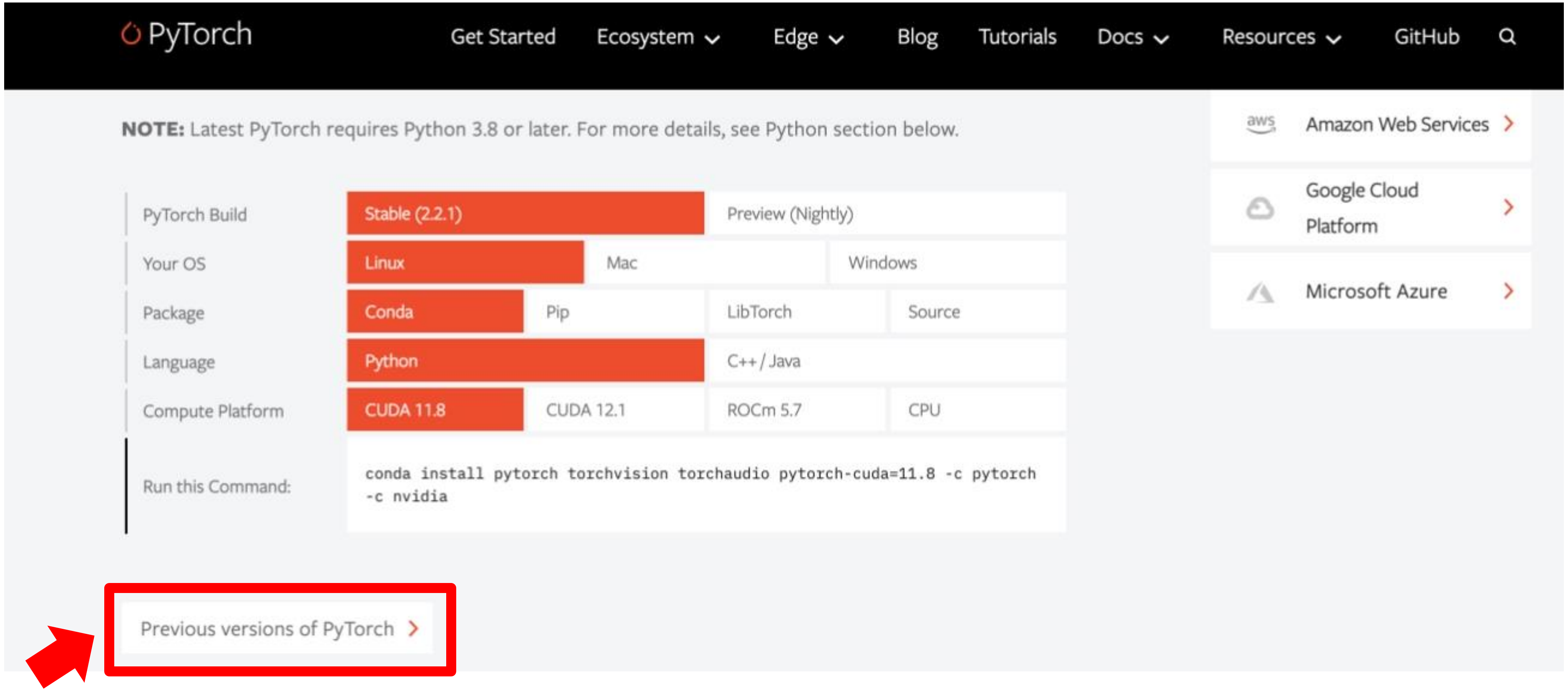
Install Pytorch

安裝pytorch: https://pytorch.org/



Install Pytorch

也可以選擇更早版本的，主要是cuda的版本電腦的硬體要能夠支持



Training

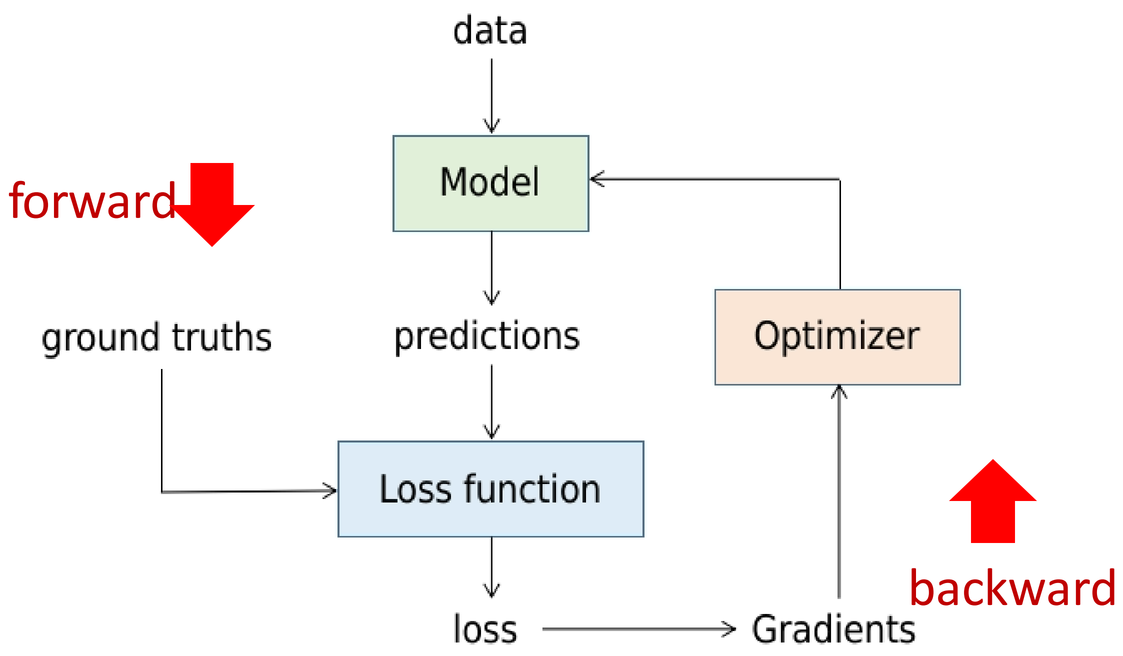
Forward: Obtain output logits

`model.forward()`

Backward: Back-propagation

Use Autograd

`loss.backward()`

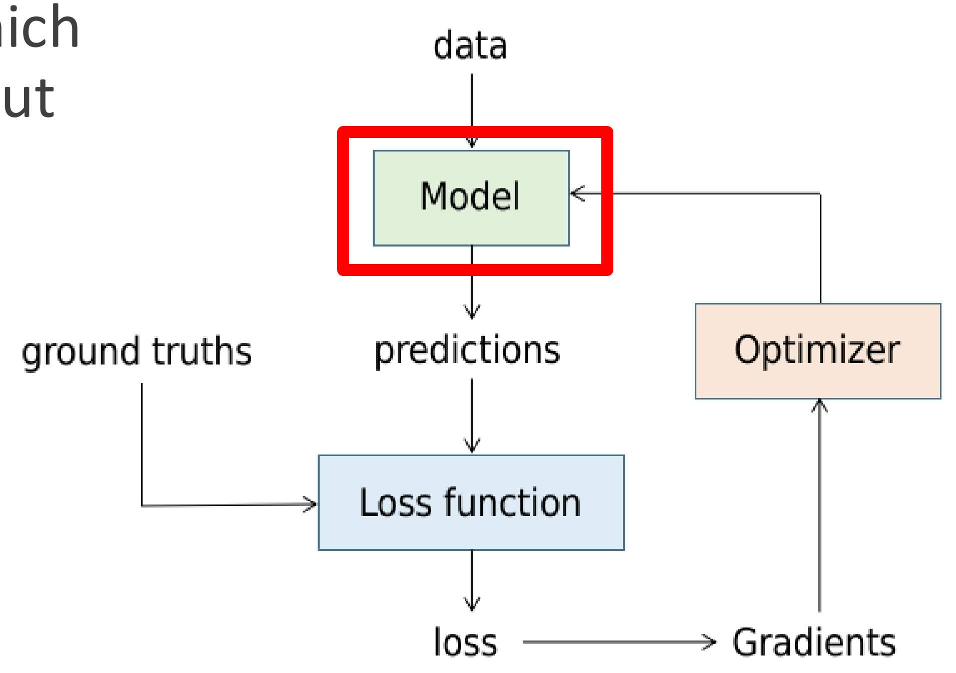


What is a Model

A model is like a special function which define the relationship between input and output.

```
def func(x):
    return 3*x+4
```

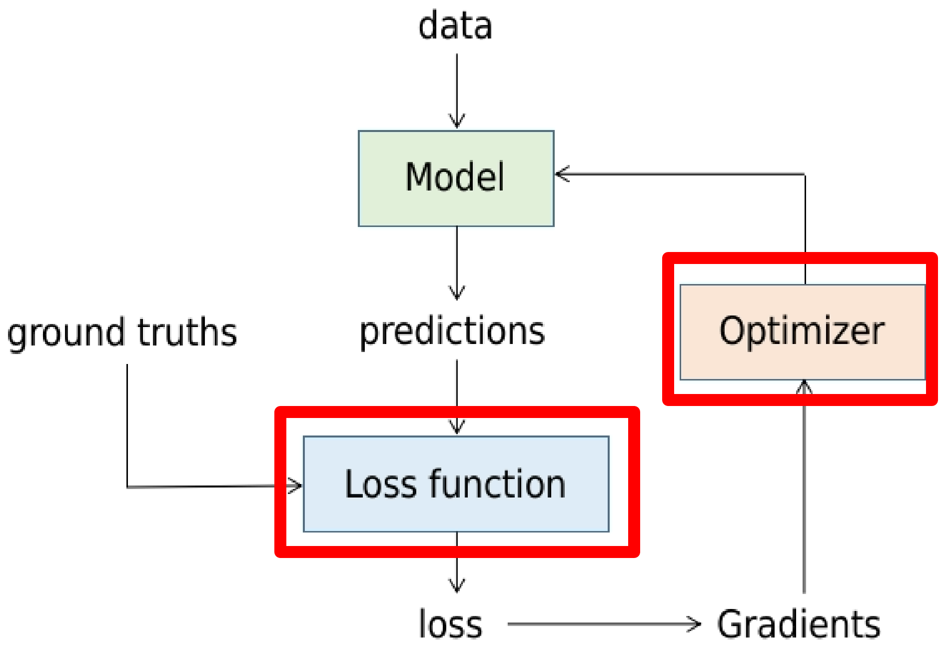
parameters



What is Loss function&Optimizer

Loss Function defines the distance between model's prediction & expected answer (ground truth).

Optimizer defines the way to optimize the parameters.



A simple example

We have a linear "model": $y = 2x+1$

Input a data pair: (1,1)

When x=1 we get: $y = 2*1+1 = 3$

Loss function: What is the difference between expected value 1 and model output 3? loss = 3-1=2 ?

Optimizer: How to fix the difference between expected value 1 and model output 3?

Forward & Back-propagation

model.forward()

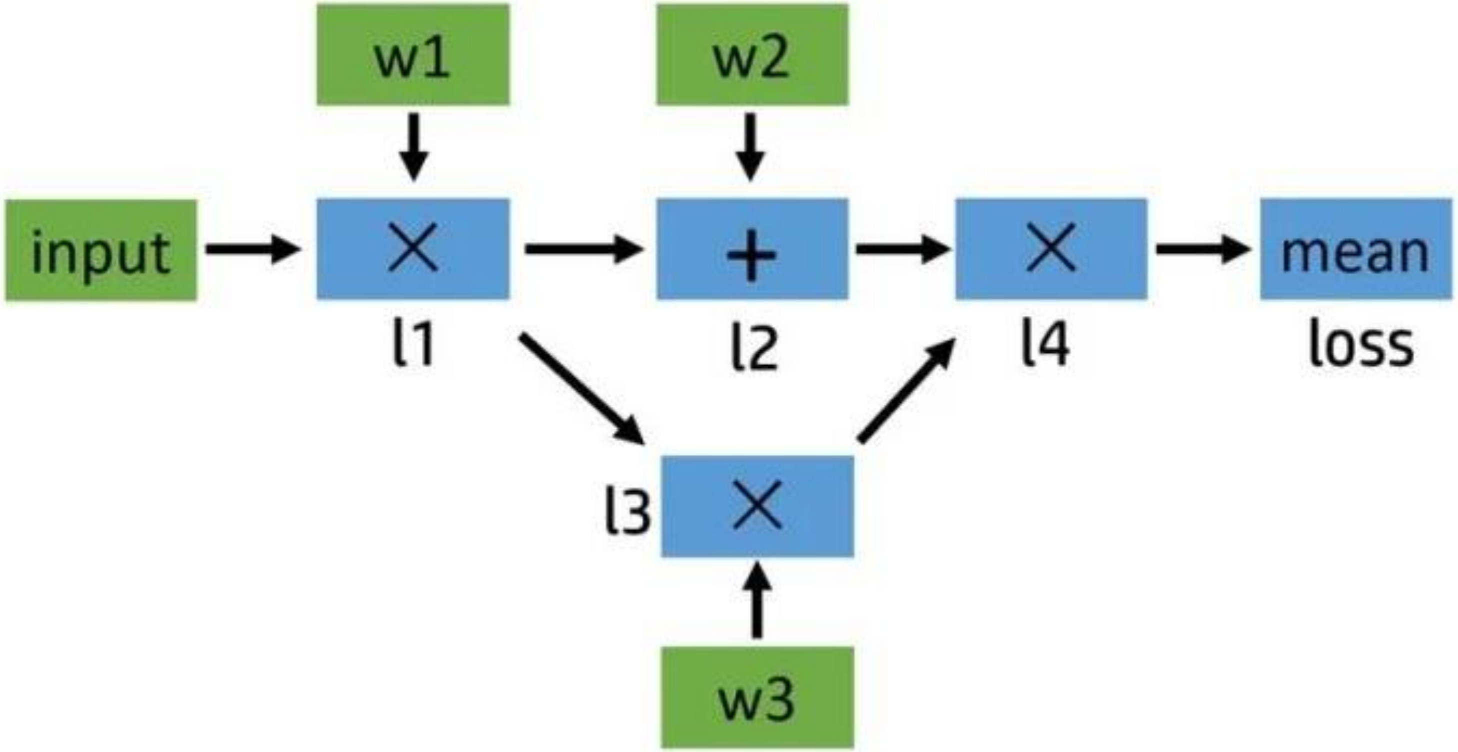
l1 = input x w1

l2 = l1 + w2

l3 = l1 x w3

l4 = l2 x l3

loss = mean(l4)



Forward & Back-propagation

loss.backward()

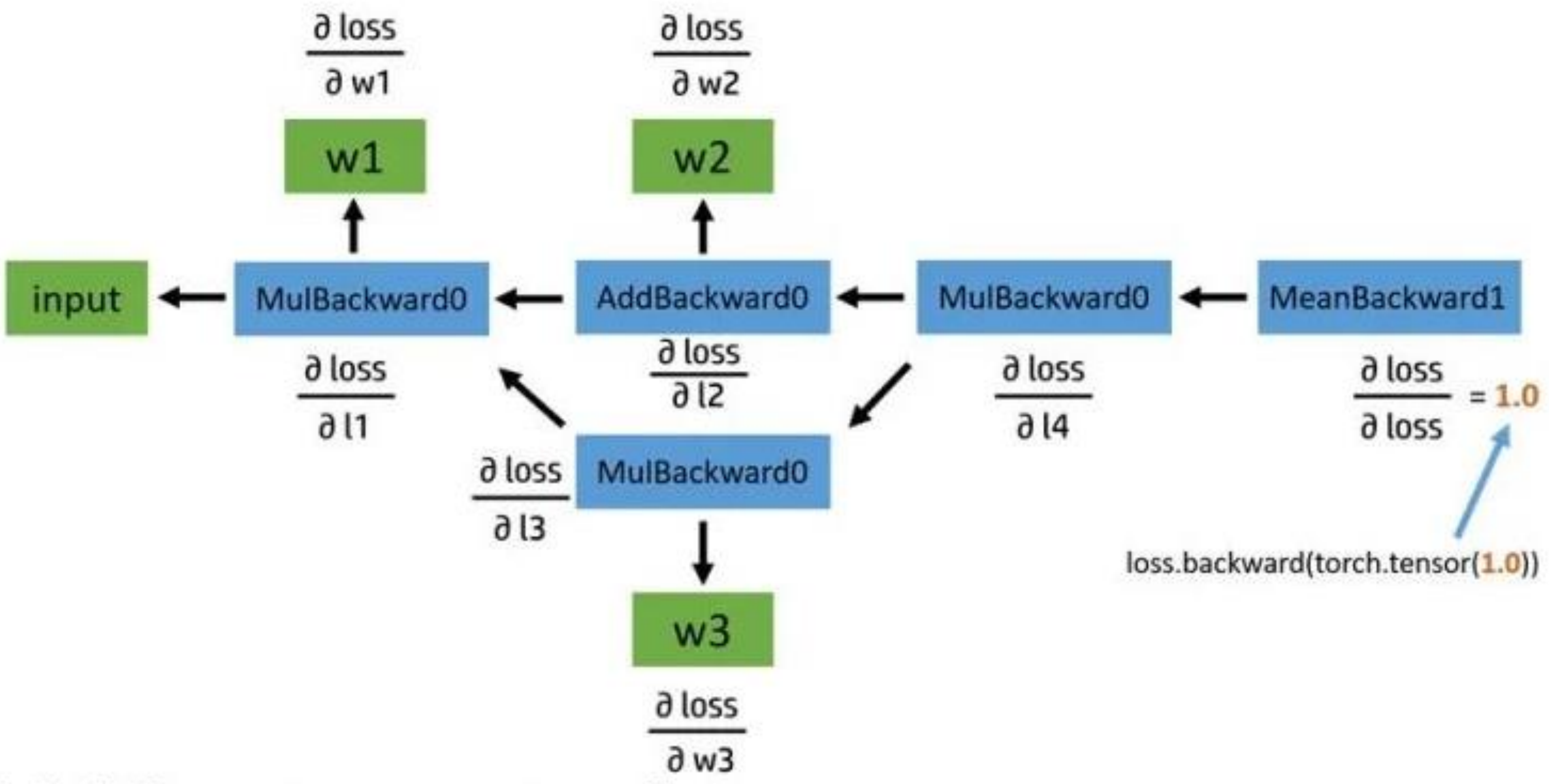
l1 = input x w1

l2 = l1 + w2

l3 = l1 x w3

l4 = l2 x l3

loss = mean(l4)



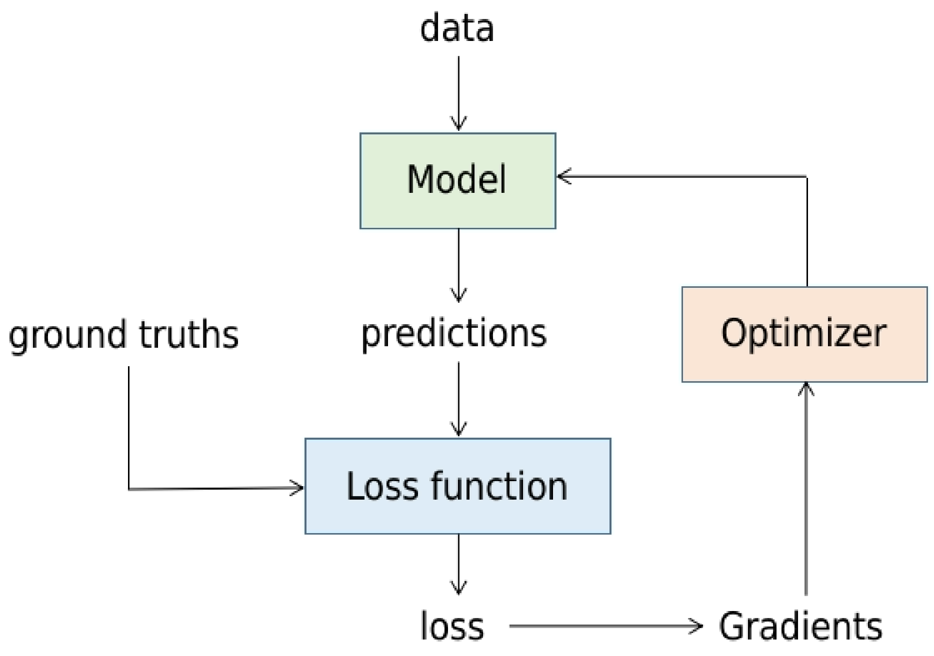
Problem description

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.

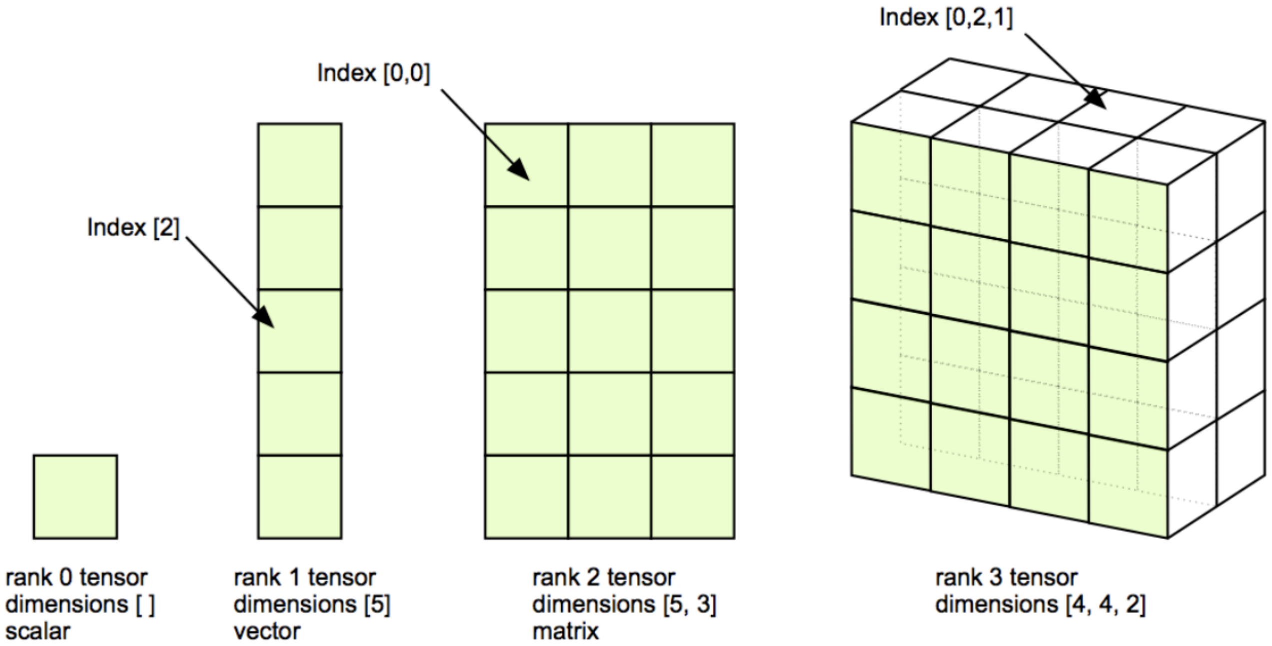
➡ A special data structure was invented.

Called "**Tensor**"(張量)



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

```
T = torch.randn(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 (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([...])

Matrix operations

Tensor & scalar

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

A+1 => tensor([[2, 3], [4, 5]])

A*2 => tensor([[2, 4], [6, 8]])

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

The operation is applied equaly to every elements in the tensor.

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 multipliy

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

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

Squeeze, Unsqueeze

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

Transpose

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

Matrix operations

Reshape

```
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

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

```python
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
```

Train a Model

Step1: Prepare the dataset

Step2: Construct the model

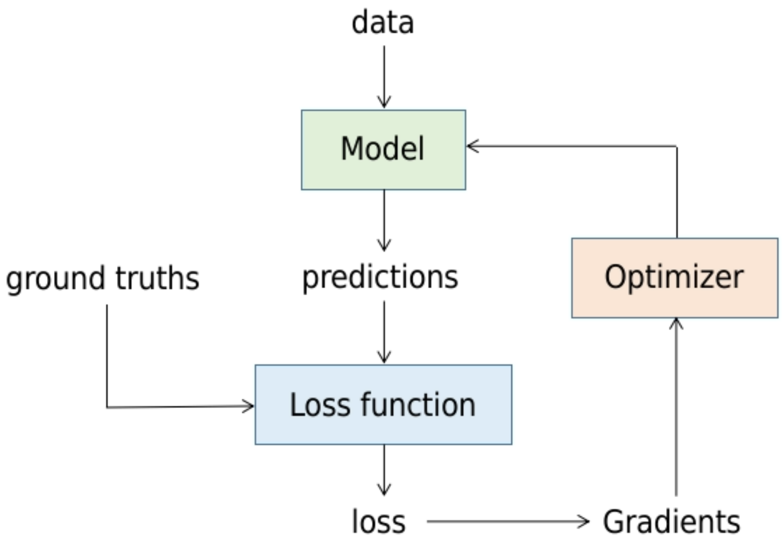
Step3: Define Optimizer

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.



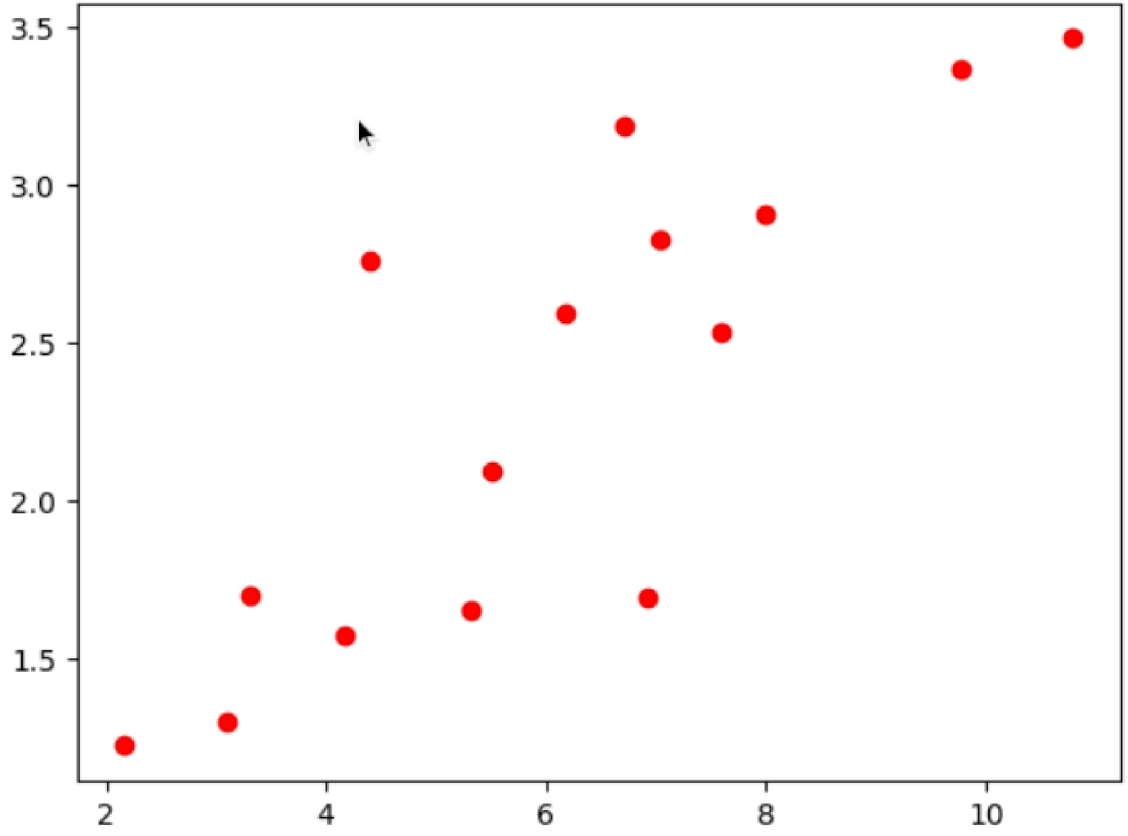
Training

1. clear gradients	`optimizer.zero_grad()`
2. Input data to the model	`output = model(**batch)`
3. compute loss	`loss = loss_fn(output)`
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

Model

Loss

Optimizer

load data

forward

backward

optimize

```
# 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):
    # Convert numpy arrays to torch tensors
    inputs = torch.from_numpy(x_train)
    targets = torch.from_numpy(y_train)

    # Forward pass
    outputs = model(inputs)
    loss = criterion(outputs, targets)

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

    if (epoch+1) % 5 == 0:
        print ('Epoch [{}/{}], Loss: {:.4f}'.format(epoch+1, num_epochs, loss.item()))
```

Check the parameters&gradients

```
inputs = torch.from_numpy(x_train)
targets = torch.from_numpy(y_train)

# Forward pass
outputs = model(inputs)
loss = criterion(outputs, targets)
```

```
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
```

Check the gradients

```
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])
```

```
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])
```

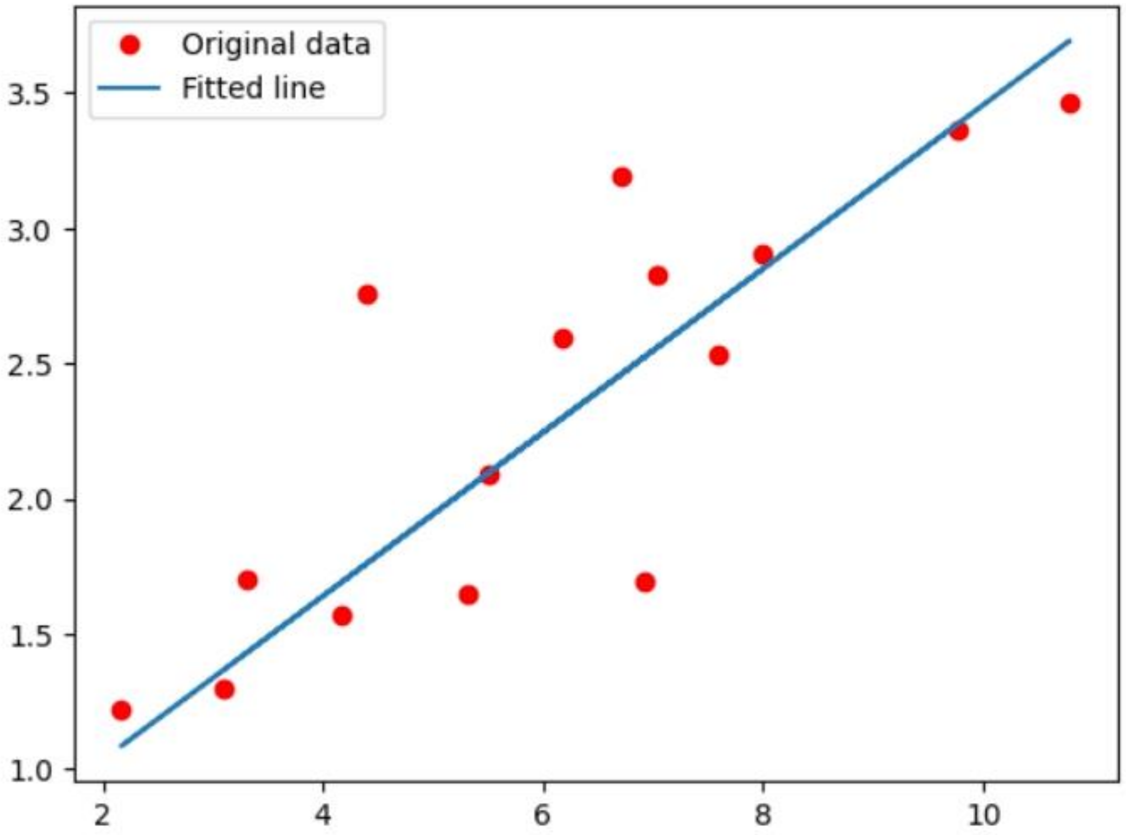
```
optimizer.zero_grad()
print_grads(model)
```

```
weight: Parameter containing:
tensor([[0.4065]], requires_grad=True)
weight grad: None

bias: Parameter containing:
tensor([0.4805], requires_grad=True)
bias grad: None
```

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
```



NLP Tasks

In NLP tasks …

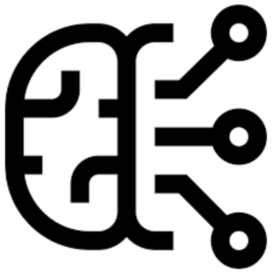
torch.utils.data.Dataset

Text dataset usually very big

torch.utils..data.DataLoader

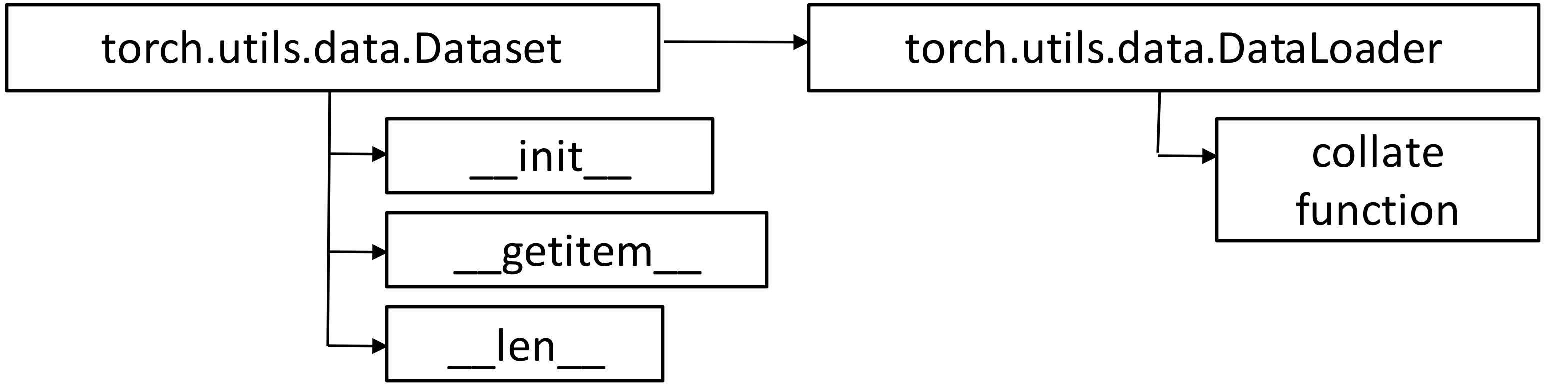
We need to divide the dataset into numerous data batches and gradually feed them to a big complex model.

torch.nn.Module



a big complex language model.

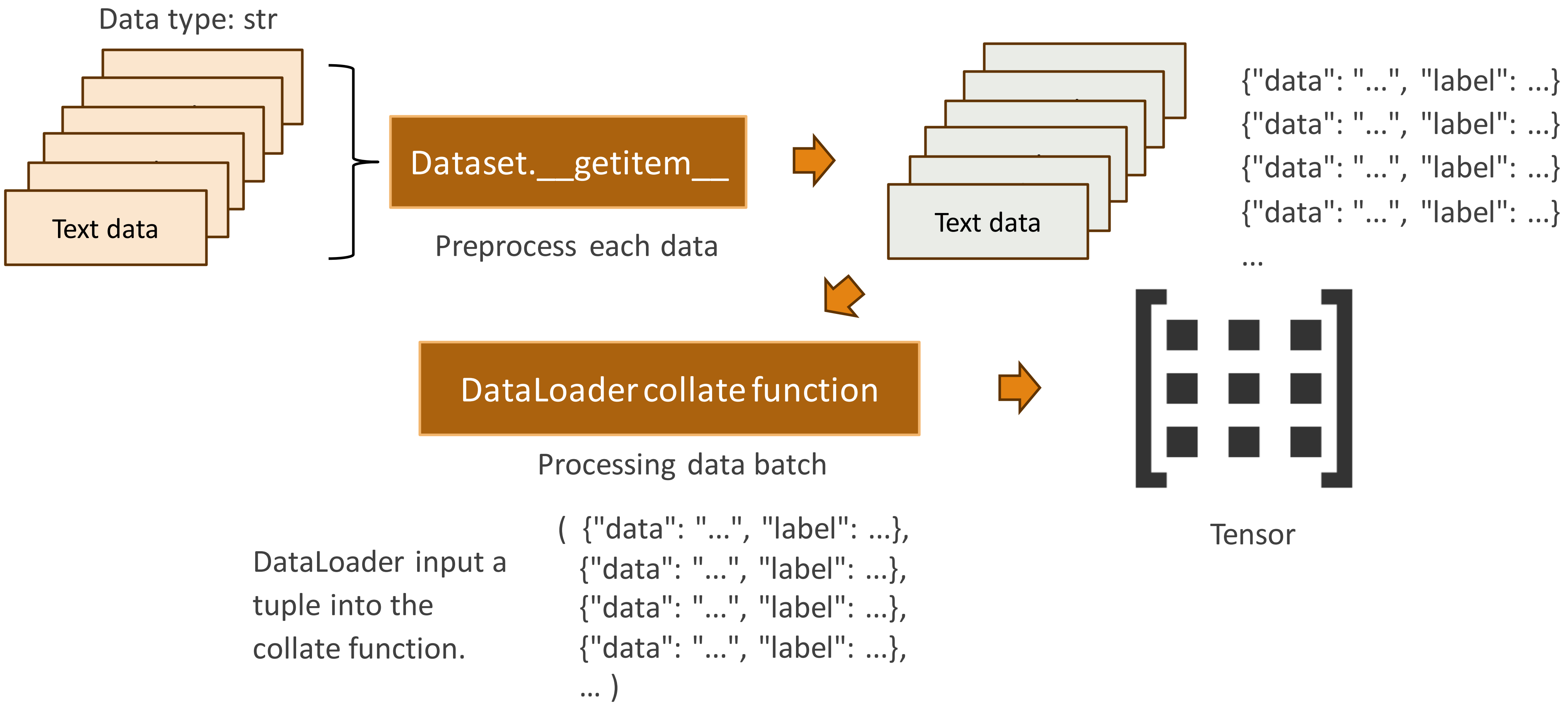
Dataset & DataLoader




```
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)
```

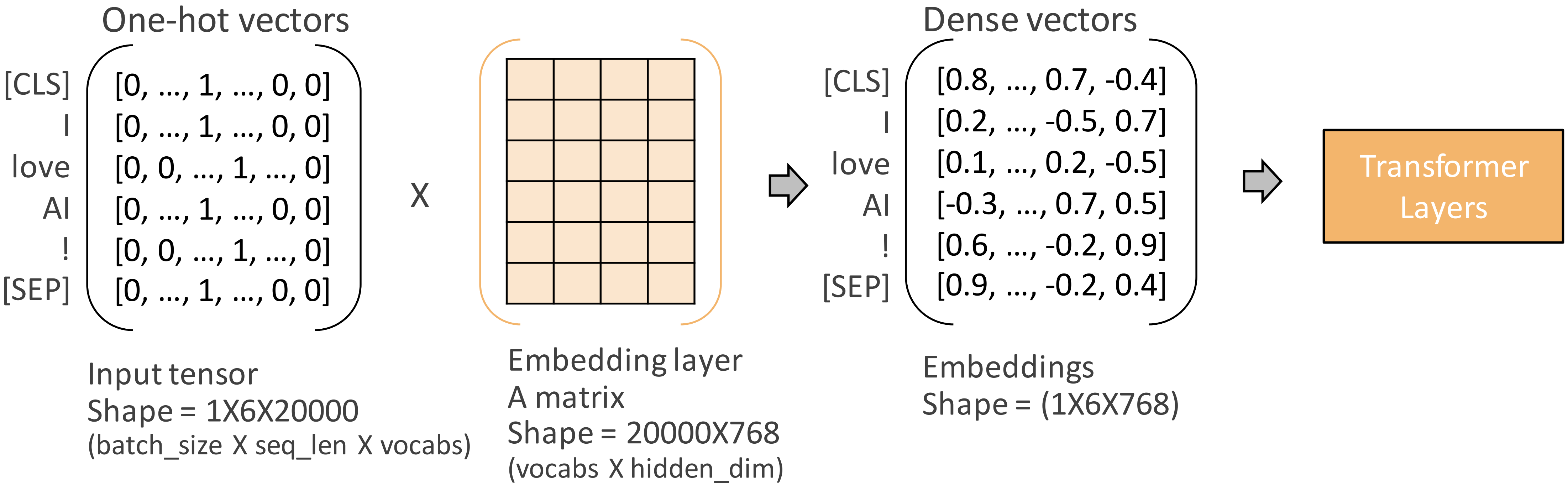
```
def get_batch(sample):
    …

dl = DataLoader(myDataset(split=…, ), batch_size=…,
                collate_fn=get_batch, shuffle=…)

for batch in dl:
    …
```

Data Flow in NLP tasks



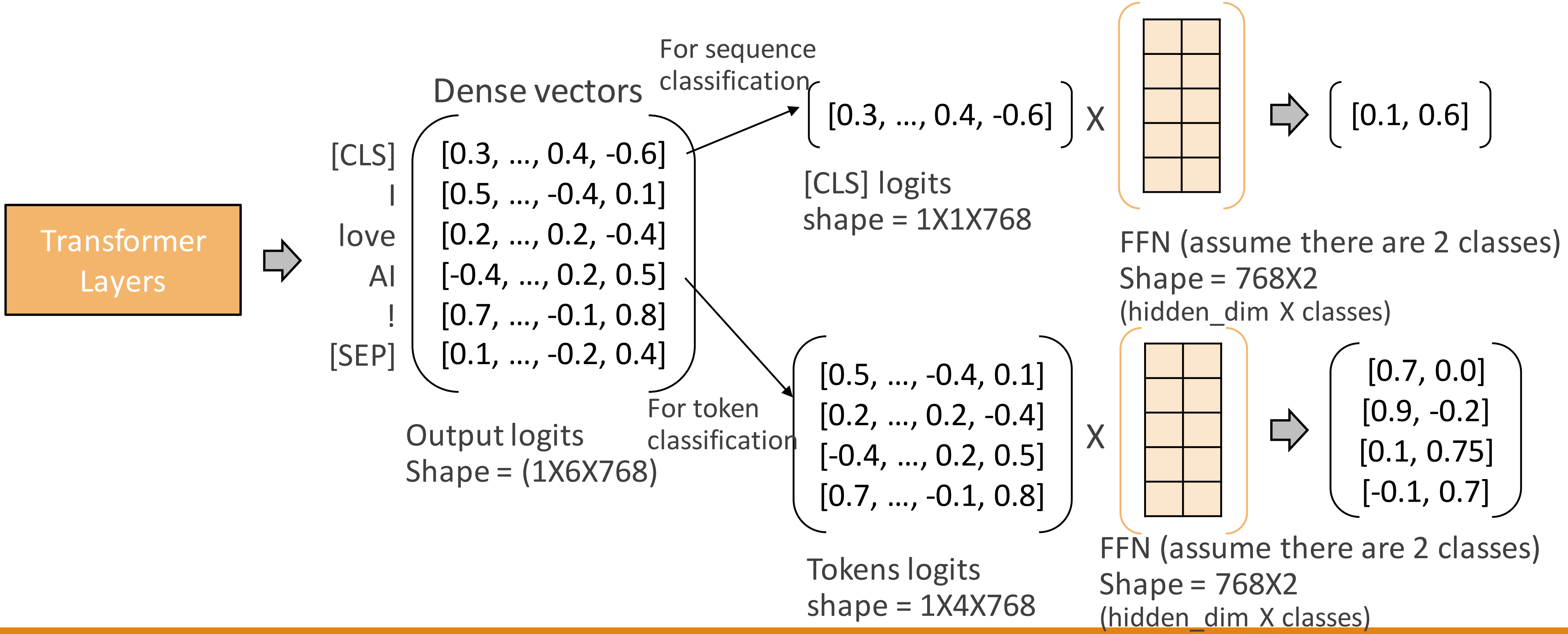
Data Flow in NLP tasks

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

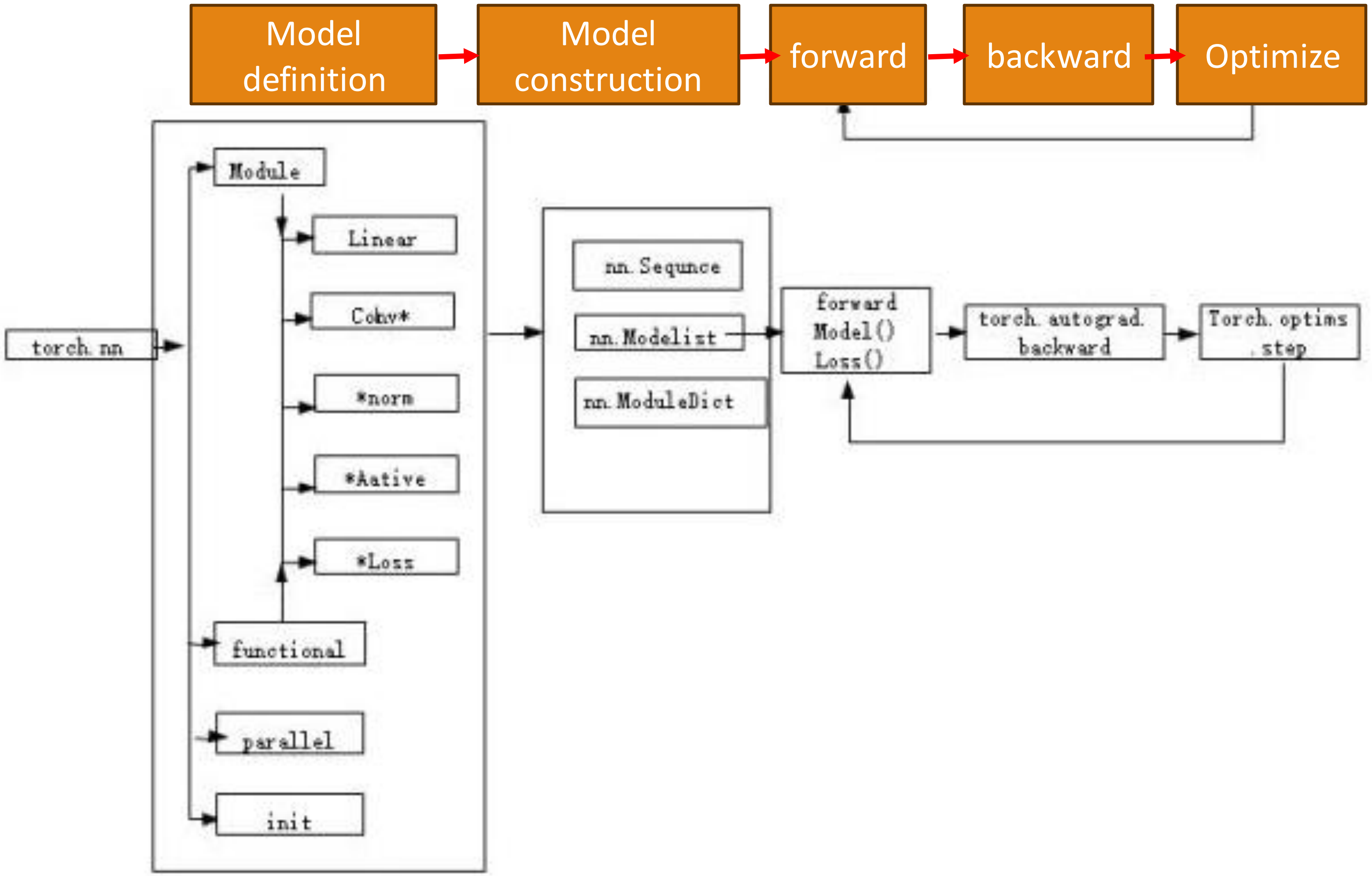


Data Flow in NLP tasks

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



Construct & Train a Model



Construct a Model

A model is defined as:

```
class myModel(torch.nn.Module):
    def __init__(self):
        Super().__init__()
        self.layers = …

    def forward(self, inputs):
        …
        return …
```

Obtain the output logits

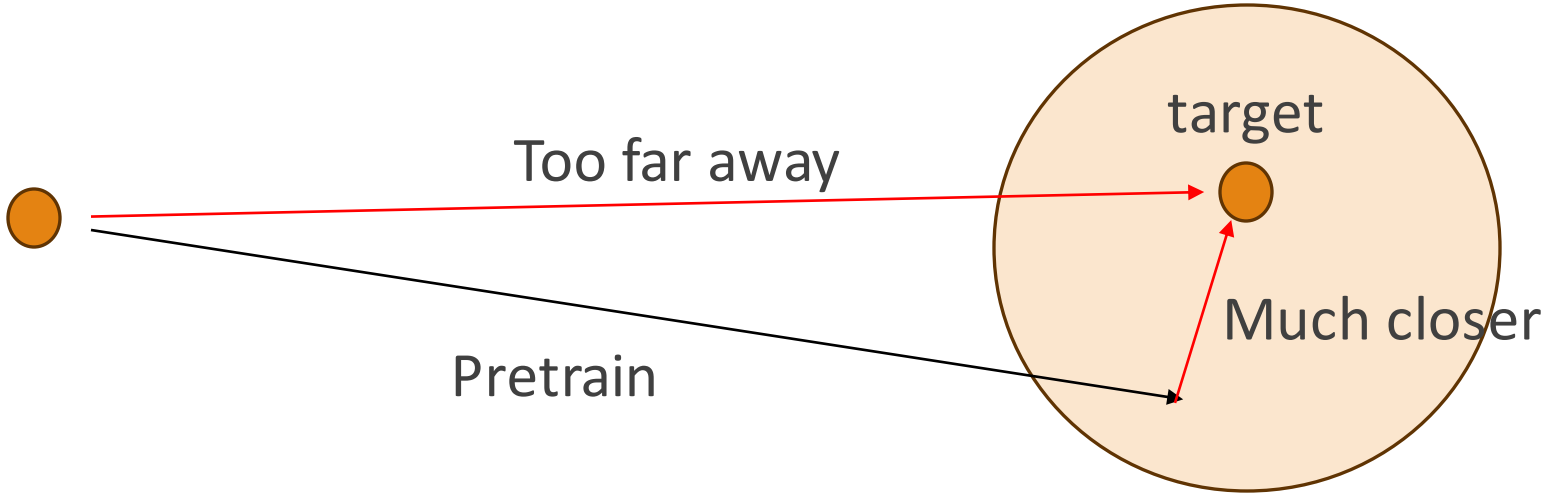
```
model = myModel()

logits = model(inputs)
```

```
torch.nn.Module.__call__
```

Pretrain & Finetuning

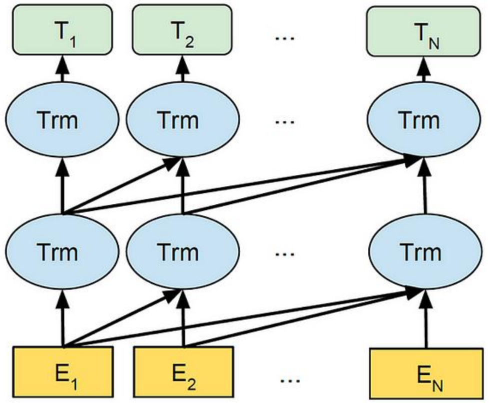
Prof said that transformers have to learn far more features than previous networks, so we have to pretrain the model.

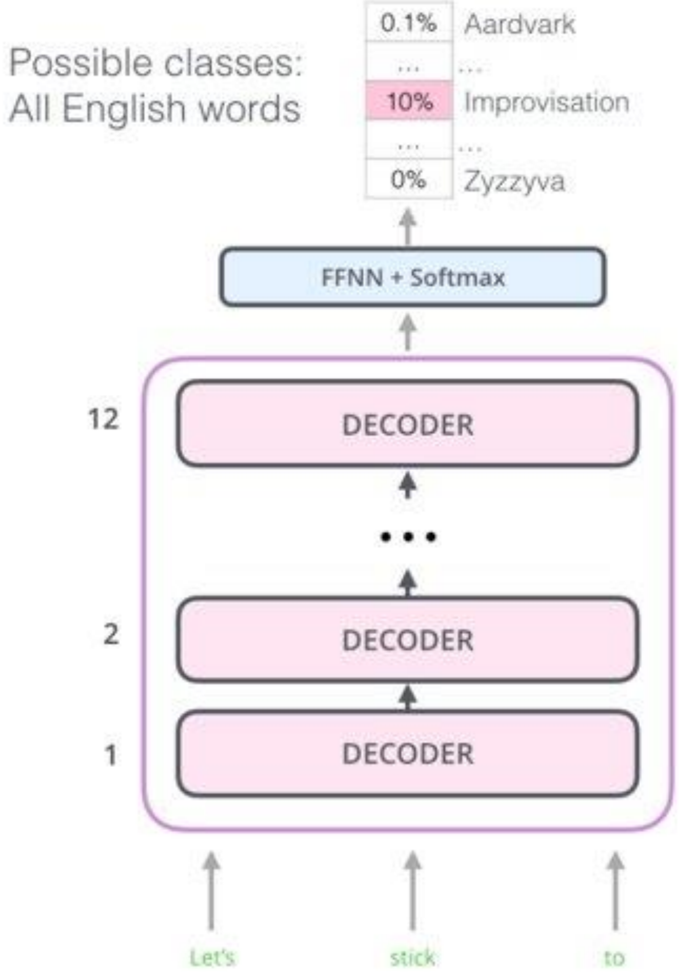
Usually we do not have enough data to train the model directly to the target.



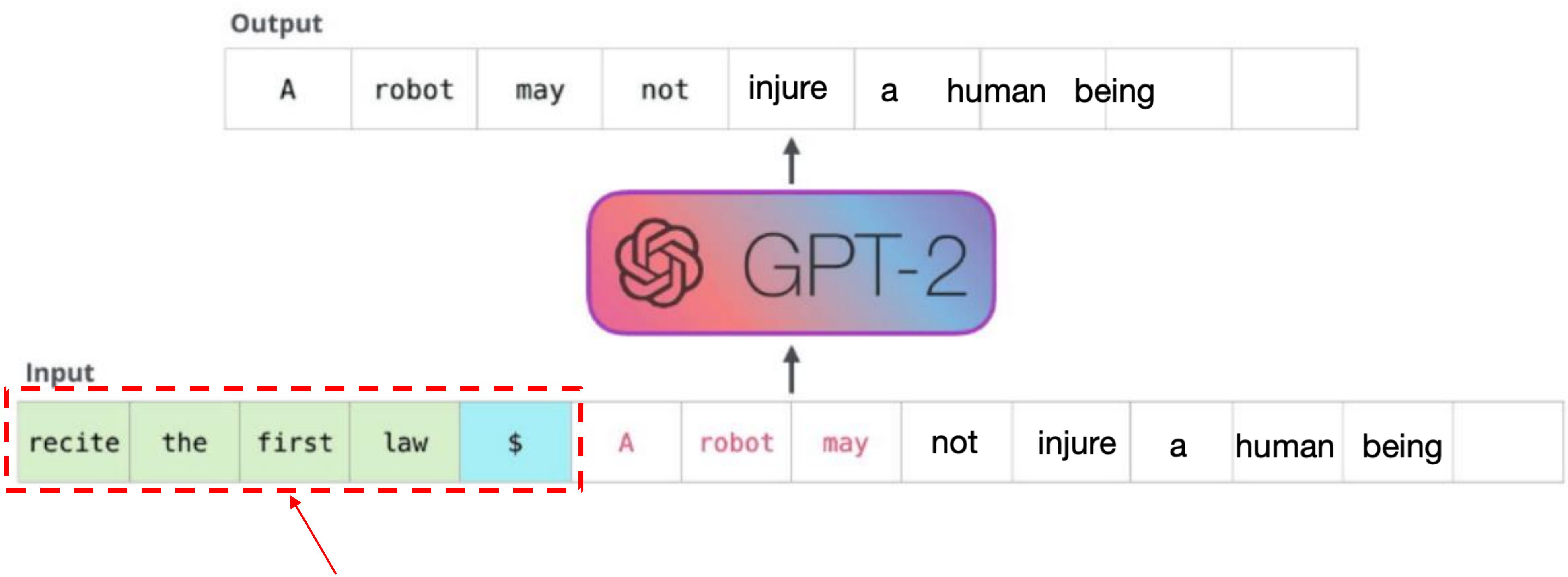
GPT2 Review

- Decoder-only
  - The GPT-2 model consists of multiple layers of decoders.
- Masked Self-attention
  - The masking mechanism in GPT allows the model to look only forward.





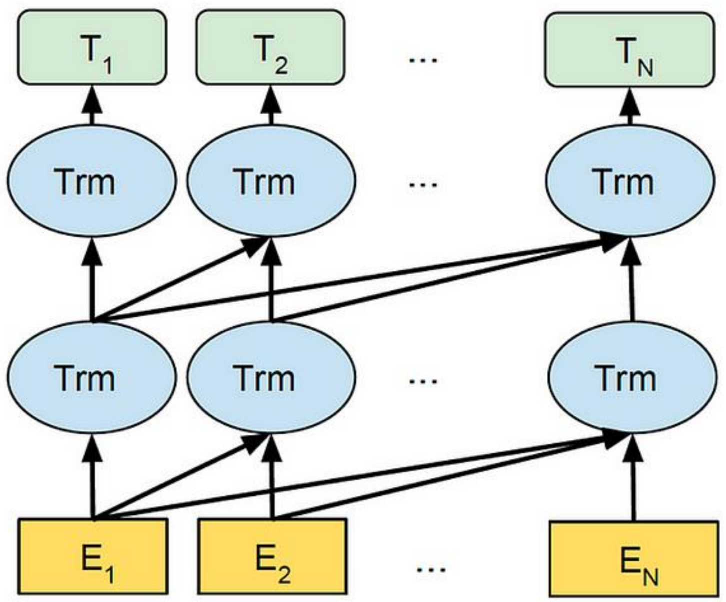
GPT2 Generation



Instruction (prompt)

GPT2 Generation

<s>. Recite human



output

Input Recite the <s>

Huggingface API

Install huggingface tools:

- open the anaconda prompt
- activate [your_env_name]
- pip install transformers
- pip install datasets

"transformers" provides tokenizer and models.

"datasets" provides datasets.

https://huggingface.co/

Tutorials:

https://huggingface.co/docs

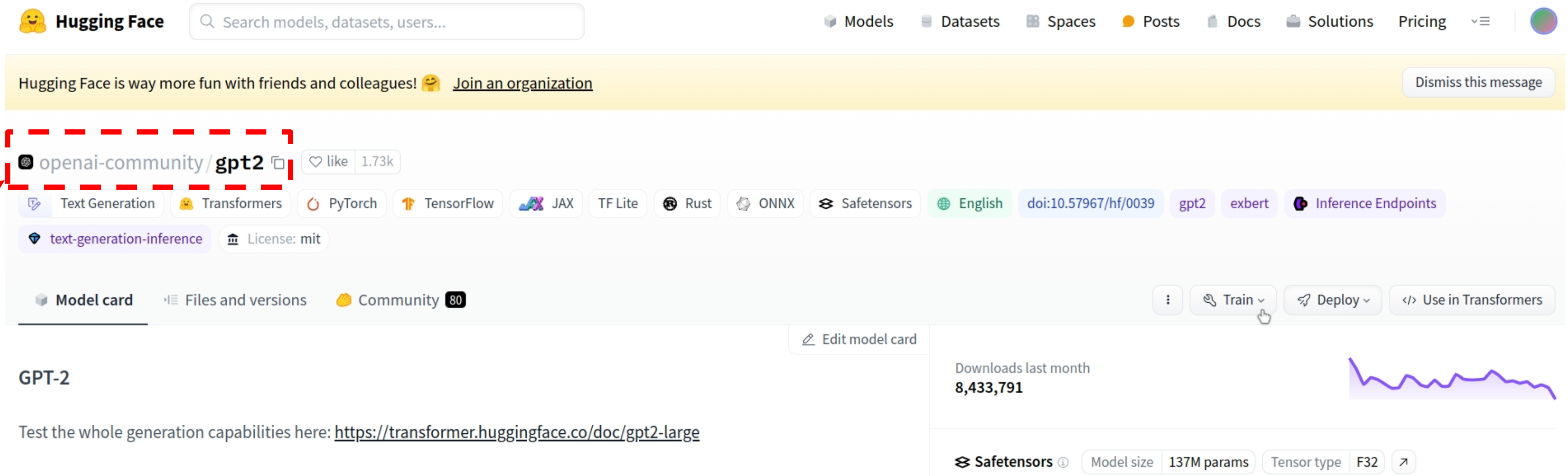
Load Model&Tokenizer

Load model:

```
gpt2_model = transformers.GPT2LMHeadModel.from_pretrained("model_name")
```

Load tokenizer:

```
gpt2_tokenizer = transformers.AutoTokenizer.from_pretrained("model_name")
```



model name is here

Load Model&Tokenizer

In text generation, sometimes we should set padding_side='left', when loading tokenizer.

<s>	Recite	the	first	law	<s>	<pad>	<pad>	<pad>	
<s>	How	are	you	<s>	<pad>	<pad>	<pad>	<pad>	
<s>	Who	is	the	first	president	of	U.S.	<s>	

If padding is at the end the prompt, it would be unreasonable.

Left-padding

<pad>	<pad>	<pad>	<s>	Recite	the	first	law	<s>	
<pad>	<pad>	<pad>	<pad>	<s>	How	are	you	<s>	
<s>	Who	is	the	first	president	of	U.S.	<s>	

Aligning the new tokens.

Tokenize the sequence

The tokenizer API can transform the text into ids.

It can also pack the text batch into tensors:

```
gpt2_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)

Forward

Input the "BatchEncoding" object to the model

```
loss = gpt2_model(**inputs, labels=inputs["input_ids"]).loss
```

If the model is initialized for a specific task (e.g. `GPT2ForSequenceClassification`), the label can be directedly input here and the API compute the loss automaticly.

However, if you just:

```
gpt2_model = T.GPT2Model.from_pretrained("model_name")
```

Then you have to define the loss function by yourself.

e.g. `torch.nn.CrossEntropyLoss()`

Visualization

tqdm

During training, tqdm can provide a visual progress bar.

A simple example

import package

Process bar

```python
from tqdm import tqdm
list = [1,2,3,4,5,6]
bar = tqdm(list)
for i in bar:
    pass
```

```
100%|██████████| 6/6 [00:00<00:00, 15069.36it/s]
```

tqdm

Set description to the front:

`bar.set_description("List number: ")`

```
List number: : 100%|██████████| 6/6 [00:00<00:00, 4707.41it/s]
```

Add item to the end:

`bar.set_postfix(number=i)`

```
List number: : 100%|██████████| 6/6 [00:00<00:00, 750.90it/s, number=6]
```

Tools for recording results

Tensorboard

Install:

```
pip install tensorboard
```

Import:

```
from torch.utils import tensorboard as tb
```

Documentation:

https://pytorch.org/docs/stable/tensorboard.html

Weights & Bias

Install:

```
pip install wandb
```

Import:

```
import wandb
```

Documentation:

https://docs.wandb.ai/