pytorch

Ramin Zarebidoky (LiterallyTheOne)

02 Sep 2025

Introduction

What is PyTorch

PyTorch is an open-source framework for **Machine Learning**. It is developed by **Facebook AI Research** (**Meta**). This framework is mostly used to build and train **Deep Learning** models. Because of its flexibility and **Pythonic inference**, it has become so popular, especially among researchers. So, let's write a **hello world** to understand **PyTorch** better. Then, in the future, we will complete this **hello world** example step by step.

Hello world

Problem definition

Imagine that we have `3` samples of data. Each sample has `8` features. We want to classify this data into `4` classes. So, the shape of our data would be `[3, 8]` and the shape of our result should be `[3, 4]`. Now, our plan is to just make a model that we can feed our data to. The simplest way to do that is to have a **fully connected layer**, with the input size of `8` and the output size of `4`, like the image below:

Implementation

At first, we should import the necessary modules as below:

```
# -------------------[ Imports ]-------------------
import torch
from torch import nn
```

In the code above, we import `torch` and `nn` (neural network). Now, let's create random data:

```
# -------------------[ Data ]-------------------
data = torch.rand((3, 8))  # (number_of_samples, features)
```

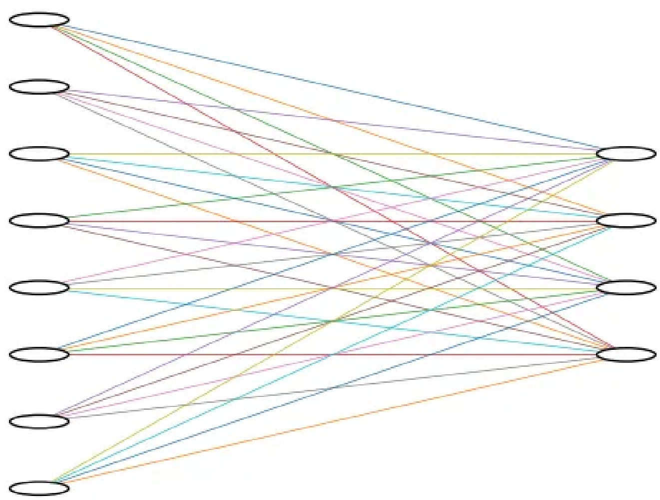

Figure 1: pytorch_hello_world

Now, we have random data that has 3 samples, and each sample has `8` features. After that, let's create a simple linear model like the image that we provided in the implementation section.

```
# -------------------[ Model ]-------------------
model = nn.Linear(8, 4)  # (features, number_of_classes)
```

The code above creates a fully connected neural network layer that takes `8` features as its input and produces `4` classes as its output. For the next step, let's feed that data to our model.

```
# -------------------[ Feed the data to the model
 ↪  ]-------------------
logits = model(data)
print(logits)

"""
--------
output:

tensor([[ 5.3127e-01,  6.7324e-01, -1.7548e-01, -2.0279e-02],
        [ 5.3984e-01,  1.0462e+00, -1.0124e-01,  8.4969e-03],
        [ 4.6493e-01,  1.0864e+00, -3.6424e-01,  8.6406e-04]],
       grad_fn=<AddmmBackward0>)
"""
```

As you can see, we could simply call the model with our data. The output would be something like the probability of each class in each sample. As you might have noticed, there is a `grad_fn` in the output. When we call the model like this, `PyTorch` stores the `gradient` that we are going to use it in the future. So, if we want to get the class that we want, we should just report the index of the maximum probability.

```
result = logits.argmax(1)
print(result)

"""
--------
output:

tensor([1, 1, 1])
"""
```

In the code above, I took the `argmax` of the dimension `1`, which had the probabilities on it. Because we haven't trained our model yet, the results are biased and all of them are predicting that this sample belongs to class `1`.

In further we are going to explain each of them and try to complete our code step by step. You must have so many questions right now, but don't worry, they will be answered soon.

Tensor

What is Tensor

`Tensor` is the fundamental of `PyTorch`. Input, output, and the parameters of the model are all in `Tensors`. `Tensor` is like an array (`Numpy array`) but with more power.

- It can be run on `GPU`
- It supports automatic gradients

`Tensor` operations in `Pytorch` are pretty similar to `Numpy array`. So, if you have worked with `Numpy array` before, you are a step ahead.

In our `Hello world` example, we have created random data using `torch.rand((3, 8))` also we got the index of the maximum probability using `logits.argmax(1)`. In this tutorial, we are going to explain more about the main operations in `Tensor` and learn how to use them.

Create a Tensor

There are so many ways that we can create a `Tensor`. One of the simplest ways to create a tensor is as below:

```
data = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
]
t1 = torch.tensor(data)
print(t1)

"""
--------
output:

tensor([[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]])
"""
```

As you can see, we had a 2-dimensional matrix, and we gave it to `torch.tensor` as an argument and stored the result in a variable called `t1`. When we print `t1`, the output would be a `Tensor` of that matrix.

We can also create a `Tensor` by knowing its shape. For example, in our `Hello World` example, we created a random dataset using `torch.rand` function. We also have other functions that we can give the shape of `Tensor` to them and get a `Tensor`. You can see the examples in the code below:

```
s1 = torch.rand((3, 8))
print(s1)
print(s1.shape)

"""
--------
output:

tensor([[0.6667, 0.7057, 0.7670, 0.7719, 0.7298, 0.5729, 0.8281,
 ↪  0.5963],
        [0.1056, 0.5377, 0.3380, 0.4923, 0.0246, 0.8192, 0.3945,
 ↪  0.1150],
        [0.3885, 0.4211, 0.2655, 0.6766, 0.5082, 0.6465, 0.9499,
 ↪  0.2008]])
torch.Size([3, 8])
"""
```

```
s2 = torch.zeros((3, 8))
print(s2)
print(s2.shape)

"""
--------
output:

tensor([[0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0.]])
torch.Size([3, 8])
"""
```

```
s3 = torch.ones((3, 8))
print(s3)
print(s3.shape)

"""
--------
output:

tensor([[1., 1., 1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1., 1., 1.]])
```

```
torch.Size([3, 8])
"""
```

In the above examples, we have used 3 functions:

- `torch.rand`: Creates random data
- `torch.ones`: Fills with one
- `torch.zeros`: Fills with zero

As you can see, the shape of all of them is `[3, 8]`, like the way that we gave them. (You can access the shape of a tensor by `.shape` variable)

We can also create a `Tensor` from other `Tensors`.

```
l1 = torch.zeros_like(t1)
print(l1)
print(l1.shape)

"""
--------
output:
tensor([[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]])
torch.Size([3, 3])
"""
```

The first `Tensor` that we created was called `t1` and its shape was `[3, 3]`. In the example above, we created a `Tensor` like `t1`, which is filled with zeros.

Attributes of a Tensor

`Tensor` has different attributes that define how it is stored. We mentioned one of them, which was `shape`. Now we learn two more of them, `dtype` and `device`.

- `shape`: shape of the tensor
- `dtype`: data type of the tensor
- `device`: device of the tensor, like `cpu` or `cuda` (for `gpu`)

```
print(f"shape: {t1.shape}")
print(f"dtype: {t1.dtype}")
print(f"device: {t1.device}")

"""
--------
output:

shape: torch.Size([3, 3])
```

```
dtype: torch.int64
device: cpu
"""
```

Control the device

To find if our system has any available accelerators, we can use the code below:

```
if torch.accelerator.is_available():
    device = torch.accelerator.current_accelerator()
else:
    device = "cpu"

print(device)

"""
--------
output:

mps
"""
```

The code above first checks if there are any accelerators like `cuda` or `mps` (for MacBook). Then puts the current accelerator in a variable called `device`. If there wasn't any available, the value of `device` would be `cpu`. In my case, the output is `mps`. If you run this code on `Google Colab` with the `GPU` on, you would get `cuda`.

We can change the device of any `Tensor` by using a function called `.to()`. For example:

```
t1 = t1.to(device)
print(t1.device)

"""
--------
output:

mps:0
"""
```

In the code above, we changed the device of the `Tensor` called `t1` to the current accelerator, which in my case is `mps`.

Operations on Tensor

The syntax of `Tensor` operations is pretty much like the `Numpy Arrays`. As you recall, we had a `Tensor` called `t1` that we cast it to run on `gpu`. `t1` was a 2D matrix with the shape of `[3, 3]` and the content of it was like below:

```
"""
tensor([[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]], device='mps:0')
"""
```

If we want to only select the first row of it, we can use the code below:

```
t1_first_row = t1[0]
print(t1_first_row)

"""
--------
output:

tensor([1, 2, 3], device='mps:0')
"""
```

If we want to select its first column, we can use the code below:

```
t1_first_column = t1[:, 0]
print(t1_first_column)

"""
--------
output:

tensor([1, 4, 7], device='mps:0')
"""
```

If we want to select a slice of that tensor, for example, the second row till the end, and the second column till the end, the code below would be useful:

```
t1_slice = t1[1:, 1:]
print(t1_slice)

"""
--------
output:

tensor([[5, 6],
        [8, 9]], device='mps:0')
```

```
"""
```

We can join (concatenate) two tensors using `torch.cat`. For example, let's make two 2D tensors and concatenate them.

```
c1 = torch.zeros((5, 4))
c2 = torch.ones((5, 2))

c3 = torch.concat((c1, c2), dim=1)
print(c3)

"""
--------
output:

tensor([[0., 0., 0., 0., 1., 1.],
        [0., 0., 0., 0., 1., 1.],
        [0., 0., 0., 0., 1., 1.],
        [0., 0., 0., 0., 1., 1.],
        [0., 0., 0., 0., 1., 1.]])
"""
```

We can transpose a tensor, using `.T`.

```
a1 = torch.tensor([
    [1, 2, 3],
    [4, 5, 6],
])
a1t = a1.T

print(a1)
print(a1t)

"""
--------
output:

tensor([[1, 2, 3],
        [4, 5, 6]])
tensor([[1, 4],
        [2, 5],
        [3, 6]])

"""
```

We can do arithmetic operations on `Tensors` as well. For example, let's create 2 matrices and multiply them.

```python
matrix_1 = torch.Tensor([
    [1.0, 2.0, 3.0],
    [4.0, 5.0, 6.0],
])

matrix_2 = torch.tensor([
    [1.0],
    [2.0],
    [3.0],
])

result = matrix_1 @ matrix_2
print(result)

"""
--------
output:

tensor([[14.],
        [32.]])
"""
```

As you can see, `1x1+2x2+3x3=1+4+9=14` and `4x1+5x2+6x3=4+10+18=32`.

Also, we can calculate the sum of a matrix using `.sum`.

```python
sum_matrix_1 = matrix_1.sum()
print(sum_matrix_1)

"""
--------
output:

tensor(21.)
"""
```

In the `hello world` example, we used `argmax`. Now, let's use the `max` function, which calculates the maximum and the index of the maximum as well.

```python
b1 = torch.tensor([
    [3, 1, 7, 2],
    [2, 4, 1, 3],
    [9, 1, 2, 5],
])

max_of_each_row = b1.max(dim=1)
```

```
print(max_of_each_row)

"""
--------
output:

torch.return_types.max(
values=tensor([7, 4, 9]),
indices=tensor([2, 1, 0]))
"""
```

As you can see, in the code above, the maximum number in the first row is `7`, and it is in the index of the `2` (we start with `0`) and so on.

Now, let's use the `argmax` function and compare the results.

```
argmax_of_each_row = b1.argmax(dim=1)

print(argmax_of_each_row)

"""
--------
output:

tensor([2, 1, 0])
"""
```

As you can see, the indices of both results are the same.

Conclusion

In this tutorial, we learned more about `Tensor`, which is the core concept of `PyTorch`. We learned how to create them, what their most important attributes are, how to control the device, also how to perform an operation on them. There are so many things that you can do with tensors, and these were only some of them to show the concept of a `Tensor`.

Model

What is model

Model in PyTorch can be seen as a function that maps `inputs` to `outputs`. It consists of different layers, each of which has its own requirements. In our `hello world` example, we had a simple `linear` model that required its input to have 8 features, and produced output of 4 features.

```python
# -------------------[ Model ]-------------------
model = nn.Linear(8, 4)  # (features, number_of_classes)
```

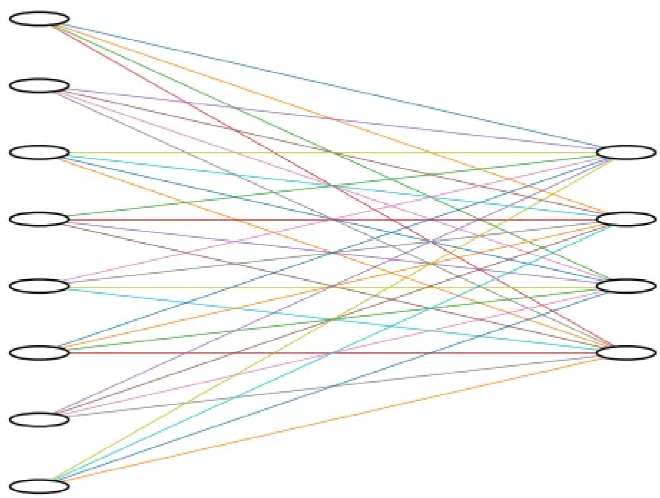

Figure 2: PyTorch hello world model

Now, let's make the model a little bit more complex.

Sequential Model

One of the ways that we can stack up some layers in **PyTorch** is by using `nn.Sequential`. So, let's make our model a little bit more complicated, like below:

```python
model_2 = nn.Sequential(
    nn.Linear(8, 16),
    nn.Linear(16, 4),
)
```

As you can see, our model right now takes `8` features as its input. Then, it maps it to `16`. And finally, it produces `4` output. As it's shown in the image above, we have some circles that lines are connected to. We call these circles `neurons`. The first layer is called an **input layer**. The middle layer, which has `16` neurons called a **hidden layer**. And the last layer is called an output layer.

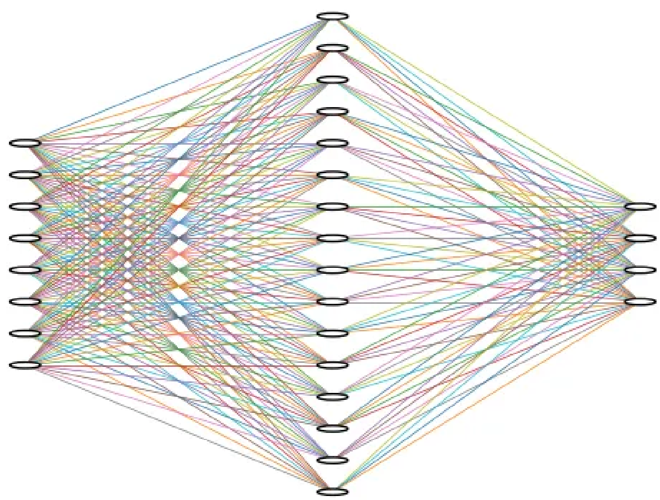

Figure 3: model-8-16-14

So, for this model we have:

- 1 Input layer
- 1 Hidden layer
- 1 Output layer

Now, let's make some random data and see if it works correctly or not.

```
data = torch.rand((3, 8))
result = model_2(data)

print(result)

"""
--------
output:
tensor([[ 0.3901, -0.0124, -0.1982,  0.4792],
        [ 0.6230,  0.0920, -0.0491,  0.5871],
        [ 0.4019,  0.0620, -0.2312,  0.4669]],
 ↪  grad_fn=<AddmmBackward0>)
"""
```

As you could see, we have the output in a way that we wanted, and the model is functioning correctly. Now, let's make 2 hidden layers.

```
model_3 = nn.Sequential(
    nn.Linear(8, 16),
    nn.Linear(16, 32),
    nn.Linear(32, 4),
)
```

As you can see, we have 2 hidden layers now. One with 16 neurons and the other with 32 neurons. Let's test this model as well to see if it functions correctly.

```
data = torch.rand((3, 8))
result = model_3(data)

print(result)

"""
--------
output:
tensor([[-0.2115, -0.2278, -0.0586, -0.0266],
        [-0.1489, -0.0981,  0.0675, -0.0184],
        [-0.1330, -0.1383,  0.1055, -0.0529]],
 ↪  grad_fn=<AddmmBackward0>)
"""
```

As it's shown above, it is functioning correctly.

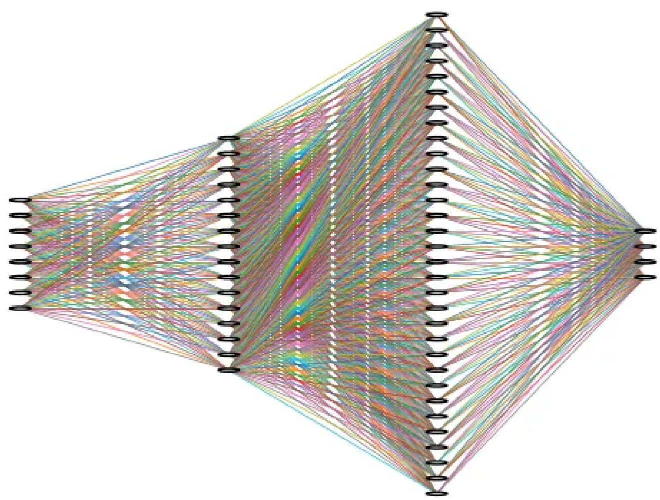

Figure 4: model-8-16-32-4

Standard way to define a model

In **Pytorch**, we define our model by creating a subclass of `nn.Module`. We put all the layers in the `__init__` function. We also put the way that we want to process our data in `forward` function. For example, we can define a model like below:

```
class MyModel(nn.Module):
    def __init__(self):
        super().__init__()

        self.layers = nn.Sequential(
            nn.Linear(8, 16),
            nn.Linear(16, 32),
            nn.Linear(32, 4),
        )

    def forward(self, x):
        x = self.layers(x)
        return x
```

In the code above, we have defined a model like `model_3`. We put the layers in `__init__` function and put the way that we want to process the input in `forward` function. Let's create an instance of that model and print it.

```
my_model = MyModel()

print(my_model)

"""
--------
output:

MyModel(
  (layers): Sequential(
    (0): Linear(in_features=8, out_features=16, bias=True)
    (1): Linear(in_features=16, out_features=32, bias=True)
    (2): Linear(in_features=32, out_features=4, bias=True)
  )
)
"""
```

As you can see, it shows the layers that we have created. So, let's create some random data and feed it to our model to see if it functions correctly.

```
data = torch.rand((3, 8))

result = my_model(data)
```

```
print(result)

"""
--------
output:
tensor([[ 0.1615, -0.0514,  0.0914, -0.1007],
        [ 0.1709, -0.0739,  0.1314, -0.2231],
        [ 0.0905, -0.0171,  0.1184, -0.1016]],
 ↪  grad_fn=<AddmmBackward0>)
"""
```

And as you can see, it functions as intended. Now, let's see another example:

```
class MyModel2(nn.Module):
    def __init__(self):
        super().__init__()

        self.layers_1 = nn.Sequential(
            nn.Linear(8, 16),
            nn.Linear(16, 32),
        )

        self.layers_2 = nn.Sequential(
            nn.Linear(32, 16),
            nn.Linear(16, 4),
        )

    def forward(self, x):
        x = self.layers_1(x)
        x = self.layers_2(x)
        return x
```

In this model, we have 2 sequential layers. When we give data to this model, at first it goes through `layers_1` and then `layers_2`. Let's create an instance of this model and print it.

```
my_model_2 = MyModel2()

print(my_model_2)

"""
--------
output:

MyModel2(
  (layers_1): Sequential(
    (0): Linear(in_features=8, out_features=16, bias=True)
```

```
    (1): Linear(in_features=16, out_features=32, bias=True)
  )
  (layers_2): Sequential(
    (0): Linear(in_features=32, out_features=16, bias=True)
    (1): Linear(in_features=16, out_features=4, bias=True)
  )
)
"""
```

Now let's test it to see if it functions correctly.

```
data = torch.rand((3, 8))

result = my_model_2(data)
print(result)

"""
--------
output:

tensor([[-0.2815,  0.2154,  0.0795, -0.0977],
        [-0.2823,  0.2451,  0.0843, -0.0942],
        [-0.2267,  0.2214,  0.0792, -0.0364]],
 ↪  grad_fn=<AddmmBackward0>)
"""
```

As you can see it works as it should be.

Run on Accelerator

To run our model on the available accelerator, we should first find it. To do so, we can use the code below:

```
if torch.accelerator.is_available():
    device = torch.accelerator.current_accelerator()
else:
    device = "cpu"

print(device)

"""
--------
output:

mps
"""
```

For me, the available accelerator was `mps`. Now, we should cast both the **data** and the **model** to the device that we have. To make the code more clean, I have created them again.

```
data = torch.rand((3, 8))
my_model_2 = MyModel2()

data = data.to(device)
my_model_2 = my_model_2.to(device)

result = my_model_2(data)

print(result)

"""
--------
output:
tensor([[ 0.1318,  0.0968, -0.0257, -0.3693],
        [ 0.0812,  0.0943, -0.0765, -0.4375],
        [ 0.1081,  0.0463, -0.0657, -0.4549]], device='mps:0',
       grad_fn=<LinearBackward0>)
"""
```

As you can see, now I ran our model on our available accelerator and the output's device is the available accelerator.

Conclusion

In this tutorial we have learned how to define a model. First, we learned how to make our layers more complex with `nn.Sequential`. Then, we learned how to make a model in standard way with `nn.Module` and fill the `forward` function. At this time, we only know about one layer, which is `Linear` layer. Moving forward, we learn more about different layers and how to use them. Also, you might say the outputs are pretty random. In the next tutorials we are going to learn how to train our model.

Data

Load a dataset

We can work with all kinds of data in **Pytorch**. For this example, we are going to work with the data called IRIS. Let's load it together using a package called `scikit-learn`. It is pre-installed on `Google Colab`, but if you want to install it, you can use: `pip install scikit-learn`.

```
from sklearn.datasets import load_iris
```

```python
iris = load_iris()
```

After we run the code above, it downloads the dataset, and all the data are in a variable called `iris`. If we want to see what features it has, we can use the code below:

```python
print("feature names:")
print(iris.feature_names)

"""
--------
output:

feature names:
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',
 ↪  'petal width (cm)']
"""
```

As you can see, it has `4` features:

- sepal length (cm)
- sepal width (cm)
- petal length (cm)
- petal width (cm)

If we want to see what the target classes are, we can use the code below:

```python
print("target names:")
print(iris.target_names)

"""
--------
output:

target names:
['setosa' 'versicolor' 'virginica']
"""
```

As it is shown, it has `3` classes, which are the names of the flowers:

- setosa
- versicolor
- virginica

To access the data, we can use `iris.data`, and to access the targets of each sample, we can use `iris.targets`. Let's see how many samples we have:

```
print("Number of samples:", len(iris.data))

"""
--------
output:

Number of samples: 150
"""
```

As you can see, it has 150 samples. Let's show some of the samples using the code below:

```
chosen_indexes = np.linspace(0, len(iris.data), 10, dtype=int,
 ↪  endpoint=False)
print("Chosen indexes:")
print(chosen_indexes)
print()

print("10 sample of data:")
print(iris.data[chosen_indexes])
print()

print("10 sample of target:")
print(iris.target[chosen_indexes])
print()

"""
--------
output:

Chosen indices:
[  0  15  30  45  60  75  90 105 120 135]

10 samples of data:
[[5.1 3.5 1.4 0.2]
 [5.7 4.4 1.5 0.4]
 [4.8 3.1 1.6 0.2]
 [4.8 3.  1.4 0.3]
 [5.  2.  3.5 1. ]
 [6.6 3.  4.4 1.4]
 [5.5 2.6 4.4 1.2]
 [7.6 3.  6.6 2.1]
 [6.9 3.2 5.7 2.3]
 [7.7 3.  6.1 2.3]]
```

```
10 samples of target:
[0 0 0 0 1 1 1 2 2 2]

"""
```

In the code above, I have chosen `10` samples of data using `np.linspace`. After that, I printed the chosen indices.

Make the data ready for the model

In our `hello world` example, we had `3` samples of data with `8` features. Now, for this dataset, we have `150` samples of data with `4` features. So, our job is pretty much the same; we should only transform our dataset and targets to `Tensors`. To do so, we can use the code below:

```
data = torch.tensor(iris.data).to(torch.float)
target = torch.tensor(iris.target)
```

Now, both the data and the target are in `Tensors`. Also, I changed the type of data to `float`. For the next step, let's prepare a model that can work with this data.

```
class IRISClassifier(nn.Module):
    def __init__(self):
        super().__init__()

        self.layers = nn.Sequential(
            nn.Linear(4, 16),
            nn.Linear(16, 8),
            nn.Linear(8, 3),
        )

    def forward(self, x):
        return self.layers(x)
```

As you can see, I have created a model, called `IRISClassifier`, that has:

- 4 neurons for the input layer (because we have 4 input features)
- 16 neurons for the first hidden layer
- 8 neurons for the second hidden layer
- 3 neurons for the output layer (because we have to classify them into 3 classes)

So, let's create an instance of that model and print it.

```
iris_classifier = IRISClassifier()
print(iris_classifier)
```

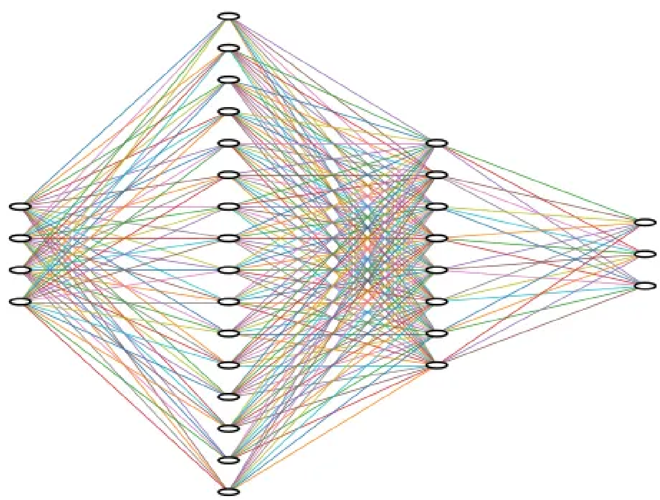

Figure 5: model-4-16-8-3

```
"""
--------
output:

IRISClassifier(
  (layers): Sequential(
    (0): Linear(in_features=4, out_features=16, bias=True)
    (1): Linear(in_features=16, out_features=8, bias=True)
    (2): Linear(in_features=8, out_features=3, bias=True)
  )
)
"""
```

Then, let's feed the chosen indices of our data to it.

```
logits = iris_classifier(data[chosen_indexes])
print(logits)

"""
--------
output:

tensor([[ 0.7939, -0.1909,  0.1670],
        [ 0.8980, -0.1740,  0.1619],
        [ 0.7493, -0.1995,  0.1764],
        [ 0.7270, -0.2024,  0.1689],
        [ 0.7400, -0.2674,  0.1978],
        [ 0.9774, -0.2836,  0.1797],
        [ 0.8546, -0.2658,  0.2126],
        [ 1.1355, -0.3332,  0.1992],
        [ 1.0169, -0.2975,  0.2015],
        [ 1.1078, -0.3330,  0.1814]], grad_fn=<AddmmBackward0>)
"""
```

Now, we have an output. Let's compare it with the targets that we have.

```
predictions = logits.argmax(dim=1)
for prediction, true_label in zip(predictions,
 ↪  target[chosen_indexes]):
    print(prediction.item(), true_label.item())

"""
--------
output:

0 0.0
0 0.0
```

```
0 0.0
0 0.0
0 1.0
0 1.0
0 1.0
0 2.0
0 2.0
0 2.0
"""
```

In the code above, at first, I used `argmax` as we used in the `Hello World` example. Then, zipped the `predictions` and the chosen `targets` to iterate through them. After that, I printed them beside each other to see how close my predictions are to the true labels. (`.item` function returns the value of a single tensor) As you can see, all the prediction classes are `0`. The reason behind that is that we haven't trained our model yet.

Dataset

The standard way of creating a **dataset** in **PyTorch** is by using `torch.utils.data.Dataset`. In this way, data is more manageable and can be dealt with in so many different ways. Let's make a `Dataset` class for our `IRIS` dataset.

```
class IRISDataset(Dataset):
    def __init__(self, data, target):
        super().__init__()
        self.data = data
        self.target = target

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        data = torch.tensor(self.data[idx]).to(torch.float)
        target = torch.tensor(self.target[idx])
        return data, target
```

In the code above, we have a class that is an abstract of `Dataset`, called `IRISDataset`. As you can see, we gave `data` and `target` as arguments to this class. When we implement a `Dataset` in **PyTorch**, we have to implement `__len__` and `__getitem__` as well. The function `__len__` returns the size of our data (`len(self.data)`). Also, the function `__getitem__` returns each data and target with the given index. We should make sure that we transform our data and target correctly before returning. To do so, I transformed `data` to a `float Tensor` and `target` to a `Tensor`. This function is used when we want to iterate over our dataset. Let's load our data again and create an instance of our `IRISDataset`.

```
iris = load_iris()

iris_dataset = IRISDataset(iris.data, iris.target)
```

Now, if we want to iterate over our dataset, we can use a simple for. For example, in the code below, we iterate over our dataset and `break` the loop after `one` element.

```
for one_data, one_target in iris_dataset:
    print(one_data)
    print(one_target)
    break

"""
--------
output:

tensor([5.1000, 3.5000, 1.4000, 0.2000])
tensor(0.)
"""
```

DataLoader

In **PyTorch**, we have a class called `DataLoader`. This class is super useful when you want to train your model. It gives you so many options that you can control pretty easily. Let's create a `DataLoader` for our `iris_dataset`.

```
from torch.utils.data import DataLoader

iris_loader = DataLoader(iris_dataset, batch_size=10,
 ↪  shuffle=True)
```

In the code above, we created an instance of `DataLoader` and stored it in `iris_loader`. We set the `batch_size` to `10`. This means in each iteration, our `Dataloader`, returns `10` samples of data. Also, we set `suffle` to true. This argument shuffles the order of data every time, which is super useful in training. Now, let's make a loop that iterates over `iris_loader`, and shows only the first element.

```
for batch_of_data, batch_of_target in iris_loader:
    print(batch_of_data)
    print(batch_of_target)
    break
```

```
"""
--------
output:

tensor([[6.4000, 2.9000, 4.3000, 1.3000],
        [6.4000, 3.1000, 5.5000, 1.8000],
        [7.7000, 2.6000, 6.9000, 2.3000],
        [4.8000, 3.4000, 1.9000, 0.2000],
        [4.6000, 3.2000, 1.4000, 0.2000],
        [6.7000, 3.1000, 4.4000, 1.4000],
        [6.2000, 2.8000, 4.8000, 1.8000],
        [6.1000, 3.0000, 4.6000, 1.4000],
        [5.7000, 2.8000, 4.1000, 1.3000],
        [5.4000, 3.9000, 1.3000, 0.4000]])
tensor([1., 2., 2., 0., 0., 1., 2., 1., 1., 0.])
"""
```

As you can see, there are 10 samples of `data` with their `target`. If you run this loop multiple times, you will get different output every time. The reason behind that is that we set the `suffle` to `True` in our data loader.

Train, Validation, and Test data

When we want to train our model, it is recommended to have 3 sets of data:

- **Train**: The data that the model is trained on
- **Validation**: The data that the model doesn't train on, and it is being used to evaluate the model after each `epoch`
- **Test**: The completely unseen data to evaluate our model after the training is over.

There are so many different ways that we can split our data. One of the ways is using `random_split` in `pytorch.utils.data`. To do so, we can use the code below:

```
from torch.utils.data import random_split

g1 = torch.Generator().manual_seed(20)
train_data, val_data, test_data = random_split(iris_dataset,
 ↪  [0.7, 0.2, 0.1], g1)
```

In the code above, at first, we create a `seed`. This `seed`, makes sure that every time we use our code, we get the same `train`, `validation`, and `test` subsets of our data. Then we split our data using `random_split`. As you can see, `70%` of the data goes for `training`, `20%` goes for `validation`, and `10%` goes for `testing`. Now, let's print the size of each subset to see if it works correctly.

```python
print("train_data length:", len(train_data))
print("val_data length:", len(val_data))
print("test_data length:", len(test_data))

"""
--------
output:
train_data length: 105
val_data length: 30
test_data length: 15
"""
```

As you can see, the data lengths are correct. Now, let's create a `DataLoader` for each of them.

```python
train_loader = DataLoader(train_data, batch_size=10,
 ↪  shuffle=True)
val_loader = DataLoader(val_data, batch_size=10, shuffle=False)
test_loader = DataLoader(test_data, batch_size=10, shuffle=False)
```

As you can see, now we have 3 dataloaders for each subset. Let's write a for loop to feed our training data to our model.

```python
for batch_of_data, batch_of_target in train_loader:
    logits = iris_classifier(batch_of_data)

    predictions = logits.argmax(dim=1)
    for prediction, true_label in zip(predictions,
     ↪  batch_of_target):
        print(prediction.item(), true_label.item())
    break

"""
--------
output:

1 1.0
1 2.0
0 0.0
1 1.0
0 0.0
1 1.0
1 1.0
0 0.0
1 2.0
1 2.0
"""
```

In the code above, we have a for loop that iterates over the `train_loader`. We feed each `batch_of_data` to our model to give us the `logits`. Then, we compare our predictions with the true labels. We put a `break` at the end of the for loop, to only show the first result. Now, we have everything to train our model.

Conclusion

In this tutorial, we have learned how to control data in **PyTorch**. We downloaded a traditional dataset. Then, we load that dataset as a `PyTorch Dataset`. After that, we created a `DataLoader` for that `Dataset`. Finally, we split our dataset into `train`, `validation`, and `test`. Now, we are ready to train our model.

AutoGrad, loss function, and optimizer

Introduction

Training a model is one of the most important features in **PyTorch**. In the previous tutorials, we prepared our **data** and our **model**. Now, we should learn about training fundamentals.

AutoGrad

One of the fundamental parts of each `Tensor` in `PyTorch` is that they can store gradients, using `requires_grad` argument. Let's define an equation with some tensors:

```
a = torch.tensor(3.0, requires_grad=True)
b = torch.tensor(2.0, requires_grad=True)

y = a ** 2 + b
```

In the code above, we have tensor `a` and tensor `b` with the values of `3` and `2`. As you can see, I set the `requires_grad` argument to true for both of them. Then, I have defined an equation, where:

$$y = a^2 + b$$

Now, let's calculate the gradient. To do so, we can use a function called `.backward()`. This function looks at the computational graph of the tensor and calculates the gradient of the tensors that require gradient. So, if I call the `.backward()` function for y, these gradients would be calculated $\frac{\delta y}{\delta a}$ and $\frac{\delta y}{\delta b}$. Before calling that function, let's calculate it ourselves.

$$\frac{\delta y}{\delta a} = \frac{\delta(a^2 + b)}{\delta a} = 2a \xrightarrow{a=3} 6$$

$$\frac{\delta y}{\delta b} = \frac{\delta(a^2 + b)}{\delta b} = 1$$

Now, let's see if we get the same results when we call the `.backward()` function for `y`.

```
y.backward()

print("dy/da: ", a.grad.item())  # d(a**2 + b)/da = 2*a
↪  ----a=3----> 6
print("dy/db: ", b.grad.item())  # d(a**2+b)/db = 1

"""
--------
output:

dy/da:  6.0
dy/db:  1.0
"""
```

As you can see, our results are the same. In **Deep Learning**, we use **gradient** to update the weights of our model. To do so, we can define a `loss function` as below:

$$l = (y - \hat{y})^2$$

- l: loss function
- y: true label
- $\hat{y}$: prediction

Now, let's have another example that is closer to what we want to do in **Deep Learning**.

```
w = torch.tensor(5.0, requires_grad=True)  # weight
b = torch.tensor(2.0, requires_grad=True)  # bias

x = 2  # input
y_true = 7  # true output

y_hat = w * x + b  # prediction

loss = (y_hat - y_true) ** 2  # calculate loss
loss.backward()  # calculate gradients
```

```python
print(f"d(loss)/dw: {w.grad.item()}")
print(f"d(loss)/db: {b.grad.item()}")

"""
--------
output:

d(loss)/dw: 20.0
d(loss)/db: 10.0
"""
```

In the example above, we have `w` that represents `weight`, and we also have `b` that represents `bias`. Our input is `2` and our expected output is `7`. We predict the output by multiplying the input (`x`) by `w`, and then add it to `b` to get the prediction that we want. For our loss function, we have the difference between the prediction and true output powered by 2. Then, we calculate the gradient of `loss` with respect to `w` and `b` and print them. Let's calculate the gradients ourselves to be able to check the results.

$$\frac{\delta l}{\delta w} = \frac{\delta (wx+b-y)^2}{\delta w} = \frac{\delta (wx+b-y)^2}{\delta (wx+b-y)} \frac{\delta (wx+b-y)}{\delta w} = 2(wx+b-y)x \xrightarrow{w=5,b=2,x=2,y=7} 2(5\times2+2-7)\times2 = 4(10$$

$$\frac{\delta l}{\delta b} = \frac{\delta (wx+b-y)^2}{\delta b} = \frac{\delta (wx+b-y)^2}{\delta (wx+b-y)} \frac{\delta (wx+b-y)}{\delta b} = 2(wx+b-y) \xrightarrow{w=5,b=2,x=2,y=7} 2(5\times2+2-7) = 2(10+2-$$

As you can see, the results are the same as our calculations.

Loss function

Now that we have an idea of how `AutoGrad` works, let's talk about a **loss function**. We have different **loss functions**, the one that we are going to explain right now is `CrossEntropyLoss`. If you want to know more about `CrossEntropyLoss`, you can check out this link: Cross Entropy Loss PyTorch. Now, let's define our loss function and test it to see how it works.

```python
y_true = torch.tensor([0, 1])
y = torch.tensor([
    [2.0, 8.0],
    [5.0, 5.0],
])

loss_fn = nn.CrossEntropyLoss()
loss = loss_fn(y, y_true)
```

```
print(loss.item())

"""
--------
output:
3.347811460494995
"""
```

In the code above, I have 2 classes (`1` and `0`). As you can see, the class of the first sample is `0` and the sample is `1`. My prediction for the first sample has a higher value for the class `1`. My second prediction has equal value for both of them. So, the loss output is not equal to zero. If I want my loss output to be zero, my predictions should look something like this:

```
y_true = torch.tensor([0, 1])
y = torch.tensor([
    [100.0, 0.0],
    [0.0, 100.0]
])

loss_fn = nn.CrossEntropyLoss()
loss = loss_fn(y, y_true)

print(loss.item())

"""
--------
output:

0.0
"""
```

As you can see, the prediction on each sample has a higher value with regard to its true class. So, as a result, the output of our loss function would be zero.

Optimizer

We have learned how to calculate the gradients of our loss function. Now, let's talk about how to update the weights of our model. To do that, we can use an `Optimizer`. One of the most famous `optimizers` is `Adam`. If you want to know more about it, you can take a look at this link: Pytorch Adam. When we want to create an instance of an `optimizer`, we should give it the tensors that it has to `optimize`. Let's define a simple model and make an `optimizer`.

```
from torch.optim import Adam
```

```
model = nn.Linear(4, 2)

optimizer = Adam(model.parameters())
```

In the code above, we have a simple linear model. We gave the parameters of that model to our `optimizer`. `Optimizer` will try to decrease the loss, using the calculated `gradients`. So, for each step of `optimization`, we should do something like below:

```
x = torch.tensor([
    [1.0, 2.0, 3.0, 4.0],
    [-1.0, -2.0, -3.0, -4.0],
])  # simple data
y_true = torch.tensor([0, 1])  # simple targe

for step in range(10):
    optimizer.zero_grad()  # clear the gradients

    logits = model(x)  # make a prediction

    loss = loss_fn(logits, y_true)  # calculate the loss
    print(f"step {step}, loss: {loss.item()}")

    loss.backward()  # calculate the gradients with respect to
 ↪  loss

    optimizer.step()  # optimize the weights

"""
--------
output:
step 0, loss: 0.021350994706153870
step 1, loss: 0.020931493490934372
step 2, loss: 0.020520454272627830
step 3, loss: 0.020117828622460365
step 4, loss: 0.019723571836948395
step 5, loss: 0.019337747246026993
step 6, loss: 0.018960254266858100
step 7, loss: 0.018590923398733140
step 8, loss: 0.018229883164167404
step 9, loss: 0.017876904457807540
"""
```

As you can see in the code above, we defined a simple dataset and a simple target. We run our `optimization` steps 10 times. In each step, first, we clear the previously calculated gradients using `optimizer.zero_grad()`.

Then, we make a prediction and calculate the loss with the `loss function` we have defined earlier (`Cross Entropy Loss`). After that, we calculate the gradients using `loss.backward()`. And finally, we `optimize` the `weights` using `optimizer.step()`. As you can see in the output, the loss is decreasing in each step, which means our `optimization` is working correctly.

Conclusion

In this tutorial, we have learned about training fundamentals. At first, we explained how to calculate the gradient. Then, we introduced the loss function. Finally, we programmed a simple optimization step to show how we can optimize our model's parameters.

Train

Introduction

In the previous tutorials, we have learned about **Model**, **Data**, and **Training fundamentals**. Now, let's combine them and train our model on **IRIS dataset**.

Load the data and make the model

Let's go step by step and load our data, and make our model, like the previous tutorial, to train it. First, let's load our data with the code below:

```
iris = load_iris()
```

Now, let's make a `Dataset` for our data.

```
class IRISDataset(Dataset):
    def __init__(self, data, target):
        super().__init__()
        self.data = data
        self.target = target

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        data = torch.tensor(self.data[idx]).to(torch.float)
        target = torch.tensor(self.target[idx])
        return data, target


iris_dataset = IRISDataset(iris.data, iris.target)
```

Then, it is time to split it into `train`, `validation`, and `test`.

```
g1 = torch.Generator().manual_seed(20)
train_data, val_data, test_data = random_split(iris_dataset,
 ↪  [0.7, 0.2, 0.1], g1)

train_loader = DataLoader(train_data, batch_size=10,
 ↪  shuffle=True)
val_loader = DataLoader(val_data, batch_size=10, shuffle=False)
test_loader = DataLoader(test_data, batch_size=10, shuffle=False)
```

Let's create our model as well.

```
class IRISClassifier(nn.Module):
    def __init__(self):
        super().__init__()

        self.layers = nn.Sequential(
            nn.Linear(4, 16),
            nn.Linear(16, 8),
            nn.Linear(8, 3),
        )

    def forward(self, x):
        return self.layers(x)


model = IRISClassifier()
```

Now, we are ready to start learning how to train our model.

Train the model

Right now, we know how to train our model in `PyTorch`. So, let's write an optimization step for our model. First, we need to define `loss function` and `optimizer`.

```
loss_fn = nn.CrossEntropyLoss()
optimizer = Adam(model.parameters())
```

Now, let's write our training loop.

```
model.train()

for batch_of_data, batch_of_target in train_loader:
    optimizer.zero_grad()

    logits = model(batch_of_data)
```

```
    loss = loss_fn(logits, batch_of_target)
    print(f"loss: {loss.item()}")

    loss.backward()

    optimizer.step()

"""
--------
output:

loss: 1.181538462638855
loss: 1.1570122241973877
loss: 1.1441924571990967
loss: 1.1753343343734741
loss: 1.1002519130706787
loss: 1.1666862964630127
loss: 1.0838695764541626
loss: 1.1226308345794678
loss: 1.1205450296401978
loss: 1.1404510736465454
loss: 1.0940010547637939
"""
```

At first, we make sure that our model is in `train` mode by using `model.train()` (When we freshly create a model, it is in `train` mode). Then, we write the code for the `optimization`. As you can see, for each batch of data, we calculated the loss and the gradients and optimized the weights. You might have noticed that the loss in each batch is not necessarily improving. Don't worry about it, because we are going to address it pretty soon.

Evaluate the model

Now, let's write a code to evaluate our model on validation dataset.

```
model.eval()

with torch.inference_mode():
    total_loss = 0

    for batch_of_data, batch_of_target in val_loader:
        logits = model(batch_of_data)

        loss = loss_fn(logits, batch_of_target)
        total_loss += loss.item()
```

```
    print(f"average_loss: {total_loss / len(val_loader)}")

"""
--------
output:

average_loss: 1.0044949253400166
"""
```

In the code above, at first, we set the model to the `evaluation` mode, using `model.eval()`. With `torch.inference_mode()`, we disable all the gradient calculations, because we don't need to train our model; we only need to evaluate it. Then, we iterate over our validation dataset. We give each `batch_of_data` to the model to predict the output. After that, we calculate the loss and add it to the total_loss. And finally, we calculate the `average_loss` by dividing `total_loss` by the number of batches, which can be accessed by `len(val_loader)`.

Now, let's add accuracy to this as well. We can calculate the accuracy by dividing the number of correct predictions by the total number of all samples. To do so, we can change our code as below:

```
model.eval()

with torch.inference_mode():
    total_loss = 0
    total_correct = 0

    for batch_of_data, batch_of_target in val_loader:
        logits = model(batch_of_data)

        loss = loss_fn(logits, batch_of_target)
        total_loss += loss.item()

        predictions = logits.argmax(dim=1)
        total_correct +=
↪   predictions.eq(batch_of_target).sum().item()

    print(f"average_loss: {total_loss / len(val_loader)}")
    print(f"accuracy: {total_correct / len(val_loader.dataset)}")

"""
--------
output:
```

```
average_loss: 1.0044949253400166
accuracy: 0.6333333333333333
"""
```

As you can see, I added a variable called `total_correct` which calculates the total number of correct predictions. To calculate if our prediction is wrong or right, as we have done before, at first, we can perform `argmax` on dimension 1 (Right now, we have two dimensions, 0 and 1). Then, we check our prediction against the correct target. Finally, we divide `total_correct` by the total number of samples, which can be accessed with `len(val_loader.dataset)`.

make train_step and val_step

Now, for convenience, let's put our **Training step** and **Validation step** into their functions. Let's start with **Training step**.

```
def train_step():
    model.train()

    total_loss = 0

    for batch_of_data, batch_of_target in train_loader:
        optimizer.zero_grad()

        logits = model(batch_of_data)

        loss = loss_fn(logits, batch_of_target)
        total_loss += loss.item()

        loss.backward()

        optimizer.step()

    print(f"training average_loss: {total_loss /
     ↪  len(train_loader)}")
```

As you can see, in the example above, I copied the code that we had written before, with only two changes. First, I removed the printing of `loss` in each batch, to make the output more clean. Second, I calculate `average_loss` like we did in the evaluation. Now, let's add **Validation step**.

```
def val_step():
    model.eval()

    with torch.inference_mode():
        total_loss = 0
        total_correct = 0
```

```
        for batch_of_data, batch_of_target in val_loader:
            logits = model(batch_of_data)

            loss = loss_fn(logits, batch_of_target)
            total_loss += loss.item()

            predictions = logits.argmax(dim=1)
            total_correct +=
↪   predictions.eq(batch_of_target).sum().item()

        print(f"validation average_loss: {total_loss /
         ↪  len(val_loader)}")
        print(f"validation accuracy: {total_correct /
         ↪  len(val_loader.dataset)}")
```

As you can see in the code above, I just copied the code we have written previously. Now, let's test them to see if they are working correctly.

```
train_step()

"""
--------
output:

training average_loss: 1.1503298336809331
"""
```

```
val_step()

"""
--------
output:

validation average_loss: 1.2322160800298054
validation accuracy: 0.23333333333333334
"""
```

Epoch

Now that we have our **Training and Validation step** ready, let's talk about `epoch`. When we train our model on all the batches for one time, we take one `epoch`. If we repeat this loop for `n` times, we took `n epochs`. Let's create a fresh model, define our loss function, give the model's parameters to our optimizer, and train our model for `5 epochs`

```python
model = IRISClassifier()

loss_fn = nn.CrossEntropyLoss()
optimizer = Adam(model.parameters())

for epoch in range(5):
    print("-" * 20)
    print(f"epoch: {epoch}")
    train_step()
    val_step()

"""
--------
output:

--------------------
epoch: 0
training average_loss: 1.1236063025214456
validation average_loss: 1.0980798403422039
validation accuracy: 0.2
--------------------
epoch: 1
training average_loss: 1.0682959123091265
validation average_loss: 1.043296257654826
validation accuracy: 0.5333333333333333
--------------------
epoch: 2
training average_loss: 1.0306043733250012
validation average_loss: 1.0079283316930134
validation accuracy: 0.6
--------------------
epoch: 3
training average_loss: 0.991635187105699
validation average_loss: 0.9691224495569865
validation accuracy: 0.8333333333333334
--------------------
epoch: 4
training average_loss: 0.9554464546116915
validation average_loss: 0.9225329160690308
validation accuracy: 0.7666666666666667
"""
```

As you can see, in the code above, we have trained and evaluated our model in each `epoch`. Your results and outputs might be different from mine. Because we are working on a small dataset, we haven't learned all the layers and training techniques, so the training results might seem a little bit random. But don't worry about it, we are going to fix that pretty soon.

Run on Accelerator

We learned how to find the available accelerator in the previous tutorials. Now, we are going to do that, and also make some changes in the code in order to train and evaluate our model on the accelerator.

```python
if torch.accelerator.is_available():
    device = torch.accelerator.current_accelerator()
else:
    device = "cpu"

print(device)

"""
--------
output:

mps
"""
```

In the code above, I have found the current accelerator, which for me is `mps`. Now, let's change our `train_step`.

```python
def train_step():
    model.train()

    total_loss = 0

    for batch_of_data, batch_of_target in train_loader:
        batch_of_data = batch_of_data.to(device)
        batch_of_target = batch_of_target.to(device)

        optimizer.zero_grad()

        logits = model(batch_of_data)

        loss = loss_fn(logits, batch_of_target)
        total_loss += loss.item()

        loss.backward()

        optimizer.step()

    print(f"training average_loss: {total_loss /
    ↪  len(train_loader)}")
```

As you can see, I changed the `device` of `batch_of_data` and `batch_of_target` to the current device. I should do the same for my `val_step` as well.

```
def val_step():
    model.eval()

    with torch.inference_mode():
        total_loss = 0
        total_correct = 0

        for batch_of_data, batch_of_target in val_loader:
            batch_of_data = batch_of_data.to(device)
            batch_of_target = batch_of_target.to(device)

            logits = model(batch_of_data)

            loss = loss_fn(logits, batch_of_target)
            total_loss += loss.item()

            predictions = logits.argmax(dim=1)
            total_correct +=
 ↪  predictions.eq(batch_of_target).sum().item()

        print(f"validation average_loss: {total_loss /
         ↪  len(val_loader)}")
        print(f"validation accuracy: {total_correct /
         ↪  len(val_loader.dataset)}")
```

Now, I should only change the `device` of the `model` too and run the training procedure again.

```
model = IRISClassifier()
model.to(device)

loss_fn = nn.CrossEntropyLoss()
optimizer = Adam(model.parameters())

for epoch in range(5):
    print("-" * 20)
    print(f"epoch: {epoch}")
    train_step()
    val_step()

"""
--------
output:
```

```
--------------------
epoch: 0
training average_loss: 1.1559315432201733
validation average_loss: 1.0502928098042805
validation accuracy: 0.36666666666666664
--------------------
epoch: 1
training average_loss: 1.078606204553084
validation average_loss: 1.0400715271631877
validation accuracy: 0.36666666666666664
--------------------
epoch: 2
training average_loss: 1.0253016406839544
validation average_loss: 1.0179588794708252
validation accuracy: 0.2
--------------------
epoch: 3
training average_loss: 0.9952371987429532
validation average_loss: 0.9651865760485331
validation accuracy: 0.6
--------------------
epoch: 4
training average_loss: 0.9447547793388367
validation average_loss: 0.9108109474182129
validation accuracy: 0.7666666666666667
"""
```

As you can see, everything is working correctly.

Save and load our model

Now, to save our model, we can use `torch.save` function.

```
torch.save(model.state_dict(), "model.pth")
```

With the code above, we save all the weights of our model to a file called `model.pth`. Now, let's load it into a new model, using `torch.load`.

```
new_model = IRISClassifier()

weights = torch.load("model.pth")

new_model.load_state_dict(weights)

new_model = new_model.to(device)
```

In the code above, I have created a new instance of our model with the name of `new_model`. Then, I loaded the saved weights with `torch.load`. After that, I used `load_state_dict` to load the weights. Finally, I changed the device of our model to the current accelerator. To test if we have done everything correctly, we can use the code below:

```
for key in new_model.state_dict().keys():
    if key not in model.state_dict().keys():
        print(f"Key {key} not in model.state_dict()")
        break

    if not torch.allclose(new_model.state_dict()[key],
     ↪  model.state_dict()[key]):
        print("Values are different")
        break
```

In the code above, we check if all the layers and weights that we loaded are the same as the model that we used for saving.

Conclusion

In this tutorial, we have trained a simple model with simple layers. The outputs right now are pretty random. But moving forward, we are going to learn more about the different layers and how to get better results. Right now, we know what a simple **Deep Learning** project looks like. We trained our model and then evaluated it. We learned about **Epoch** and learned how to use the accelerator. Finally, we learned how to save our model and load it again.

Train 2

Introduction

In the previous tutorial, we have learned how to train our model. But our model wasn't getting properly trained. In this tutorial, we want to address that problem and try to solve it.

Modular train step and validation step

In the previous tutorial, we wrote a code to train our model as below:

```
# -------------------[ Imports ]-------------------
import torch
from torch import nn
from torch.optim import Adam
from torch.utils.data import Dataset, DataLoader, random_split

from sklearn.datasets import load_iris
```

```python
# -------------------[ Find the device ]-------------------
if torch.accelerator.is_available():
    device = torch.accelerator.current_accelerator()
else:
    device = "cpu"

print(device)

# -------------------[ Load the data ]-------------------
iris = load_iris()


class IRISDataset(Dataset):
    def __init__(self, data, target):
        super().__init__()
        self.data = data
        self.target = target

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        data = torch.tensor(self.data[idx]).to(torch.float)
        target = torch.tensor(self.target[idx])
        return data, target


iris_dataset = IRISDataset(iris.data, iris.target)

# -------------------[ Split the data to train, validation, and
↪ test ]-------------------
g1 = torch.Generator().manual_seed(20)
train_data, val_data, test_data = random_split(iris_dataset,
↪ [0.7, 0.2, 0.1], g1)

train_loader = DataLoader(train_data, batch_size=10,
↪ shuffle=True)
val_loader = DataLoader(val_data, batch_size=10, shuffle=False)
test_loader = DataLoader(test_data, batch_size=10, shuffle=False)


# -------------------[ Define model ]-------------------
class IRISClassifier(nn.Module):
    def __init__(self):
        super().__init__()
```

```python
        self.layers = nn.Sequential(
            nn.Linear(4, 16),
            nn.Linear(16, 8),
            nn.Linear(8, 3),
        )

    def forward(self, x):
        return self.layers(x)


# -------------------[ Train step ]-------------------
def train_step():
    model.train()

    total_loss = 0

    for batch_of_data, batch_of_target in train_loader:
        batch_of_data = batch_of_data.to(device)
        batch_of_target = batch_of_target.to(device)

        optimizer.zero_grad()

        logits = model(batch_of_data)

        loss = loss_fn(logits, batch_of_target)
        total_loss += loss.item()

        loss.backward()

        optimizer.step()

    print(f"training average_loss: {total_loss /
     ↪  len(train_loader)}")


# -------------------[ Validation step ]-------------------
def val_step():
    model.eval()

    with torch.inference_mode():
        total_loss = 0
        total_correct = 0

        for batch_of_data, batch_of_target in val_loader:
            batch_of_data = batch_of_data.to(device)
            batch_of_target = batch_of_target.to(device)
```

```
            logits = model(batch_of_data)

            loss = loss_fn(logits, batch_of_target)
            total_loss += loss.item()

            predictions = logits.argmax(dim=1)
            total_correct +=
 ↪  predictions.eq(batch_of_target).sum().item()

        print(f"validation average_loss: {total_loss /
         ↪  len(val_loader)}")
        print(f"validation accuracy: {total_correct /
         ↪  len(val_loader.dataset)}")


# -------------------[ Create a model ]-------------------
model = IRISClassifier()
model.to(device)

# -------------------[ Define loss function and optimizer
 ↪  ]-------------------
loss_fn = nn.CrossEntropyLoss()
optimizer = Adam(model.parameters())

# -------------------[ Train the model ]-------------------
for epoch in range(5):
    print("-" * 20)
    print(f"epoch: {epoch}")
    train_step()
    val_step()
```

Let's put that code in a file called train_v1.py. Right now, `train_step` and `val_step` only work with the global variables. Let's make them more modular.

```
# -------------------[ Define Training step ]-------------------
def train_step(
        data_loader: DataLoader,
        model: nn.Module,
        optimizer: Optimizer,
        loss_fn: nn.Module,
        device: str,
) -> tuple[float, float]:
    model.train()

    total_loss = 0
```

```
    total_correct = 0

    for batch_of_data, batch_of_target in data_loader:
        batch_of_data = batch_of_data.to(device)
        batch_of_target = batch_of_target.to(device)

        optimizer.zero_grad()

        logits = model(batch_of_data)

        loss = loss_fn(logits, batch_of_target)
        total_loss += loss.item()

        predictions = logits.argmax(dim=1)
        total_correct +=
 ↪  predictions.eq(batch_of_target).sum().item()

        loss.backward()

        optimizer.step()

    return total_loss / len(data_loader), total_correct /
     ↪  len(data_loader.dataset)


# -------------------[ Define Validation Step
 ↪  ]-------------------
def val_step(
        data_loader: DataLoader,
        model: nn.Module,
        loss_fn: nn.Module,
        device: str,
) -> tuple[float, float]:
    model.eval()

    with torch.inference_mode():
        total_loss = 0
        total_correct = 0

        for batch_of_data, batch_of_target in data_loader:
            batch_of_data = batch_of_data.to(device)
            batch_of_target = batch_of_target.to(device)

            logits = model(batch_of_data)

            loss = loss_fn(logits, batch_of_target)
```

```
            total_loss += loss.item()

            predictions = logits.argmax(dim=1)
            total_correct +=
↪   predictions.eq(batch_of_target).sum().item()

        return total_loss / len(data_loader), total_correct /
        ↪   len(data_loader.dataset)
```

As you can see, in the code above, we now give the needed arguments to `train_step` and `val_step` to work with. Also, instead of printing the results in each function, now I return the results. For both functions, I return `average loss` and `accuracy`. Now, let's make our code more organized and put it in a file named train_v2.py.

```
# -------------------[ Imports ]-------------------
import torch
from torch import nn
from torch.optim import Adam, Optimizer
from torch.utils.data import Dataset, DataLoader, random_split

from sklearn.datasets import load_iris


# -------------------[ Define Dataset ]-------------------
class IRISDataset(Dataset):
    def __init__(self, data, target):
        super().__init__()
        self.data = data
        self.target = target

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        data = torch.tensor(self.data[idx]).to(torch.float)
        target = torch.tensor(self.target[idx])
        return data, target


# -------------------[ Define Model ]-------------------
class IRISClassifier(nn.Module):
    def __init__(self):
        super().__init__()

        self.layers = nn.Sequential(
```

```
            nn.Linear(4, 16),
            nn.Linear(16, 8),
            nn.Linear(8, 3),
        )

    def forward(self, x):
        return self.layers(x)


# -------------------[ Define Training step ]-------------------
def train_step(
        data_loader: DataLoader,
        model: nn.Module,
        optimizer: Optimizer,
        loss_fn: nn.Module,
        device: str,
) -> tuple[float, float]:
    model.train()

    total_loss = 0
    total_correct = 0

    for batch_of_data, batch_of_target in data_loader:
        batch_of_data = batch_of_data.to(device)
        batch_of_target = batch_of_target.to(device)

        optimizer.zero_grad()

        logits = model(batch_of_data)

        loss = loss_fn(logits, batch_of_target)
        total_loss += loss.item()

        predictions = logits.argmax(dim=1)
        total_correct +=
 ↪  predictions.eq(batch_of_target).sum().item()

        loss.backward()

        optimizer.step()

    return total_loss / len(data_loader), total_correct /
     ↪  len(data_loader.dataset)


# -------------------[ Define Validation Step
 ↪  ]-------------------
```

```python
def val_step(
        data_loader: DataLoader,
        model: nn.Module,
        loss_fn: nn.Module,
        device: str,
) -> tuple[float, float]:
    model.eval()

    with torch.inference_mode():
        total_loss = 0
        total_correct = 0

        for batch_of_data, batch_of_target in data_loader:
            batch_of_data = batch_of_data.to(device)
            batch_of_target = batch_of_target.to(device)

            logits = model(batch_of_data)

            loss = loss_fn(logits, batch_of_target)
            total_loss += loss.item()

            predictions = logits.argmax(dim=1)
            total_correct +=
 ↪  predictions.eq(batch_of_target).sum().item()

        return total_loss / len(data_loader), total_correct /
         ↪  len(data_loader.dataset)


def main():
    # -------------------[ Find the accelerator
     ↪  ]-------------------
    if torch.accelerator.is_available():
        device = torch.accelerator.current_accelerator()
    else:
        device = "cpu"

    print(device)

    # -------------------[ Load the data ]-------------------
    iris = load_iris()

    iris_dataset = IRISDataset(iris.data, iris.target)

    # -------------------[ Split the data to train, validation,
     ↪  and test ]-------------------
```

```
    g1 = torch.Generator().manual_seed(20)
    train_data, val_data, test_data = random_split(iris_dataset,
↪   [0.7, 0.2, 0.1], g1)

    train_loader = DataLoader(train_data, batch_size=10,
↪   shuffle=True)
    val_loader = DataLoader(val_data, batch_size=10,
↪   shuffle=False)
    test_loader = DataLoader(test_data, batch_size=10,
↪   shuffle=False)

    # -------------------[ Create the model ]-------------------
    model = IRISClassifier()
    model.to(device)

    # -------------------[ Define loss function and optimizer
    ↪   ]-------------------
    loss_fn = nn.CrossEntropyLoss()
    optimizer = Adam(model.parameters())

    # -------------------[ Train and evaluate the model
    ↪   ]-------------------
    for epoch in range(5):
        print("-" * 20)
        print(f"epoch: {epoch}")
        train_loss, train_accuracy = train_step(train_loader,
↪   model, optimizer, loss_fn, device)
        val_loss, val_accuracy = val_step(val_loader, model,
↪   loss_fn, device)
        print(f"train: ")
        print(f"\tloss: {train_loss:.4f}")
        print(f"\taccuracy: {train_accuracy:.4f}")

        print(f"validation: ")
        print(f"\tloss: {val_loss:.4f}")
        print(f"\taccuracy: {val_accuracy:.4f}")

    print("-" * 20)
    test_loss, test_accuracy = val_step(test_loader, model,
↪   loss_fn, device)
    print(f"test: ")
    print(f"\tloss: {test_loss:.4f}")
    print(f"\taccuracy: {test_accuracy:.4f}")


if __name__ == "__main__":
```

```
    main()

"""
--------
output:
    mps
--------------------
epoch: 0
train:
    loss: 1.0473
    accuracy: 0.3714
validation:
    loss: 1.0471
    accuracy: 0.2333
--------------------
epoch: 1
train:
    loss: 0.9799
    accuracy: 0.4857
validation:
    loss: 0.9770
    accuracy: 0.6667
--------------------
epoch: 2
train:
    loss: 0.9447
    accuracy: 0.6571
validation:
    loss: 0.9077
    accuracy: 0.6667
--------------------
epoch: 3
train:
    loss: 0.9004
    accuracy: 0.7143
validation:
    loss: 0.8768
    accuracy: 0.6333
--------------------
epoch: 4
train:
    loss: 0.8546
    accuracy: 0.6857
validation:
    loss: 0.8063
    accuracy: 0.6667
```

```
--------------------
test:
    loss: 0.8586
    accuracy: 0.6000
"""
```

In the code above, I organized the code. I defined a main function, and separated the classes and functions with the code for running. I made the logging look more appealing. Also, at the end, I evaluated our model on `test` subset as well. As you can see, `training loss` and `training accuracy` are improving, but `validation loss` and `validation accuracy` might not necessarily.

Better splitting

We have learned how to split our dataset into `3` subsets (`train`, `validation`, `test`), using `random_split` in PyTorch, as below:

```python
class IRISDataset(Dataset):
    def __init__(self, data, target):
        super().__init__()
        self.data = data
        self.target = target

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        data = torch.tensor(self.data[idx]).to(torch.float)
        target = torch.tensor(self.target[idx])
        return data, target


# -------------------[ Load the data ]-------------------
iris = load_iris()

iris_dataset = IRISDataset(iris.data, iris.target)

# -------------------[ Split the data to train, validation, and
↪ test ]-------------------
g1 = torch.Generator().manual_seed(20)
train_data, val_data, test_data = random_split(iris_dataset,
↪ [0.7, 0.2, 0.1], g1)
```

Now, let's see how the labels are distributed.

```python
label_count = {
    0: 0,
```

```
    1: 0,
    2: 0,
}

for data, target in train_data:
    label_count[target.item()] += 1

print(f"train label count: {label_count}")

"""
--------
output:

train label count: {0: 33, 1: 39, 2: 33}
"""
```

```
label_count = {
    0: 0,
    1: 0,
    2: 0,
}

for data, target in val_data:
    label_count[target.item()] += 1

print(f"validation label count: {label_count}")

"""
--------
output:

validation label count: {0: 13, 1: 6, 2: 11}
"""
```

```
label_count = {
    0: 0,
    1: 0,
    2: 0,
}

for data, target in test_data:
    label_count[target.item()] += 1

print(f"test label count: {label_count}")

"""
```

```
--------
output:

test label count: {0: 4, 1: 5, 2: 6}
"""
```

As you can see, the distribution of the labels isn't perfect. Let's fix that by using the `train_test_split` function in `scikit-learn`.

```
iris = load_iris()

data = iris.data
target = iris.target

train_subset, val_subset, train_target, val_target =
 ↪   train_test_split(
    data,
    target,
    test_size=0.3,
    random_state=42,
    stratify=target,
)
val_subset, test_subset, val_target, test_target =
 ↪   train_test_split(
    val_subset,
    val_target,
    test_size=0.33,
    random_state=42,
    stratify=val_target,
)

print("size of each subset: ")
print(f"\ttrain: {train_subset.shape[0]}")
print(f"\tval: {val_subset.shape[0]}")
print(f"\ttest: {test_subset.shape[0]}")

print("target distribution:")
print(f"\ttrain: {np.unique(train_target, return_counts=True)}")
print(f"\tval: {np.unique(val_target, return_counts=True)}")
print(f"\ttest: {np.unique(test_target, return_counts=True)}")

"""
--------
output:

size of each subset:
```

```
    train: 105
    val: 30
    test: 15
target distribution:
    train: (array([0, 1, 2]), array([35, 35, 35]))
    val: (array([0, 1, 2]), array([10, 10, 10]))
    test: (array([0, 1, 2]), array([5, 5, 5]))
"""
```

In the code above, first, we split our data into 2 subsets (`train`, `val`). As a result, our `train` would be 70 of the data, and `val` would be 30. Then we split the `val` into `val` and `test`. Then, our `val` would be 30 of all data, and `test` would be 30 of all the data. As you can see, we used the `stratify` argument as well. This argument forces the splitting to have equal distribution. As you can see, now we have `35` samples of each label for `train`, `10` samples of each label for `val`, and `5` samples of each label for `test`. Now, let's make a dataset out of them.

```
train_data = IRISDataset(train_subset, train_target)
val_data = IRISDataset(val_subset, val_target)
test_data = IRISDataset(test_sebset, test_target)
```

I have applied all the changes to train_v3.py.

Standard Scaler

One of the usual techniques in **Deep Learning** is to **Normalize** our data. Right now, every feature has a different **average** and **standard deviation** (**std**). Let's print them out.

```
print(f"Mean of the features:\n\t {train_subset.mean(axis=0)}")
print(f"Standard deviation of the features:\n\t
 ↪  {train_subset.std(axis=0)}")

"""
--------
output:

Mean of the features:
     [5.87333333 3.0552381  3.7847619  1.20571429]
Standard deviation of the features:
     [0.85882164 0.45502087 1.77553646 0.77383751]
"""
```

We want to change the **average** of each feature to `0` and their **std** to `1`. To do so, we can use `NormalScaler` in `scikit-learn`.

```python
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaler.fit(train_subset)

train_subset_normalized = scaler.transform(train_subset)
val_subset_normalized = scaler.transform(val_subset)
test_subset_normalized = scaler.transform(test_subset)

print(f"Mean of the features after scaling:")
print(f"\ttrain: {train_subset_normalized.mean(axis=0)}")
print(f"\tval: {val_subset_normalized.mean(axis=0)}")
print(f"\ttest: {test_subset_normalized.mean(axis=0)}")
print(f"Standard deviation of the features after scaling:")
print(f"\ttrain: {train_subset_normalized.std(axis=0)}")
print(f"\tval: {val_subset_normalized.std(axis=0)}")
print(f"\ttest: {test_subset_normalized.std(axis=0)}")

"""
--------
output:

Mean of the features after scaling:
    train: [ 2.38327876e-15 -1.12145742e-15 -1.37456184e-16
↪   -6.97854473e-17]
    val: [-0.14360762  0.06174494 -0.04398402 -0.02030696]
    test: [-0.06210059 -0.07744281 -0.06275769 -0.04184464]
Standard deviation of the features after scaling:
    train: [1. 1. 1. 1.]
    val: [0.88306745 0.81063775 0.97257027 0.93027831]
    test: [0.80131426 0.8871022  0.96009651 0.9513319 ]
"""
```

In the code above, I have created an instance of `StandardScaler`. Then, I fitted my `scaler` only with `train_subset`. The reason for that was that we want `validation` and `test` subsets to be unseen. Then I have used the `transform` function to normalize each subset. As you can see, the average of each feature in `train_subset`, is now super close to zero, and the std of each of them is `1`. Because we only trained our scaler on `train_subset`, the average and the std of the `validation` and `test` subsets are not perfect. Now, let's make datasets from our normalized subsets.

```python
train_data = IRISDataset(train_subset_normalized, train_target)
val_data = IRISDataset(val_subset_normalized, val_target)
test_data = IRISDataset(test_subset_normalized, test_target)
```

Now, it's time to train and evaluate our model to see what happens. I have applied all the changes in train_v4.py. Let's run train_v4.py.

```
"""
--------
output:
mps
--------------------
epoch: 0
train:
    loss: 1.1541
    accuracy: 0.1238
validation:
    loss: 1.0946
    accuracy: 0.2667
--------------------
epoch: 1
train:
    loss: 1.0744
    accuracy: 0.3810
validation:
    loss: 1.0350
    accuracy: 0.6667
--------------------
epoch: 2
train:
    loss: 1.0080
    accuracy: 0.6571
validation:
    loss: 0.9773
    accuracy: 0.7000
--------------------
epoch: 3
train:
    loss: 0.9450
    accuracy: 0.7810
validation:
    loss: 0.9198
    accuracy: 0.7000
--------------------
epoch: 4
train:
    loss: 0.8759
    accuracy: 0.8000
validation:
    loss: 0.8617
    accuracy: 0.7333
```

```
--------------------
test:
    loss: 0.8406
    accuracy: 0.8000
"""
```

As you can see, right now our evaluation results are not random anymore.

Conclusion

In this tutorial, we have discussed 2 techniques that are being used to enhance our training. First, we explained how to split our data in a way that labels are equally distributed. Then, we introduced `StandardScaler`, which is one of the most important preprocessing techniques. Although they are not specifically `PyTorch` modules, they are being used in `PyTorch` projects.

Plot and TensorBoard

Introduction

In the previous tutorials, we were just printing our training and evaluation results. When our training epochs become larger or when we want to compare two methods with each other, looking at the numbers becomes devastating. One of the best ways to do that is to plot them. In this tutorial, we are going to first plot the results using `matplotlib`, then we will be using `TensorBoard` to achieve a better result.

Plot using matplotlib

To plot our results using `matplotlib`, the first thing that we should do is to make a list of our previous results in our training loop, like below:

```python
train_losses = []
train_accuracies = []

val_losses = []
val_accuracies = []

for epoch in range(20):
    print("-" * 20)
    print(f"epoch: {epoch}")

    train_loss, train_accuracy = train_step(train_loader, model,
    optimizer, loss_fn, device)
    train_losses.append(train_loss)
    train_accuracies.append(train_accuracy)
```

```
    val_loss, val_accuracy = val_step(val_loader, model, loss_fn,
↪   device)
    val_losses.append(val_loss)
    val_accuracies.append(val_accuracy)

    print(f"train: ")
    print(f"\tloss: {train_loss:.4f}")
    print(f"\taccuracy: {train_accuracy:.4f}")

    print(f"validation: ")
    print(f"\tloss: {val_loss:.4f}")
    print(f"\taccuracy: {val_accuracy:.4f}")
```

In the code above, I have created 4 lists (`train_losses`, `train_accuracies`, `val_losses`, `val_accuracies`). Each list is for a different result. As you can see, I have increased the epoch range to `20` as well. Now, let's plot our results.

```
from matplotlib import pyplot as plt

# -------------------[ Plot our results ]-------------------
plt.figure()
plt.title("loss")
plt.plot(train_losses, label="train")
plt.plot(val_losses, label="val")
plt.legend()

plt.figure()
plt.title("accuracy")
plt.plot(train_accuracies, label="train")
plt.plot(val_accuracies, label="val")
plt.legend()

plt.show()
```

In the code above, I plot losses and accuracies in different figures. I have put all the changed parts in train_plot.py. So, the output would be something like below:

As you can see, analyzing the plots is so much easier than examining the numbers.

TensorBoard

`TensorBoard` is one of the most used and greatest tools to keep track of our training. It has so many features, but we are going to focus on only plotting. To do so, we should have a `TensorBoard writer`. So, let's make one.

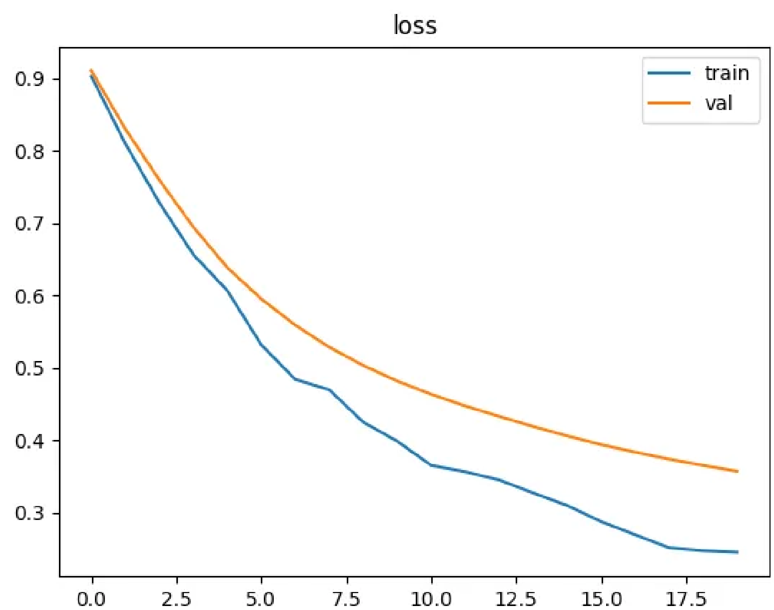

Figure 6: Loss

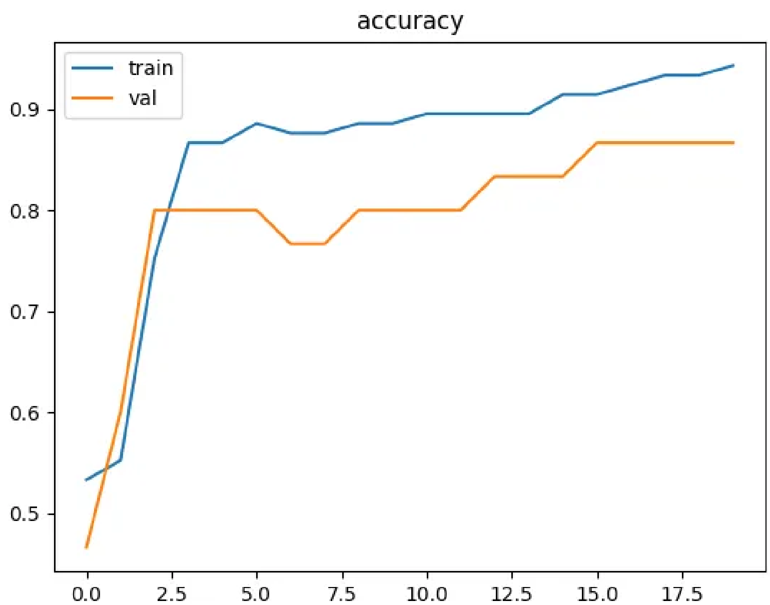

Figure 7: Accuracy

```python
from torch.utils.tensorboard import SummaryWriter

# -------------------[ Setup TensorBoard ]-------------------
writer = SummaryWriter()
```

In the code above, I have imported the `SummaryWriter` from `torch.utils.tensorboard`. Then, I have created an instance of `SummaryWriter` and called it `writer`. By default, `SummaryWriter` creates a log directory with the name of `runs` and stores the data into that directory. Now, let's write our training and evaluation results with `writer`.

```python
# -------------------[ Train and evaluate the model
↪  ]-------------------
for epoch in range(20):
    print("-" * 20)
    print(f"epoch: {epoch}")

    train_loss, train_accuracy = train_step(train_loader, model,
↪   optimizer, loss_fn, device)
    writer.add_scalar("loss/train", train_loss, epoch)
    writer.add_scalar("accuracy/train", train_accuracy, epoch)

    val_loss, val_accuracy = val_step(val_loader, model, loss_fn,
↪   device)
    writer.add_scalar("loss/val", val_loss, epoch)
    writer.add_scalar("accuracy/val", val_accuracy, epoch)

    print(f"train: ")
    print(f"\tloss: {train_loss:.4f}")
    print(f"\taccuracy: {train_accuracy:.4f}")

    print(f"validation: ")
    print(f"\tloss: {val_loss:.4f}")
    print(f"\taccuracy: {val_accuracy:.4f}")
```

As you can see, in our `training loop`, I used the `add_scalar` function of the `writer` to write the results. For each result, I have chosen different names.

- `train_loss` -> `loss/train`
- `train_accuary` -> `loss/accuracy`
- `val_loss` -> `loss/val`
- `val_accuary` -> `loss/accuracy`

I have applied the changes to train_tensorboard.py. Now, let's write our training script multiple times. For example, I have run it `3` times. After that, let's run our `TensorBoard` to see the results. To do so, we can run the command below:

```
tensorboard --log_dir runs
```

Or if you want to see the results on a notebook, you can use the code below:

```
%load_ext tensorboard
%tensorboard --logdir runs
```

The output would be something like below:

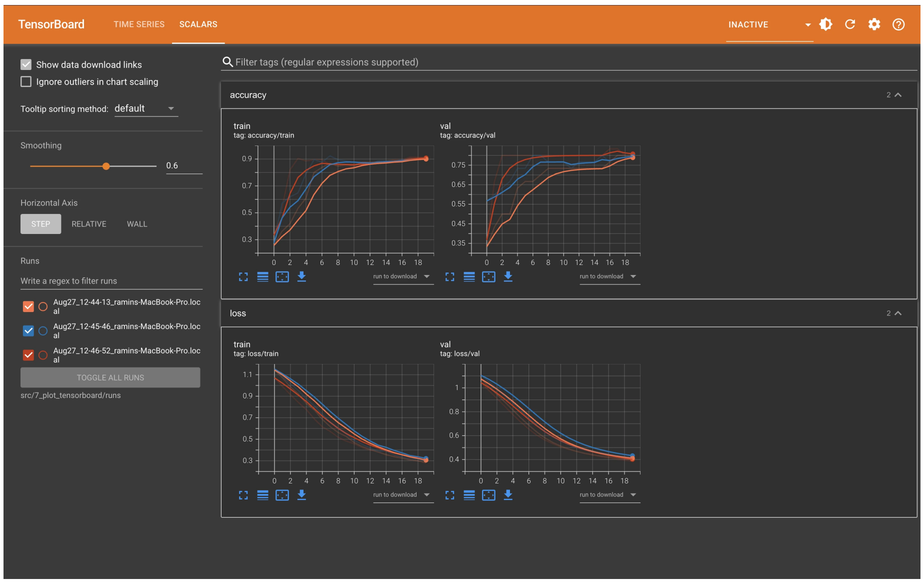

Figure 8: tensorboard result

As you can see, my `3` runs are being displayed separately. Right now, I can easily analyze my training. Also, because I write the results into files, I can see the results during the training process, which is extremely helpful

Conclusion

In this tutorial, we have learned how to plot our training and evaluation results. First, we plotted our results using `matplotlib`. Then, we learned how to use `TensorBoard` for better analysis.

Work with Images

Introduction

In the previous tutorials, we have learned how to work with one-dimensional data. In this tutorial, we are going to learn how to make a `dataloader` out of images.

Load a dataset

PyTorch has a built-in way to download and load some important datasets. This functionality is available with their TorchVision package. Let's download a minimal Dataset called MNIST. This dataset contains 0 to 9 handwritten numbers. To do so, we can use the code below:

```
from torchvision.datasets import MNIST

train_data = MNIST("data/", train=True, download=True)
test_data = MNIST("data/", train=False, download=True)
```

In the code above, we loaded MNIST in two subsets: train and test. The first argument is the path of the data that we want to load. In our case, we set that to data/. With the train argument, we can control whether we want to download train subset or test subset. When we set download to True, if the data is not available in the given path, it would download it. These subsets are the instances of Dataset. To make sure, we can check them with the code below:

```
print(isinstance(train_data, Dataset))

"""
--------
output:

True
"""
```

So, knowing this, we can do all the things with Dataset that we would do before. Let's now see the size of each dataset.

```
print(f"train_data's size: {len(train_data)}")
print(f"test_data's size: {len(test_data)}")

"""
--------
output:

train_data's size: 60000
test_data's size: 10000
"""
```

As you can see, we have 60000 data for training and 10000 data for testing. Now let's display one of the images.

```python
from matplotlib import pyplot as plt

for image, label in train_data:
    plt.imshow(image, cmap="gray")
    print(label)
    break

"""
--------
output:

5
"""
```

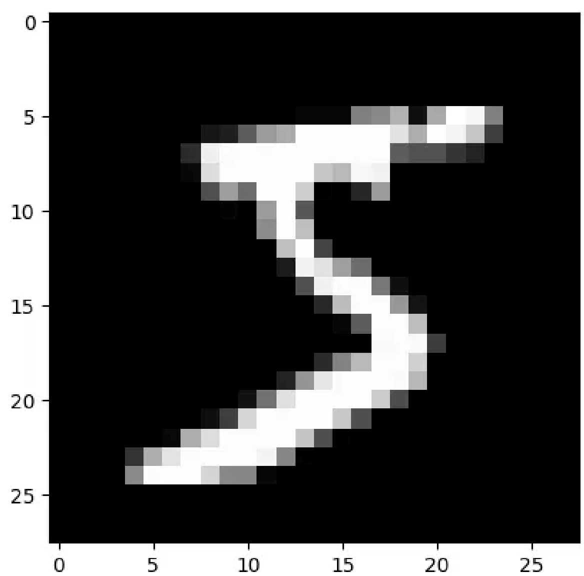

Figure 9: mnist sample

In the code above, we have displayed one sample of `MNIST` with its label.

Transforms

As you recall, in the previous tutorials, we had created a `Dataset` like below:

```python
class IRISDataset(Dataset):
    def __init__(self, data, target):
        super().__init__()
        self.data = data
        self.target = target

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        data = torch.tensor(self.data[idx]).to(torch.float)
        target = torch.tensor(self.target[idx])
        return data, target
```

In the `__getitem__` function, we were transforming our `data` and `target` to tensors to make them ready for our model. In `PyTorch`, it is a good practice to implement two more arguments for our `Dataset` called: `transform` and `target_transform`. `transform` is being used for transforming each sample of data, and `target_transform` is being used for transforming each target. In the code above, we have:

- `transfrom`: `torch.tensor(self.data[idx]).to(torch.float)`
- `target_transform`: `torch.tensor(self.target[idx])`

If we want to change our dataset to have these two arguments, we can do something like below:

```python
class IRISDataset(Dataset):
    def __init__(self, data, target, transform=None,
    ↪   target_transform=None):
        super().__init__()
        self.data = data
        self.target = target

        if transform is None:
            transform = lambda x: torch.tensor(x).to(torch.float)

        if target_transform is None:
            target_transform = lambda x: torch.tensor(x)

        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
```

```python
        data = self.transform(self.data[idx])
        target = self.target_transform(self.target[idx])
        return data, target
```

As you can see, in the code above, we have defined `transform` and `target_transform` as arguments. If they were `None`, we would have defined them as they were, using `lambda` function. `TorchVision` has provided us with some built-in transforms for images. You can find all the transforms in this link: TorchVision Transforms At first, we are going to use the `ToTensor` transform. This module transforms the `image` to `Tensor`. So, when we want to load our `MNIST`, we are going to add that as a transform.

```python
from torchvision import transforms

train_data = MNIST("data/", train=True, download=True,
 ↪  transform=transforms.ToTensor())
test_data = MNIST("data/", train=False, download=True,
 ↪  transform=transforms.ToTensor())
```

Now, let's see if it's applied or not.

```python
for image, label in train_data:
    print(type(image))
    break

"""
--------
output:

<class 'torch.Tensor'>
"""
```

As you can see, the type of our image is `Tensor`.

We can make a sequence of `transforms` using `transforms.Compose`. For example, let's first resize each image to `[14, 14]` (our current size is `[28, 28]`). Then, transform them into tensors.

```python
transform_compose = transforms.Compose(
    [
        transforms.Resize([14, 14]),
        transforms.ToTensor()
    ]
)
```

Now, let's test it to see if it works or not.

```python
# -------------------[ Before transform compose
 ↪  ]-------------------
for image, label in train_data:
    print(f"Before transform compose: {image.shape}")
    break

train_data = MNIST("data/", train=True, download=True,
 ↪  transform=transform_compose)
test_data = MNIST("data/", train=False, download=True,
 ↪  transform=transform_compose)

# -------------------[ After transform compose
 ↪  ]-------------------
for image, label in train_data:
    print(f"After transform compose: {image.shape}")
    break

"""
--------
output:

Before transform compose: torch.Size([1, 28, 28])
After transform compose: torch.Size([1, 14, 14])
"""
```

As you can see in the code above, it works as intended.

Train, validation, and test

We had `60000` data to train and `10000` data for testing. Now, let's make a `validation subset` as well. One of the ways to do that is to split `test subset` into two subsets.

```python
g1 = torch.Generator().manual_seed(20)
val_data, test_data = random_split(test_data, [0.7, 0.3], g1)

print(f"val_data's size: {len(val_data)}")
print(f"test_data's size: {len(test_data)}")

"""
--------
output:
val_data's size: 7000
test_data's size: 3000

"""
```

In the code above, I have divided the `test_data` into `val_data` and `test_data`. So, `70%` of the `10000` (10000×70) goes for validation, and the rest goes for testing. Now, let's make data loaders from them.

```
train_loader = DataLoader(train_data, batch_size=64,
 ↪  shuffle=True)
val_loader = DataLoader(val_data, batch_size=64, shuffle=False)
test_loader = DataLoader(test_data, batch_size=64, shuffle=False)
```

As you can see, we now have all 3 `dataloaders` which we needed to train our model.

ImageFolder

One of the ways to load an image dataset is with `ImageFolder`. `ImageFolder` requires your data to be in this structure:

- main_folder
  - class_1
    - image_1
    - image_2
    - …
  - class_2
    - image_3
    - image_4
    - …
  - …

As you can see, each class has its own directory and all its data is in that directory. Let's download a dataset from `Kaggle` with the name of `Tom and Jerry` in this link: Tom and Jerry. We can use the code below to do that:

```
import kagglehub
from pathlib import Path

path = kagglehub.dataset_download(⌋
 ↪  "balabaskar/tom-and-jerry-image-classification")
path = Path(path) / "tom_and_jerry/tom_and_jerry"
```

In the code above, I have downloaded the dataset using `kagglehub`, also I changed the path to the correct path to have the structure that we wanted. Now, let's see what classes we have:

```
for x in path.iterdir():
    print(x.name)

"""
--------
```

```
output:

tom
jerry
tom_jerry_1
tom_jerry_0
"""
```

As you can see, we have four classes:

- `tom`: when only Tom is in the picture
- `jerry`: when only Jerry is in the picture
- `tom_jerry_1`: when both of them are on the picture
- `tom_jerry_0`: when none of them are on the picture

Let's load this dataset using `ImageFolder`.

```
tom_and_jerry_transforms =
↪  transforms.Compose([transforms.Resize([90, 160]),
↪  transforms.ToTensor()])

all_data = ImageFolder(path, transform=tom_and_jerry_transforms)
```

In the code above, I have defined two transforms, one for resizing and one to transform each image into a tensor. Then, I loaded the data using `ImageFolder`. Now, let's display one of the images.

```
for image, label in all_data:
    plt.figure()
    plt.imshow(transforms.ToPILImage()(image))
    print(label)
    break

"""
--------
output:

0
"""
```

In the code above, I have displayed one image of our dataset. Images are currently in tensor format. To change them back to images, I used a transform called: `ToPILImage()`. Now, let's split them and make dataloaders:

```
g1 = torch.Generator().manual_seed(20)
train_data, val_data, test_data = random_split(all_data, [0.7,
↪  0.2, 0.1], g1)
```

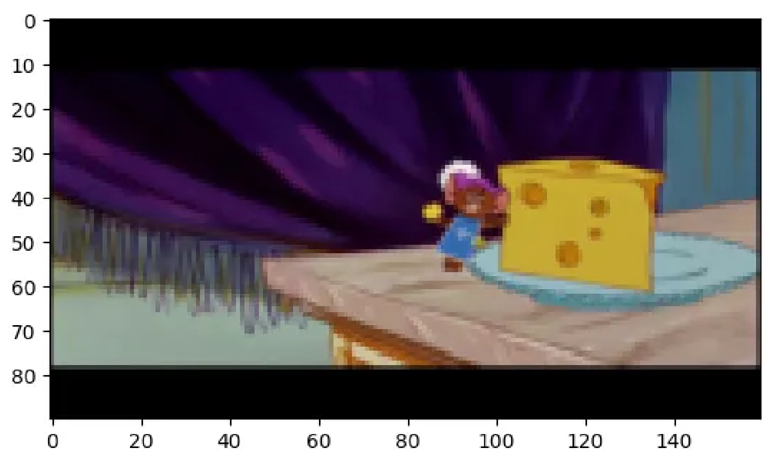

Figure 10: tom and jerry sample

```
train_loader = DataLoader(train_data, batch_size=16,
 ↪  shuffle=True)
val_loader = DataLoader(val_data, batch_size=16, shuffle=False)
test_loader = DataLoader(test_data, batch_size=16, shuffle=False)
```

And here you have it, we have our 3 dataloaders that we can work with.

Conclusion

In this tutorial, we have learned how to load and prepare image datasets. First, we used the built-in modules in `TorchVision`. Then, we explained `transforms` to prepare our dataset. Finally, we have learned how to work with `ImageFolder`.

Convolution and ReLU

Introduction

In the previous tutorial, we learned how to work with images. We learned how to load an image dataset and how to transform its images into tensors. In this tutorial, we are going to learn about a layer that is being widely used for images in **Deep Learning** called **Convolution**. Also, we are going to talk about `ReLU` and make you more familiar with how to work with any **layer**.

Convolution

Convolution is an operation in which we slide a smaller matrix (kernel) over a bigger matrix and calculate the weighted sum. Let's explain its concepts using an example. In our example, we have a `6x6` image, and our kernel is `3x3`, like below:

```
image_size = (6, 6)
kernel_size = (3, 3)

image = np.arange(image_size[0] *
 ↪  image_size[1]).reshape(image_size)
kernel = np.ones(kernel_size) / (kernel_size[0] * kernel_size[1])

print("image:")
print(image)
print("kernel:")
print(kernel)

"""
--------
output:
```

```
image:
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]
 [24 25 26 27 28 29]
 [30 31 32 33 34 35]]
kernel:
[[0.11111111 0.11111111 0.11111111]
 [0.11111111 0.11111111 0.11111111]
 [0.11111111 0.11111111 0.11111111]]
"""
```

As you can see, our image is the numbers from `0` to `35`, and our kernel is working as an average kernel. If we apply convolution, we are going to have a result like below:

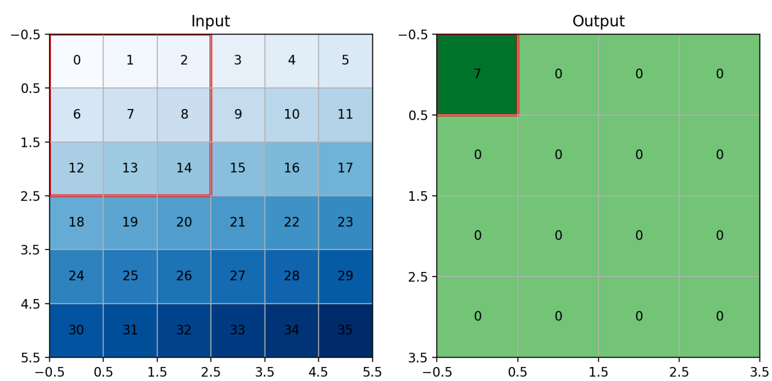

Figure 11: conv

As you can see in the GIF above, the kernel is being slid on our image, and we are getting the average of each `3x3` block as an output. Let's calculate the first block.

$$0 \times \frac{1}{9} + 1 \times \frac{1}{9} + 2 \times \frac{1}{9} + 6 \times \frac{1}{9} + 7 \times \frac{1}{9} + 8 \times \frac{1}{9} + 12 \times \frac{1}{9} + 13 \times \frac{1}{9} + 14 \times \frac{1}{9} = 7$$

As you can see, the calculations have the same results as the code. Also, our input's shape is `6x6`, but our output's shape is `4x4`. The reason behind that is our kernel is `3x3`. So, we can only slide it `4` times on our input. For now, we can calculate it like below:

$$W_{out} = (W_{in} - K_w) + 1$$

$$H_{out} = (H_{in} - K_h) + 1$$

- W: Width
- H: Height
- K: Kernel

Now, let's talk about 3 important things in **Convolution**. If you want to experience different convolutions with different options, you can use this code: conv_gif.py.

Stride

Right now, we are sliding our kernel `1` square at a time. If we decide to slide it with a number different from one, we can use `stride`.

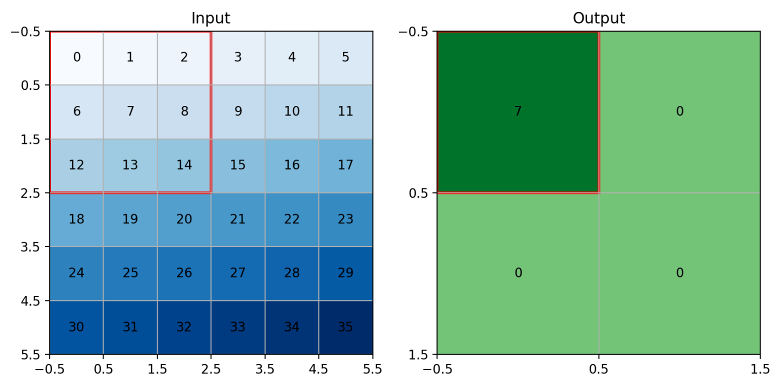

Figure 12: conv stride

As you can see in the GIF above, we put the stride to `2`. So, it slides `2` squares instead of `1` in both `x` and `y` axis. As a result, our output's shape becomes half of what it was. We can calculate the output's shape as below:

$$W_{out} = \frac{(W_{in} - K_w)}{S_w} + 1$$

$$H_{out} = \frac{(H_{in} - K_h)}{S_h} + 1$$

- W: Width

- H: Height
- K: Kernel
- S: Stride

padding

Padding is a technique that we use to fill the surrounding of the input with some values. The most common value for padding is `0`, which is called `zero padding`. The main reason for that is to prevent our image from being shrunk after some convolutions. In the previous example, you saw that the image with `6x6` becomes `4x4`. If the input shape and output shape are the same, it is called `zero-padding`.

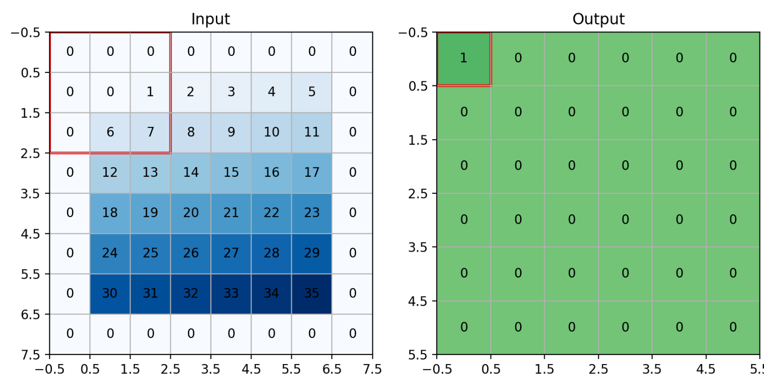

Figure 13: conv pad 1

As you can see in the GIF above, we have added zeros to the surroundings of our input. As a result, our output has the same shape as our input (`6x6`). We can calculate the output size as below:

$$W_{out} = \frac{(W_{in} + 2P_w - K_w)}{S_w} + 1$$

$$H_{out} = \frac{(H_{in} + 2P_h - K_h)}{S_h} + 1$$

- W: Width
- H: Height
- K: Kernel
- S: Stride
- P: Padding

Dilation

Dilation is a technique that we use to make the kernel bigger to cover a bigger area. To do so, we insert gaps between our kernel. For example, if our kernel is like below:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

After `dilation=2`, it becomes like below:

$$\begin{bmatrix} 1 & 0 & 2 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 5 & 0 & 6 \\ 0 & 0 & 0 & 0 & 0 \\ 7 & 0 & 8 & 0 & 9 \end{bmatrix}$$

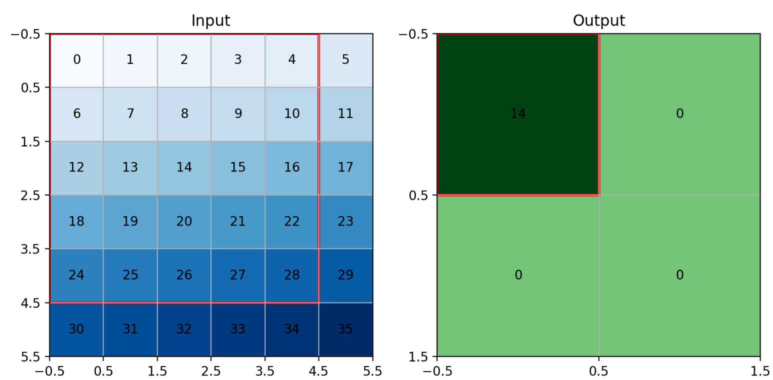

Figure 14: conv dilation 2

As you can see in the GIF above, we have `dilation=2`, so our kernel becomes `5x5`. We can calculate the output shape with the formula below:

$$W_{out} = \frac{(W_{in} + 2P_w - D_w \times (K_w - 1) - 1)}{S_w} + 1$$

$$H_{out} = \frac{(H_{in} + 2P_h - D_h \times (K_h - 1) - 1)}{S_h} + 1$$

- W: Width

- H: Height
- K: Kernel
- S: Stride
- P: Padding
- D: Dilation

Load MNIST

Now, let's load **MNIST** again like we did in the previous tutorial.

```
train_data = MNIST("data/", train=True, download=True,
 ↪  transform=transforms.ToTensor())
test_data = MNIST("data/", train=False, download=True,
 ↪  transform=transforms.ToTensor())
```

Now let's make `train`, `validation`, and `test` data loaders and see the shape of a batch of our data.

```
g1 = torch.Generator().manual_seed(20)
val_data, test_data = random_split(test_data, [0.7, 0.3], g1)

train_loader = DataLoader(train_data, batch_size=64,
 ↪  shuffle=True)
val_loader = DataLoader(val_data, batch_size=64, shuffle=False)
test_loader = DataLoader(test_data, batch_size=64, shuffle=False)

images, labels = next(iter(train_loader))

print(f"images shape : {images.shape}")
print(f"labels shape : {labels.shape}")

"""
--------
output:

images shape : torch.Size([64, 1, 28, 28])
labels shape : torch.Size([64])
"""
```

As you can see, we have a batch of our data with a batch size of `64`. Each image is grayscale, so it has `1` channel, and the size of the image is `28x28`.

Convolution layer

Earlier, we learned how `convolution` works. Now, let's talk about how to use it in **PyTorch**. We can define a `Convolution layer` in **PyTorch** like below:

```
conv_1 = nn.Conv2d(
    in_channels=1,
    out_channels=3,
    kernel_size=3,
    stride=1,
    padding=1,
    dilation=1,
)
```

In the code above, we have defined a `convolution layer`. This layer takes `1` channel as its input (because our data has `1` channel). For its output, it creates `3` channels. Also, it has a `3x3` kernel. As you can see, we have control over `stride`, `padding`, and `dilation`. Now, let's feed our loaded images to `conv_1`, to see what happens.

```
result = conv_1(images)
print(f"input shape : {images.shape}")
print(f"output shape : {result.shape}")

"""
--------
output:
input shape : torch.Size([64, 1, 28, 28])
output shape : torch.Size([64, 3, 28, 28])

"""
```

The results above show that the width and height of our inputs and outputs are the same. The reason behind that is that we put `padding` to `1`. Also, we have 3 channels for the results as expected.

ReLU

`ReLU` stands for `Rectified Linear Unit`. It is one of the most used activation functions in **Deep Learning**. The logic behind that is pretty simple. It only changes the negative values to `0`. Here is its formula:

$$ReLU(x) = max(0, x)$$

We can define `ReLU` in **PyTorch** as below:

```
relu = nn.ReLU()
```

Now let's test it to see how it works:

```
a1 = torch.arange(-5, 6)
result = relu(a1)
```

```python
print(f"input: {a1}")
print(f"output: {result}")

"""
--------
output:

input: tensor([-5, -4, -3, -2, -1,  0,  1,  2,  3,  4,  5])
output: tensor([0, 0, 0, 0, 0, 0, 1, 2, 3, 4, 5])
"""
```

In the code above, we have created a tensor called `a1` which has values in the range of `[-5, 5]`. We fed `a1` to `relu` and as a result, all the negative values have become zeros.

Flatten

`Flatten` is a layer that we use to change the multidimensional input to one dimension. It is pretty useful when we want to change the dimension of the output of our **convolution layers** to one dimension and feed it to our **linear layers** in order to classify them. We can define a `Flatten` layer in **PyTorch** like below:

```python
flatten = nn.Flatten()
```

Now, let's test it to see if it works as intended.

```python
a2 = torch.arange(0, 16).reshape((2, 2, 4)).unsqueeze(0)
result = flatten(a2)

print(f"input: {a2}")
print(f"input shape : {a2.shape}")
print(f"output: {result}")
print(f"output shape : {result.shape}")

"""
--------
output:

input: tensor([[[[ 0,  1,  2,  3],
          [ 4,  5,  6,  7]],

         [[ 8,  9, 10, 11],
          [12, 13, 14, 15]]]])
input shape: torch.Size([1, 2, 2, 4])
output: tensor([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,
 ↪  12, 13, 14, 15]])
```

```
output shape: torch.Size([1, 16])
"""
```

In the code above, we have defined an input called `a2` with the shape of `2x2x4`. The values in `a2` are in range of `[0, 16]`. Then we used `unsqueeze(0)` to add a dimension to the start of the tensor. We did that because each layer in **PyTorch** requires a batch of data, not a single data by itself. Then we fed that data to the `flatten` layer. As a result, we can see the input shape has changed from `2x2x4` to `16`. Also, all the data is untouched.

Make a convolution model

Now that we know how convolution works and know how to connect convolution with a linear model for classification, let's make a convolution model to classify the MNIST dataset.

```
# --------------------[ Define Model ]--------------------
class IRISClassifier(nn.Module):
    def __init__(self):
        super().__init__()

        self.conv_layers = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=32,
    kernel_size=3, padding=1, stride=2),   # 32x14x14
            nn.ReLU(),
            nn.Conv2d(in_channels=32, out_channels=64,
    kernel_size=3, padding=1, stride=2),   # 64x7x7
            nn.ReLU(),
            nn.Conv2d(in_channels=64, out_channels=128,
    kernel_size=3, padding=1, stride=3),   # 128x3x3
            nn.ReLU(),
        )

        self.classification_layers = nn.Sequential(
            nn.Flatten(),
            nn.Linear(128 * 3 * 3, 128),
            nn.ReLU(),
            nn.Linear(128, 10),
        )

    def forward(self, x):
        x = self.conv_layers(x)
        x = self.classification_layers(x)
        return x
```

In the code above, we have 2 parts for our model. The first part consists of

Convolution layers (`conv_layers`), and the other part has **Classification layers** (`classification_layers`). When we feed data to this model, it first goes through `conv_layers`, then it goes through `classification_layer`. For `conv_layers`, we have `3` **Convolution layers**. The first one takes the data with `1` channel and creates `32` channels as its output. Its kernel size is `3` with padding `1` and a stride of `2`, so we can calculate its output shape as below:

$$W_{out} = \frac{(W_{in} + 2P_w - K_w)}{S_w} + 1 \rightarrow \frac{(28 + 2 \times 1 - 3)}{2} + 1 = 13 + 1 \rightarrow \boxed{W_{out} = 14}$$

$$H_{out} = \frac{(H_{in} + 2P_h - K_h)}{S_h} + 1 \rightarrow \frac{(28 + 2 \times 1 - 3)}{2} + 1 = 13 + 1 \rightarrow \boxed{H_{out} = 14}$$

For the second convolution, we take 32 channels and make 64 channels. Kernel size is 3, padding is 1, and stride is 2. So, we can calculate the output shape as below:

$$W_{out} = \frac{(W_{in} + 2P_w - K_w)}{S_w} + 1 \rightarrow \frac{(14 + 2 \times 1 - 3)}{2} + 1 = 6 + 1 \rightarrow \boxed{W_{out} = 7}$$

$$H_{out} = \frac{(H_{in} + 2P_h - K_h)}{S_h} + 1 \rightarrow \frac{(14 + 2 \times 1 - 3)}{2} + 1 = 6 + 1 \rightarrow \boxed{H_{out} = 7}$$

And the third convolution has `64` input channels and makes `128` output channels. Its kernel size is `3`, its padding is `1`, and its stride is `3`. So, let's calculate the output shape of this convolution to:

$$W_{out} = \frac{(W_{in} + 2P_w - K_w)}{S_w} + 1 \rightarrow \frac{(7 + 2 \times 1 - 3)}{3} + 1 = 2 + 1 \rightarrow \boxed{W_{out} = 3}$$

$$H_{out} = \frac{(H_{in} + 2P_h - K_h)}{S_h} + 1 \rightarrow \frac{(7 + 2 \times 1 - 3)}{3} + 1 = 2 + 1 \rightarrow \boxed{H_{out} = 3}$$

Our classification layer has 2 **linear layers**. At first, we flatten the output of our `conv_layers`. The output was in the shape of $128 \times 3 \times 3$, so the flatten of that would be the multiplication of them. First, **linear layer** takes the $128 \times 3 \times 3$ and makes an output with 128 neurons. And the last **linear layer** takes 128 as its input shape and outputs the 10 class that we have for **MNIST**. Now, let's give a batch of **MNIST** images to see if it works or not:

```
model = IRISClassifier()
model(images)

"""
--------
output:

tensor([[-0.0223,  0.0049, -0.0598, -0.0597, -0.0689, -0.0711,
 ↪  0.0565, -0.0623,
          0.0433,  0.0466],
        [-0.0215,  0.0064, -0.0591, -0.0567, -0.0690, -0.0680,
 ↪  0.0531, -0.0552,
          0.0441,  0.0499],
          ...
        [-0.0225,  0.0070, -0.0598, -0.0565, -0.0709, -0.0740,
 ↪  0.0536, -0.0624,
          0.0413,  0.0421]], grad_fn=<AddmmBackward0>)
"""
```

As you can see, our model predicts 10 classes for each image, which is the thing that we wanted.

Train the model

Now, let's change the last code (train_tensorboard.py) And change the data to **MNIST** and change the model to our new **convolution model**. I have already done that, and the changes are in train_mnist_conv.py. So let's run it for 5 epochs and see the output.

```

"""
--------
output:

mps
--------------------
epoch: 0
train:
    loss: 0.2567
    accuracy: 0.9219
validation:
    loss: 0.0748
    accuracy: 0.9757
--------------------
epoch: 1
train:
```

```
    loss: 0.0736
    accuracy: 0.9773
validation:
    loss: 0.0575
    accuracy: 0.9816
--------------------
epoch: 2
train:
    loss: 0.0501
    accuracy: 0.9843
validation:
    loss: 0.0592
    accuracy: 0.9813
--------------------
epoch: 3
train:
    loss: 0.0363
    accuracy: 0.9887
validation:
    loss: 0.0389
    accuracy: 0.9859
--------------------
epoch: 4
train:
    loss: 0.0289
    accuracy: 0.9912
validation:
    loss: 0.0409
    accuracy: 0.9854
--------------------
test:
    loss: 0.0465
    accuracy: 0.9863

"""
```

As you can see, we have reached a pretty good accuracy, and our loss is pretty low.

Conclusion

In this tutorial, we learned how **Convolution** works and how we can use it for image datasets. First, we explained the methodology of **Convolution**. Then, we showed how we can use **Convolution**, **ReLU**, and **Flatten** in **PyTorch**. After that, we made a model and calculated the output of each **Convolution layer**. Finally, we trained our model and saw the output.

Fine-tuning

Introduction

Fine-tuning is one of the most used techniques in **deep learning**. In this tutorial, we are going to learn how to load a pretrained model. Then, how to do **Transfer learning**. Finally, **Fine-tune** our model.

Load a dataset from Kaggle

In previous tutorials, we learned how to load a dataset from Kaggle. We have loaded a dataset called **Tom and Jerry image classification** and made the three subsets of **train**, **validation**, and **test**. Now, let's do it again.

```
path = kagglehub.dataset_download(
    "balabaskar/tom-and-jerry-image-classification")
path = Path(path) / "tom_and_jerry/tom_and_jerry"

tom_and_jerry_transforms =
    transforms.Compose([transforms.Resize([90, 160]),
    transforms.ToTensor()])

all_data = ImageFolder(path, transform=tom_and_jerry_transforms)

g1 = torch.Generator().manual_seed(20)
train_data, val_data, test_data = random_split(all_data, [0.7,
    0.2, 0.1], g1)

train_loader = DataLoader(train_data, batch_size=16,
    shuffle=True)
val_loader = DataLoader(val_data, batch_size=16, shuffle=False)
test_loader = DataLoader(test_data, batch_size=16, shuffle=False)
```

Let's plot on batch of its data:

```
images, labels = next(iter(train_loader))

fig, axes = plt.subplots(4, 4)

axes_ravel = axes.ravel()

for i, (image, label) in enumerate(zip(images, labels)):
    axes_ravel[i].imshow(transforms.ToPILImage()(image))
    axes_ravel[i].set_title(label.item())
    axes_ravel[i].set_axis_off()
```

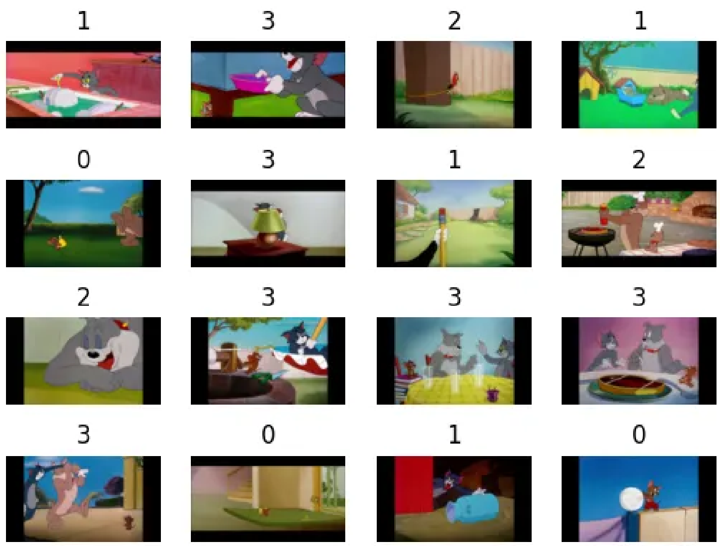

Figure 15: Tom and Jerry Batch

Load a pretrained model

TorchVision has prepared some of the most famous vision models with pre-trained weights. In this tutorial, we are going to use a model called **MobileNetV2**. To load that model, we can use the code below:

```
from torchvision.models import mobilenet_v2, MobileNet_V2_Weights

model = mobilenet_v2(weights=MobileNet_V2_Weights.IMAGENET1K_V1)
print(model)

"""
--------
output:

MobileNetV2(
  (features): Sequential(
    (0): Conv2dNormActivation(
      (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2),
 ↪  padding=(1, 1), bias=False)
      (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
 ↪  track_running_stats=True)
      (2): ReLU6(inplace=True)
    )
    ...
   (17): InvertedResidual(
      (conv): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(160, 960, kernel_size=(1, 1), stride=(1,
 ↪  1), bias=False)
          (1): BatchNorm2d(960, eps=1e-05, momentum=0.1,
 ↪  affine=True, track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (1): Conv2dNormActivation(
          (0): Conv2d(960, 960, kernel_size=(3, 3), stride=(1,
 ↪  1), padding=(1, 1), groups=960, bias=False)
          (1): BatchNorm2d(960, eps=1e-05, momentum=0.1,
 ↪  affine=True, track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (2): Conv2d(960, 320, kernel_size=(1, 1), stride=(1, 1),
 ↪  bias=False)
        (3): BatchNorm2d(320, eps=1e-05, momentum=0.1,
 ↪  affine=True, track_running_stats=True)
      )
    )
```

```
    (18): Conv2dNormActivation(
      (0): Conv2d(320, 1280, kernel_size=(1, 1), stride=(1, 1),
↪   bias=False)
      (1): BatchNorm2d(1280, eps=1e-05, momentum=0.1,
↪   affine=True, track_running_stats=True)
      (2): ReLU6(inplace=True)
    )
  )
  (classifier): Sequential(
    (0): Dropout(p=0.2, inplace=False)
    (1): Linear(in_features=1280, out_features=1000, bias=True)
  )
)

"""
```

In the code above, we have imported `mobilenet_v2` and `MobileNet_V2_Weights`. Then we loaded the mobile net with the weights that are pretrained on a dataset called **ImageNet**. Then, we printed the model to see the layers. As you can see, it consists of 2 **Sequential layers**. The first one is to extract features from the image. The second one is for classification.

Transfer Learning

Transfer learning is a technique of using a pretrained model (called the base model), on a new dataset with a different purpose. We don't train all the layers; instead, we only train the **classification layers**. So, the first step is to freeze all the layers.

```
# -------------------[ Freeze the model weights
↪  ]-------------------
for param in model.parameters():
    param.requires_grad = False
```

With the code above, we can freeze all the parameters. Now, let's replace the **classification layer** with our layer.

```
print("classifier before the change:")
print(model.classifier)
print("-" * 20)
# -------------------[ Change the classifier layer
↪  ]-------------------
model.classifier = nn.Linear(in_features=1280, out_features=4)

print("classifier after the change:")
print(model.classifier)
```

```
"""
--------
output:

classifier before the change:
Sequential(
  (0): Dropout(p=0.2, inplace=False)
  (1): Linear(in_features=1280, out_features=1000, bias=True)
)
--------------------
classifier after the change:
Linear(in_features=1280, out_features=4, bias=True)
"""
```

As you can see, in the code above, we have replaced the **classification layer** with our layer. As you recall, the dataset that we are using has 4 classes. So, the output of our final layer should be 4. Also, the output of the previous layer is 1280, so we should set our `in_features` to 1280 as well. Now let's see which layers are trainable:

```
for name, param in model.named_parameters():
    if param.requires_grad:
        print(f"{name} {param.shape}")

"""
--------
output:

classifier.weight torch.Size([4, 1280])
classifier.bias torch.Size([4])
"""
```

In the code above, I have used `named_parameters` to go over the model parameters and also get their names. As you can see, the only parameters that are trainable are related to the `classifier`. So, let's replace **dataset** and the **model** that we had in train_mnist_conv.py and train our model. I have already applied the changes in transfer_learning.py. Now, let's run it for 20 epochs and see the results on `Tensorboard`.

```


"""
--------
output:
```

```
mps
--------------------
epoch: 0
train:
    loss: 1.0890
    accuracy: 0.5330
validation:
    loss: 0.9879
    accuracy: 0.5894
--------------------
epoch: 1
train:
    loss: 0.9273
    accuracy: 0.6248
validation:
    loss: 0.8319
    accuracy: 0.6542
--------------------
...
--------------------
epoch: 19
train:
    loss: 0.6706
    accuracy: 0.7379
validation:
    loss: 0.6769
    accuracy: 0.7199
--------------------
test:
    loss: 0.7269
    accuracy: 0.7130

"""
```

Transfer Learning Train Accuracy

Transfer Learning Validation Accuracy

As you can see, in the results above, we have reached acceptable accuracies.

subset	accuracy
train	73.79
validation	71.99
test	71.30

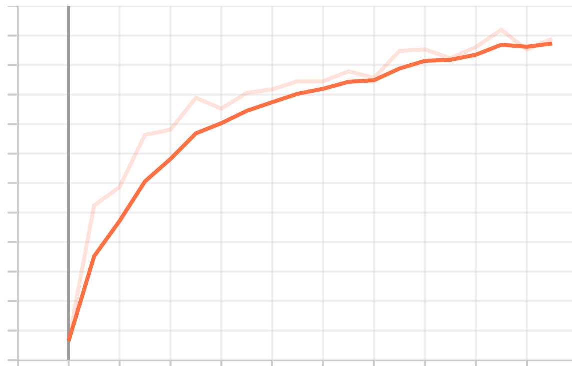

Figure 16: Transfer Learning Accuracy Train

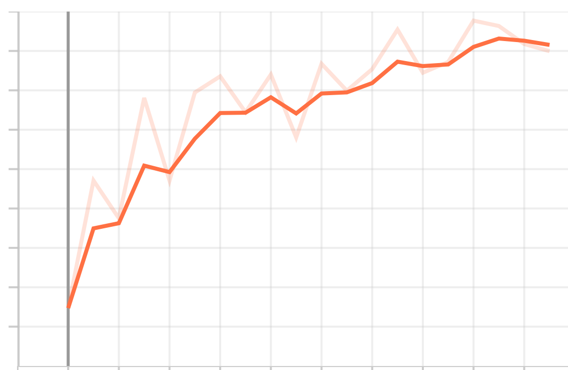

Figure 17: Transfer Learning Accuracy Validation

The results are pretty close, so our model is not overfitting. Also, the charts are ascending.

Fine-tuning

Fine-tuning has the same purpose as **Transfer Learning**. The only exception is that we train more layers. So, let's load our model again and freeze all layers except the last two (17 and 18).

```
model = mobilenet_v2(weights=MobileNet_V2_Weights.IMAGENET1K_V1)

# -------------------[ Freeze the model weights
 ↪  ]-------------------
for name, param in model.named_parameters():
    if not ("18" in name or "17" in name):
        param.requires_grad = False
```

In the code above, I iterated over the model's parameters. If they had 18 or 17 in their names, I didn't freeze them. Now, let's change the classifier layer and print the trainable parameters.

```
model.classifier = nn.Linear(in_features=1280, out_features=4)

for name, param in model.named_parameters():
    if param.requires_grad:
        print(f"{name} {param.shape}")

"""
--------
output:

features.17.conv.0.0.weight torch.Size([960, 160, 1, 1])
features.17.conv.0.1.weight torch.Size([960])
features.17.conv.0.1.bias torch.Size([960])
features.17.conv.1.0.weight torch.Size([960, 1, 3, 3])
features.17.conv.1.1.weight torch.Size([960])
features.17.conv.1.1.bias torch.Size([960])
features.17.conv.2.weight torch.Size([320, 960, 1, 1])
features.17.conv.3.weight torch.Size([320])
features.17.conv.3.bias torch.Size([320])
features.18.0.weight torch.Size([1280, 320, 1, 1])
features.18.1.weight torch.Size([1280])
features.18.1.bias torch.Size([1280])
classifier.weight torch.Size([4, 1280])
classifier.bias torch.Size([4])
"""
```

As you can see, the last two layers of our model are still trainable, and we have a classifier that works with our dataset. I have already applied the required changes in fine_tuning.py. Let's run it to see the results.

```
"""
--------
output:

mps
--------------------
epoch: 0
train:
    loss: 0.9683
    accuracy: 0.6188
validation:
    loss: 0.7647
    accuracy: 0.6870
--------------------
epoch: 1
train:
    loss: 0.6625
    accuracy: 0.7458
validation:
    loss: 0.5958
    accuracy: 0.7737
--------------------
...
--------------------
epoch: 18
train:
    loss: 0.1789
    accuracy: 0.9335
validation:
    loss: 0.5616
    accuracy: 0.8650
--------------------
epoch: 19
train:
    loss: 0.1397
    accuracy: 0.9518
validation:
    loss: 0.6718
    accuracy: 0.8577
--------------------
test:
```

```
    loss: 0.6284
    accuracy: 0.8537

"""
```

Fine-tuning Train Accuracy

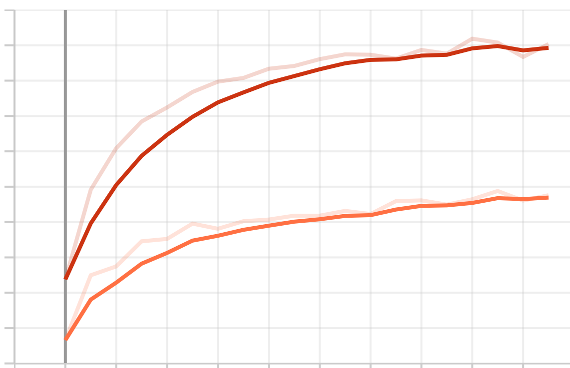

Figure 18: Fine-tuning Train accuracy

- Orange: Transfer Learning
- Red: Fine-tuning

Fine-tuning Validation Accuracy

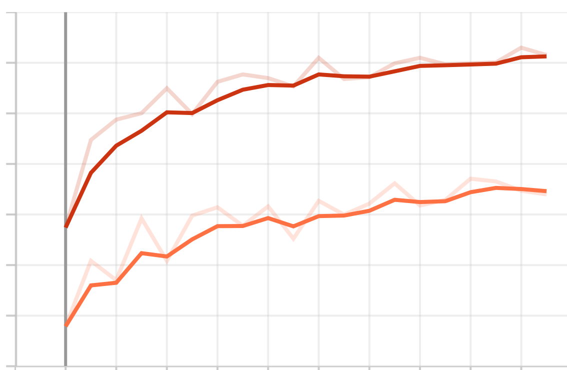

Figure 19: Fine-tuning Validation Accuracy

- Orange: Transfer Learning

- Red: Fine-tuning

As you can see in the results above, we have achieved better results than **Transfer Learning**.

subset	accuracy
train	95.18
validation	85.77
test	85.37

Conclusion

In this tutorial, we learned how to use a pretrained model on a new dataset. This is one of the most used techniques in deep learning. At first, we learned about **Transfer Learning** and saw the results. Then, we learned about **Fine-tuning** and compared it with **Transfer Learning**.

Next Steps

Final words

So far, we have learned the basics of **PyTorch**. Now, we are ready to explore more. There are so many packages that can help us make the training and evaluation easier and more efficient. Two of the most famous ones are: PyTorch Ignite and Lightning Also, there are sites like Kaggle which host **competitions**, **datasets**, **pretrained models** and more that are extremely helpful. If you want to have more **PyTorch** pretrained image models, you can use Timm. Timm is a really great way to share your model and explore other people's models. I strongly recommend that you take a look at Transformers as well. There are tons of great models, and **Large Language models** are available on it. Also, it is super easy to use them and fine-tune them in a way that you want. **Deep Learning** is evolving extremely fast, and so many packages and new tools are being released to help make the experience of using **Deep Learning** models easier. These were only some tools and packages, but there are so many others available to use. Hope you enjoyed these tutorials.

Regards,

Ramin ZareBidoky (LiterallyTheOne)