# AutoGrad, loss function, and optimizer

Ramin Zarebidoky (LiterallyTheOne)

18 Aug 2025

## 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

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.

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

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

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.

```
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: 0.02135099470615387
step 1, loss: 0.020931493490934372
step 2, loss: 0.020520454272627783
step 3, loss: 0.020117828622460365
step 4, loss: 0.019723571836948395
step 5, loss: 0.019337747246026993
step 6, loss: 0.0189602542668581
step 7, loss: 0.01859092339873314
step 8, loss: 0.018229883164167404
step 9, loss: 0.017876904457807 54
"""
```

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.