Real-world data representation using tensors

This chapter covers:

- Understanding how algorithms can learn from data
- Reframing learning as parameter estimation, using differentiation and gradient descent
- Walking through a simple learning algorithm
- How PyTorch supports learning with autograd

This chapter: Explains how to model a function from given data

- How the weights of a model are updated
- How less loss is what we want (loss function, objective function)

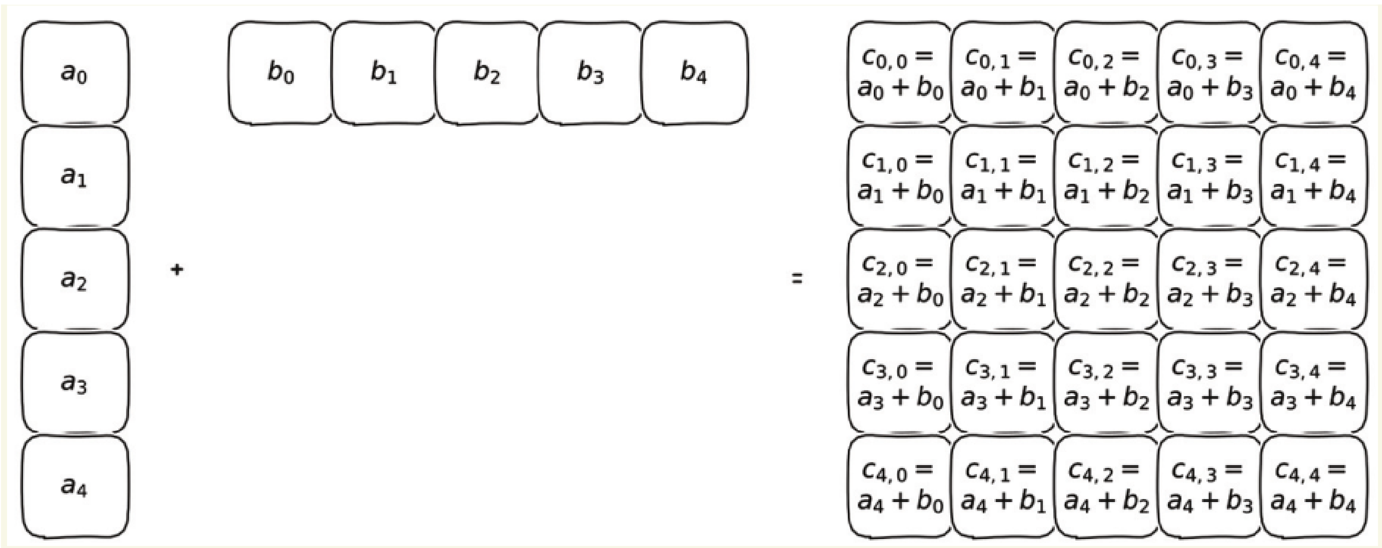
Broadcasting

Usually, we can only use element-wise binary operations (addition, subtraction, multiplication, division) for arguments of the **same shape**.

Broadcasting Relaxes this assumption. The following rules are used to match tensor elements:

1. For Each index dimension counted from the back, if one of the operands is size 1 in that dimension, PyTorch will use the single entry along this dimension with each of the entries in the other tensor along this dimension.
2. If both sizes are greater than 1, they must be the same, ant natural matching is used.
3. If one of the tensors has more index dimensions than the other, the entirety of the other tensor (smaller dimensions) will be used for each entry along these dimensions.

```
shapes: x: torch.Size([]), y: torch.Size([3, 1])
z: torch.Size([1, 3]), a: torch.Size([2, 1, 1])
x * y: torch.Size([3, 1])
y * z: torch.Size([3, 3])
y * z * a: torch.Size([2, 3, 3])
```



- Explanations on forward pass & backward pass (and thus, backpropagation)
- Explanations of how too large/small values of lr may blow/stall the learning process

In the book example (page 119~), the first-epoch gradient for the weight is about 50 times larger thant eh gradient for the bias.

- Weight and bias live in differently scaled spaces
- A LR that's large enough to meaningfully update one will be so large as to be unstable for the other
- a rate that's apt for the other won't be large enough to meaningfully change the first

Simple way to keep things in check: changing the inputs so that the gradients aren't quite so different

- by making sure the range of the input doesn't get too far from the range of -1.0 to 1.0.
- i.e. normalization

> Plotting our data; Seriously, this is the first thing anyone doing data science should do. Always plot the heck out of the data.
>
> - Learn matplotlib/pyplot, other visualization tools

Python argument unpacking can be used for PyTorch tensors as well:

- `*params` means to pass the elements of `params` as individual arguments.
- split along the leading dimension.
  - `model(t_un, *params) <> model(t_un, params[0], params[1])`

Autograd

In general, all PyTorch tensors have an attribute named `grad`. Normally, it's `None`

- start with a tensor with `requires_grad` set to True
- call the model and compute the loss
- Call `backward` on the `loss` tensor

```
# In[7]:
loss = loss_fn(model(t_u, *params), t_c)
loss.backward()
params.grad
# Out[7]:
tensor([4517.2969, 82.6000])
```

- Calling `backward` will lead derivatives to ACCUMULATE at leaf nodes. Need to `zero the gradient explicitly` after using it for parameter updates.

```
# In[9]:
def training_loop(n_epochs, learning_rate, params, t_u, t_c):
    for epoch in range(1, n_epochs + 1):
        if params.grad is not None:
            params.grad.zero_() # any point in the loop prior to loss.backw
ard()

        t_p = model(t_u, *params)
        loss = loss_fn(t_p, t_c)
        loss.backward()

        with torch.no_grad():
            params -= learning_rate * params.grad # cumbersome, but not an
 issue in practice (optimizer available)

        if epoch % 500 == 0:
            print('Epoch %d, Loss %f' % (epoch, float(loss)))
    return params
```

Optimizers

```
# In[5]:
import torch.optim as optim
dir(optim)
# Out[5]:
['ASGD',
'Adadelta',
'Adagrad',
'Adam',
'Adamax',
'LBFGS',
'Optimizer',
 'RMSprop',
'Rprop',
'SGD',
'SparseAdam',
...
]
```

Every optimizer takes a list of params (PyTorch tensors, typically with `requires_grad` set to True) as the first input.

- `zero_grad` : zeroes the `grad` attribute of all the parameters passed to the optimizer upon construction.
- `step` : updates the value of those parameters according to the optimization strategy implemented by the specific optimizer.

```
# Loop ready code
params = torch.tensor([1.0, 0.0], requires_grad=True)
learning_rate = 1e-2
optimizer = optim.SGD([params], lr=learning_rate)

t_p = model(t_un, *params)
loss = loss_fn(t_p, t_c)

optimizer.zero_grad()
loss.backward()
optimizer.step()

params

# Out[8]:
tensor([1.7761, 0.1064], requires_grad=True)
```

> We have touched on a lot of the essential concepts that will enable us to train complicated deep learning models while knowing what's going on under the hood: backpropagation to estimate gradients, autograd, and optimizing weights of models using gradient descent or other optimizers. Really, there isn't a lot more. The rest is mostly filling in the blanks, however extensive they are.

Overfitting, and Validation

Better to have more data

Assuming we have enough data points:

- add penalization terms to the loss function
- add noise to the input samples (create new data points in between training data samples and force the model to try to fit those too)
- Make the model simpler
  - Increase the size until it fits
  - Scale it down until it stops overfitting

Upon integrating validation as well, **ONLY CALL BACWARD() ON THE TRAIN LOSS**

- calling backward() on the valid loss will include correponding gradients in the training
- But "not calling backward() alone" will still construct the graphs (which is an unnecessary overhead)
  - Therefore use `torch.no_grad()`
  - the opposite is `torch.set_grad_enabled(Boolean)`

```
with torch.no_grad():
    val_t_p = model(val_t_u, *params)
    val_loss = loss_fn(val_t_p, val_t_c)
    assert val_loss.requires_grad == False # check that requires_grad is fo
rced to False inside block
```