Autograd (Backpropagation)

track → `x = torch.zeros(1, requires_grad=True)`

[`with torch.no_grad():`
turns off autograd

we can get the grad_fn with `x.grad_fn`
finally, we do `z.backward()`

Loss and Autograd together:

```
Code as before
:
loss = .....
```

`loss.backward()` → compute gradients

Using gradients:

we use optimizers :

before forward pass ← `optimizer.zero_grad()`

`optimizer.step()` ↳ after getting the loss

Summary of Training Steps:

1. forward pass
2. compute loss
3. `loss.backward()` to compute gradients
4. take a step with optimizer to update weights

Full training code:

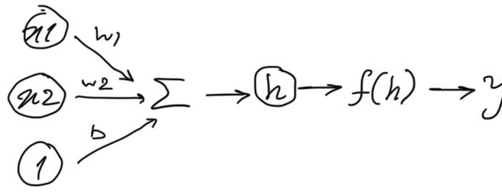
```
criterion = nn.NLLLoss()
optimizer = optim.SGD(model.parameters(), lr=0.003)
for e in range(epochs):
    running_loss = 0
    for images, labels in trainloader:
        images = images.view(images.shape[0], -1)
    * optimizer.zero_grad()

        Forward pass

    * loss.backward()
    * optimizer.step()

        running_loss += loss.item()
    else:
        training_loss = running_loss/len(trainloader)
```

Tensors



$$y = f(w_1 x_1 + w_2 x_2 + b)$$

$$h = [x_1 \cdots x_n] \cdot \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix}$$

```
def sigmoid(x): return 1/(1+torch.exp(-x))
torch.manual_seed(7)
features = torch.randn((1,5))   <- five random normal vars
weights = torch.randn_like(features)   <- a vector with the shap of features, random.
bias = torch.randn((1,1))   <- a 1x1 bias term
```

elem-by-elem mult

```
y = sigmoid(torch.sum(features * weights) + bias)
```

output of a single layer

`torch.mm` and `torch.matmul` → mat multiplication

for changing the shape of tensors:

1. `weights.reshape(a,b)` → returns new tensor
2. `weights.resize_(a,b)` → in place operation
3. `weights.view(a,b)` → return new data with same data

alt

```
y = sigmoid(torch.mm(features, weights.view(5,1)) + bias)
```

Stacking up units:

hidden units →

input units ↓

$$\vec{h} = [h_1, h_2] = [x_1 \cdots x_n] \cdot \begin{bmatrix} w_{11} & w_{12} \\ \vdots & \\ w_{n1} & w_{n2} \end{bmatrix}$$

```
h = sigmoid(torch.mm(features, W1) + B1)
y = output = sigmoid(torch.mm(h, W2) + B2)
```

Going between Torch and Numpy.

memory is shared between them

```
a = np.random.rand(4,3)
b = torch.from_numpy(a)
b.numpy()
```

Inference and Validation

we had this code for testing the prediction:

validation step

```
model = Classifer()
images, labels = next(iter(testloader))
ps = torch.exp(model(images))
```

→ (64, 10)
- 64: # images
- 10: # classes

```
top_p, top_class = ps.topk(1, dim=1)
```

k: highest value

```
equals = top_class == labels.view(*top_class.shape)

accuracy = torch.mean(equals.type(torch.FloatTensor))
```

because equals is a bit tensor and we have to convert it.

Implementing Validation Pass:

```
for e in epochs:
    for image, label in trainloader:
        as before
    else:
        with torch.no_grad():
            above code
```

the validation pass

Using nn module to build neural nets:

```
from torch import nn

class Network(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden = nn.Linear(784, 256)
        self.output = nn.Linear(256, 10)
```

* `self.sigmoid = nn.Sigmoid`
* `self.softmax = nn.Softmax(dim=1)`

```
    def forward(self, x):
        x = self.hidden(x)
        x = self.sigmoid(x)
        x = self.output(x)
        x = self.softmax(x)
        return x
```

For making things easier: `import torch.nn.functional as F`

then we can remove self.sigmoid and self.softmax from the class above.

```
    def forward(self, x):
        x = F.sigmoid(self.hidden(x))
        x = F.softmax(self.output(x), dim=1)
        return x
```

Avoiding Overfitting:

<< dropout >>

```
class Classifier(nn.Module):
    def __init__(self):
        ==========
        self.dropout = nn.Dropout(p=0.2)
```

0.2 ← drop probability

```
    def forward(self, x):
        ----------
        x = self.dropout(F.relu(self.fc1(x)))
        x = self.dropout(...
        ----------
        x = F.log_softmax(self.fc4(x), dim=1)
```

apply dropout at every hidden layer

no dropout for output layer

```
        return x
```

Note: when we do validation, we don't want any dropouts, So:

```
with torch.no_grad():
    model.eval()
    ==========
model.train()
```

`model.eval()` → turns off dropouts

`model.train()` → revert to train mode

Saving and Loading Models:

Save
```
torch.save(model.state_dict(), 'checkpoint')
```
↑ all weights and biases of model

Load
```
state_dict = torch.load('checkpoint')
model.load_state_dict(state_dict)
```

When loading a checkpoint, it also needs other info such as input_size, output_size, and number of hidden_layers in addition to the state_dict. So:

Saving checkpoint
```
checkpoint = {'input_size': 784,
              :
                                /
              'state_dict': model.state_dict()}
torch.save(checkpoint, 'checkpoint')
```

loading checkpoint
```
def load_checkpoint(filepath):
    checkpoint = torch.load(filepath)
    model = fc_model.Network(checkpoint['input'],
                             ....)
    model.load_state_dict(chkpoint['stat_dict'])
    return model
```

Loading Image Data:

```
dataset = datasets.ImageFolder('path',
                               transform = transforms)
```

image folder structure:

```
root/dog/xxx.png
root/dog/xxy.png
   :
root/cat/xxz.png
root/cat/xxw.png
   :
```

one folder for each class

Transforms:

```
transforms = transforms.Compose([
                transforms.Resize(255),
                transforms.CenterCrop(224),
                transforms.ToTensor()])
```

Converting to tensor ← `transforms.ToTensor()`

Data Loaders:

```
data_loader = torch.utils.data.DataLoader
                (dataset, batch_size=32,
                 shuffle = True)
```

```
next(iter(data_loader))
```

`iter(data_loader)`: generator

`next(iter(data_loader))`: iterator

Data Augmentation:

Introducing randomness in input data itself.

transforms ← RandomRotation, RandomResizedCrop, etc.

Normalization:

`transforms.Normalize`

```
input[channel] = (input[channel] - mean[channel]) / std[channel]
```

Helps keep data near zero, making backpropagation more stable.

Transfer Learning:

The use of pre-trained models.

torch ← models.densenet121(pretrained = True)

then, we have to freeze model params:

```
for param in model.parameters():
    param.requires_grad = False
```

} we don't want to affect these.

Replace their classifier with our classifier:

```
from collections import OrderedDict
classifier = nn.Sequential(OrderedDict([
    ('fc1', nn.Linear(1024, 500)),
    ('relu', nn.ReLU()),
    ('fc2', nn.Linear(500, 2)),
    ('output', nn.LogSoftmax(dim=1))
]))
```

a new, untrained classifier

```
model.classifier = classifier
```

Using GPU:

```
model.cuda()    # move the model params to GPU
images.cuda()   # move tensor to gpu
```

```
model.cpu()
images.cpu()
```

} returning the computations to cpu.

Putting Everything Together:

```
device = torch.device("cuda"
                      if torch.cuda.is_available()
                      else
                      "cpu")
model = models.resnet50(pretrained = True)
for param in model.parameters():
    param.requires_grad = False
```

freeze

replace the pre-trained classifier

```
classifier = nn.Sequential(nn.Linear(2048, 512),
                           nn.ReLU(),
                           nn.Dropout(p=0.2),
                           nn.Linear(512, 2),
                           nn.LogSoftmax(dim=1))
model.fc = classifier
```

```
criterion = nn.NLLLoss()
optimizer = optim.Adam(model.fc.parameters(),
                       lr= 0.003)
```

move the model to the available device.

```
model.to(device)
```

set hyper params

```
epochs=1
steps = 0
running_loss = 0
print_every = 5
```

start actual training

```
for epoch in range(epochs):
    for ims, labs in trainloader:
        steps += 1
```

move to the available device

```
        ims, labs = ims.to(device), labs.to(device)
```

```
        optimizer.zero_grad()
        logps = model(images)
        loss = criterion(logps, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
```

frequency for validation

```
        if steps % print_every == 0:
            model.eval()
            test_loss = 0
            accuracy = 0
```

remember to transfer to gpu:

```
ims, labs = ims.to(device)
labs.to(device)
```

Validation

```
            for ims, labs in testloader:
                logps = model(ims)
                loss = criterion(logps, labs)
                test_loss += loss.item()

                ps = torch.exp(logps)
                top_ps, top_class = ps.topk(1,
                                            dim=1)
                equality = top_class ==
                           labels.view(*top_cls.shape)
                accuracy += torch.mean(
                    equality.type(torch.FloatTensor
                    )).item()
```

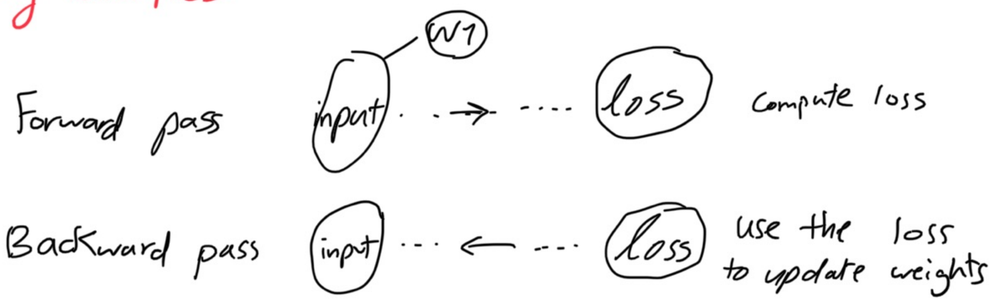
```
        running_loss = 0
        model.train()
```

Training Neural Networks

loss function ⟶ measure prediction error

$$\ell = \frac{1}{2n} \sum_{i}^{n} (y_i - \hat{y}_i)^2$$

- n: number of training examples
- y_i: true labels
- $\hat{y}_i$: predicted labels



Loss in Pytorch:

nn.CrossEntropyLoss ↗ nn.LogSoftmax()
nn.CrossEntropyLoss ↘ nn.NLLLoss()

input: Scores (not probabilities)

In summary

1. Build a feed forward network:

```
model = nn.Sequential( nn.Linear(784, 128),
                       nn.ReLU,
                       nn.Linear(128, 64),
                       nn.ReLU,
                       nn.Linear(64, 10))
```

ReLU is almost always used for hidden layers

2. Define the loss:

```
criterion = nn.CrossEntropyLoss()
```

and set this to nn.NLLLoss()

we can add a `nn.LogSoftmax(dim=1)`

3. Get the data:

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

4. flatten images:

```
images = images.view(images.shape[0], -1)
```

5. forward pass — get logits

```
logits = model(images)
```

6. Calculate the loss:

```
loss = criterion(logits, labels)
```

Neural Networks with Pytorch

```
from torchvision import datasets, transforms
```

for normalizing data

```
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5,0.5,0.5), (0.5,0.5,0.5))
                               ])
```

download mnist dataset

```
trainset = datasets.MNIST('MNIST_data/',
                          download=True,
                          train=True,
                          transform=transform)
```

define a loader

```
trainloader = torch.utils.data.DataLoader(trainset,
                                          batch_size=64,
                                          shuffle=True)
```

each time we get a batch, it's shape is (64, 1, 28, 28)

- 64 images
- 1 color channel
- image size

```
dataiter = iter(trainloader)
images, labels = dataiter.next()
```

we usually flatten the input:

from (64, 1, 28, 28) to (64, 784)

Build a network for identifying images:

```
inputs = images.view(images.shape[0], -1)
```

choose the appropriate size

input units

```
w1 = torch.randn(784, 256)
b1 = torch.randn(256)
w2 = torch.randn(256, 10)
b2 = torch.randn(10)
```

hidden units

one output digit for each digit.

one hidden layer

```
h = sigmoid(torch.mm(inputs, w1) + b1)
out = torch.mm(h, w2) + b2
```

```
def softmax(x):
    return torch.exp(x)/torch.sum(torch.exp(x), dim=1)
                                                  .view(-1,1)

probs = softmax(out)  # shape = (64,10)
```