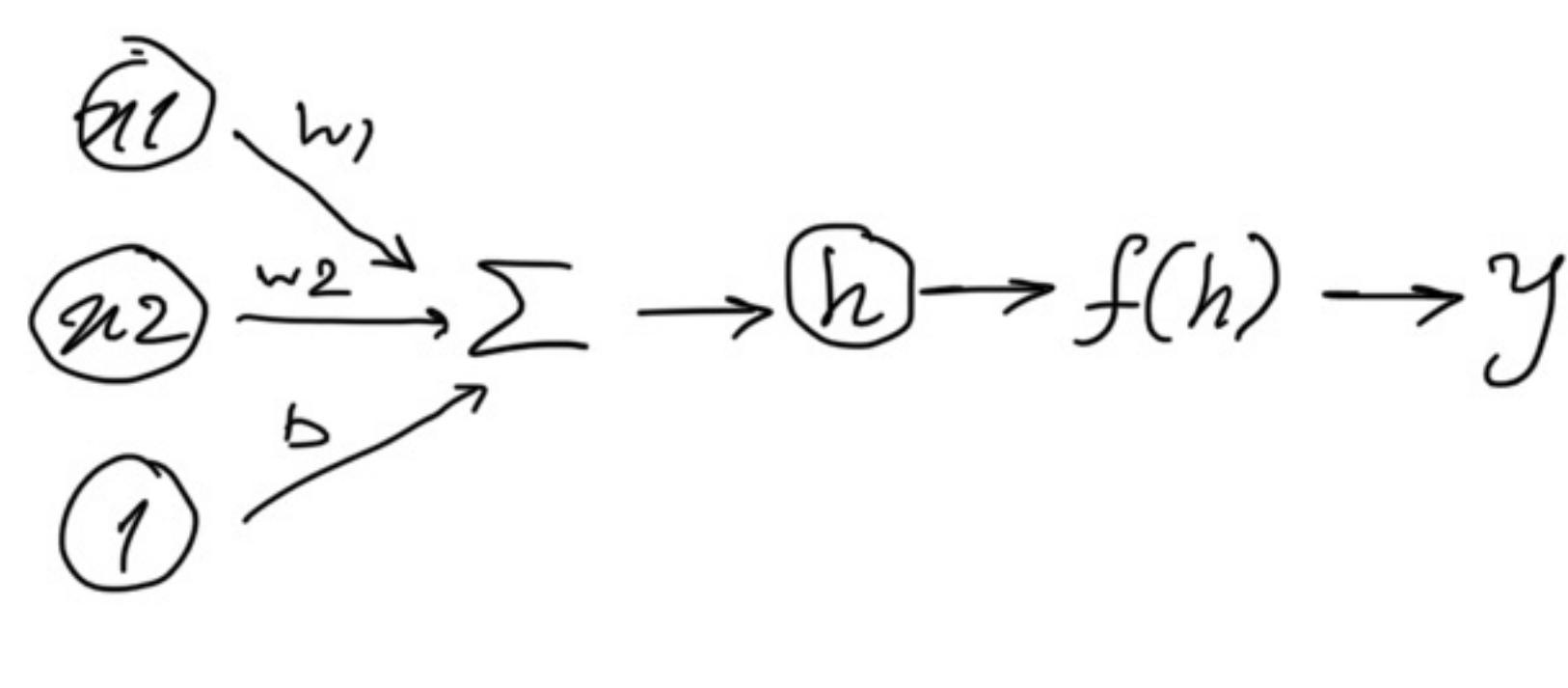


## Tensors



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

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

def sigmoid(x): return 1/(1+torch.exp(-x))

torch.manual\_seed(7)

features = torch.randn((1, 5))  $\leftarrow$  five random normal vars

weights = torch.randn\_like(features)  $\leftarrow$  a vector with  
the shape of features, random.

bias = torch.randn((1, 1))  $\leftarrow$  a 1x1 bias term

elem-by-elem mult

$y = \text{sigmoid}(\text{torch.sum}(\text{features} * \text{weights}) + \text{bias})$

output of a single layer

torch.mm and torch.matmul  $\rightarrow$  mat multiplication

for changing the shape of tensors:

1. weights.reshape(a, b)  $\rightarrow$  returns new tensor

2. weights.resize\_(a, b)  $\rightarrow$  in place operation

3. weights.view(a, b)  $\rightarrow$  return new data with same data

alt

$y = \text{sigmoid}(\text{torch.mm}(\text{features}, \text{weights.view}(5, 1)) + \text{bias})$

Stacking up units:

$$\vec{h} = [h_1, h_2] = [x_1, \dots, x_n] \cdot \begin{bmatrix} \overset{\text{input units}}{\downarrow} & \overset{\text{hidden units}}{\rightarrow} \\ w_{11} & w_{12} \\ \vdots & \vdots \\ w_{n1} & w_{n2} \end{bmatrix}$$

$h = \text{sigmoid}(\text{torch.mm}(\text{features}, w_1) + b_1)$

$y = \text{output} = \text{sigmoid}(\text{torch.mm}(h, w_2) + b_2)$

Going between Torch and Numpy.

memory is shared between them

$a = np.random.rand(4, 3)$

$b = \text{torch.from_numpy}(a)$

$b.numpy()$

# Neural Networks with Pytorch

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

for  
normalizing transform = transforms.Compose([transforms.ToTensor(),  
data  
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,  
each time we get a batch, it's shape is (64, 1, 28, 28)  
batch-size = 64, shuffle = True)  
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)

↑  
input units

w1 = torch.randn(784, 256)

b1 = torch.randn(256)

w2 = torch.randn(256, 10)

b2 = torch.randn(10)

one output digit for each digit.

one hidden layer

h = sigmoid(torch.mm(inputs, w1) + b1)

out = torch.mm(h, w2) + b2

choose the appropriate size

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

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

## 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
```

# Training Neural Networks

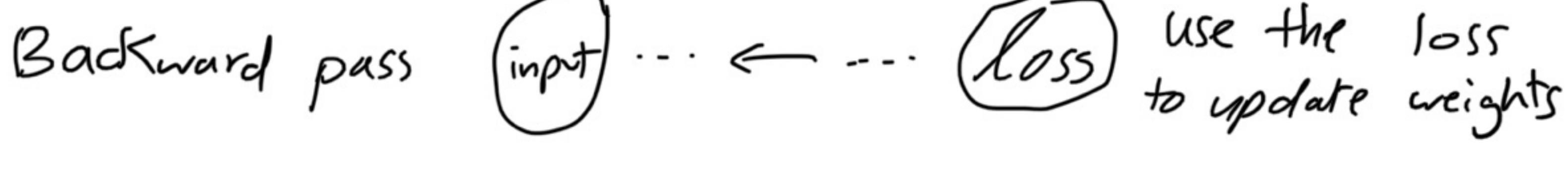
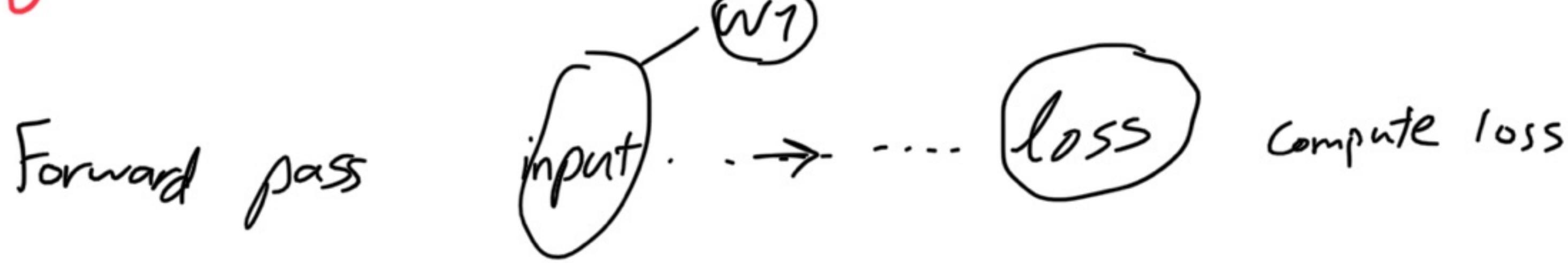
loss function → measure prediction error

$$l = \frac{1}{2n} \sum_i^n (y_i - \hat{y}_i)^2$$

number of training examples

true labels

predicted labels



## Loss in Pytorch:

`nn.CrossEntropyLoss` ↗  
    ↗ `nn.LogSoftmax()`  
    ↘ `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 :

and set this to `nn.NLLLoss()`

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

`criterion = nn.CrossEntropyLoss()`

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)`

# Autograd (Backpropagation)

track →  $x = \text{torch.zeros}(1, \text{requires_grad=True})$   
[with  $\text{torch.no\_grad}()$ :  
turns off autograd]

we can get the grad\_fn with  $x.\text{grad_fn}$   
finally, we do  $z.\text{backward}()$

## Loss and Autograd together:

Code as before  
:  
 $\text{loss} = \dots$

$\text{loss}.\text{backward}()$  → compute gradients

Using gradients:  $\xrightarrow{\text{forward pass}} \text{optimizer}.\text{zero\_grad}()$   
we use optimizers :  $\begin{cases} \text{optimizer}.\text{step}() \\ \hookrightarrow \text{after getting the loss} \end{cases}$

## Summary of Training Steps:

1. forward pass
2. compute loss
3.  $\text{loss}.\text{backward}()$  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)

\*  $\text{optimizer}.\text{zero\_grad}()$

**Forward pass**

\*  $\text{loss}.\text{backward}()$

\*  $\text{optimizer}.\text{step}()$

running-loss += loss.item()

else:

training-loss = running-loss / len(trainloader)

## Inference and Validation

we had this code for testing the prediction:

```
model = Classifier()
images, labels = next(iter(testloader))
ps = torch.exp(model(images))
top_p, top_class = ps.topk(1, dim=1)
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

## Avoiding Overfitting:

« dropout »

```
class Classifier(nn.Module):
```

```
    def __init__(self):
```

====

0.2

drop probability ← self.dropout = nn.Dropout( $p=0.2$ )

```
    def forward(self, x):
```

=====

apply  
dropout ← {  
at every layer  
hidden }  $x = \text{self.dropout}(\text{F.relu}(\text{self.fc1}(x)))$   
 $x = \text{self.dropout}(\dots)$

no dropout ↴  $x = \text{F.log-softmax}(\text{self.fc4}(x), \text{dim}=1)$

for output layer

return x

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

with torch.no\_grad():

model.eval() → turns off dropouts

====

model.train() → revert to train mode

## Saving and Loading Models:

all weights and biases of model

Save [ `torch.save(model.state_dict(), 'checkpoint')` ]

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(checkpoint['state_dict'])`  
                  `return model` }

## Loading Image Data:

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

image folder structure :

root/dog/m.png

one folder

root/dog/m~.png

for each

:

root/cat/m.png

class

root/cat/m~.png

:

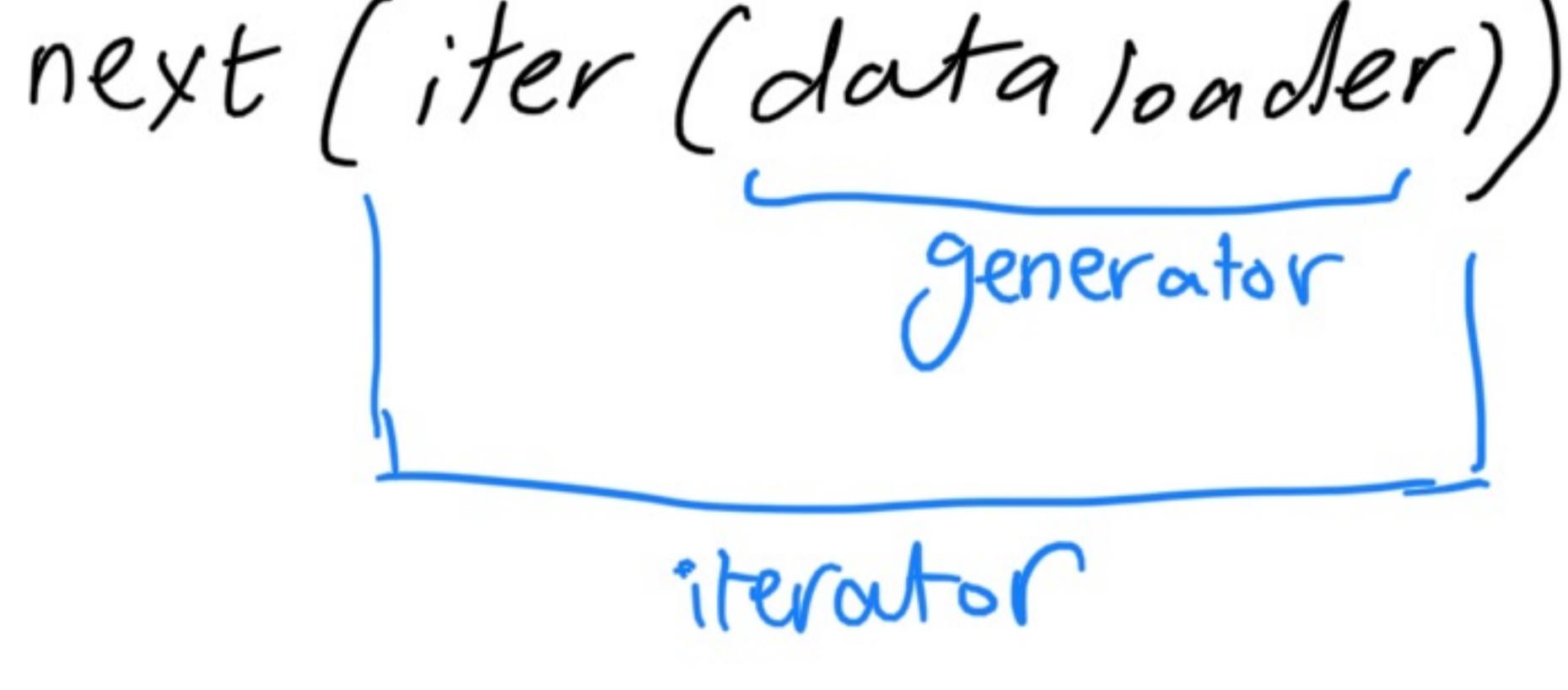
## Transforms :

```
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)
```



## Data Augmentation :

Introducing randomness in input data itself.

transforms ← RandomRotation, RandomResizedCrop, etc.

## Normalization :

transforms.Normalize

$\text{input[channel]} = (\text{input[channel]} - \text{mean[channel]}) / \text{std[channel]}$

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():` } we don't  
  `param.requires_grad = False`      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 to GPU  
`images.cuda()` # move tensor to gpu

`model.cpu()`      } returning the computations  
`images.cpu()`      to cpu.

## Putting Everything Together:

```
device = torch.device("cuda"  
                     if torch.cuda.is_available()  
                     else  
                     "cpu")
```

```
model = models.resnet50(pretrained=True)
```

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

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.

set  
hyper  
params } 

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

Start → for epoch in range(epochs):  
actual  
training      for ims, labs in trainloader:  
                  steps += 1  
                  move to ← ims, labs = ims.to(device), labs.to(device)  
                  the available device

```
optimizer.zero_grad()
```

```
logps = model(images)
```

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

```
loss.backward()
```

```
optimizer.step()
```

```
running_loss += loss.item()
```

frequency → if step % print\_every == 0:  
for validation      model.eval()  
                  test\_loss = 0  
                  accuracy = 0

remember to transfer  
to gpu:  
ims, labs = ims.to(device)  
              labs.to(device)

```
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()
```