lesson-04-review

June 10, 2024

[ ]:
```
!pip install -Uqq fastbook
import fastbook
fastbook.setup_book()

from fastai.vision.all import *
from fastbook import *

matplotlib.rc('image', cmap='Greys')
```

Code Summary

Pixel Similarity Code

[ ]:
```
#Load the training images into lists
path = untar_data(URLs.MNIST_SAMPLE)
Path.BASE_PATH = path
threes = (path/'train'/'3').ls().sorted()
sevens = (path/'train'/'7').ls().sorted()

#Convert images into tensors (from lists of imgs to lists of tensors)
seven_tensors = [tensor(Image.open(o)) for o in sevens]
three_tensors = [tensor(Image.open(o)) for o in threes]

#Convert lists of tensors into higher dimensional tensors with stacked imgs
stacked_sevens = torch.stack(seven_tensors).float()/255
stacked_threes = torch.stack(three_tensors).float()/255

#Apply all the same operations for the validation sets
valid_3_tens = torch.stack([tensor(Image.open(o)) for o in (path/'valid'/'3').
 ↪ls()])
valid_3_tens = valid_3_tens.float()/255
valid_7_tens = torch.stack([tensor(Image.open(o)) for o in (path/'valid'/'7').
 ↪ls()])
valid_7_tens = valid_7_tens.float()/255

#Calculate the average 3 and the average 7
mean3 = stacked_threes.mean(0) #0 is the first dimension
mean7 = stacked_sevens.mean(0)
```

```
#Function that calculates the MAE/L1 norm between an two images, the input and␣
 ↪the mean3 or mean7 img
def mnist_distance(test_img, mean_img):
    return (test_img-mean_img).abs().mean((-1,-2))

#Create function that predicts if an img is a 3 or a 7, based on the distance␣
 ↪to mean3 and mean7
def is_3(x):
    return mnist_distance(x,mean3) < mnist_distance(x,mean7)

accuracy_3s = is_3(valid_3_tens).float().mean()
accuracy_7s = (1 - is_3(valid_7_tens).float()).mean()

print(f"3s accuracy (validation set): {accuracy_3s:.4f}")
print(f"7s accuracy (validation set): {accuracy_7s:.4f}")
print(f"avg accuracy (validation set): {(accuracy_3s+accuracy_7s)/2:.4f}")
```

```
3s accuracy (validation set): 0.9168
7s accuracy (validation set): 0.9854
avg accuracy (validation set): 0.9511
```

Linear Model Code

[ ]:
```
###### PREP DATA ######
#Concatenate the rank-3 tensors end to end
train_x = torch.cat([stacked_threes, stacked_sevens]).view(-1, 28*28)
#Create target (y) label tensor where 1 is label for image of a 3 and 0 for␣
 ↪image of a 7
train_y = tensor([1]*len(threes) + [0]*len(sevens)).unsqueeze(1)
#Create dataset by combining each input with with corresponding label into a␣
 ↪tuple
dset = list(zip(train_x,train_y))
#Create Dataloader
dl = DataLoader(dset, batch_size=256)

#Do it all for the validation set as well
valid_x = torch.cat([valid_3_tens, valid_7_tens]).view(-1, 28*28)
valid_y = tensor([1]*len(valid_3_tens) + [0]*len(valid_7_tens)).unsqueeze(1)
valid_dset = list(zip(valid_x,valid_y))
valid_dl = DataLoader(valid_dset, batch_size=256)


###### INIT WEIGHTS/PARAMS ######
#Create a function to randomly initialize a set of parameters
def init_params(size, std=1.0):
    return (torch.randn(size)*std).requires_grad_()
```

```
#Create a weights 784x1 matrix (column vector) and a scalar bias
weights = init_params((28*28,1))
bias = init_params(1)


###### DEFINE MODEL ######
#Linear model function, makes the predictions
def linear1(xb):
    return xb@weights + bias


###### DEFINE LOSS FUNCTION ######
#Sigmoid function to distribute the outputs/predictions between 0 and 1
def sigmoid(x):
    return 1/(1+torch.exp(-x))

#Function that calculates loss. If target/label is 1, we need the prediction to␣
 ↪be as close to 1 as possible.
#Distance between predicition and 1 is then loss. For 0, we need predicition to␣
 ↪be as close to 0 so loss is prediction value directly.
def mnist_loss(predictions, targets):
    predictions = predictions.sigmoid()
    return torch.where(targets==1, 1-predictions, predictions).mean()


###### DEFINE HUMAN METRICS ######
#Function that calculates the accuracy of a single batch
def batch_accuracy(xb, yb):
    preds = xb.sigmoid()
    correct = (preds>0.5) == yb
    return correct.float().mean()

#Function that calculates the accuracy of the model on the entire validation set
def validate_epoch(model):
    accs = [batch_accuracy(model(xb), yb) for xb,yb in valid_dl]
    return round(torch.stack(accs).mean().item(), 4)


###### TRAIN THE MODEL ######
#Function to calculcate gradients after making predicitions and calculating␣
 ↪loss automatically.
def calc_grad(xb, yb, model):
    preds = model(xb)
    loss = mnist_loss(preds, yb)
    loss.backward()

def train_epoch(model, lr, params):
```

```
    for xb,yb in dl:
        calc_grad(xb, yb, model)
        for p in params:
            p.data -= p.grad*lr
            p.grad.zero_()

lr = 1.
params = weights,bias
for i in range(20):
    train_epoch(linear1, lr, params)
    print(validate_epoch(linear1), end=' ')
```

```
0.7303 0.8505 0.9008 0.9291 0.9389 0.9443 0.9526 0.9545 0.9589 0.9614 0.9623
0.9623 0.9633 0.9638 0.9648 0.9662 0.9667 0.9672 0.9672 0.9677
```

Alternate code for the linear model using the pytorch nn.Linear function and a custom Optimizer object (which is the same as the fastai built-int SGD() optimizer).

[ ]:
```
#Using pytorch/fastai nn.Linear() module
linear_model = nn.Linear(28*28,1)
#Parameter initilization is automatic for nn.Linear
w,b = linear_model.parameters()

#Custom optimizer. Does the same thing as the previous train_epoch
class BasicOptim:
    def __init__(self,params,lr): self.params,self.lr = list(params),lr

    def step(self, *args, **kwargs):
        for p in self.params: p.data -= p.grad.data * self.lr

    def zero_grad(self, *args, **kwargs):
        for p in self.params: p.grad = None
opt = BasicOptim(linear_model.parameters(), lr)

#Update train_epoch to use the BasicOptim instead of doing the stepping itself
def train_epoch(model):
    for xb,yb in dl:
        calc_grad(xb, yb, model)
        opt.step()
        opt.zero_grad()

def train_model(model, epochs):
    for i in range(epochs):
        train_epoch(model)
        print(validate_epoch(model), end=' ')

train_model(linear_model,10)
```

```
0.4932 0.9179 0.7974 0.9048 0.9312 0.9448 0.9551 0.9614 0.9653 0.9678
```

Neural Network Code

The linear model above can be used to create a simple NN by taking the previous (linear) model as the first linear layer res = xb@w1 + b1, using the output of that linear layer as input to another linear layer (after the activation function).

One difference for the first layer is that we are using 30 sets of weights to produce 30 activations/numbers instead of just one like in the linear model.

[ ]:
```
def simple_net(xb):
    res = xb@w1 + b1
    res = res.max(tensor(0.0)) #ReLU
    res = res@w2 + b2
    return res
```

[ ]:
```
w1 = init_params((28*28,30))
b1 = init_params(30)
w2 = init_params((30,1))
b2 = init_params(1)
```

Code for the simple NN using PyTorch/fastai functions/modules:

[ ]:
```
###### PREP DATA ###### (same as linear model)
#Concatenate the rank-3 tensors end to end
train_x = torch.cat([stacked_threes, stacked_sevens]).view(-1, 28*28)
#Create target (y) label tensor where 1 is label for image of a 3 and 0 for␣
 ↪image of a 7
train_y = tensor([1]*len(threes) + [0]*len(sevens)).unsqueeze(1)
#Create dataset by combining each input with with corresponding label into a␣
 ↪tuple
dset = list(zip(train_x,train_y))
#Create Dataloader
dl = DataLoader(dset, batch_size=256)

#Do it all for the validation set as well
valid_x = torch.cat([valid_3_tens, valid_7_tens]).view(-1, 28*28)
valid_y = tensor([1]*len(valid_3_tens) + [0]*len(valid_7_tens)).unsqueeze(1)
valid_dset = list(zip(valid_x,valid_y))
valid_dl = DataLoader(valid_dset, batch_size=256)

#Create a dataloaders that combines the training and validation dataset
dls = DataLoaders(dl, valid_dl)


##### DEFINE METRICS and LOSS ######
#Loss function (same as linear model)
def mnist_loss(predictions, targets):
    predictions = predictions.sigmoid()
    return torch.where(targets==1, 1-predictions, predictions).mean()
```

```
#Accuracy metric function (same as linear model)
def batch_accuracy(xb, yb):
    preds = xb.sigmoid()
    correct = (preds>0.5) == yb
    return correct.float().mean()

###### DEFINE MODEL ######
#Define the model architecture
simple_net = nn.Sequential(
    nn.Linear(28*28,30),
    nn.ReLU(),
    nn.Linear(30,1)
)

#Create Learner object with the dls, the simple_net architecture defined above␣
 ↪and the metrics/loss
learn = Learner(dls, simple_net, opt_func=SGD,
                loss_func=mnist_loss, metrics=batch_accuracy)

###### TRAIN MODEL ######
learn.fit(4, 0.1)
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

Detailed summary

Training hand written digit classifier on MNIST dataset

We start with simple binary classification (only 2 possible predicitions) of 3 and 7s. More precisely it is a classifier that identifies if an image is a 3 or NOT a 3, and since the only NOT 3 images we use for training it are 7s, if it's not 3, it's 7.

[ ]:
```
path = untar_data(URLs.MNIST_SAMPLE)
```

FASTAI_HOME which is by default ~/.fastai contains a config.ini file and some folders where it downloads data sets and models. So when we use untar_data we download the specified data set (URLs.MNIST_SAMPLE) into:

[ ]:
```
type(path),path
```

[ ]:
```
(pathlib.PosixPath, Path('/home/alex/.fastai/data/mnist_sample'))
```

The line bellow sets a base Path, so that all Path objects are refered from that BASE_PATH, to avoid specifying it repeatedly.

[ ]:
```
Path.BASE_PATH = path
```

The downloaded data set contains the following folders:

```
[ ]: (path/'train').ls()
```

```
[ ]: (#2) [Path('train/7'),Path('train/3')]
```

```
[ ]: len((path/'train/3').ls())
```

```
[ ]: 6131
```

```
[ ]: len((path/'train/7').ls())
```

```
[ ]: 6265
```

Load list of all 3s and 7s training images:

```
[ ]: threes = (path/'train'/'3').ls().sorted()
sevens = (path/'train'/'7').ls().sorted()
threes,sevens
```

```
[ ]: ((#6131) [Path('train/3/10.png'),Path('train/3/10000.png'),Path('train/3/10011.p
ng'),Path('train/3/10031.png'),Path('train/3/10034.png'),Path('train/3/10042.png
'),Path('train/3/10052.png'),Path('train/3/1007.png'),Path('train/3/10074.png'),
Path('train/3/10091.png')…],
 (#6265) [Path('train/7/10002.png'),Path('train/7/1001.png'),Path('train/7/10014
.png'),Path('train/7/10019.png'),Path('train/7/10039.png'),Path('train/7/10046.p
ng'),Path('train/7/10050.png'),Path('train/7/10063.png'),Path('train/7/10077.png
'),Path('train/7/10086.png')…])
```

Here is the size of each image:

```
[ ]: Image.open(threes[0]).width,Image.open(threes[0]).height
```

```
[ ]: (28, 28)
```

Below, we can view the first 3 image as a 28,28 tensor:

```
[ ]: df = pd.DataFrame(tensor(Image.open(threes[0])))
df.style.set_properties(**{'font-size':'6pt'}).background_gradient('Greys')
```

```
[ ]: <pandas.io.formats.style.Styler>
```

First approach, pixel similarity

The idea is to create a simple baseline model by calculating an average 3 and an average 7 image. This is done by averaging all pixels across all 3s and 7s. First step, loading all the 3s and 7s images in some tensors:

```
[ ]: seven_tensors = [tensor(Image.open(o)) for o in sevens]
three_tensors = [tensor(Image.open(o)) for o in threes]
print(f"Type of seven_tensors ---> {type(seven_tensors)}")
print(f"Type of elements in seven_tensors list ---> {type(seven_tensors[0])}")
print(f"Shape of elements in seven_tensors list ---> {seven_tensors[0].shape}")
```

```
Type of seven_tensors ---> <class 'list'>
Type of elements in seven_tensors list ---> <class 'torch.Tensor'>
Shape of elements in seven_tensors list ---> torch.Size([28, 28])
```

Code above uses list comprehension, which is equivalent to:

```
[ ]: three_tensors = []
     for t_img in threes:
         three_tensors.append(tensor(Image.open(t_img)))

     seven_tensors = []
     for s_img in sevens:
         seven_tensors.append(tensor(Image.open(s_img)))
```

```
[ ]: print(f"Type of seven_tensors ---> {type(seven_tensors)}")
     print(f"Type of elements in seven_tensors list ---> {type(seven_tensors[0])}")
     print(f"Shape of elements in seven_tensors list ---> {seven_tensors[0].shape}")
```

```
Type of seven_tensors ---> <class 'list'>
Type of elements in seven_tensors list ---> <class 'torch.Tensor'>
Shape of elements in seven_tensors list ---> torch.Size([28, 28])
```

Display one of the images tensors:

```
[ ]: three_tensors[4]
```

```
[ ]: 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,   0,   0,   0,   0],
             [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
     0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
             [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
     0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
             [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
     0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
             [  0,   0,   0,   0,   0,   0,   0,   5,  60, 136, 136, 147, 254, 255,
     199, 111,  18,   9,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
             [  0,   0,   0,   0,   0,   0,  25, 152, 253, 253, 253, 253, 253, 253,
     253, 253, 253, 124,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
             [  0,   0,   0,   0,   0,   0, 135, 225, 244, 253, 202, 200, 181, 164,
     216, 253, 253, 211, 151,   0,   0,   0,   0,   0,   0,   0,   0,   0],
             [  0,   0,   0,   0,   0,   0,   0,  30, 149,  78,   3,   0,   0,   0,
     20, 134, 253, 253, 224,   0,   0,   0,   0,   0,   0,   0,   0,   0],
             [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
     28, 206, 253, 253, 224,   0,   0,   0,   0,   0,   0,   0,   0,   0],
             [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
     78, 253, 253, 253, 224,   0,   0,   0,   0,   0,   0,   0,   0,   0],
             [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   5,
     99, 234, 253, 253, 224,   0,   0,   0,   0,   0,   0,   0,   0,   0],
             [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  14, 142, 220,
```

```
219, 236, 253, 253, 240, 121,   7,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  24, 253, 253,
253, 253, 235, 233, 253, 253, 185,  53,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   8, 150, 194,
194, 194,  53,  40,  97, 253, 253, 170,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
  0,   0,   0,   0, 122, 253, 253, 170,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
  0,   0,   0,  55, 237, 253, 253, 170,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
  0,   0,   0, 130, 253, 253, 253, 170,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
  4,  12, 120, 193, 253, 253, 214,  28,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   7,
153, 253, 253, 253, 253, 212,  30,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,  33, 136,  70,   6,   0,  27,  67, 186,
253, 253, 253, 253, 234,  31,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,  26, 231, 253, 253, 191, 183, 223, 253, 253,
253, 253, 172, 216, 112,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,  36, 215, 253, 253, 253, 253, 253, 253, 253,
253, 253,  47,  25,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   5,  87, 223, 253, 253, 253, 244, 152, 223,
223, 109,   4,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,  67,  50, 176, 148,  78,  16,   0,  12,
 12,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0]],
dtype=torch.uint8)
```

Use the fastai show_image function to display the tensor as an image (https://fastai1.fast.ai/vision.image.html#show_image):

```
[ ]: show_image(three_tensors[4]);
```

Our three_tensors and seven_tensors are of list type (they are lists of 2-dimensional/rank-2 28x28 tensors). In order to do effiecient computations accross all of those tensors, we need to stack them into a 3-dimensional/rank-3 tensor.

Also, since we will need to take the mean/avg of all those tensor values, we need to cast them (change their data type) to float (because we will be doing division and using ints would result in loss of precision).

Finally, we will normalize the pixel values to be between 0 and 1. This is good practice that will allow the neural network to train better (by using the same scale across all values, to avoid saturation and vanishing gradients) and faster (because our weights/parameters will converge faster towards ideal values).

[ ]:
```
stacked_sevens = torch.stack(seven_tensors).float()/255
stacked_threes = torch.stack(three_tensors).float()/255
print(f"stacked_thress/stacked_sevens TYPES ---> {type(stacked_threes)} /␣
 ↪{type(stacked_sevens)}")
print(f"stacked_thress/stacked_sevens RANKS ---> {len(stacked_threes.shape)} /␣
 ↪{len(stacked_sevens.shape)}")
print(f"stacked_thress/stacked_sevens SHAPES ---> {stacked_threes.shape} /␣
 ↪{stacked_sevens.shape}")
```

```
stacked_thress/stacked_sevens TYPES ---> <class 'torch.Tensor'> / <class
'torch.Tensor'>
stacked_thress/stacked_sevens RANKS ---> 3 / 3
stacked_thress/stacked_sevens SHAPES ---> torch.Size([6131, 28, 28]) /
torch.Size([6265, 28, 28])
```

So instead of having a list of 6131 2-d 28x28 tensors of threes (three_tensors), we have a 3-d tensor of 6131 tensors of 28x28.

FYI: - rank –> Number of axes/dimensions of a tensor (len(stacked_threes.shape)) - shape –> The size of each axis/dimension of the tensor (stacked_threes.shape)

Calculate average three and seven

The stacked_thress/stacked_sevens tensors are 3-dimensional/rank-3. The last 2 dimensions are 28,28 which indicate the width and height of each image. The first dimensions is thus the "list" of all threes or sevens. To calculate the average/mean 3 and 7, we simply need to calculate the average of each pixel across the first dimension. So essentially, we take the value of pixel 1,1 across all threes, add them up and divide by the number of images. So this will give us the value of pixel 1,1 in our average 3. Then we do the same for pixel 1,2 and so on:

[ ]:
```
mean3 = stacked_threes.mean(0) #0 is the first dimension
mean7 = stacked_sevens.mean(0)
show_image(mean3);show_image(mean7);
```

Since we added all images accross the first dimension, we end up with a 2-d tensor of size 28,28:

[ ]:
```
mean3.shape,mean7.shape
```

[ ]:
```
(torch.Size([28, 28]), torch.Size([28, 28]))
```

Make predictions

So in order to predict if a given image is a 3 or a 7, we compare it (pixel by pixel) to the average 3 and the average 7. If our given image is "closer" to the average 3, it means its a 3, and vice versa.

We can't directly substract the difference between each pixel because we would end up with positive and negative values which would cancel each other out. For instance, if our text image has pixel 1,1 value of 0 and our average 3 has pixel 1,1 of value 0.5, then we would end up with -0.5 and then if pixel 1,2 has a value of 0.5 in our test image, and value of 0 in the average image, we end up with a value of 0.5. When we add up all the values at the end, those would cancel each other out, and we would lose information.

To calculate the difference or distance, between two data sets (a test image and an average image in our case), we need to cancel out the negatives. In other words, we don't want to know the direction of the difference, only the magnitude. The two main ways of doing this are:

- L1 Norm aka MAE –> Mean Absolute Error (where we take the absolute value of the difference to eleminate negative differences, so in the example above, both values would be +0.5) –> (test_img - mean_img).abs().mean()
- L2 Norm aka RMSE –> Root Mean Squared Error (where we take the difference between two pixels, square it which eliminates negative values, add all of them up and then take the square root of the result) –> ((test_img - mean_img)**2).mean().sqrt()

These two approaches are similar but the RMSE will penalize higher differences more because of the squaring of those differences. So for instance, if a given image of a 3 has random black pixels all over (noise), the error will be higher with RMSE than MAE because it heavily penalizes outliers.

[ ]:
```
a_3 = stacked_threes[1] #A random image of a 3 from the training set
show_image(a_3);
```

Calculate MAE and RMSE between a random 3 and the mean 3

[ ]:
```
dist_3_abs = (a_3 - mean3).abs().mean() #Calculate MAE distance (outputs a 1-d␣
 ↪tensor)
dist_3_sqr = ((a_3 - mean3)**2).mean().sqrt() #Calculate RMSE distance (outputs␣
 ↪a 1-d tensor)
print(f"type(dist_3_abs) / type(dist_3_sqr) --> {type(dist_3_abs)} /␣
 ↪{type(dist_3_sqr)}")
print(f"The MAE distance between a random 3 and the mean 3 ---> {dist_3_abs:.
 ↪4f}")
print(f"The RMSE distance between a random 3 and the mean 3 ---> {dist_3_sqr:.
 ↪4f}")
```

```
type(dist_3_abs) / type(dist_3_sqr) --> <class 'torch.Tensor'> / <class
'torch.Tensor'>
The MAE distance between a random 3 and the mean 3 ---> 0.1114
The RMSE distance between a random 3 and the mean 3 ---> 0.2021
```

As mentioned above, most of the time, RMSE will be higher than MAE distance/error because of the squaring factor

Calculate MAE and RMSE between a random 3 and the mean 7

[ ]:
```
dist_7_abs = (a_3 - mean7).abs().mean() #Calculate MAE distance (outputs a 1-d␣
 ↪tensor)
dist_7_sqr = ((a_3 - mean7)**2).mean().sqrt() #Calculate RMSE distance (outputs␣
 ↪a 1-d tensor)
print(f"type(dist_7_abs) / type(dist_7_sqr) --> {type(dist_7_abs)} /␣
 ↪{type(dist_7_sqr)}")
print(f"The MAE distance between a random 3 and the mean 3 ---> {dist_7_abs:.
 ↪4f}")
print(f"The RMSE distance between a random 3 and the mean 3 ---> {dist_7_sqr:.
 ↪4f}")
```

```
type(dist_7_abs) / type(dist_7_sqr) --> <class 'torch.Tensor'> / <class
'torch.Tensor'>
The MAE distance between a random 3 and the mean 3 ---> 0.1586
The RMSE distance between a random 3 and the mean 3 ---> 0.3021
```

So if we compare the MAE between the 3 and the mean 3 with the MAE between the 3 and the mean 7, we can see it is smaller, which means that our random 3 image is "closer" to the average 3 than the average 7, and it is likely a 3.

[ ]:
```
dist_3_abs < dist_7_abs
```

[ ]:
```
tensor(True)
```

Same for the RMSE distance:

[ ]:
```
dist_3_sqr < dist_7_sqr
```

[ ]:
```
tensor(True)
```

PyTorch already provides both of these as *loss functions*. You'll find these inside `torch.nn.functional`, which the PyTorch team recommends importing as `F` (and is available by default under that name in fastai):

[ ]:
```
F.l1_loss(a_3.float(),mean7), F.mse_loss(a_3,mean7).sqrt()
```

[ ]:
```
(tensor(0.1586), tensor(0.3021))
```

[ ]:
```
(a_3 - mean7).abs().mean(),((a_3 - mean7)**2).mean().sqrt()
```

[ ]:
```
(tensor(0.1586), tensor(0.3021))
```

NumPy vs PyTorch - NumPy arrays and PyTorch tensors are similar in functionality, they are both mutlidimensional arrays, but tensors have extra limitations and extra capabilities.

- Tensors need to have regular, rectangular shapes, whereas np arrays can be jagged (for example a list of matrices where the matrices have different sizes).
- Both libraries use compiled C code to run computations much faster than plain vanilla python. In addition, tensors can run on GPUs, which allows for even faster computations and more importantly, they can compute gradients very effieciently.
- Both libraries use (mostly) the same syntax, and support the same operations (it's their backend implementation that differs).

Calculating Metrics

To determine how good/accurate this baseline model is, we need to test it on the validation set. The MNIST dataset has already split the data into training and validation sets (so we don't need to split it ourselves). We simply need to load those validation images into tensors and compare them to our mean 3 and 7 to see how often we predict the correct label/target.

Just like we did for the training data set, we will load all the valid/3 and valid/7 images into two lists of 2-d tensors, which we then stack into 3-d tensors before normalizing all pixel values (between 0 and 1):

[ ]:
```
valid_3_tens = torch.stack([tensor(Image.open(o))
                            for o in (path/'valid'/'3').ls()])
valid_3_tens = valid_3_tens.float()/255
```

```
valid_7_tens = torch.stack([tensor(Image.open(o))
                            for o in (path/'valid'/'7').ls()])
valid_7_tens = valid_7_tens.float()/255
valid_3_tens.shape,valid_7_tens.shape
```

[ ]: (torch.Size([1010, 28, 28]), torch.Size([1028, 28, 28]))

So we have 1010 28x28 images of 3s and 1028 28x28 images of 7s in the validation sets

Now we need to define a function that calculate the MAE distance between two images (a test image and one of the mean images):

[ ]:
```
def mnist_distance(test_img, mean_img):
    return (test_img-mean_img).abs().mean((-1,-2)) #equivalent to (a_3 - mean7).
 ↪abs().mean() from above
```

We specify .mean((-1,-2)) to tell pytorch to take the mean across the last 2 dimensions (width and height of the image). This allows us to pass in either 2-dimensional tensors for test_img and mean_img (aka one test image and one mean image), or a 3-dimensional tensor for test_img, which will be a tensor of image tensors.

[ ]:
```
mnist_distance(a_3, mean3) #Calculating the distance between one image and the␣
 ↪mean 3
```

[ ]: tensor(0.1114)

[ ]:
```
valid_3_dist = mnist_distance(valid_3_tens, mean3) #Calculating the distance␣
 ↪between ALL our valid 3s and the mean 3

print(f"valid_3_tens.shape --> {valid_3_tens.shape}")
print(f"mean3.shape --> {mean3.shape}")
print(f"\n(valid_3_tens - mean3).shape --> {(valid_3_tens - mean3).shape}")
print(f"(valid_3_tens - mean3).abs().shape --> {(valid_3_tens - mean3).abs().
 ↪shape}")
print(f"\nvalid_3_dist.shape --> {valid_3_dist.shape}")
```

```
valid_3_tens.shape --> torch.Size([1010, 28, 28])
mean3.shape --> torch.Size([28, 28])

(valid_3_tens - mean3).shape --> torch.Size([1010, 28, 28])
(valid_3_tens - mean3).abs().shape --> torch.Size([1010, 28, 28])

valid_3_dist.shape --> torch.Size([1010])
```

This works by using broadcasting. We pass in valid_3_tens, which is a 3-d tensor containing 1010 28x28 images of 3s from the validation set, and a single mean3 image. PyTorch will implicitly broadcast the mean3 accross each row (image) of the 3-d tensor: - (test_img-mean_img) –> outputs 1010,28,28 tensor, where each 28x28 matrix is filled with values that represent the difference between each pixel of each image in our valid_3_tens tensor, and the mean3 tensor - (test_img-mean_img).abs() –> the shape doesn't change (1010,28,28) the abs() is applied to each individual

value in all tensors (by making it positive) - (test_img-mean_img).abs().mean((-1,-2)) –> outputs a 1010 tensor where each value is the average/mean difference of that particular image and the mean3 (so for each of the 1010 images, it sums up the differences in the 28x28 matrix and divides them by the total number of pixels in an image, 784)

Now we define a function that indicates whether a particular image is a 3 or a 7. If the distance between the image and mean3 is smaller than the distance between the image and mean7, then the image is a 3 (True). If not, the image is not a 3 (False) and is thus a 7.

[ ]:
```
def is_3(x):
    return mnist_distance(x,mean3) < mnist_distance(x,mean7)
```

[ ]:
```
is_3(a_3), is_3(a_3).float() #For a single image
```

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

[ ]:
```
is_3(valid_3_tens),is_3(valid_3_tens).shape #For all the 1010 images of 3s in␣
↪the validation set
```

[ ]:
```
(tensor([True, True, True,  …, True, True, True]), torch.Size([1010]))
```

[ ]:
```
accuracy_3s = is_3(valid_3_tens).float().mean()
accuracy_3s
```

[ ]:
```
tensor(0.9168)
```

So is_3(valid_3_tens).float() returns a tensor of size 1010, where each value is True (1.) or False (0.) if the image is closer to the mean3 or the mean7 respectively. By taking the mean, we get the number of images identified as 3s. Since we are using the valid_3_tens tensor, all images are 3s, so if the model were perfect, the 1010 tensor would be all True (1.) and the accuracy would be 1 (100%).

[ ]:
```
accuracy_7s = (1 - is_3(valid_7_tens).float()).mean()
accuracy_7s
```

[ ]:
```
tensor(0.9854)
```

Since is_3() function returns True(1.) if the image is a 3, and False(0.) if the image is a 7, when we pass in valid_7_tens, it returns a tensor where all the 7s are identified as False(0.). We can transform it into a tensor where all the 7s are identified as True by substracting each value from 1. So if an image is identified as a 7, the value would be 0, and 1-0 would equal 1, so it would be True, and if the image is a 3, it would be identified as True(1.) and 1-1 is 0, so False. It baiscally flips all True and False values.

So the total accuracy of the model, is the average accuracy of the 3s and the 7s:

[ ]:
```
(accuracy_3s+accuracy_7s)/2
```

[ ]:
```
tensor(0.9511)
```

SGD: Stochastic Gradient Descent

An alternative approach to the pixel similarity model, is to use a learnable linear function. So instead of defining and average 3 and an average 7 and comparing each pixel to an input image, we define a linear function, that takes each pixel as input and produces a number/scalar output. We assign each pixel a weight/parameter that will be adjusted and learned by the model during training.

ouput = (w1 * pixel1) + (w2 * pixel2) + (w3 * pixel3) ... + (w784 * pixel784)

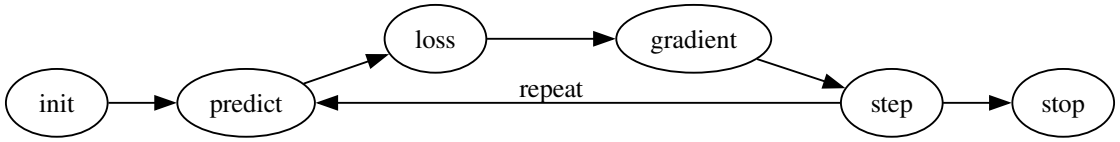
*Our images are 28x28 for a total of 784 pixels. We can represent that 28x28 matrix as a vector by taking each row of 28 pixels and appending it at the end of the previous row, so we end up with a 1x784 vector that represents an image.

During training, the model will learn and adjust the values of all the w parameters, so that it gives out some value for 3s and some other value for 7s. So the outputs for all 3 images would be close to each other and the outputs for all 7s would be close to each other.

To be more specific, here are the steps that we are going to require, to turn this function into a machine learning classifier:

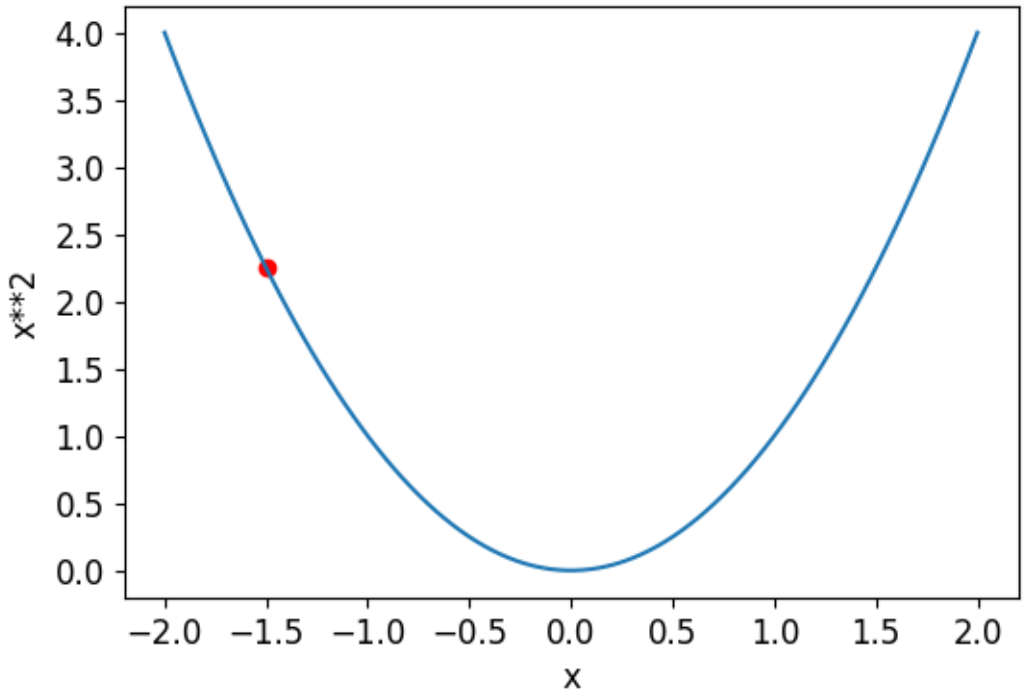
- Initialize the weights.
- For each image, use these weights to predict whether it appears to be a 3 or a 7.
- Based on these predictions, calculate how good the model is (its loss).
- Calculate the gradient, which measures for each weight, how changing that weight would change the loss
- Step (that is, change) all the weights based on that calculation.
- Go back to the step 2, and repeat the process.
- Iterate until you decide to stop the training process (for instance, because the model is good enough or you don't want to wait any longer).

[ ]:
```
gv('''
init->predict->loss->gradient->step->stop
step->predict[label=repeat]
''')
```

[ ]:


To understand gradient descent, we have to think of the loss as a function of the parameters. In other words, lets say we have a 2-d plot where the x axis is the weight vector (all the weights) and the y axis is the loss. Every time we calculate the loss, we get a point on that plot, that point represents the value of the loss for that particular set of weights. Then we adjust the weights and do another pass and calculate the loss again, so we get a second point in our plot, that represents the new loss based on the adjusted weights. The goal is to find the set of weights, where the value of the loss (the y axis in our plot) is at a minimum.

```
[ ]: def f(x): return x**2
     plot_function(f, 'x', 'x**2')
     plt.scatter(-1.5, f(-1.5), color='red');
```



In the plot above, the x and y axes are scalars (a single number) so it is easy to see that by modifying x to get closer to 0, we get a smaller value on the y axis.

Another way of figuring out how to adjust the value of x to minimize y is to use the derivative of our function. The derivative of a function is another function that indicates the direction and magnitude of change at each given point. The derivative is rise/run where rise is the change of the ouput (y) and run is the change of the input (x). So it is a function that tells us how much and in what direction the output (y) will change based on a change in the input (x). Calculating the derivative (which is a function) at a specific point, it is the same thing as calculating the tangent in the original function at that given point. So in the plot above, at the red dot (x=-1.5), the tangent is a straight line going down towards the right. Which means that at x=-1.5, in order to minimize the value of y, we need to increase x (closer to 0).

In our case, the x axis is not a scalar, but a vector with 784 values in it. It's not so easy to identify how we should change that vector to minimize loss, which is where gradients come in. Gradients are essentially derivatives in higher dimensions. If we have a weight vector with 4 values, the gradient would be another vector with 4 values, and each of those values represent how much the corresponding weight contirbuted the current value of the loss.

To calculate the gradient of our first weight, we would calculate the derivative of that weight vs the loss, while treating the other 3 weights as constants. Then to calculate the gradient of the second weight, we do the same thing and keep all other weights constant, and so on.

```
[ ]: xt = tensor([3.,4.,10.]).requires_grad_()
     xt
```

```
[ ]: tensor([ 3.,  4., 10.], requires_grad=True)
```

In the example above, we define a vector with 3 elements, and call requires_grad_() on it to flag that tensor. That flag tells pytorch to start tracking operations on xt so we can evenntually calculate its derivative.

```
[ ]: def f(x): return (x**2).sum()

     yt = f(xt)
     yt
```

```
[ ]: tensor(125., grad_fn=<SumBackward0>)
```

```
[ ]: yt.backward()
     xt.grad
```

```
[ ]: tensor([ 6.,  8., 20.])
```

Once we finish all operations on our starting xt tensor, we call backward() on the output/last tensor calculated from xt. This will propagate the gradient calculation backwards from yt to xt.

Our gradient vector is equivalent to a multidimensional tangent to the loss vs parameters function. That vector always points in the direction of the highest (local) loss. If you think of a vector with 3 elements, in 3-dimensions, each element of the vector indicates the units of the x, y and z axes that make up that vector. The idea is the same for gradients, which are vectors with (many) more dimensions, and each element of that vector indicates the units that make up that gradient vector. Since that gradient vector points in the direction of the highest loss, if we take the negative of it (the opposite vector), we get a vector that points towards the lowest loss.

Stepping and learning rate

Once we calculate the gradient of the loss, which is a vector that points to the highest local loss, we can take its inverse (-gradient) which will be a vector that points towards the lowest local loss. Each elements of that -gradient vector tells us how much we need to adjust the corresponding parameter/weight (in the original weight vector) to push our loss towards a lower value. In other words, once we have a gradient vector, we adjust each of the original weight elements by substracting its corresponding value in the gradient vector:

weights = [a, b, c]

...Computations...loss.backward()...

weights.grad = [gra, grb, grc]

...Step weights...

weights = [(a-gra), (b-grb), (c-grc)]

The weights have been slightly adjusted based on the gradients we calculated and they should allow the model to make better predicitions on the next pass, and minimize the loss.

The learning rate comes in during the weight adjustement step (a-gra, b-grb, etc.). Instead of substracting the full gradient from the weight, we will substract a fraction of each gradient. The fraction is called the learning rate. So the last step becomes:

weights = [(a-(LR * gra)), (b-(LR * grb)), (c-(LR * grc))]

LR, the Learning Rate, is a hyperparameter, a parameter that defines how we train the other parameters (the weights). Hyperparameters are defined by us, unlike the weights which are learned automatically by the model. The LR is typically a small value between 0.00001 and 0.1, but can be anything (more on this in future chapters).

Why use a LR, why not adjust the weights with the full gradient values?

Because the gradients are calculated on small batches of inputs (images) and are fare from perfect. They are noisy and the elements can have very different magnitudes, which can lead to unstable training where the loss jumps around without actually converging towards a lower value.

MNIST Loss Function

Create a training input/image set by transforming the 6131x28x28 (rank-3) tensor of 3s and the 6265x28x28 tensor of 7s into a 2-D/rank-2 tensor of 12396x784 (aka (6131+6265)x(28x28=784))

[ ]:
```
stacked_threes.shape,stacked_sevens.shape,torch.cat([stacked_threes,␣
↪stacked_sevens]).shape
```

[ ]:
```
(torch.Size([6131, 28, 28]),
 torch.Size([6265, 28, 28]),
 torch.Size([12396, 28, 28]))
```

[ ]:
```
train_x = torch.cat([stacked_threes, stacked_sevens]).view(-1, 28*28)
train_x.shape
```

[ ]:
```
torch.Size([12396, 784])
```

Each row is a vector image, where all the rows of the origianl 28x28 image/tensors are transformed into a single row vector image of a 3 or a 7 by appending the 28 rows end to end for each image to get a vector of size [1,784]

the view(-1, 28 * 28) essentially describes how we want to view our tensor. Initially, we have a rank-3 tensor of shape 12396x28x28, the arguments to view (-1, 28 * 28) say that we want to view our rank-3 tensor as a rank-2 tensor (matrix), where the second dimension (the columns) are of size 784 (28 * 28) and the -1 tells view to infer the first dimension on its own, based on the rest of the parameters. So we end up with a 12396x784 –> https://pytorch.org/docs/stable/generated/torch.Tensor.view.html

The labels/targets are abitrary, we can define them ourselves. In this case, the sevens are represented as category 0 and the threes as category 1.

[ ]:
```
train_y = tensor([1]*len(threes) + [0]*len(sevens)).unsqueeze(1)
train_y.shape
```

[ ]:
```
torch.Size([12396, 1])
```

Since train_x contains all images of 3s and 7s in the same tensor, train_y needs to be built the same way:

[1]*len(threes) –> creates a tensor filled with 6131 (the number of 3 images) ones (1) –> [1,1,1,1,1,1,1,1,1…,1]

[0]*len(sevens) –> creates a tensor filled with 6265 zeros (0) that represent the category of the 7s –> [0,0,0,0,0…,0]

tensor([1]*len(threes) + [0]*len(sevens)) –> creates a tensor of size 12396, where the first 6131 are 1s and the rest (6265) are zeros –> [1,1,1,1,1…0,0,0,0]

unsqueeze(1) –> adds a new dimension at the speicifed index, 1 so the rank-1 tensor of 12396 becomes a rank-2 tensor of 12396x1, aka a column vector; unsqueeze(0) would add the dimension in the first position and create a horizontal vector of shape 1x12396

A dataset needs to be a list of tuples, where each tuple has the input and the corresponding target output/category (1 or 0)

[ ]:
```
dset = list(zip(train_x,train_y))
dset[0][0].shape,dset[0][1].shape,len(dset)
```

[ ]:
```
(torch.Size([784]), torch.Size([1]), 12396)
```

zip takes each row in the 12396x784 tensor of image vectors and combines it with the corresponding row in the 12396x1 target label (category) matrice into a tuple (like a list)

Do the same for the validation set:

[ ]:
```
valid_x = torch.cat([valid_3_tens, valid_7_tens]).view(-1, 28*28)
valid_y = tensor([1]*len(valid_3_tens) + [0]*len(valid_7_tens)).unsqueeze(1)
valid_dset = list(zip(valid_x,valid_y))
```

Create a function to initialize a set of parameters, and initialize a random (column) weight vector (784x1) and a random scalar bias:

[ ]:
```
def init_params(size, std=1.0):
    return (torch.randn(size)*std).requires_grad_()

weights = init_params((28*28,1))
bias = init_params(1)
weights.shape,bias.shape
```

[ ]:
```
(torch.Size([784, 1]), torch.Size([1]))
```

Create a function to make the predictions. Remeber, the model is linear function of the form

y = w * x + b

aka

output = input_vector * weights_vector + bias

aka (the shapes)

[1] = [1,784] * [784,1] + [1]

So in other words we do a matrix multiplication between the 1x784 matrix (a row vector) representing a single flattened image and the 784x1 matrix (column vector) of weights. In matrix multiplication, the inner dimensions must match and get eliminated in the output, so 1x784 x 784x1 becomes a 1x1 scalar output (to which we add a scalar bias to offset from 0).

[ ]:
```
def linear1(xb):
    return xb@weights + bias

#efficient way of doing: (train_x[0]*weights.T).sum() + bias

preds = linear1(train_x)
preds.shape, preds
```

[ ]:
```
(torch.Size([12396, 1]),
 tensor([[14.6833],
         [17.2826],
         [15.7854],
         …,
         [ 1.1032],
         [ 4.8017],
         [-3.1000]], grad_fn=<AddBackward0>))
```

Gradients are calculated on a function of the value of the loss over a specific weights vector. Since the gradient is the rise/run

(y_new - y_old) / (x_new - x_old)

since we do very small adjustments to the weights at each step we need the loss function to be very sensitive. If we calculate the loss based on accuracy, and accuracy only has 2 possible values:

True/1 when prediciton = label False/0 when predicition != label

so the accuracy values are always 0 or 1 and they do not change often. In order for a given predicition to flip from 0 to 1 or vice versa, it would likely require multiple/bigger steps (updates to the weight), very small updates would rarely change the predicition of a given input, which means (y_new - y_old) would very often be 0. A gradient vector full os 0s is useless for weight adjustements.

Instead of using accuracy as the loss function, we can use prediction confidence, aka how confident (and right/wrong) the model is about each predicition. The prediction confidence is a continuous value which is of course much more sensitive than a binary (True/False) value.

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

def mnist_loss(predictions, targets):
    predictions = predictions.sigmoid()
    return torch.where(targets==1, 1-predictions, predictions).mean()
```

We defined out categories/targets as 0 represents a 7 and 1 represents a 3. So if the targets==1 (meaning the image is a 3), the loss is 1-prediction (because we are trying to push the predicitions of 3 towards the category/label 1) and if targets!=1 (so targets==0) the loss is the prediction itself (because we are trying to push the predictions of 7s towards the category/label 0).

In plain English, this function will measure how distant each prediction is from 1 if it should be 1, and how distant it is from 0 if it should be 0, and then it will take the mean of all those distances.

The sigmoid function is applied to the output to ensure it is in the form of a probability, between 0 and 1.

SGB and MiniBatches

The loss is calculated on each data point (image), for each predicition but the weight are not adjusted after each loss calculation. The optimization step is applied once per batch (typically 32 or 64), after accumulating the gradients of the batche's losses (using the entire dataset would be too inneficient and using a single data point is too imprecise).

One of the ways of improving a model's generalization capabilities is to vary the data during training by using data augmentation, but also by shuffling the inputs of a mini-batch so they are always different from previous epochs. This is handled by the DataLoader and DataSet pytorch/fastai objects.

The Dataset is made from a list of (input,output) tuples:

ds = [(i1,o1),(i2,o2),(i3,o3),(i4,o4)]

which is passed to the Dataloader in order to obtain mini-batches:

dl = DataLoader(ds, batch_size=2, shuffle=True)

list(dl) –> [(tensor(i1, i2), tensor(o1, o2)), (tensor(i3, i4), tensor(o3, o4))]

Final model

Initialize weights and bias parameters (randomly) and create the training and validation data loaders:

[ ]:
```
weights = init_params((28*28,1))
bias = init_params(1)
```

[ ]:
```
valid_dl = DataLoader(valid_dset, batch_size=256)
dl = DataLoader(dset, batch_size=256)
xb,yb = first(dl)
xb.shape,yb.shape
```

[ ]:
```
(torch.Size([256, 784]), torch.Size([256, 1]))
```

Create function to calculate the gradient of the loss for a given input,ouput:

[ ]:
```
def calc_grad(xb, yb, model):
    preds = model(xb)
    loss = mnist_loss(preds, yb)
    loss.backward()
```

Create a function that trains for one epoch and adjusts the weights at each batch:

[ ]:
```
def train_epoch(model, lr, params):
    for xb,yb in dl:
        calc_grad(xb, yb, model)
        for p in params:
            p.data -= p.grad*lr
            p.grad.zero_()
```

Create function that returns the accuracy (human metric) for a single batch:

[ ]:
```
def batch_accuracy(xb, yb):
    preds = xb.sigmoid()
    correct = (preds>0.5) == yb
    return correct.float().mean()
```

Create a function that calculates the accuracy of the validation set after a given epoch by averaging the accuracy of each batch of prediction from the validaiton set:

[ ]:
```
def validate_epoch(model):
    accs = [batch_accuracy(model(xb), yb) for xb,yb in valid_dl]
    return round(torch.stack(accs).mean().item(), 4)
```

Here we are ready to train. We define the learning rate (lr) and combine our weights and bias into a tuple called params. Then we train and print the validation set accuracy after each epoch:

[ ]:
```
lr = 1.
params = weights,bias

for i in range(20):
    train_epoch(linear1, lr, params)
    print(validate_epoch(linear1), end=' ')
```

```
0.9687 0.9692 0.9702 0.9702 0.9702 0.9707 0.9707 0.9722 0.9727 0.9727 0.9731
0.9727 0.9727 0.9727 0.9731 0.9741 0.9741 0.9741 0.9741 0.9741
```

Optimizer

We can use PyTorch's nn.Linear to replace the linear() and init_params() functions.

[ ]:
```
linear_model = nn.Linear(28*28,1)
w,b = linear_model.parameters()
w.shape,b.shape
```

[ ]:
```
(torch.Size([1, 784]), torch.Size([1]))
```

Then we can create an optimizer class that takes care of the stepping of the parameters and the reset of the accumulated gradients:

[ ]:
```
class BasicOptim:
    def __init__(self,params,lr): self.params,self.lr = list(params),lr
```

```
    def step(self, *args, **kwargs):
        for p in self.params: p.data -= p.grad.data * self.lr

    def zero_grad(self, *args, **kwargs):
        for p in self.params: p.grad = None
```

[ ]:
```
opt = BasicOptim(linear_model.parameters(), lr)
```

This is the same as the fastai SGD optimizer class:

opt = SGD(linear_model.parameters(), lr)

Update the train_epoch function to use the BasicOptim object:

[ ]:
```
def train_epoch(model):
    for xb,yb in dl:
        calc_grad(xb, yb, model)
        opt.step()
        opt.zero_grad()
```

Create utiliy function to train for a number of epochs:

[ ]:
```
def train_model(model, epochs):
    for i in range(epochs):
        train_epoch(model)
        print(validate_epoch(model), end=' ')
```

This can be replaced by a Learner:

learn = Learner(dls, nn.Linear(28*28,1), opt_func=SGD, loss_func=mnist_loss, metrics=batch_accuracy)

learn.fit(10, lr=lr)

Nonlinearity

Simplest Neural Network is two linear layers (xb@w1 and res@w2) between a non-linear activation function, in this case, a Rectified Linear Unit aka ReLU that replaces negative numbers by a 0.

The non-linear activation is essential to model complex functions because a series of any number of linear layers in a row can be replaced with a single linear layer with a different set of parameters. Without the ReLU the two linear layers would just become a single different linear layer.

[ ]:
```
def simple_net(xb):
    res = xb@w1 + b1
    res = res.max(tensor(0.0)) #ReLU
    res = res@w2 + b2
    return res
```

[ ]:
```
w1 = init_params((28*28,30))
b1 = init_params(30)
```

```
w2 = init_params((30,1))
b2 = init_params(1)
```

The difference with previous model is that we are using mutliple activations. In the previous model, we had an input image with 1x784 pixels (in a vector) which we multiplied by a 784x1 weights (column) vector to obtain one single output after we summed it all up. In this example we have 30 vectors of shape 784x1 which each represent a set of weights, a weight vector, a neuron. So the 1x784 image is mutliplied by each of those 30 weight vectors and the first layer outputs a vector containing 30 activations which is fed into the second layer (after the ReLU).

In PyTorch, the code above is equivalent to:

[ ]:
```
simple_net = nn.Sequential(
    nn.Linear(28*28,30),
    nn.ReLU(),
    nn.Linear(30,1)
)
```

nn.Sequential creates a module that will call each of the listed layers or functions in turn. nn.ReLU is a PyTorch module that does exactly the same thing as the F.relu function. Most functions that can appear in a model also have identical forms that are modules. Generally, it's just a case of replacing F with nn and changing the capitalization. When using nn.Sequential, PyTorch requires us to use the module version. Since modules are classes, we have to instantiate them, which is why you see nn.ReLU() in this example.

[ ]:
```
dls = DataLoaders(dl, valid_dl)

learn = Learner(dls, simple_net, opt_func=SGD,
                loss_func=mnist_loss, metrics=batch_accuracy)

learn.fit(4, 0.1)
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

At this point we have something that is rather magical: - A function that can solve any problem to any level of accuracy (the neural network) given the correct set of parameters - A way to find the best set of parameters for any function (stochastic gradient descent)

Deeper models (with more layers) give out better performance by allowing us to use less parameters (smaller matrices).

Questionnaire

[ ]:
```
my_tensor = tensor(range(1,10)).view(3,3)
my_tensor, my_tensor*2, my_tensor[1:,1:]
```

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

```
tensor([[ 2,  4,  6],
        [ 8, 10, 12],
        [14, 16, 18]]),
tensor([[5, 6],
        [8, 9]]))
```

1. How is a grayscale image represented on a computer? How about a color image?

   A grayscale image is a matrix (rank-2 tensor H x W) where each pixel's value is its 0-255 grayscale. A color image is a rank-3 tensor of (CH x H x W) where each pixel in the (H x W) matrix making up the image has multiple channels (values)

2. Explain how the "pixel similarity" approach to classifying digits works.

   Compute an average image for each digit (by averaging all the pixels at a given position in all the training images). Then for predicition, compare an input image to the average image of each digit by calculating the average magnitude of the distance between them. The lowest distance is the prediction.

3. What is a list comprehension? Create one now that selects odd numbers from a list and doubles them.

   [2*number for number in list_numbers if number%2!=0]

4. What is a "rank-3 tensor"?

   List of matrices, aka a 3-dimensional tensor

5. What is the difference between tensor rank and shape? How do you get the rank from the shape?

   Rank indiciates the number of dimensions, the shape indicates the size of each dimension. The number of elements in the shape of the tensor is its rank.

6. What are RMSE and L1 norm?

   They are formulas for calculating the distance between two data points, aka loss functions. RMSE squares the difference between each data point in two datasets, averages them up and takes the square root. The L1 norm, the Mean Absolute Error simply takes the average of all the absolute errors in the two datasets. In our example we calulated the distance between the target label and the prediction of each image which measure how confident a model is about a given prediction.

7. Create a 3×3 tensor or array containing the numbers from 1 to 9. Double it. Select the bottom-right four numbers.

   my_tensor = tensor(range(1,10)).view(3,3)

   my_tensor, my_tensor*2, my_tensor[1:,1:]

8. What is broadcasting?

   A basic operation (+, -, *, etc.) with tensors or numpy arrays where the basic operation is applied elementwise between the 2 matrices following some compatibilty rules and automatically expanding the smaller array/tensor to match the shape of the other one.

9. Are metrics generally calculated using the training set, or the validation set? Why?

   Validation set because the training set is prone to overfitting (memorizing the data), using an unseen validation set allows to test wether the model generalizes well (to unseen data) or overfits.

10. What is SGD?

    Stochastic Gradient Descent. Stochastic (aka random) because we are doing gradient descent on each batch and not the entire dataset which introduces some level of randomness in the gradient descent.

11. Why does SGD use mini-batches?

    Because using the entire dataset is too inefficient (slow) and using a single data point has too much randomness, it would slow down convergence of the weights

12. What are the seven steps in SGD for machine learning?

    - initialize random weights
    - make predicitions on a batch of data
    - calculate loss for each predicition (and track the average loss of the batch)
    - accumulate gradients of the loss of each prediciton
    - step parameters once per batch based on the accumulated gradients and the learning rate
    - repeat for all batches and for all epochs, suffling data between epochs
    - stop once validation accuracy starts getting worse or reach number of epochs needed

13. How do we initialize the weights in a model?

    Randomly of from pre-trained model

14. What is "loss"?

    A highly sensitive metric of performance that the model itself can use to calculate gradients and optimize its parameters through gradient descent.

15. Why can't we always use a high learning rate?

    Because the loss would jump around and converge towards minimum loss very slowly if ever.

16. What is a "gradient"?

    The derivative of an parameter while other parameters are kept constant which indicates how and by how much did that particular parameter contribute to our output (the loss value).Each parameter has its own gradient.

17. Why can't we use accuracy as a loss function?

    It is not sensitive enough. The weights are adjusted by very small amounts at each step, but it unlikely that a small change in the weights would change if a given image is a 3 or a 7 (this would require multiple or bigger weight updates) and the value of accuracy only changes when the prediction for an image changes. Which means the accuracy delta would often be 0, which gives us 0 gradients that cannot be used for optimization.

18. What is the difference between a loss function and a metric?

Loss function is a sensitive measure of performance used by the model to optimize its weights and the metric (like accuracy) is a metric of performance meant for humans/users.

19. What is the function to calculate new weights using a learning rate?

    SGD(linear_model.parameters(), lr)

20. What does the `DataLoader` class do?

    A convenience class that converts raw data into tensors, and stacks them into higher dimension tensors, allows pre-processing and data augmentation and collates the data into batches

21. Write pseudocode showing the basic steps taken in each epoch for SGD.

    - for each data point in a batch calulate loss for that datapoint
    - accumulate the gradient for each element in the batch
    - at end of batch, adjust weights (params -= params.grad*lr)

22. Create a function that, if passed two arguments `[1,2,3,4]` and `'abcd'`, returns `[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]`. What is special about that output data structure?

    func1 (list1, list2): return list(zip(list1, list2))

23. What does `view` do in PyTorch?

    Allows to reshape a tensor, to add or remove dimensions

24. What are the "bias" parameters in a neural network? Why do we need them?

    It allows to offset the predictions from 0, to avoid feeding 0 values to subsequent layers.

25. What does the `@` operator do in Python?

    Matrix multiplication

26. What does the `backward` method do?

    Calculates the gradients of parameters from an output tensor by applying the derivative of each operation in reverse order until it reaches the initial parameters tensor.

27. Why do we have to zero the gradients?

    Because gradients are added to the exisiting value of the parameter gradients. Once we used a gradient to adjust the weights (at the end of a batch), we need to re-initialize them (to 0) for the next batch

28. What information do we have to pass to `Learner`?

    The dataloader(s), the model, the loss, optimization and metric functions

29. What is "ReLU"? Draw a plot of it for values from `-2` to `+2`.

    Activation function that applies non-linearity to the outputs by setting any negative value to 0.

30. What is an "activation function"?

    A function that separates two linear layers by adding non-linearity. Without it, the two linear functions could just be converted into a single different linear function.

31. What's the difference between `F.relu` and `nn.ReLU`?

    nn.ReLU is a module/class and can be used in nn.Sequential to build layered models. F.relu is an utility/quick function version of relu.

32. The universal approximation theorem shows that any function can be approximated as closely as needed using just one nonlinearity. So why do we normally use more?

    For better performance. More layers allow for less parameters (smaller matrices).