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

Greyscale ranges from 0-255. White is 0, Black is 255. The image is just scored as a table of these numbers.

1. How are the files and folders in the MNIST\_SAMPLE dataset structured? Why?

Separate folders for training set and validation set.

This is standard.

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

For example, if you wanted to classify ‘3’, get all the training images of 3, and find the average pixel value of the each pixel on the pixel map. Then to classify, simply match the input image to the number that has the most similar pixel values.

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

A list comprehension looks like this: new\_list = [f(o) for o in a\_list if o>0]. This will return every element of a\_list that is greater than 0, after passing it to the function f. There are three parts here: the collection you are iterating over (a\_list), an optional filter (if o>0), and something to do to each element (f(o)).

OddNumbersDoubled = [d(o) for o in a\_list if o is odd]

This will iterate through a\_list, find an odd number, then pass it to d(o), a function to double numbers, then append it. It will then iterate.

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

A single three-dimensional tensor. I guess if a 2D image is just a X-Y plane, then a 3D image is a X-Y-Z plane. In our case, the rank-2 tensor would be a single image of ‘3’, and the rank-3 tensor is a stack of all the images of ‘3’.

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

The rank is the number of dimensions.

The shape is the length of each axis.

The length of a tensor’s shape is its rank.

For our example, the shape is [6131, 28, 28]. The length, 3, is the rank. The shape tells us that there are 6131 images of ‘3’, and each image is size 28x28 pixels. But the order we arbitrarily set.

Watch out though, sometimes it can be confusing. A 3D vector tensor is rank 1, it’s a vertical [Vx, Vy, Vz]. It’s rank 1 because it has a length of one/because it has one axis of length of three.

1. What are RMSE and L1 norm?

Root mean squared error: find the difference between each value and it’s ideal, then square it, add them all together, then square root it. Divide this by the number of values to find the mean.

L1 norm is mean absolute difference. Just find the mean of the absolute value of the difference between each value and it’s ideal.

The main difference between these two loss functions is that RMSE penalises larger differences more.

1. How can you apply a calculation on thousands of numbers at once, many thousands of times faster than a Python loop?

NumPy and PyTorch are similar modules but PyTorch can use the GPU and calculate gradients. Both however can finish computations many thousands of times faster than pure python. So use them.

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

If tns = tensor([[1, 2, 3], [4, 5, 6]]), then tns[:,1] means to select all the values in the 1st axis.

Tns[1] = [4,5,6], naturally. But then consider [start:end], it makes Tns[1,1:3] mean “from Tns[1], aka [4,5,6], start at [1] and end at [3], so = [5,6]

For the question:

Tns = [[[1,2,3],[4,5,6],[7,8,9]], [[1,2,3],[4,5,6],[7,8,9]], [[1,2,3],[4,5,6],[7,8,9]]]

Tns = Tns\*2

The bottom right corner: Tns[0,2,2]. The others: Tns[0,2,1], Tns[0,1,2] and Tns[1,2,2]

1. What is broadcasting?

Broadcasting is automatically expanding the tensor with a smaller rank to have the same size as the one with the larger rank.

This is hard to get. In practice broadcasting is super useful. If we had a function to calculate the loss of a single image, we can simply pass the entire stack of images to that function and be returned a list of all their loses.

This is indeed the same as the fact that you can subtract tensors from each other, or add scalar values to tensors. The smaller one expands to match.

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

Validation set. Because training data is not generalisable to new data.

1. What is SGD?

It’s the way we make a model learn by updating weights automatically.

Gradient decent is different because each iteration it uses all the training data to update the weights. SGD technically uses a mini batch size of 1, so it updates after each piece of data. The SGD we use uses mini batches though of sizes we set.

1. Why does SGD use mini-batches?

Mini-batch size = 1 means that the model makes predictions on one piece of training data, finds the loss, and updates the weights based on it.

Mini-batch size = training data size means that the model makes predictions on all the training data and updates the weights based on it.

Neither are ideal, because we wouldn’t necessarily want to update the weights largely based on one data point, and obviously updating it based on all the data at one time wouldn’t make sense.

Thus we use mini\_batches, we pick a number of training data points to update on, we find their combined mean loss and update based on that.

A higher mini\_batch size would give a more accurate and stable estimate of how to update the weights, but takes longer.

A GPU works better when doing multiple pieces of work simultaneously,so mini-batches utilise this. So ironically, both small batch sizes and large batch sizes take a long time.

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

Initialise weights (usually randomly).

Make a prediction on a piece of training data.

Calculate the loss.

Find the gradient of the loss, I.E. how does changing the weights affect the loss.

Update the weights in the opposite direction of this. (The ‘STEP’ part).

Repeat the prediction to update parts.

Stop entirely.

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

Either randomly, or we use pretrained weights from a different model. In the latter case for a neural network we usually remove the last node and retrain.

1. What is "loss"?

Loss is a measure of how good the model is doing. It’s for the computer to use to update weights. Metrics like accuracy are what we as humans evaluate and are related to loss.

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

Because the gradient of the loss is not constant. For example, at a given setup, if increasing a weight W1 by 1 decreases the loss by 10, increasing W1 by more than 1 could actually increase the loss.

1. What is a "gradient"?

A gradient is a value that describes how the function will change by varying a parameter. For example, the gradient of x^2 is 2x, telling you that at x = 1, the gradient is 2, saying that (if the gradient stayed constant) increasing x by 1 will increase x^2 by 2.

In our case, the gradient of the loss tells us for the weights, how changing the weights at their current position will affect the loss. But remember, this is only the theoretical assumption that the gradient stays constant, which it doesn’t, so we only change the weights by small amounts every time to improve loss, and then as the gradient has changed, we recalculate it.

1. Do you need to know how to calculate gradients yourself?

No. PyTorch can do it.

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

Because for classification, for a training image, you are either 100% accurate or 0% accurate, right or wrong. You cant update weights with that.

A loss function should imply better performance with a lower value. Accuracy does not.

1. Draw the sigmoid function. What is special about its shape?

Icon

Description automatically generated

In our maths for deep learning, sometimes our loss function assumes our prediction is between 0 and 1.

The sigmoid function always outputs a value between 0 and 1. For both negative and positive values. And as it’s a smooth curve that goes up, SGD can more easily find meaningful gradients.

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

Loss function is for the computer to use to update weights, metrics are for humans to use to evaluate.

A loss function must have a meaningful derivative. If it had big flat sections and or large jumps, then it would not be. As we use the derivative of the loss function to update weights, if it had these sections, then updating the weights to make better predictions would be difficult. For this reason, we use the sigmoid function as/part of a loss function.

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

W = W – W\*-Lr

1. What does the DataLoader class do?

A DataLoader takes any Python collection and burns it into an iterator over mini-batches.

In practice, this means taking a dataset, and randomly splitting it into smaller subsections, mini-batches, of a given size.

While training a model, it improves generalisation if we vary some training elements. We can use DataLoader to do this. Each epoch (run through all the training data), we can use DataLoader to randomise how the mini-batches are constituted.

You can then create a DataLoaders with DataLoader(s). A DataLoaders is just the training and validation DataLoader(s).

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

First split the dataset into training and validation DataLoader(s). Then load them into DataLoaders, which will put them into randomised batches of batch sizes we set.

Then train the model. For each batch, put each item into a prediction function and then put the prediction into a loss function. Then calculate the mean loss, and update the weights with that.

An epoch is one run through the training data. For instance, if we had 100 training items with batch sizes of 10, an epoch would be done after 10 weight updates.

Run the training for as many epochs as you desire.

1. 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?

def tuplemaker(X,Y):

List = []

for i in range len(X):

List.append((X[i],Y[i])

Return List

This function isn’t amazing since python loops are very slow, a way to do it much better is via list comprehension.

I’m not sure if List = [(X,Y) for x in X and for y in Y]] works.

Tuples are important because for training, we want each tuple to contain a piece of training data. It contains an independent and dependent item. For example, the image and it’s classification. In PyTorch, a collection of said tuples is called a Dataset, which we pass into a DataLoader to get training data organised in random batches.

1. What does view do in PyTorch?

It changes the shape of a tensor without changing its contents. For example, turning a list of matrices (rank-3 tensor) to a list of vectors (rank-2 tensor).

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

Recall that the way the model makes a prediction is SUM(weights\*inputs). If an input = 0, then that input contributes nothing. This is a problem for example, if two different inputs had the value 0, they both contribute nothing to the prediction. In an image problem, this would mean white pixels. But a white pixel in a certain part of an image might need a lot more than a white pixel in another part of an image.

This is why we need bias parameters, to resolve this, although im not sure how.

In more general and easier terms, a bias would shift the prediction function in a way the weights could not. Imagine a quadratic prediction function but without a +c, it would lose flexibility.

1. What does the @ operator do in Python?

It represents matrix multiplication.

1. What does the backward method do?

You apply it to the loss function, it calculates the gradient of the loss w.r.t to the weights, so later on your can do weights.grad to access them and update the loss.

But it actually doesn’t just calculate the gradients, it adds the gradients to any gradients previously calculated by it.

1. Why do we have to zero the gradients?

Because loss.backwards adds the gradient to any previously calculated gradient(s). So set it to 0 to reset it when using a new mini batch.

1. What information do we have to pass to Learner?

The model used, the optimisation function (which passes the parameters), the loss function, and any metrics you want to use.

1. Show Python or pseudocode for the basic steps of a training loop.

For x,y in dl: # for the training item and its label (in the training data)

Pred = model(x) # make a prediction using the model

Loss = loss\_func(pred,y) # find the loss for that prediction

Loss.backward() # calculate the loss’ gradient

Parameters -= parameters.grad \* lr # update the parameters/weights

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

A rectified linear unit.

It’s just a linear function, a line, that makes all negative values 0.

A picture containing shape

Description automatically generated

It turns out that every function can be constructed out of a number of ReLU functions. Each one adds another line.

1. What is an "activation function"?

Say we’re trying to make a neural network.

The first layer we set is linear X\*W1 + b1, which has 10 nodes. It takes 30 inputs and produces 10 outputs.

The second layer/node is also linear X\*W2 + b2, which has one node. It takes the 10 outputs and produces 1 final output.

In this form, there’s redundancy, because the composition of two linear functions is another linear function: these two layers can produce the same output as one layer when trained.

Thus we put a layer in between the two layers. We put an activation function, a ReLU in this case, to split them up.

Now our layers are linear,nonlinear/activation function, linear. They can now approximate any function to an arbitrarily high level of accuracy, given the right weights and enough nodes?

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

They do the exact same thing. nn.ReLU is the PyTorch module version, which you have to pair with nn.Sequential if you want to use it.

1. 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?

UAT says that with a linear layer, nonlinearity/activation function, linear layer, neural network with enough nodes can approximate any function.

I’m guessing you need more layers because it simply just takes too long to find the correct weights with so many nodes around.

The more layers there are, the more parameters/weights there are, and the harder it is in practice to optimise them.

But still, a neural network with more than two linear layers can approximate a function with less parameters than a neural network with only two that has a billion nodes.