my understanding of the workflow

Credits: Clark (recitations), Claude Sonnet 4.6, reddit, youtube (I did not keep track of my searches)

Conclusion

Linear regression:
Train Loss: 0.0543
Valid Loss: 0.0711
Test Loss: 0.0673
KNN:
Train Loss: 0.0474
Valid Loss: 0.0517
Test Loss: 0.0558
MLP:
Train Loss: 0.0348
Valid Loss: 0.0497
Test Loss: 0.0460

Unsurprisingly, mlp did the best and linear regression did the worst!

Methodology

What is the point of this problem set?

Use ML models to predict the LOGG (stands for log g ie the surface gravity of a star, on a log scale because the range of values for surface gravity is huge, this makes it more manageable)
Get familiar with the ML workflow by doing this with three different models
Compare the 3 different ML models to see which does better
Learn how to tune hyperparameters

Step 1: Getting the data

First acquire a data set. In our case Hogg has kindly downloaded the data for us from the SDSS so we don't have to suffer the wait times.

- **The output:** First we need a labels file (labels.fits) which is essentially a huge catalog of all the stars which contains the stellar properties of each star, something like in the table below. The labels are needed to essentially provide the answers for the data. So in our case we are interested in the LOGG parameter of the star. So the labels file should have the LOGG parameter of a star and it's identifying features so it can be matched to the downloaded data.
- **The input:** when downloading, we split the data into 3 parts: train, validation, and test. Most of the data should be in train (say 60-70%), and validation is about (10-15%), and test is (20-35%). In our case we did 1024 stars for training, 256 stars for validation, and 512 stars for testing. What our data consists of is the spectrum of each star (ie how much light the star emits at each wavelength).
- **What happens in get_data.py:**
  - First we import the labels. We decide we want to be able to predict the LOGG of red giant stars, so we toss all the labels.fits data that falls out of the TEFF, LOGG, and H ranges of red giants. (note: I am ignoring all sanity check plots, for sanity's sake). Call this RGB_labels.
  - Now we make the train, validation, and test data sets, first by creating a random number generator with seed 17. This will essentially shuffle our data using the seed 17 (kind of like a type, so 17 gives you one way of shuffling the data, 18 gives you a different way of shuffling the data, 20 gives you another, and so on and so far, i.e if everyone in the class uses seed 17, then everyone's data will be shuffled exactly the same way). N_RGB counts how many stars we got when we did the filtering in the first bullet point, so that tells us how many overall stars we have now. This is NOT a list of the stars, it is just a list of the NUMBERS of stars. We then decide how many stars we want to be in each of our three datasets (see the input section above). Then we shuffle in N_RGB using the random seed 17 which we defined initially and put them in a list called I. We can then take the first 1024 numbers in the list and put them in I_train, the next 256 in I_valid, and the next 512 for I_test. Finally we have to make the train_labels, valid_labels, and test_labels, sets, which essentially take the list of numbers in I_train, and matches them to the star that that number matches with in RGB_labels.
  - Now we only care about the LOGG parameter in this particular problem, (i.e. we just want our model to be able to predict the LOGG parameter of a star, given the star spectra) so we can just toss everything from the train/valid/test_labels and keep the LOGG parameter only, so as to avoid confusing the model with too many parameters. Note that in the get_data.py code this is only done for the train_labels, and I think maybe you do this in the validation and test sections of the actual model, but to make everything simpler I am just going to modify the code and add them here.

- Recall that Hogg has already downloaded the star spectra from the SDSS website. We have the labels already, now we just need the inputs. So we can load the data that Hogg downloaded and split and save it as train_features, valid_features, and test_features. Normally what you would do if you were downloading the data yourself, is everything we did above with the labels up until this bullet point, and then you would make a function that would download the spectra for each star in the corresponding labels set. So for example get_features(train_labels) will get the spectrum of each star in the train_labels set, and so on and so forth. Then you can save this in the train_features.npy set as a numpy array.

Star number	TEFF (temperature)	LOGG (surface gravity)	H (metallicity)	Telescope, field, ID (identity)

Step 1.5: Cleaning up get_data.py

I've just removed all the extra downloading data and printing and plotting that I don't need just so everything runs smoother.

Step 2: Linear regression

What does linear regression do?
input -> linear layer -> prediction -> loss -> backprop -> optimization
It assumes that the output (in our case the LOGG parameter of a star) is a linear combination of the input (in our case the spectrum pixels of the star). so if our spectrum has N pixels, the model would look like:

$$LOGG = w1 * pixel1 + w2 * pixel2 + \ldots + wN * pixelN + b$$

where the w terms are weights and b is a bias term. The model tries to find the best values for the weights that gives the correct or close to correct output. We start off by picking random weights, giving it the spectrum, seeing what the predicted LOGG is and comparing it to the actual LOGG value using a loss function (which takes the predicted value and the true value and gives you a single number, the higher it is the worse you are doing, the most common loss function for linear regression is MSE (mean squared error), which tells you how far off you are on average, the goal is to minimize this). We then minimize the loss with backprop (see the info box below), which gives you the gradient of the loss function with respect to each weight. Then we use an optimizer to adjust the weights according to the information provided by the gradient (small/big

gradient: small/big step, positive/negative gradient: backward/forward direction (i.e. subtract/add gradient)). There are different kinds of optimizers which decide how to "react" to the gradient size differently. See the optimization info box. For our case we will be using Adam. Then we just repeat the whole process.

Backprop >

To minimize something, you want to make the gradient of that thing as close to 0 as possible or equal to 0, as we learned in idk calc 2? So we want to take the gradient of the loss function with respect to each weight, if the gradient is large, then we are far from minimizing, so we need to take a big step (i.e adjust the weight by a large amount) and if the gradient is small, then we are close to minimizing, so we need to take a small step (i.e. adjust the weight by a small amount). The sign of the gradient tells you which way to move, if the gradient is negative, this means that increasing the weight makes the loss go down (imagine it as basically moving forward in the same direction as you were going on a downward slope), but if the gradient is positive, this means that increasing the weight will increase the loss, so you want to decrease the weight (imagine it as moving forward in the same direction as you were going on an upward slope, you would get higher, which you don't want, so you want to turn around and start going backwards). Ok so we want the gradient of the loss function with respect to each weight, but the only issue is that the loss depends on the prediction which depends on the weights, so there's an additional middle step. Here is where the chain rule comes in!

$$d(loss)/d(w1) = d(loss)/d(prediction) * d(prediction)/d(w1)$$

What backpropagation does is the "multiplication" part of the chain rule. For linear regression this is pretty simple, but later when we add more "layers" (to be discussed) then you would have something like input -> math layer -> math layer -> math layer -> prediction, and you would have to take the gradient at each layer. In linear regression we only have one layer, that's the linear layer, which is explained above.

Optimization >

SGD (stochastic gradient descent) is the most basic optimizer. It does:

$$w1 = w1 - learningrate * gradient$$

where the learning rate is just a number set by us, so it is a hyperparameter that can be tuned. If my learning rate is too high then I will overshoot and my ML will not learn

anything (i.e. it will never minimize because you are always jumping away from the zero), and if it is too small then it also won't learn anything (presumable because it will take forever to update). Standard learning rate is 0.001.
Adam is a smarter optimizer which does the same basic thing, but it adapts the learning rate for each weight individually. So a weight that is getting the same gradient numbers will get a bigger learning rate for a bigger weight change, and weights that are getting very noisy gradient numbers (like jumping from -2 to 3 to 5 to -1 or something like that) will get a smaller learning rate for a smaller weight change.

Onto the actual coding process! First we are going to import the data labels and features from our get_data script, and convert them to tensors bc they are currently numpy arrays and torch works with tensors. Make sure the data is in float32 for speed, and that they are both in the same format. Since features is in a 2D array, we want the labels to also be in a 2D array format as well so that torch can line up the predictions with the labels (i.e output will be in same format as input, so need true values to be in same format as output for accurate comparison).

Then we define the model, by this we mean the structure of the model. You define the model as a child class of nn.Module. The init function defines the layer, in it we run `super.__init__()` which essentially runs all the initialization of the base class, and then we also have `self.linear=nn.Linear(num_pixels, 1)` which creates a linear layer that expects num_pixels numbers (i.e. pixels per one star) and outputs 1 number (the LOGG value of that star). Then we need `forward(self, x)` which takes the input we gives it and passes it into the Linear layer we initialized at first. Without it, nothing will go into our linear layer.

Then we define our model. Our X_train is the training data set, which contains train_features, which has the shape (1024, num_pixels), i.e 1024 stars, each with num_pixels pixel values. To get the num_pixels value, we need X_train.shape[1]. Then we can pass this through our model, so our model now knows how many inputs to expect. I got 8575 which means that my model is now a linear regression model that expects 8575 input values, and will output 1 value for that .

Then we define the loss and optimization functions, which are just one lines, we picked MSE for loss and Adam for our optimizer so we can just call them from torch.

Then we make the training loop! We pick how many times we want the cycle listed underneath step 2 to repeat, and call it num_epoch. And then we write a for loop, where in each run, we first get our prediction by passing X_train through the model and then calculate the loss with our defined function. This is called the forward pass. Then we do the backwards pass, which first makes sure that any gradients that were stored from previous loops are cleared out, then we do backprop with `loss.backward()` then

we do our optimization by calling the optimization function. When we print the loss, we can see that it decreases.

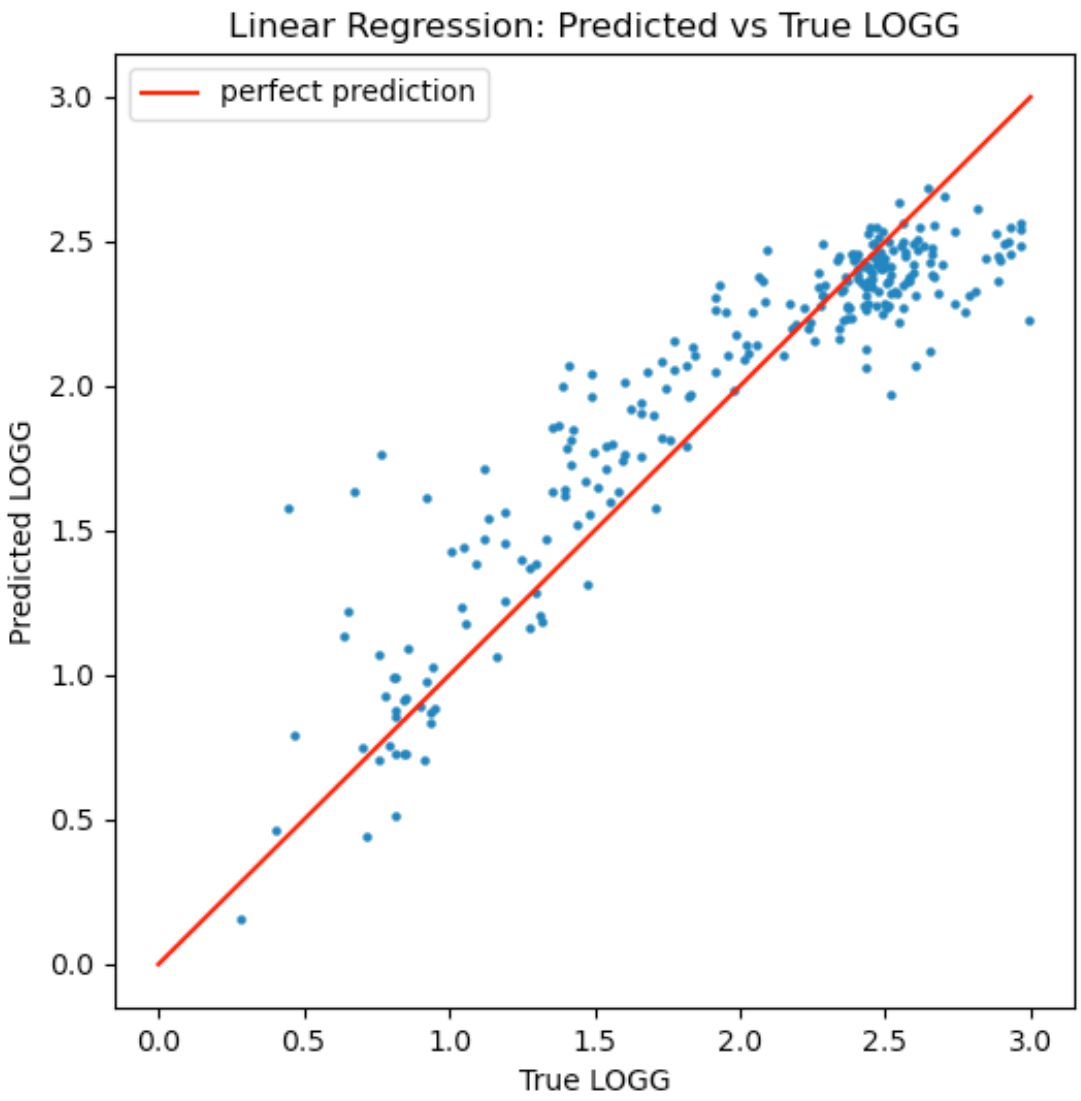
Then we validate. We make the train_preds (predictions) by running model(X_train) and valid_preds the same way. Then we calculate the loss for each. Then we can print the losses and compare them. If the We also include `torch.no_grad()` at the beginning of the validation code because we don't want torch to carry around the gradients since we are not backpropping. This speeds things up. If the training loss and valid loss are close to each other this means that the model is generalizing well to unseen data and is not overfitting. Overfitting essentially means that your model saw your train data too much and memorized it, so it can't make good predictions for new data.

Epoch counts: ›

Epoch count: 100
Train Loss: 0.2368
Valid Loss: 0.2413
Epoch count: 1000
Train Loss: 0.0512
Valid Loss: 0.0693
Epoch count: 10000
Train Loss: 0.0085
Valid Loss: 0.0493 <- you can see the validation data does worse!

We can plot the training losses and the valid losses vs the epoch number to see when they are minimized, and pick the epoch where the validation loss is the most minimal. The code for this is pretty self explanatory I think. I decided on 1000 as the most ideal range.

Finally we plot the validation predictions, first by converting the predictions and true logg values to numpy so we can plot them and then just plot them (self-explanatory). We plot them against each other because we expect to see a diagonal line in the middle (x=y). We can see in the plot below that our model fits better around higher LOGG values, and gets way worse around lower LOGG values. This probably means that our training data set contained more stars with LOGG values closer to 3 than 0.



Step 3: KNN

How it works: KNN does not require training. You simply give it the training data set and it memorizes it. Then when you give it a data set from the validation set it will find the stars most similar to that star and take the average of their logg values and output that. This is done by computing the "distance" (i.e the euclidian distance (distance between 2 points in 2d) between each pixel of the input star and each pixel of the training set stars). Sort by the distance, and pick the K closest stars (see K is a parameter which can be tuned, so how over how many stars do I want to average), and then calculate the average LOGG values, boom, output.

This isn't exactly a nn because there is no training or layers per se. So we import KNN from scikit instead (because writing it manually in torch is too much work). Then we just define k (I set it to 5 to begin with) and define our model as KNeighborsRegressor, with

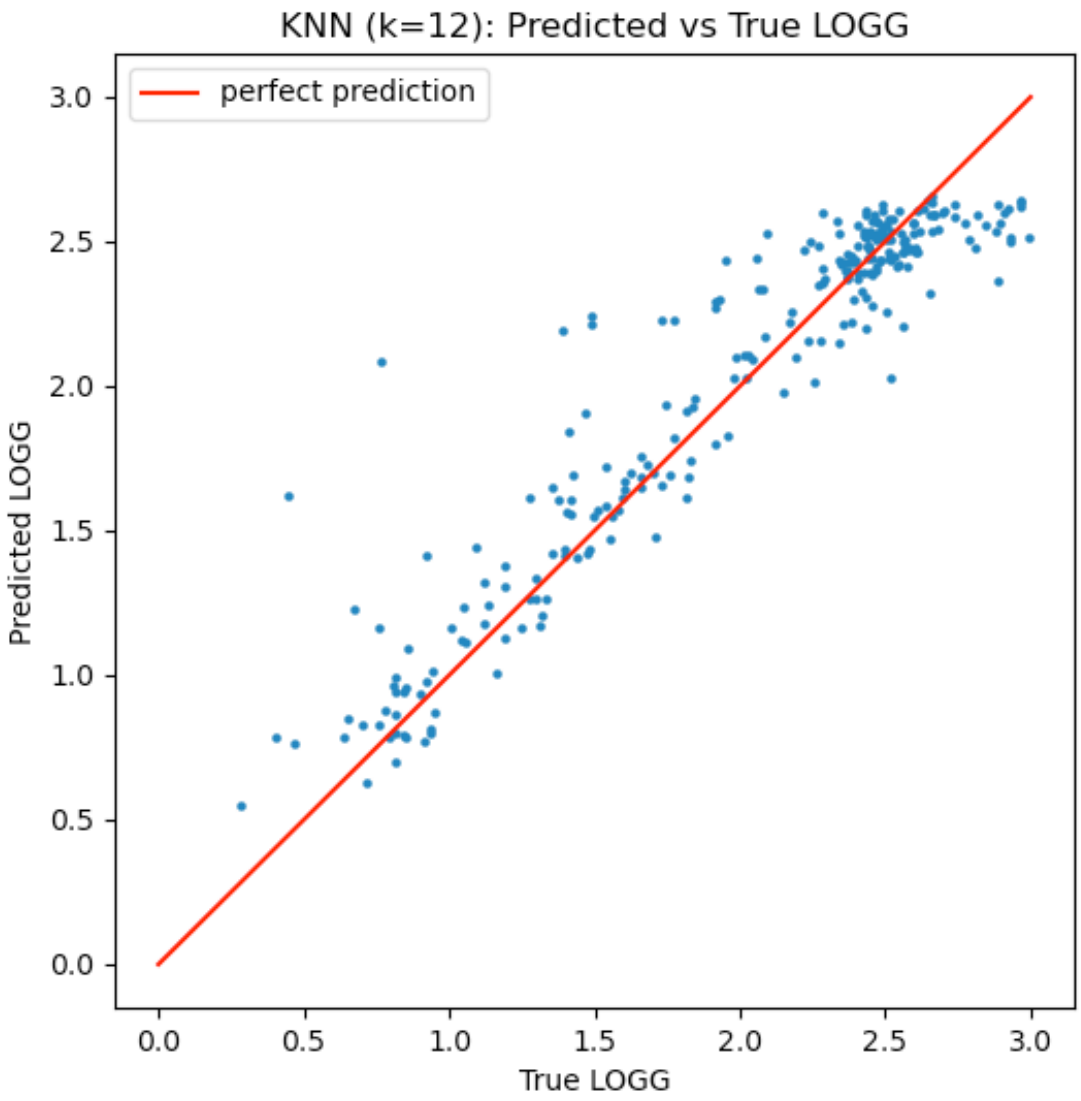
the number of neighbors being k. Then we run model.fit(training features, training labels) which essentially just stores all the training data and their output LOGG values in the model. And that's it!

The validation part of this is really easy, so we do essentially the same thing as before, where we pass the validation data through the model, and calculate the loss using MSE (which we just did manually because we aren't using torch for this), compare the training and validation loss, and plot the validation predictions vs the true validation LOGG values. Basically all exactly what we did for linear regression.

Messing around with k values ›

K value: 5
Train Loss: 0.0377
Valid Loss: 0.0512
K value: 2
Train MSE: 0.0292
Valid MSE: 0.0628
K value: 10
Train MSE: 0.0451
Valid MSE: 0.0516
K value: 15
Train MSE: 0.0514
Valid MSE: 0.0546

I ended up decided on k=12. You can see that KNN does much better than linear regression.



Step 4: MLP

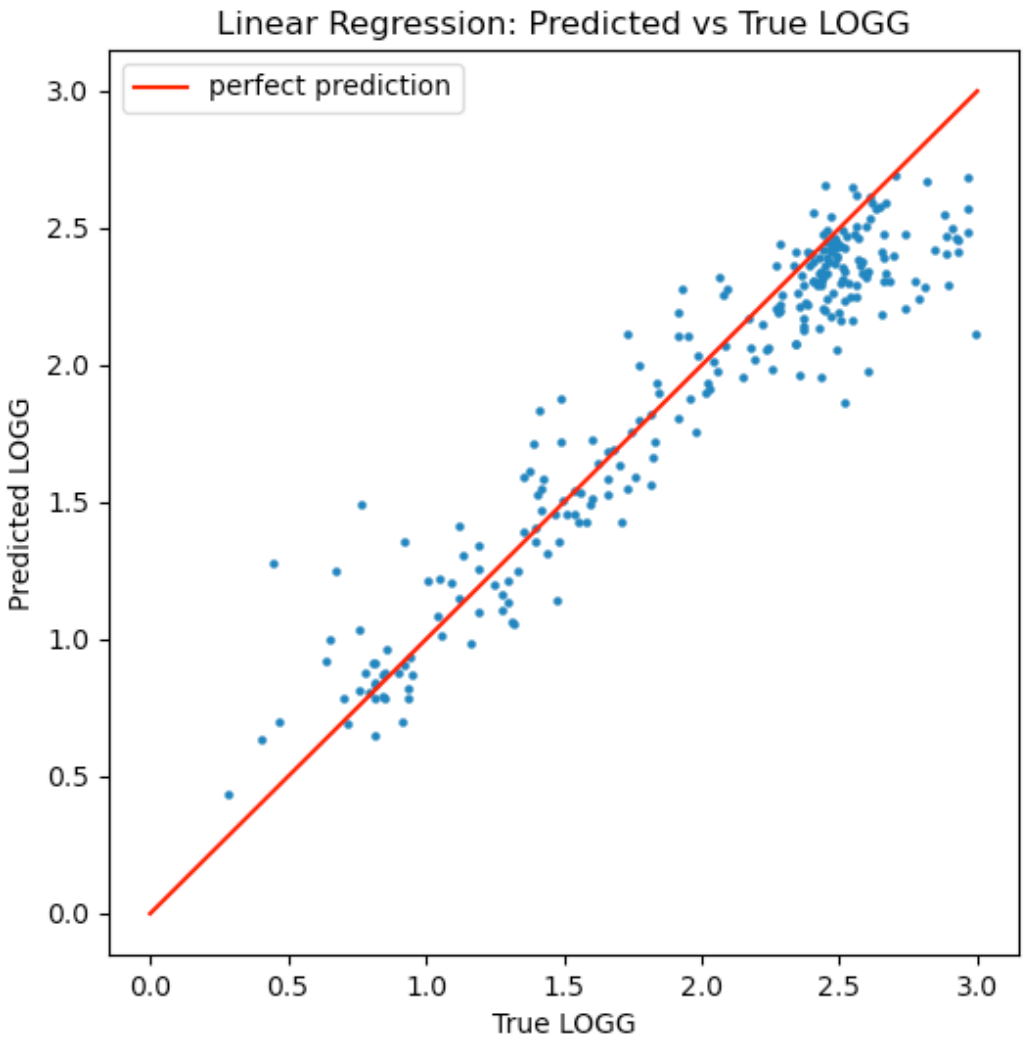
How it works: MLP is a bunch of linear layers (see linear regression) stacked together with something called an activation function in between them. The activation functions adds nonlinearity so that instead of only being able to produce a "straight line" for predictions you can now predict complex "curved" relationships. Without the activation function if you just stack a bunch of linear layers on top of each other you would still have an overall linear model.

A common activation function is called the ReLU (rectified linear unit). It looks something like `ReLU(x) = max(0,x)` and basically what it does is says, if the input number is positive, keep it, if not, make it zero. LeakyReLU let's a tiny amount of negatives through. There are lots of other activation functions, it matters less what kind of activation function you're using, and more that you are introducing nonlinearity into

the system. Of course some activation functions will work better than others and that depends on the system.

Other than that we follow pretty much the same formatting as linear regression!

- import the data as tensors
- define the model, so there are 2 main hyperparameters here, one being how many linear layers you decide to use, and also the activation function you use. I picked ReLU as we said above, and for the linear layers we will start off with 3 layers. You can see in the code a commented out version that has this hard coded into it, so you build it exactly the same way as the linear version, but instead your self.linear layer becomes self.network, and you use sequential to stack the layers on top of each other. It is easier to understand when you look at the code imo. You basically want to go down from the number of pixels being the number of inputs all the way to 1, so you intersperse the layers in between and define the number of inputs and outputs for each linear layer. Instead of doing this manually you can do a loop so you can adjust the size of each hidden layer (i.e. how many inputs it takes and how many it outputs) and also the number of layers. I chose three layers, with sizes, 512,256,128, to create a funnel sort of shape. The code for this is commented on in the file and I think is not too difficult to grasp.
- Now to train this is basically exactly the same setup as linear regression, the only difference is that we add a model.train() line in the for loop. This lets torch know that we are in training mode, because some layers in torch behave differently depending on whether you're training or evaluating. For example there is something called Dropout which randomly turns off some layers during training but during evaluation you would want all the layers to be on. IN Batch norm, it uses statistics from the current batch to train but during evaluation you want it to use stored statistics from the whole training set. These are just ways to improve your training process, and to prevent overfitting. I am not using either of those in my code right now, but it is standard practice to include them, or so I heard.
- For validation I also essentially copied my linear regression code.
- Train Loss: 0.0306 Valid Loss: 0.0461 which is pretty good initially. I also made the plot as well.
- Trying out more layers: [512, 256, 128, 64, 32]
- That gave me worse values. I'm trying three layers with all the same sizes now (512)
- That also did worse. I am going back to the og and trying different epoch ranges now.
- Okay that also did worse. I'm just going to stick to the initial values I picked. -



Step 5: Adjusting hyperparameters

I already did this for each model within the step so I won't do it again. One comment though about changing the RNG seed:
- for KNN and Linear regression it makes no difference because they are deterministic
- for mlp it does, because more complex modesl can be sensitive to initialization

Step 6: Running the tests

Linear regression:
Train Loss: 0.0543
Valid Loss: 0.0711
Test Loss: 0.0673

KNN:
Train Loss: 0.0474
Valid Loss: 0.0517
Test Loss: 0.0558

MLP:
Train Loss: 0.0348
Valid Loss: 0.0497
Test Loss: 0.0460

Unsurprisingly, mlp did the best and linear regression did the worst!