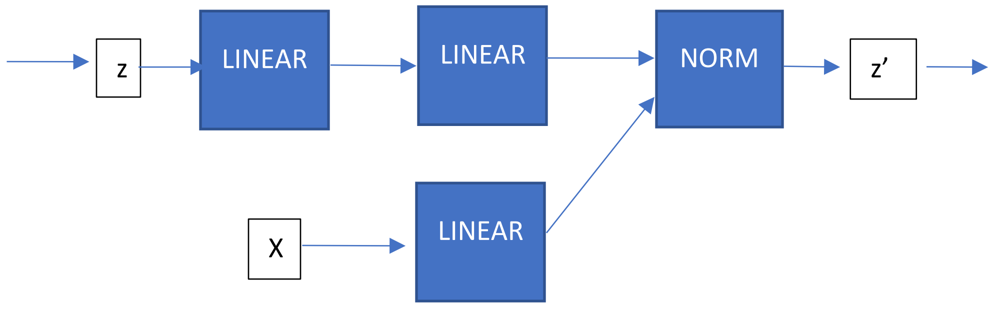
Peer

Neural Networks

My model uses a deep neural network (DNN) to estimate housing prices from the Ames Housing dataset. First, the data must be converted to a usable input for the network. It encodes non-numeric data as integer dictionary values in order of occurrence and NA values as -1. Then we select 10% of the dataset for validation and save the validation set and the remainder of the main dataset separately. This is handled by the "preprocessing" script. The script also applies the same dictionary definitions to the Kaggle test set from Kaggle that lacks price tags, but since Kaggle keeps that information private, this Kaggle set is not useful to us.

The network itself is a deep neural network with residual blocks inspired by resnet. Each residual block takes the input of the previous block and puts it through two linear layers, and puts the original input data through one linear layer. Then it combines the two outputs, performs a batch normalization, and sends them to the next block.



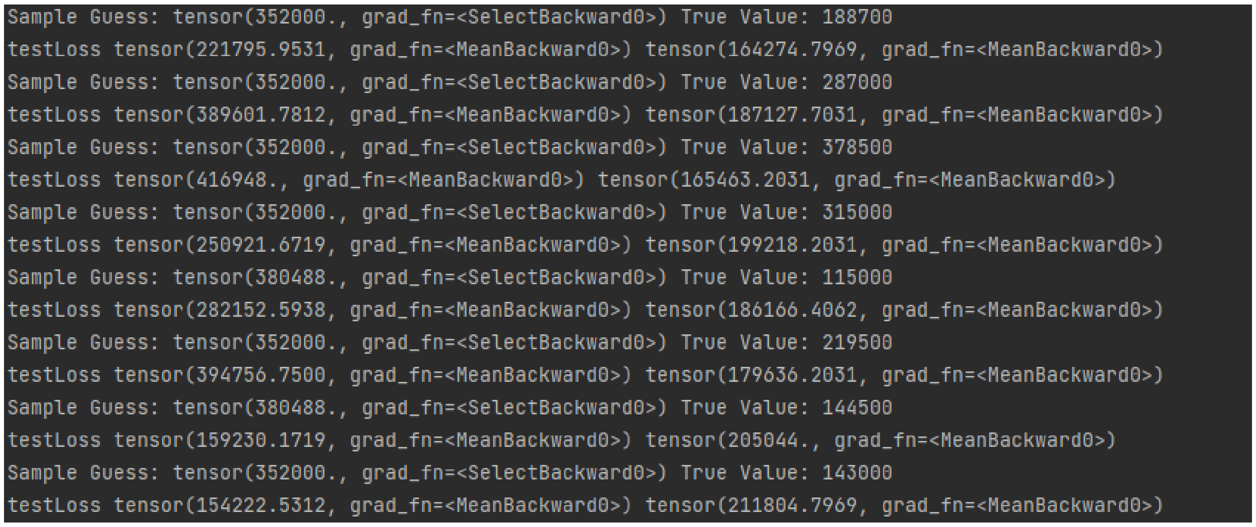
The network has a total of 16 of these blocks, followed by two linear layers and, finally, a sigmoid output. The output is a length 27 vector which is interpreted as a 29-bit binary number. Loss is calculated by converting the price to a 27 bit binary number, calculating the root mean squared error of each bit, and weighing each error by its respective value. In a 3 bit number, for the goal 5: [1,0,1] the output [.5, 0, 0] would have error ||1-.5||*4+||0-0||*2+||1-0||*1. If each output bit is rounded, this is identical to the root mean squared error, but in order for backpropagation to work, the error calculation must be continuous, so we do not round the bits.

It is worth noting that while this gives an accurate error and keeps the output size relatively small, it is not a great method for ordinal classification. The reason is the program essentially has to tune for an output space with a massive 2^27-1 degrees of freedom. It would probably have been better to use bins of prices instead of a numeric output, but the hope was that this approach would work anyways due to the large network size. Indeed, it may work well given enough training time as the network certainly has not overfit to the data in any testing and seems to settle around a mean approximation, which is, on average, about $100,000 off. It is worth noting that I used root mean squared error and not mean squared error because 100000^2 is a very large number.

The network is optimized using the Adam optimizer with an exponential scheduler to decrease the learning rate as episodes progress.

Overall, the network works but could be better. It does not guess the average every time, but it does not stray from guessing near the average.

Examples:



```
Sample Guess: tensor(352000., grad_fn=<SelectBackward0>) True Value: 188700
testLoss tensor(221795.9531, grad_fn=<MeanBackward0>) tensor(164274.7969, grad_fn=<MeanBackward0>)
Sample Guess: tensor(352000., grad_fn=<SelectBackward0>) True Value: 287000
testLoss tensor(389601.7812, grad_fn=<MeanBackward0>) tensor(187127.7031, grad_fn=<MeanBackward0>)
Sample Guess: tensor(352000., grad_fn=<SelectBackward0>) True Value: 378500
testLoss tensor(416948., grad_fn=<MeanBackward0>) tensor(165463.2031, grad_fn=<MeanBackward0>)
Sample Guess: tensor(352000., grad_fn=<SelectBackward0>) True Value: 315000
testLoss tensor(250921.6719, grad_fn=<MeanBackward0>) tensor(199218.2031, grad_fn=<MeanBackward0>)
Sample Guess: tensor(380488., grad_fn=<SelectBackward0>) True Value: 115000
testLoss tensor(282152.5938, grad_fn=<MeanBackward0>) tensor(186166.4062, grad_fn=<MeanBackward0>)
Sample Guess: tensor(352000., grad_fn=<SelectBackward0>) True Value: 219500
testLoss tensor(394756.7500, grad_fn=<MeanBackward0>) tensor(179636.2031, grad_fn=<MeanBackward0>)
Sample Guess: tensor(380488., grad_fn=<SelectBackward0>) True Value: 144500
testLoss tensor(159230.1719, grad_fn=<MeanBackward0>) tensor(205044., grad_fn=<MeanBackward0>)
Sample Guess: tensor(352000., grad_fn=<SelectBackward0>) True Value: 143000
testLoss tensor(154222.5312, grad_fn=<MeanBackward0>) tensor(211804.7969, grad_fn=<MeanBackward0>)
```

Here are some sample guesses and their ground truth on the test set (the guesses tend to be around the average of 354765.3). The testLoss is reported first as the loss the model is training with and second as the average distance from the actual guess as a round number to the true value (averaged over the bin). In this case, these numbers closely resemble each other, but tweaking certain hyper-parameters like the scaled penalty for each digit (in the code this is changed with the “off” variable scaling the loss by (2-off) ^[27,26…] to avoid overflow in early training or allow for larger penalties for guesses that are extremely far off).