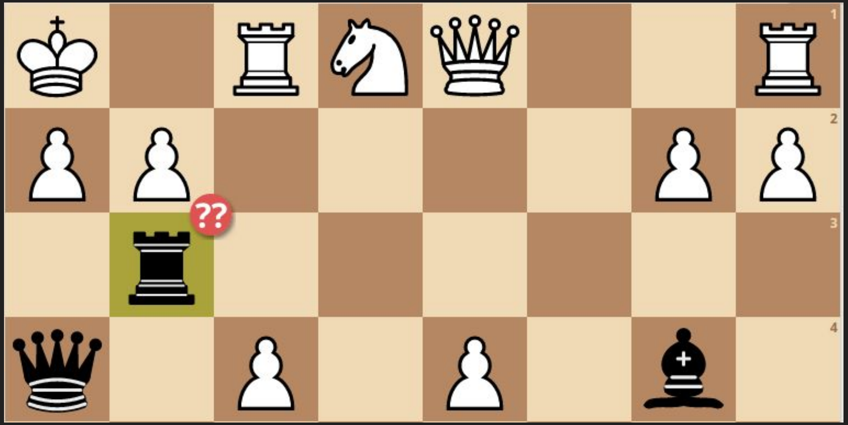
NNUE Neural Networks

Neural networks optimized for efficiency in chess engines



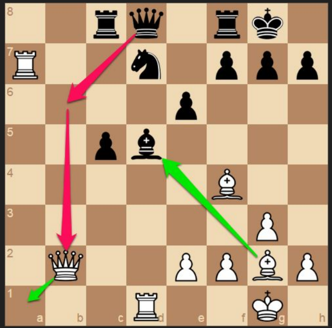
Background

Chess engine: Computer program to analyse a chess position, generating a list of moves which it thinks are strongest.

How do chess engines achieve success?

Human Grandmaster

- Has deep, conceptual knowledge about the game
- Can't calculate more than a ~dozen moves in a second
- Only considers a few moves in a given position



Computer

- Has very limited knowledge about the game
- Can calculate millions of moves in a second

Facts about chess

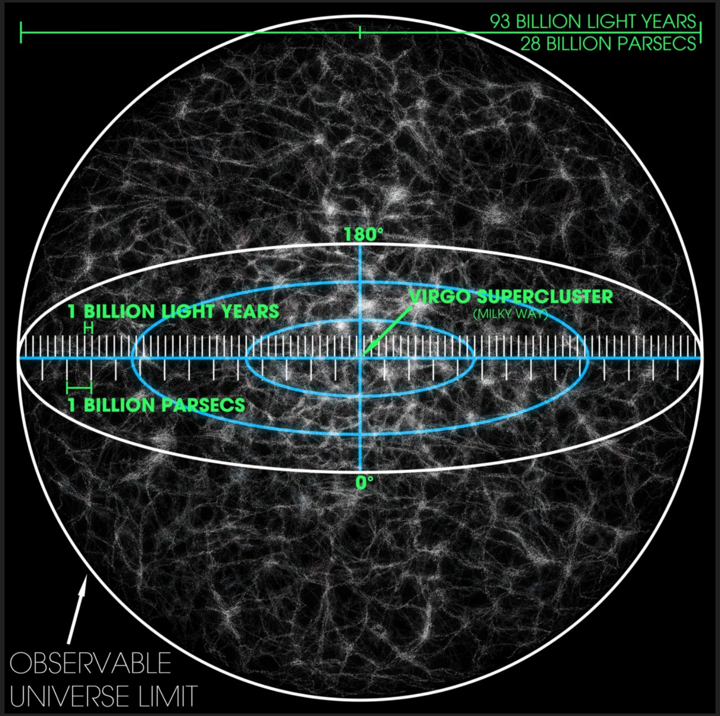
Number of possible chess games ~ **10^120**

Number of atoms in the known universe ~ **10^82**

Clock speed of modern CPUs ~ **10^9 Hz**

Chess variations after 10 moves ~ **10^15**

The brute force approach to chess is impossible



Problem: Static Evaluation

Search depth must be limited. Static evaluation is required (evaluate the position without searching moves)

Traditional approaches:

- Material - who has more pieces?
- Position - who’s pieces are in better places? (e.g. development, pawn structure, etc.)
- King safety - who’s king is more exposed?
- Mobility - who’s position is more flexible?

How do we combine these heuristics?

Problem: Static Evaluation

Hand-making a static evaluation function requires hand-tuning weights.

It is difficult to know which combination of heuristics will perform best

```
int32 EngineV1_3::evaluate(uint8 plyFromRoot)
{
    // Weights
    static constexpr int8 MATERIAL_WEIGHT = 10;
    static constexpr int8 POSITIONAL_WEIGHT = 4;
    static constexpr int8 KING_SAFETY_WEIGHT = 3;
    static constexpr int8 MOBILITY_WEIGHT = 2;
    static constexpr int8 KING_DISTANCE_WEIGHT = 2;

    // Mobility value
    static constexpr int8 PAWN_EARLY_MOBILITY_VALUE = 2;
    static constexpr int8 KNIGHT_EARLY_MOBILITY_VALUE = 3;
    static constexpr int8 BISHOP_EARLY_MOBILITY_VALUE = 3;
    static constexpr int8 ROOK_HORIZONTAL_EARLY_MOBILITY_VALUE = 2;
    static constexpr int8 ROOK_VERTICAL_EARLY_MOBILITY_VALUE = 4;
    static constexpr int8 QUEEN_EARLY_MOBILITY_VALUE = 0;
    static constexpr int8 KING_EARLY_MOBILITY_VALUE = -4;
```

```
    static constexpr int8 PAWN_END_MOBILITY_VALUE = 3;
    static constexpr int8 KNIGHT_END_MOBILITY_VALUE = 2;
    static constexpr int8 BISHOP_END_MOBILITY_VALUE = 2;
    static constexpr int8 ROOK_HORIZONTAL_END_MOBILITY_VALUE = 3;
    static constexpr int8 ROOK_VERTICAL_END_MOBILITY_VALUE = 3;
    static constexpr int8 QUEEN_END_MOBILITY_VALUE = 2;
    static constexpr int8 KING_END_MOBILITY_VALUE = 0;

    static constexpr int8 PIN_MOBILITY_PENALTY = 5;
    static constexpr int8 CASTLING_MOBILITY_BONUS = 5;
```

Example of only **some** of the weights in my evaluation function

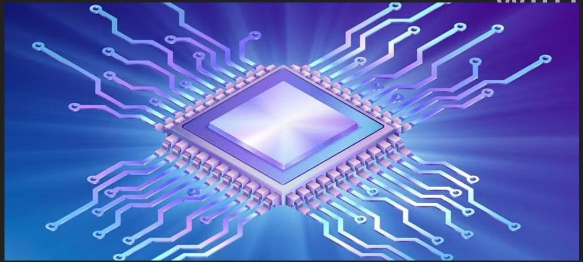
Problem: Static Evaluation - Neural Network

A neural network can learn the weights for me!

If given an chess position (encoded as a vector) a neural network could learn its own combination of heuristics

<u>Problem</u>: Neural networks are slow, especially on the CPU

- Neural networks require large matrix multiplication with floating point numbers
- CPU are best for fast, sequential operations with integer numbers



Solution: NNUE

NNUE (ƎUИИ Efficiently Updatable Neural Network)

- Popularized by Stockfish, NNUE is a type of neural network optimized for turn-based game evaluation functions. It is a fully connected neural network

Two main principles to achieve fast inferences:

- **“Efficiently updating”** - only part of the network needs to be re-evaluated after every move
- **Integer quantization** for fast evaluation on CPU

Efficiently Updating

Step 1: Ecode chess position as sparse one-hot encoded vector of “active features” (e.g. “white pawn on E5”)

Step 2: Make first layer of network be a fully connected linear layer

Step 3: Once network is evaluated, each move only requires taking the added/removed features and adding/subtracting the corresponding weights to the output of the layer (before activation). Call this the “Accumulator”

Example

```cpp
void NNUE::updateAccumulatorMove(Accumulator& input, Accumulator& output, int removedFeature, int addedFeature)
{
    // Subtract the weights vector for the removed feature
    for (int i = 0; i < HIDDEN_1_SIZE; i++) {
        output.vec[i] = input.vec[i] - SPARSE_LINEAR_WEIGHT[removedFeature * HIDDEN_1_SIZE + i];
    }

    // Add the weights vector for the added feature
    for (int i = 0; i < HIDDEN_1_SIZE; i++) {
        output.vec[i] += SPARSE_LINEAR_WEIGHT[addedFeature * HIDDEN_1_SIZE + i];
    }
}
```

The “Accumulator” is what stores the output of the first layer before activation

In the case of a capture, two features would be removed instead of one

Integer Quantization

Step 1: Train the network with floating point weights

- During training, clamp hidden layers to [0, 1] (clipped rectified-linear unit)

Step 2: Choose integer precision for hidden layers

- 8 bit precision for hidden layers means new activation [0, 127]

Step 3: Scale weights and biases into integer domain:

- Scale inputs/bias by 127 (need [0, 1] -> [0, 127]
- For each layer, additionally scale weights/bias by some factor of choice. Divide output of each layer by that factor before activation

Why Quantization is so important

CPUs are very fast with low precision integers

- Modern CPUs can perform arithmetic with 64 int8s simultaneously (SIMD)
- These "vector instructions" can be utilized in the neural network

This loop…

```
// Add the weights vector for the added feature
for (int i = 0; i < HIDDEN_1_SIZE; i++) {
    output.vec[i] += SPARSE_LINEAR_WEIGHT[addedFeature * HIDDEN_1_SIZE + i];
}
```

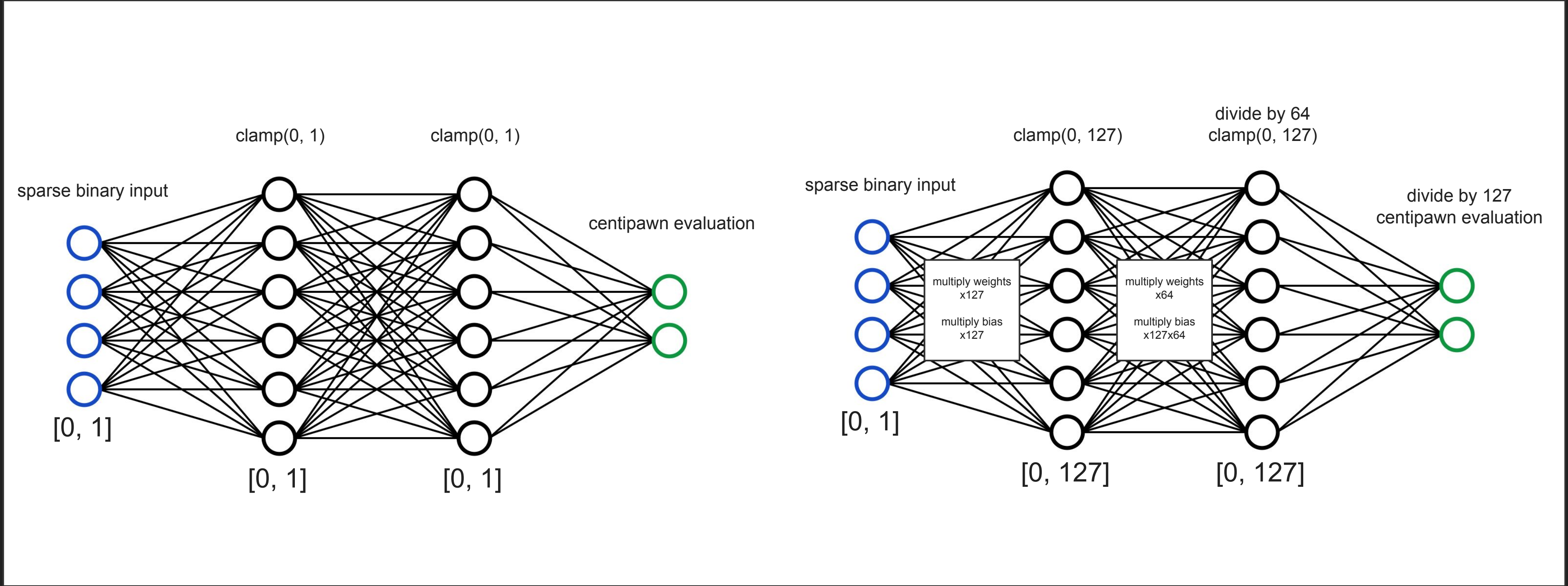
…becomes this single instruction

```
// Add the weights vector for the added feature
accumulator = _mm256_add_epi16(accumulator, _mm256_load_si256((__m256i*) & SPARSE_LINEAR_WEIGHT[addedFeature * HIDDEN_1_SIZE]));
```

Example

During training

After quantization



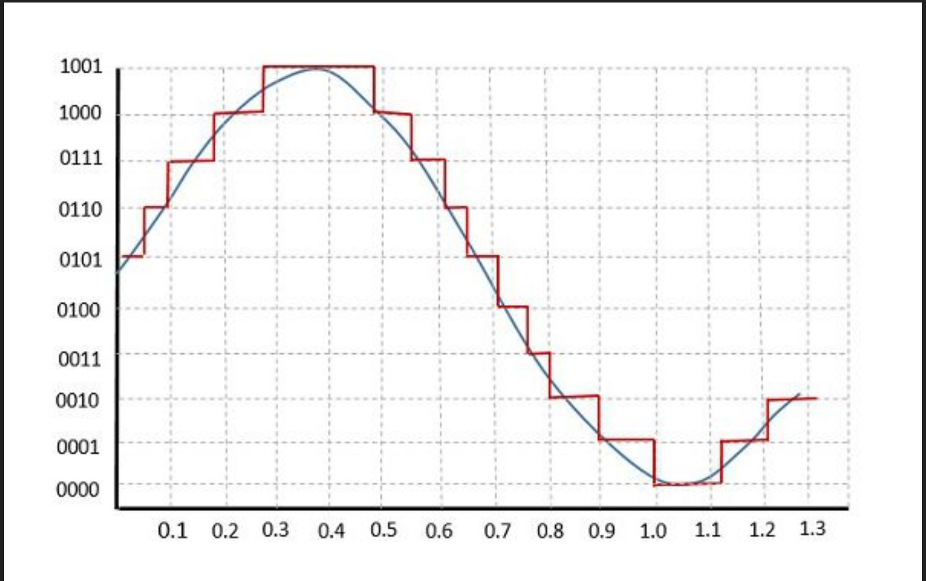
Why are weights scaled by “scaling factor”

Quantization: everything is turned to integers - **accuracy loss**

Scaling weights before converting to integer **preserves some precision**

Can’t scale too much - to have int8 weights, must be in range [-127, 127]

With scaling factor 64, maximum weight is 127/64 = 1.984375

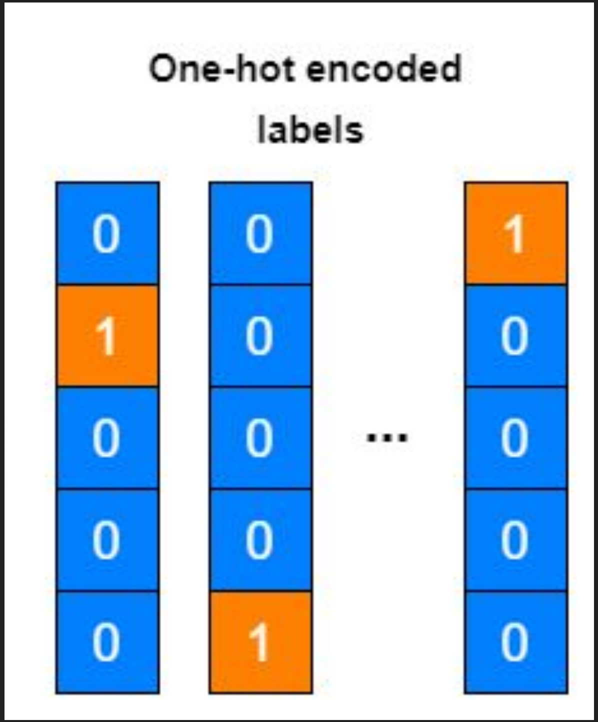


Feature Set

The feature set I decided on was simple: each entry in the one hot encoded value corresponded to a tuple:

(peice_type, color, square)

In the future: multiple perspectives



Network Shape: why?

Quantization - because of loss of accuracy, network depth must be limited

Efficiently updating principle - requires large sparse one-hot encoded input

- **result**: majority of knowledge is in first layer. Diminishing returns for additional large layers

My network:

768 -> 16 -> 16 -> 1

Stockfish network:

81,920 -> 512 -> 32 -> 32 -> 1

Data

lichess.com - online chess website

- Lichess open database of collected stockfish evaluations from games
- Wrote script to extract position / evaluation pairs, encode as vectors, and save to database
- Saved approximately 6,000,000 evaluations. 20% were reserved for testing

```
r1b2rk1/p1q1bppp/2p5/2n1p1N1/Np3P2/1B2Q3/PPP3PP/R4RK1,-69
r6r/3k1p2/p2p2b1/8/8/5NPp/Pb3P1B/3RR1K1,-76
6k1/5pp1/1p2p2p/7b/1R6/1B5P/PP1r1PP1/6K1,-2
2r2rk1/q2b1pp1/2n1p2p/pR1pP3/3P3P/P2BPN2/3Q2P1/R5K1,28
r3k1r1/p1pn1p1p/2p1b3/8/3B4/PPN1P1P1/3P1P2/R2QKB1q,-165
rn3r1k/1bp2p1p/p2p1PpQ/1p5q/3P2N1/1B5P/PPP3P1/R4RK1,436
rn1qkb1r/pp3ppp/4pn2/1bpp4/3P4/2N1PP2/PPP1N1PP/R1BQK2R,-98
r1bq4/ppB4Q/2p1kpN1/3p4/2PPn3/8/PP3PPP/4R1K1,1110
6k1/5pp1/1p2p2p/7b/1R6/1B5P/Pr3PP1/6K1,-2
1r3rk1/q2b1pp1/2n1p2p/p1RpP3/3P3P/P2BPN2/3Q2P1/R5K1,46
r4r1k/2pnRN1p/p1b2Pp1/1p1p4/3P4/1B5P/PPP3P1/5RK1,557
r7/1p2R2p/1k1p1B2/P7/2BN4/2K5/1P3PPP/n7,1298
r2n2kr/1b5p/p2N1PpB/4n3/8/2P3QB/P1P1q2P/3R2K1,206
4r2r/3k1p2/p2p2b1/8/8/5NPp/Pb3P1B/3RR1K1,-55
r3k1r1/2pn1p1p/2p5/p7/1P1B4/P1N1P1Pb/3PQP2/R3KB1q,-152
8/7k/2p3bp/2p1N1p1/4P3/1rNR3P/6PK/r7,-427
r4k1r/1p3P1p/p1n2p2/q3p3/4N3/8/PbP2PPP/R2QKB1R,-302
r3k1r1/2pn1p1p/2p5/8/1P1B4/2N1P1Pb/3PQP2/1R2KB1q,-198
8/6k1/2p3bp/2p1N1p1/4P3/1rNR3P/6PK/r7,-422
r3k3/2pn1p1p/2p3r1/8/1P1B4/2N1P1Pb/3PQP2/1R2KB1q,-234
5rk1/5ppp/8/5R2/8/1B2bq2/P1P4P/7K,32
rnb2rk1/pp2pp1p/3p2p1/q3n3/2P1P3/2PBBN1P/P2Q1PP1/R4RK1,23
r3k1r1/p1p2p1p/1np1b3/8/3B4/PP2P1P1/3P1P2/RN1QKB1q,-154
4b3/6k1/2N4p/2p3p1/4P3/1rNR3P/6PK/r7,-429
rn1qkb1r/3p1ppp/4pn2/2p5/3P4/2N1PN2/PP3PPP/R1BQ1K1R,82
r1b2rk1/1p3ppp/8/pp6/1n5q/B7/P3NPPP/1R1Q1RK1,-165
6qk/p4r1p/2p1Qp1N/2pp4/2P5/1P6/P4PPP/6K1,600
rnb2rk1/pp2pp1p/3p2p1/q3N3/2P1P3/2PBB2P/P2Q1PP1/R4RK1,24
rn1qkb1r/3p1ppp/4pn2/2pP4/8/2N1PN2/PP3PPP/R1BQ1K1R,82
rn1qkb1r/3p1ppp/5n2/2pp4/8/2N1PN2/PP3PPP/R1BQ1K1R,89
2r2rk1/pp1qppbp/2n1b1p1/3p4/3P4/BQP3P1/P2NPPBP/R3R1K1,-25
8/1r5k/2p3bp/2p1N1p1/4P2P/2N3R1/6PK/2r5,-457
r1Nq2kr/1p1Q1ppp/p1n1p3/8/8/B5P1/P1P2PBP/5RK1,600
rn1qkb1r/3p1ppp/5n2/2pN4/8/4PN2/PP3PPP/R1BQ1K1R,78
5rk1/q4pp1/2R1p2p/3pP3/p2P3P/Pr1BPN2/3Q2P1/6K1,39
r5k1/pp1n2pp/4p3/2p5/1n1pN3/5R2/q1B3PP/4B1K1,-869
8/8/5k2/2R3n1/8/6P1/5b1P/7K,-291
r1b2rk1/p1q1bppp/2p1N3/6N1/1p3p2/1B2Q3/PPP3PP/R4RK1,-70
2rr2k1/pp1qppbp/2n1b1p1/1Q1p4/3P4/B1P3P1/P2NPPBP/R3R1K1,-37
6k1/3R4/2p3bp/2p1N1p1/4P2P/2r5/6PK/2r5,-421
5rk1/1q3pp1/4p2p/2RpP3/p2P3P/r2BPN2/3Q2P1/6K1,36
1r1q1rk1/1p1bbppp/3p1n2/PN2p3/P1PnP3/3BB2P/3Q1PP1/RN3RK1,33
2rr2k1/pp1qppbp/2n1b1p1/1Q1p4/3P4/B1P3P1/P2NPPBP/2R1R1K1,-28
5rk1/1q3pp1/4p2p/1R1pP3/p2P3P/r2BPN2/3Q2P1/6K1,10
r3k2r/p1p3pp/3bbp1q/2pn4/8/PP2PNR1/1BQP1P1P/RN2K3,-268
6rk/p6p/1nr1Nb1q/3p1p2/3PpP2/4P3/PP2Q1PP/R3BRK1,144
8/4r2k/2N3bp/2p5/4P2p/2N3R1/6PK/2r5,-406
rnbq1rk1/p3ppbp/2p2np1/4P3/8/1Bp2N1P/PPP2PP1/R1BQ1RK1,-37
1r3rk1/1p1bbppp/3p1n2/q3p3/P1PnP3/2NBB2P/1Q1N1PP1/R4RK1,89
8/4r2k/2N3Rp/2p5/4P2p/2N5/6PK/2r5,-398
rnbqk1nr/pppp2pp/5p2/4p3/2PPP3/8/PP1Q1PPP/RN2KBNR,53
```

Training

pytorch on my laptop GPU

- Mean square error loss function
- Stochastic gradient descent with a batch size of 256
- During training, evals were passed through a sigmoid activation to give manageable gradients (for easier hyperparameter tuning)
- 0.001 proved to be a reasonable learning rate
- Loss converged rapidly to minimum (likely due to small model size)
- Parameters were clamped during training to suite quantization later

```python
for epoch in range(NUM_EPOCHS):
    # Training phase
    running_loss = 0.0
    for batch_num, (inputs, targets) in enumerate(train_dataloader):

        optimizer.zero_grad()
        outputs = model(inputs.to(device))
        loss = criterion(outputs.squeeze(), targets.to(device))  # Squeeze to
        loss.backward()
        optimizer.step()

        # Clamp weights of linear1 and linear2 (for quantization later)
        if batch_num % 16 == 0:
            for name, param in model.named_parameters():
                if name == 'linear1.weight':
                    param.data.clamp_(min=-1.9843, max=1.9843)
                if name == 'linear2.weight':
                    param.data.clamp_(min=-127, max=127)

        running_loss += loss.item() * inputs.size(0)

        print(f"TRAINING: Epoch [{epoch + 1}/{NUM_EPOCHS}], Batch [{batch_num

    epoch_train_loss = running_loss / len(train_dataset)
    train_losses.append(epoch_train_loss)
    print(f"{'':50}", end='\r')

    # Validation phase
    model.eval()
    running_loss = 0.0

    with torch.no_grad():
        for batch_num, (inputs, targets) in enumerate(test_dataloader):

            outputs = model(inputs.to(device))
            loss = criterion(outputs.squeeze(), targets.to(device))  # Squeez

            running_loss += loss.item() * inputs.size(0)

            print(f"TESTING: Epoch [{epoch + 1}/{NUM_EPOCHS}], Batch [{batch_


    epoch_test_loss = running_loss / len(test_dataset)
    test_losses.append(epoch_test_loss)
```

Applying Sigmoid to Evaluations

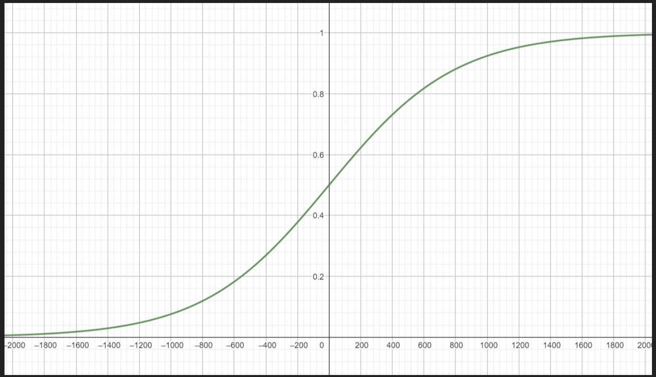
During training, it proved that using the direct evaluation to make the loss calculations caused an inferior model. Applying a sigmoid to the evaluation helped.

The sigmoid "squeezes" the extremely large evaluations together.

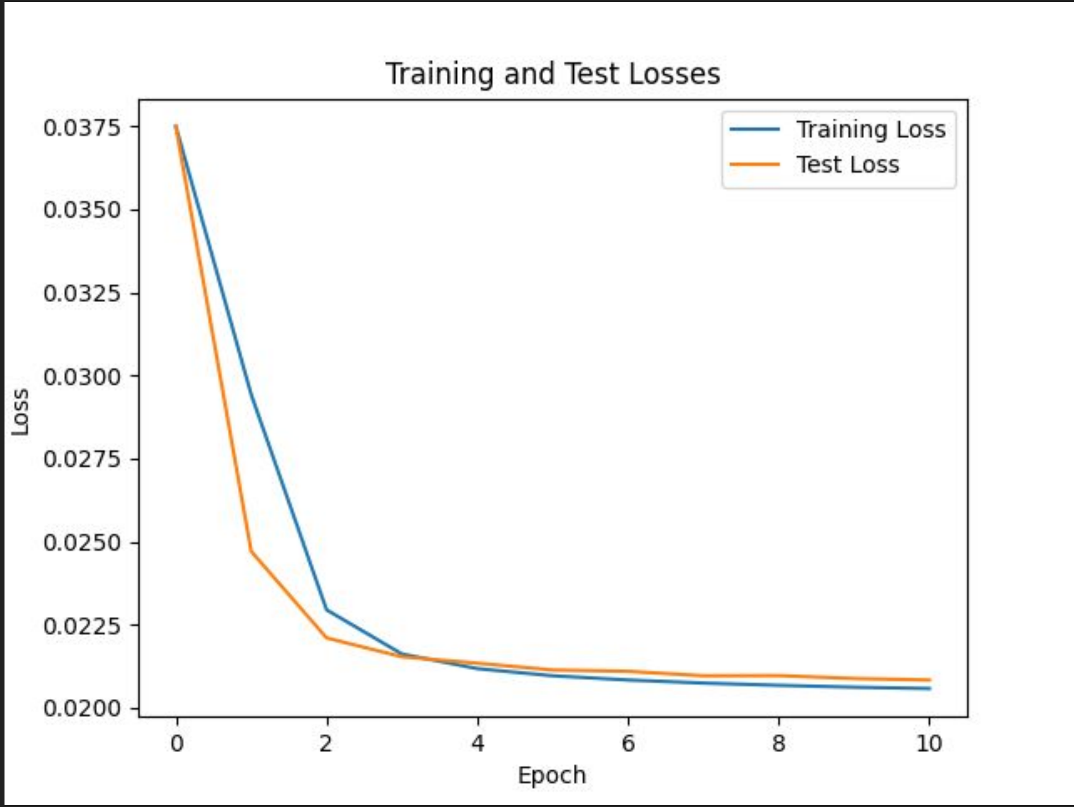
The gradient of the sigmoid is also much less in magnitude. This makes hyperparameter tuning easier

Additionally, this transition allows interpolation with game results, since the value after sigmoid can be interpreted as a probability of winning.

$$\frac{1}{1+e^{-\frac{x}{400}}}$$



Training



Training and Test Losses
Training Loss
Test Loss
0.0375
0.0350
0.0325
0.0300
0.0275
0.0250
0.0225
0.0200
Loss
0
2
4
6
8
10
Epoch

Results

After exporting the model parameters and implementing the network evaluation code, a match was run against my most recent version with the hand-crafted evaluation function

Was it a failure?

Conclusion

All things considered, I don’t believe that it is a failure

- Feature set size seems to be the largest bottleneck
- Increasing the feature set size would require **much more training data**
- More data -> bigger model -> better performance
- **Quality of data** was also lacking
- Once I discover more efficient ways to manipulate/process data, I will use everything I learned to create a larger network.

Epilogue

Obviously things worked out for Stockfish?

- Stockfish received the biggest bump in performance it had seen in a long time after with the release of Stockfish 12, the first version to come with the NNUE evaluation function

Other Neural Networks in chess engines

- A few chess engines such as Leela Chess 0 and AlphaZero utilize neural networks to a greater extent that NNUE. Both of these engines use deep convolutional neural networks, some reaching up to 40 layers deep. These engines evaluate orders of magnitude less positions, relying on the deep knowledge of the network.

Sources

Stockfish NNUE docs: https://disservin.github.io/stockfish-docs/nnue-pytorch-wiki/docs/nnue.html#converting-the-evaluation-from-cp-space-to-wdl-space

Lichess open database: https://database.lichess.org/#evals

Source code (training/data collection): https://github.com/patrickmastorga/chess-data

Source code (engine) (see src/NNUE for implementation): https://github.com/patrickmastorga/chess-old