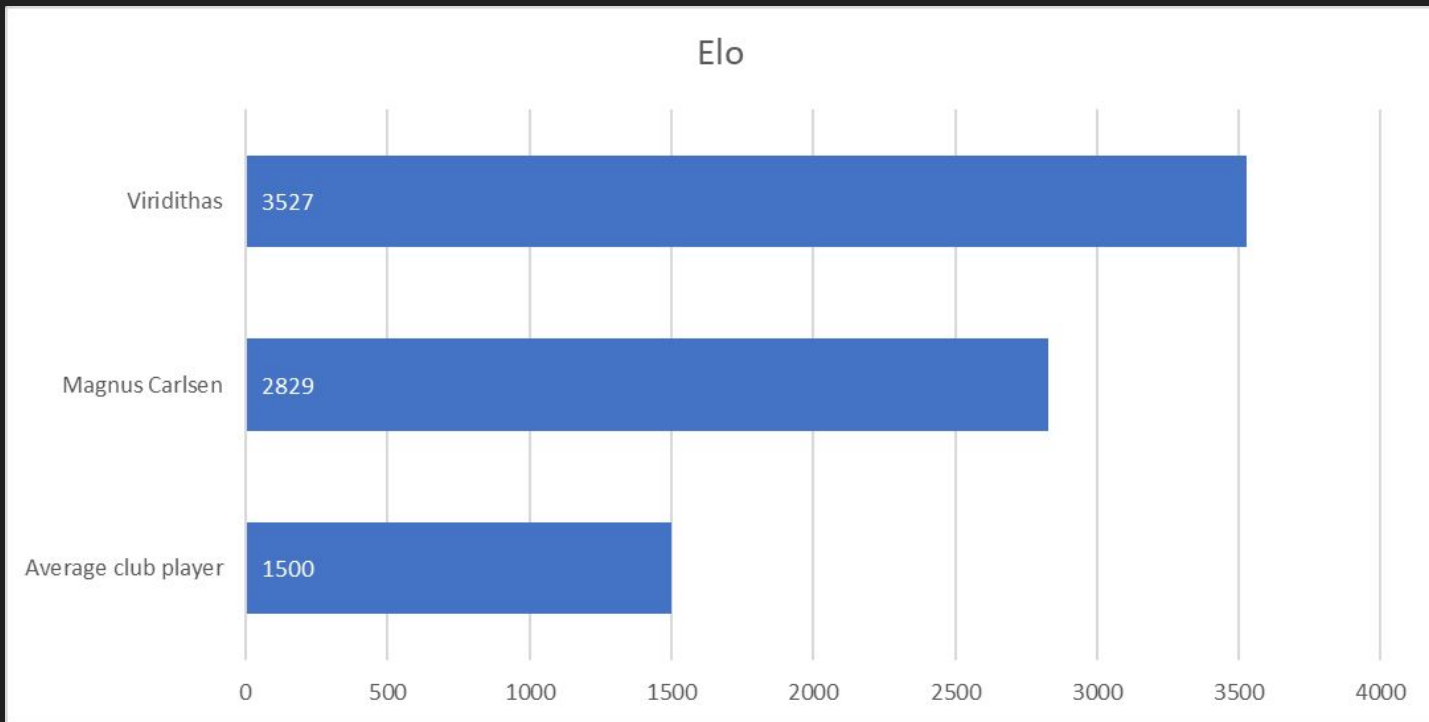


Methods in State-of-the-Art Computer Chess

The Elo rating system



Things I'm going to tell you

1. How to make a neural network evaluate chess positions really quickly
2. How to search the game-tree of Chess cleverly
3. How to make a chess program learn by playing against itself

What's a chess engine?

A *chess engine* is a computer program for calculating strong moves in the game of Chess.

Pictured: The *Bishop* automatic chessboard produced last year by my SDP team.

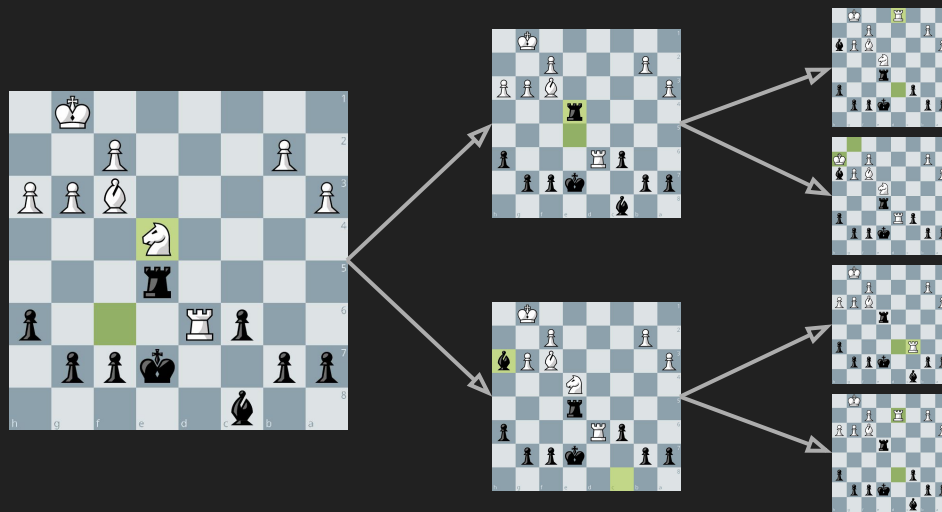


How does a chess engine work?

Two components: Evaluation and Search.

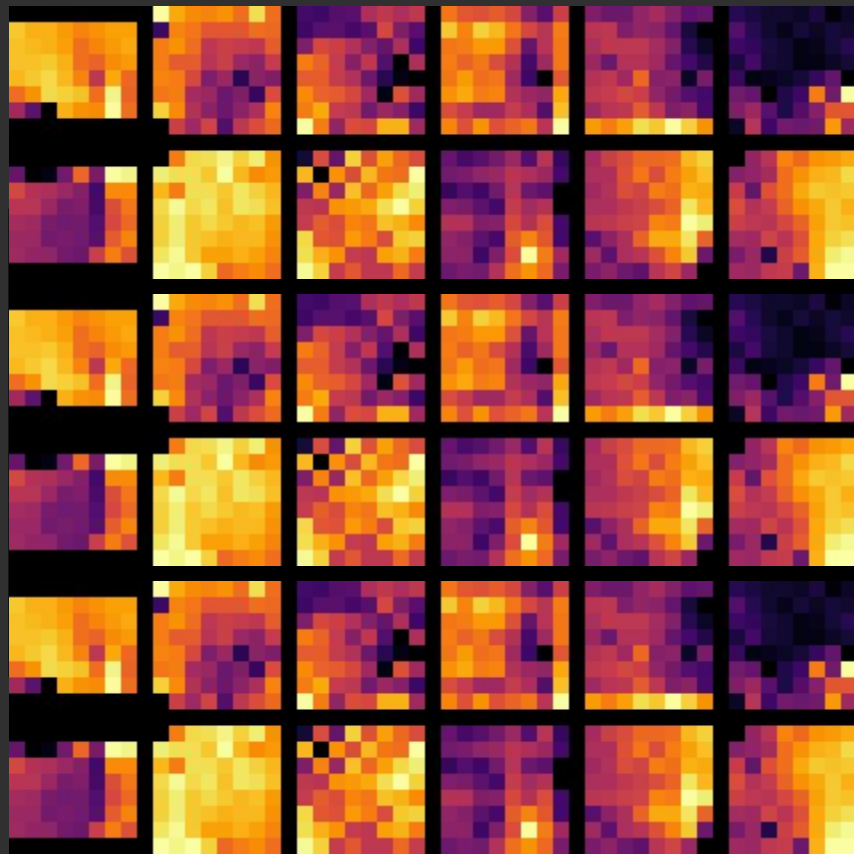
Evaluation: “How good do we think this position is?”

Search: “Look into the future and make better choices.”



Act 1

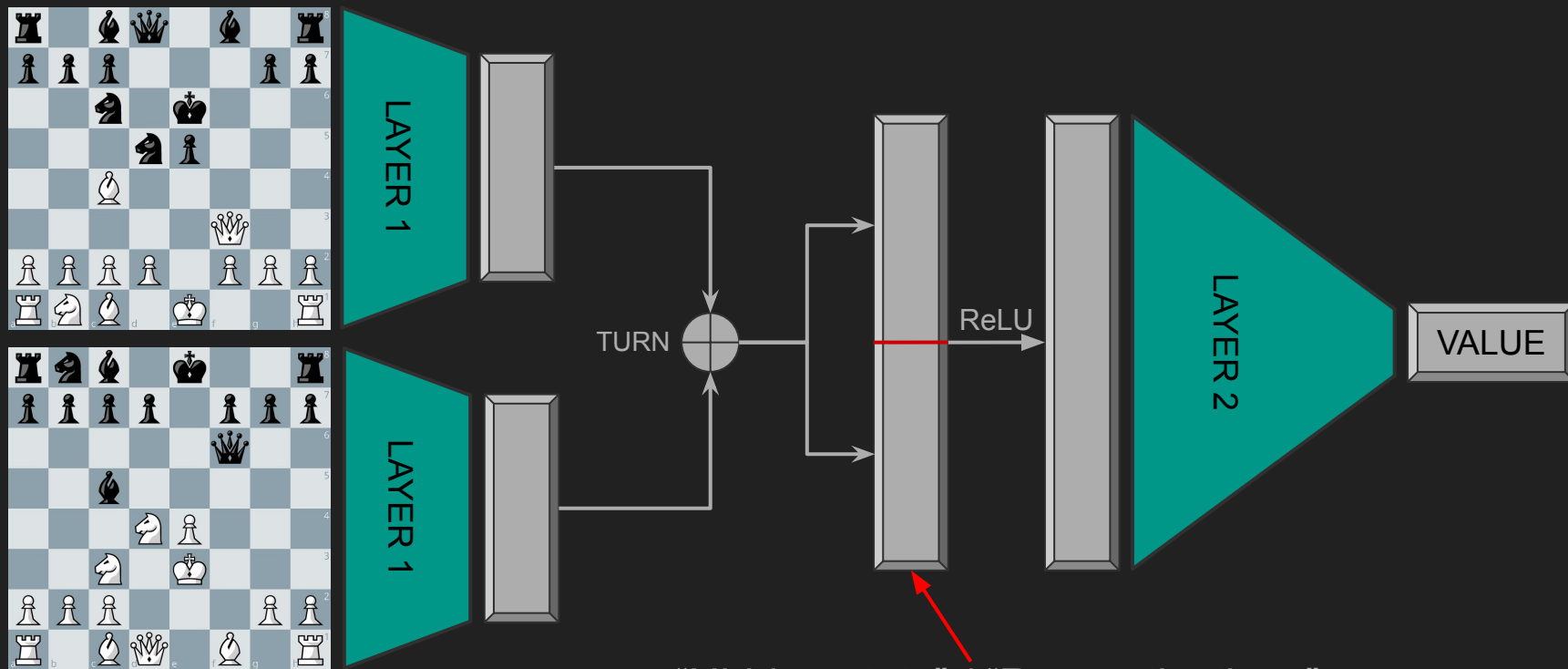
Efficient Neural Network
Inference



Evaluating chess positions using neural networks

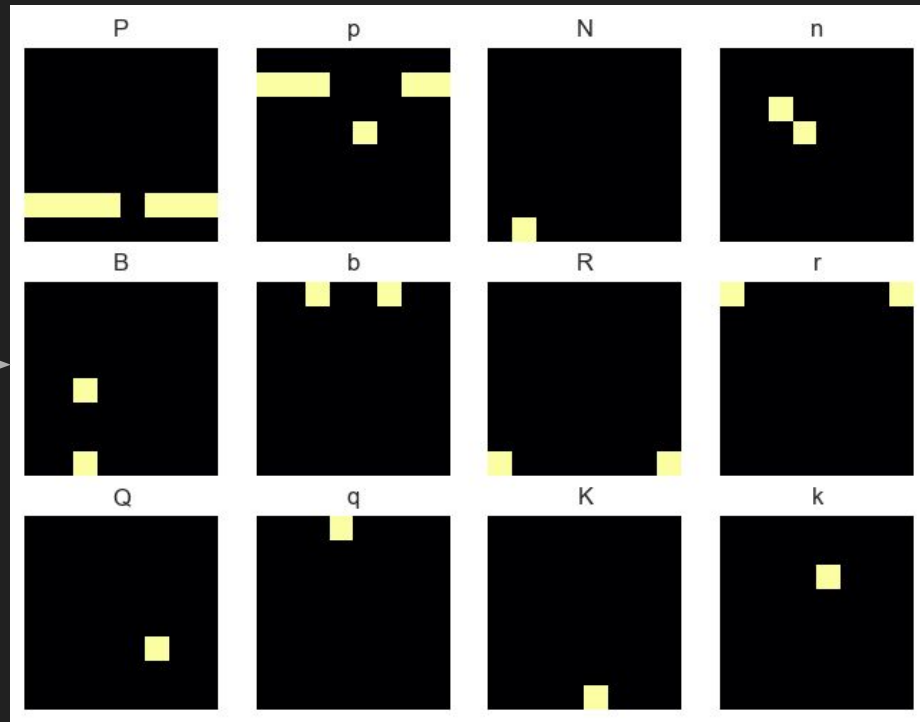


Evaluating chess positions using neural networks

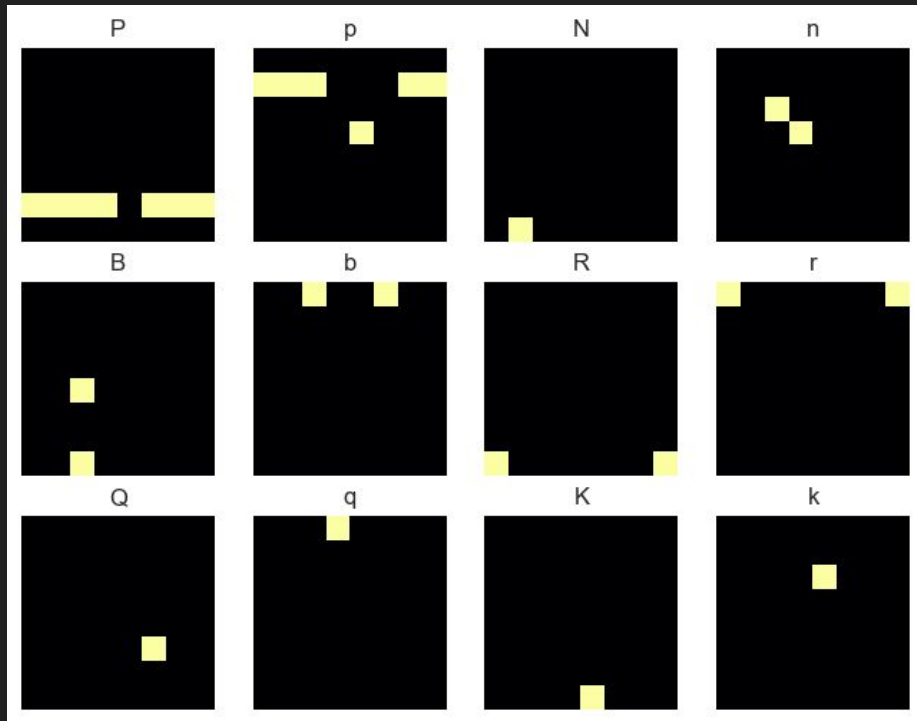


"Hidden state" / "Raw activations"

Evaluating chess positions using neural networks



Evaluating chess positions using neural networks



```
array([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,
       1, 1, 1, 1, 0, 1, 1, 1,
       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, 1, 0, 0, 0, 0, 0, 0,
       ])
```

(length = 12 x 8 x 8 = 768)

Calculating the hidden vector

$$V_i = w_i \cdot x + b_i$$

To calculate the i th element of the hidden vector, we take the dot product of the i th row of weights with the input vector, and add the i th bias term.

Calculating the hidden vector

$$V_i = w_i \cdot x + b_i$$

Important fact: As x is entirely binary (0 or 1), if a value in x changes, computing the new V requires only that one adds or subtracts the relevant values in w to or from V !

Efficiently updating the hidden state

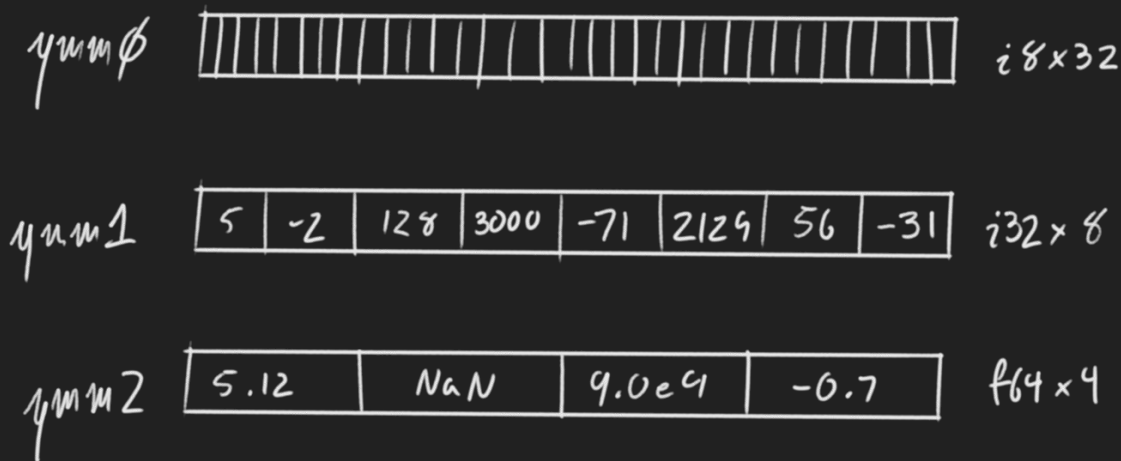
$$\begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.38 \\ 0.11 \\ 0.78 \\ 0.29 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 1 \\ \textcolor{green}{1} \\ \vdots \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.38 \\ 0.11 \\ 0.78 \\ 0.29 \end{bmatrix} + \begin{bmatrix} \textcolor{red}{-0.12} \\ \textcolor{green}{+0.04} \\ \textcolor{red}{-0.09} \\ \textcolor{red}{-0.02} \end{bmatrix}$$

Efficiently updating the hidden state

```
fn vector_add(  
    hidden_vector: &mut Align64<[i16; LAYER_1_SIZE]>,  
    weights: &Align64<[i16; INPUT * LAYER_1_SIZE]>,  
    start_index: usize,  
) {  
    let add_block = &weights[start_index..start_index + LAYER_1_SIZE];  
    for (i, d) in hidden_vector.iter_mut().zip(add_block) {  
        *i += *d;  
    }  
}
```

Efficient inference on CPUs via *Quantisation*

- Floating point arithmetic is slower than integer arithmetic
- Floating point values are larger in memory than integers (f32/f64 vs. i16)
 - If values are smaller, more of them fit in cache at the same time
 - If values are smaller, more of them fit in a SIMD lane at the same time (potential 2x speedup!)



(for 256-bit SIMD registers)

Efficient inference on CPUs via *Quantisation*

Example: $2.37 * 0.13 = 0.3081$

How would we approximate this with integers?

Answer: Fixed-Point Multiplication!

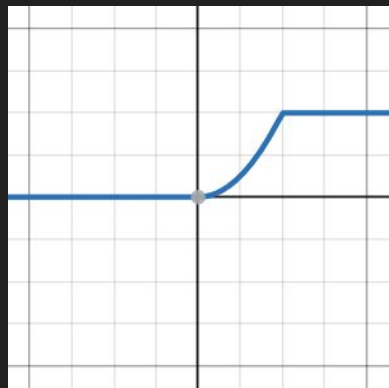
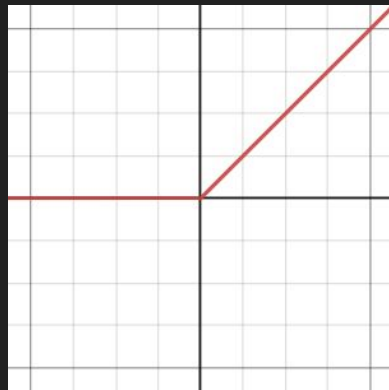
Take $2.37 = 237$ and $0.13 = 13$, then perform multiplication like so:

$$(237 * 13) / 100 = 30$$

In our fixed-point scheme, $30 = 0.30$, so this approximates floating-point reasonably well!

Writing SIMD-friendly inference code

```
fn flatten<F: ActivationFunction>(  
    us: &Align64<[i16; LAYER_1_SIZE]>,  
    them: &Align64<[i16; LAYER_1_SIZE]>,  
    weights: &Align64<[i16; LAYER_1_SIZE * 2]>,  
) -> i32 {  
    let mut sum: i32 = 0;  
    for (&i, &w) in us.iter().zip(&weights[..LAYER_1_SIZE]) {  
        sum += F::activate(i) * i32::from(w);  
    }  
    for (&i, &w) in them.iter().zip(&weights[LAYER_1_SIZE..]) {  
        sum += F::activate(i) * i32::from(w);  
    }  
    sum / F::ACCUMULATED_RENORMALISATION  
}
```



Rust source #1 ⓘ X

A ▾ Save/Load + Add new... ▾ Vim

Rust

```
1 #[derive(Debug, Copy, Clone, PartialEq, Eq)]
2 #[repr(C, align(64))]
3 pub struct Align64<T>(pub T);
4
5 pub const LAYER_1_SIZE: usize = 1536;
6
7 fn activate(x: i16) -> i32 {
8     let x = x.clamp(0, 255);
9     let x = i32::from(x);
10    x * x
11 }
12
13 /// Execute an activation on the partial activations,
14 /// and accumulate the result into a sum.
15 pub fn flatten(
16     us: &Align64<i16; LAYER_1_SIZE>,
17     them: &Align64<i16; LAYER_1_SIZE>,
18     weights: &Align64<i16; LAYER_1_SIZE * 2>,
19 ) -> i32 {
20     let mut sum: i32 = 0;
21     for (&i, &w) in us.iter().zip(&weights.0[..LAYER_1_SIZE]) {
22         sum += activate(i) * i32::from(w);
23     }
24     for (&i, &w) in them.iter().zip(&weights.0[LAYER_1_SIZE..]) {
25         sum += activate(i) * i32::from(w);
26     }
27     sum / 255
28 }
```

rustc 1.73.0 (Editor #1) ⓘ X

rustc 1.73.0

-C opt-level=2 -C target-cpu=x86-64-v3

A ▾ Output... ▾ Filter... ▾ Libraries ⓘ Overrides + Add new... ▾ Add tool... ▾

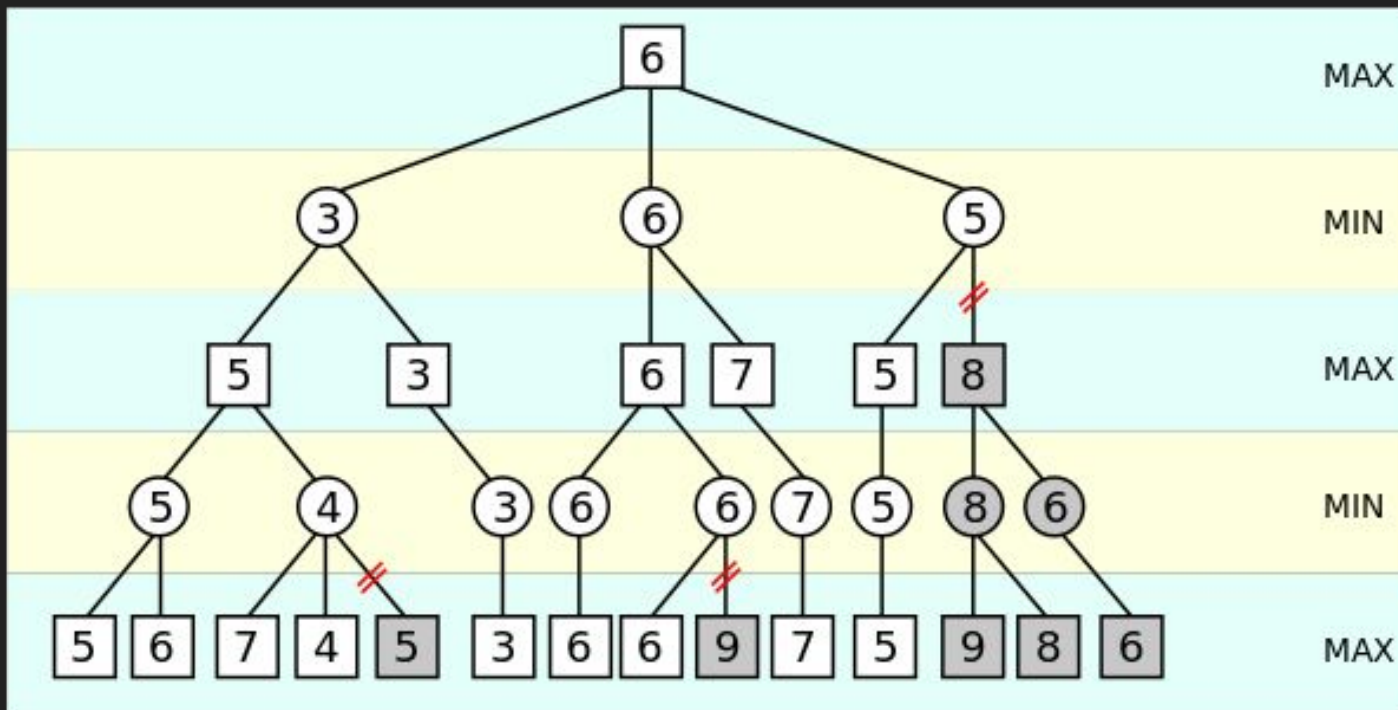
```
1 .LCPI0_1:
2     .short 255
3
4 example::flatten:
5     vpxor    xmm1, xmm1, xmm1
6     mov     eax, 24
7     vpbroadcastw    xmm0, word ptr [rip + .LCPI0_1]
8     vpxor    xmm2, xmm2, xmm2
9     vpxor    xmm3, xmm3, xmm3
10    vpxor    xmm4, xmm4, xmm4
11
12 .LBB0_1:
13    vmovdqa  xmm5, xmmword ptr [rdi + 2*rax - 48]
14    vmovdqa  xmm6, xmmword ptr [rdi + 2*rax - 32]
15    vmovdqa  xmm7, xmmword ptr [rdi + 2*rax - 16]
16    vmovdqa  xmm8, xmmword ptr [rdi + 2*rax]
17    vpmiunw  xmm9, xmm5, xmm0
18    vpsraw   xmm5, xmm5, 15
19    vpandn   xmm5, xmm5, xmm9
20    vpmiunw  xmm9, xmm6, xmm0
21    vpsraw   xmm6, xmm6, 15
22    vpandn   xmm6, xmm6, xmm9
23    vpmiunw  xmm9, xmm7, xmm0
24    vpsraw   xmm7, xmm7, 15
25    vpandn   xmm7, xmm7, xmm9
26    vpmiunw  xmm9, xmm8, xmm0
27    vpsraw   xmm8, xmm8, 15
28    vpandn   xmm8, xmm8, xmm9
29    vpmovzxd  ymm5, xmmword ptr [rdx + 2*rax - 48]
30    vpmovzxd  ymm5, xmm5
31    vpmaddwd  ymm5, ymm5, ymm5
32    vpmulld  ymm5, ymm5, ymm5
33    vpaddq   ymm1, ymm5, ymm1
34    vpmovzxd  ymm5, xmmword ptr [rdx + 2*rax - 32]
35    vpmovzxd  ymm6, xmm6
36    vpmaddwd  ymm5, ymm6, ymm5
37    vpmulld  ymm5, ymm5, ymm6
38    vpaddq   ymm2, ymm5, ymm2
39    vpmovzxd  ymm5, xmmword ptr [rdx + 2*rax - 16]
40    vpmovzxd  ymm6, xmm7
41    vpmaddwd  ymm5, ymm6, ymm5
42    vpmulld  ymm5, ymm5, ymm6
43    vpmovzxd  ymm6, xmmword ptr [rdx + 2*rax]
44    vpmovzxd  ymm7, xmm8
```

Act 2

Heuristics for Game-Tree
Search

8	r	n	b	q	k	.	.	r
7	p	p	p	.	.	p	p	p
6	.	.	.	b	.	.	.	.
5	.	.	.	p	.	.	.	.
4	.	.	.	P	.	.	.	.
3	.	.	n	B	P	N	.	.
2	P	P	.	.	.	P	P	P
1	R	.	B	Q	K	.	.	R
	a	b	c	d	e	f	g	h

Alpha-Beta Pruning



Alpha-Beta Pruning

```
fn alpha_beta(pos: Position, depth: i32, alpha: i32, beta: i32) -> i32 {  
    if depth <= 0 {  
        return pos.evaluate();  
    }  
    let mut best_score = -INFINITY;  
    for mv in pos.legal_moves() {  
        let score = -alpha_beta(pos.make_move(mv), depth - 1, -beta, -alpha);  
        best_score = max(best_score, score);  
        alpha = max(alpha, score);  
        if alpha >= beta {  
            break; // cutoff! save time by not searching the rest of the moves.  
        }  
    }  
    best_score  
}
```

Static Evaluation Pruning

```
if depth <= DEPTH_LIMIT && static_evaluation - MARGIN > beta {  
    return static_evaluation;  
}
```

Null-Move Pruning

```
if !last_move_was_nullmove() // don't let both sides null-move at each other
    && depth >= DEPTH_LIMIT
    && zugzwang_unlikely() // sufficient material is present that zugzwang is unlikely
{
    let nullmove_depth = depth - 3;
    make_nullmove();
    let null_score = -alpha_beta(pos, nullmove_depth, -beta, -beta + 1);
    unmake_nullmove();
    if null_score >= beta {
        return null_score;
    }
}
```

Act 3

Self-Play Reinforcement
Learning



Reinforcement Learning for Chess

1. Play millions of games against yourself
2. Learn from these games

Considerations:

- How do you generate sufficient variety in the games you play?
- How do you ensure that such a simple network learns something useful?

Generating variety in training games

Two primary approaches:

1. Opening randomisation: play N random moves, then perform greedy rollouts until the game ends.
2. *Double Fischer Random Openings*: Use the 921,600 different openings available in the DFRC variant (AKA double-Chess960)

Ensuring the network can learn (training setup)

Networks are trained using the AdamW optimiser, a variant of *Stochastic Gradient Descent*.

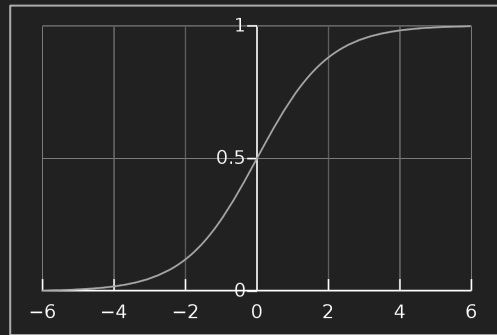
Important factors:

- Gradient updates occur on subsets of the dataset, so if all the positions are simply in-order, a batch might consist of exclusively positions from one game.
- If game outcome is the prediction target, there are only $\lg(3) \approx 1.58$ bits of signal for each game.
- If search result is the prediction target, we could hit a local minimum, only learning how to beat ourselves.

Ensuring the network can learn (training setup)

To get a training target that blends ground truth (game outcome) with a high-information signal (search result), we pass a scaled evaluation through a sigmoid and linearly interpolate with game result:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



$$TARGET(Q, V) = \sigma\left(\frac{V}{400}\right) (1 - \lambda) + Q\lambda$$

Ensuring the network can learn (training setup)

To get a training target that blends ground truth (game outcome) with a high-information signal (search result), we pass a scaled evaluation through a sigmoid and linearly interpolate with game result:

```
prediction = model(batch)
expected = torch.sigmoid(batch.cp / scale) * (1 - wdl) + batch.wdl * wdl
mse_loss = torch.mean((prediction - expected) ** 2)
```

Ensuring the network can learn (position filtering)

During self-play, only a subset of the encountered positions are saved for later training.

We exclude:

- Positions where a checkmate-score was found
- Positions where the best move is a capturing move
- Positions where the side-to-move is in check
- Positions too early in the game

```
if board.ply() > MIN_SAVE_PLY
    && !board.is_tactical(best_move)
    && !is_game_theoretic_score(score)
    && !board.in_check()
{
    single_game_buffer.push(board.pack(
        eval: score.try_into().unwrap(),
        PackedBoard::WDL_DRAW,
        extra: 0,
    ));
}
```

Putting it all together



Viridithas is my chess engine, which I have been working on for ~1.5 years now.

- >13k lines of Rust, with minimal dependencies.
- Around #15 on multiple rating lists at time of writing
- Strongest program from an author in the UK, and strongest program written in Rust

Viridithas uses all the techniques explained in this presentation, and many more! (time management heuristics, move ordering heuristics, staged move generation, racy atomic symmetric multiprocessing...)

Cool graphs

