ChatGPT

Chess Position Evaluation Network – Model Architecture and Hyperparameter Recommendations

Overview of the Task

Developing a neural network to evaluate chess positions is a challenging task that has seen various successful approaches in recent years. Your input representation (bitboard-based 8×8 planes with 23 feature channels) is similar to those used by top chess AIs like AlphaZero and Leela Chess Zero. For example, AlphaZero's network took an 8×8×*C* input (with C on the order of 100+ planes encoding pieces, history, castling rights, etc.) [1]. The goal is to output an evaluation between -1 and 1 (presumably via a $\tanh$ activation) indicating the position's favorability for White or Black. Given ~2.2 million positions (from human games) as training data and limited hardware (GeForce 4060 or Colab's T4/P100 GPUs), we need to choose an architecture and training strategy that balances *strength* (accuracy/generalization of evaluation) with *efficiency* (feasible training and fast inference for search). Below, we discuss the best model structure options – primarily **Residual CNNs vs. Transformer-based models** – and then cover recommended hyperparameters (optimizers, learning rates, etc.) for training such a network from scratch.

Model Architecture Choices

1. Convolutional Residual Networks (ResNet-Based CNNs)

Convolutional neural networks have been the cornerstone of chess evaluation networks since AlphaZero. A plain CNN (several conv layers then fully connected layers) was your initial 3M-parameter model; to improve on this, **ResNets** are a proven upgrade. Residual networks stack many conv layers with *skip connections* that help train deeper models without vanishing gradients. AlphaZero's chess network, for instance, used **19 residual blocks** (each block has two 3×3 conv layers with 256 filters, plus a skip connection) after an initial conv layer [2]. Despite the small 8×8 spatial size, stacking conv layers increases the effective receptive field so that the network can learn long-range interactions (after ~4 convolutional layers, each square's output can theoretically depend on every input square [3]). This ability to capture local patterns (tactics) as well as broader board configurations is vital for chess.

- **Recommended CNN Architecture:** Start with a **residual CNN** having on the order of 5–15 residual blocks. Each block can have, say, 64–128 filters (to limit parameters for your hardware), or more if memory allows. Even a smaller ResNet (e.g. 6 blocks × 64 filters) can outperform a plain CNN by allowing greater depth. The network would input your 23-channel 8×8 planes and process through the conv blocks with ReLU (and typically Batch Norm) activations. A possible design:
- Initial 3×3 conv layer with F filters (e.g. $F=64$ or 128)
- N residual blocks (each: 3×3 conv F filters -> BN/ReLU -> 3×3 conv F filters, then add skip)

- **Value Head:** After the conv trunk, use a 1×1 conv to reduce to a single-channel 8×8 plane, then a fully-connected layer to 256 units (with ReLU), and finally a linear layer to 1 output, with a $\tanh$ activation [4]. This head structure (conv + fc) is exactly what AlphaZero used for the value output.
- **Advantages:** CNNs excel at recognizing spatial patterns (e.g. piece contacts, pawn structures) and are relatively efficient on 8×8 inputs. A ResNet can capture complex features (pins, forks, long-range tactics) by hierarchical composition of filters. It has been *battle-tested* in AlphaZero and Leela Chess Zero – these conv nets reached superhuman play [5]. Implementing a ResNet in PyTorch is straightforward, and training should be stable with proper normalization (BatchNorm) and initialization.
- **Resource Considerations:** A model with ~5–10M parameters (which might correspond to, say, 10 residual blocks of 128 filters) should be trainable on a T4/P100 within reasonable time, especially with mixed-precision training. You may adjust filter counts to fit GPU memory (e.g. try 64 filters first, then increase). Keep an eye on inference speed: if you plan to use this network in a minimax (alpha-beta) search, evaluation speed is critical. A smaller ResNet will evaluate faster during search. In practice, top MCTS-based engines use large conv nets (but evaluate fewer nodes), whereas classical engines using neural eval prefer more lightweight networks due to millions of nodes – we discuss an alternative (NNUE) below for that scenario.

2. Transformer-Based Architectures

Transformers have recently shown remarkable results in chess position evaluation, overcoming some CNN limitations. Convolution's fixed kernel size can be a bottleneck for chess because of its many long-range interactions (e.g. pieces can influence across the board). **Transformers**, with global self-attention, can inherently capture arbitrary long-range dependencies in a single layer [6]. A recent work introduced the **"Chessformer"**, a transformer-based chess engine, and demonstrated that a well-designed transformer can **outperform AlphaZero's residual network** in playing strength using significantly less computation [7].

- **Transformer Design:** A transformer for chess typically treats the board as a **sequence of 64 tokens** (one per square). Each token can encode the piece/square information and additional features. For example, Chessformer uses a 64-token sequence where each token is a vector containing piece type info for that square (including history of past positions), side-to-move, castling, etc., concatenated into a rich 112-dimensional input per token [8] [9]. Positional encoding is crucial since the transformer is permutation-invariant by itself – Chessformer found that using a **relative positional encoding** scheme (based on Shaw et al.) inside attention gave the best results [10]. In your case, since you already represent the board as planes, you could flatten that 8×8×23 into 64 tokens of length 23 (or extend to include history if desired) and then feed through transformer layers. Each layer would have multi-head self-attention and feed-forward sublayers (with normalization). Even a **small transformer**, e.g. with embedding size ~128, 4–8 attention heads, and say 4–6 encoder layers, can amount to a few million parameters (comparable to your CNN) and might capture global patterns effectively.
- **Advantages:** Global attention means a transformer can learn relations like *"this rook on A1 is x-raying a piece on A8 if no piece in between"* more directly than a CNN which must propagate information through multiple layers. Transformers also offer flexibility: you could incorporate *non-spatial* features (like game phase, material count, etc.) into the token embeddings easily. The **Chessformer results** show that given the right position encoding, transformers achieved grandmaster-level play, detecting high-level positional concepts that even traditional engines struggled with (like fortress or

trapped piece patterns) [7] [11]. This suggests that a transformer-based evaluator can be very powerful.

- **Challenges:** Transformers generally require careful tuning and can need more data to generalize. With 2M positions (which is decent but not extremely large), you'd want to prevent overfitting via regularization (dropout, etc.) and maybe use data augmentation. Another consideration is that transformer inference might be slower per position than a similarly sized CNN due to the self-attention scaling ($O(n^2)$ over 64 tokens is not too bad, though). On limited hardware, a small transformer is feasible, but large ones (e.g. the Chessformer's largest was 240M parameters) are out of scope. The **good news** is that a transformer with only ~6M parameters (Chessformer-6M) was able to match the strength of much larger models [7], so a well-designed small transformer is an option. If you choose this path, use absolute or relative positional embeddings for squares, and consider leveraging existing code (the Chessformer authors open-sourced their code [12]). Start with a modest depth and dimension, and scale up if training goes well.

3. Hybrid Approaches (Convolution + Attention)

It's not strictly necessary to choose *either* CNN or transformer – some research indicates that combining them yields the best of both. For example, an architecture called **AlphaVile** experimented with a convolutional base followed by transformer blocks, aiming to reduce inference latency while improving accuracy [13]. Pure Vision Transformer (ViT) style models were found to have **higher loss and lower accuracy** than the standard AlphaZero CNN when trained with limited resources [14]. However, by inserting a few transformer layers on top of convolutional feature extractors (or mixing them), one can improve on the pure CNN's performance without a huge slowdown [15]. In practice, you could use a few convolutional layers (or residual blocks) to extract local features, then feed their output into a small transformer encoder that focuses on global relationships. This hybrid could improve evaluation quality, as suggested by the above research, but keep in mind it adds complexity. If you are comfortable experimenting, this is a promising direction – otherwise, a well-tuned pure ResNet or small transformer are more straightforward starting points.

4. NNUE-Style Networks (Efficient Fully-Connected Architecture)

Since you plan to integrate the network into a *minimax search*, it's worth mentioning **NNUE (Efficiently Updatable Neural Network)** architecture – the kind used in Stockfish for fast evaluation on CPU [16] [17]. NNUE networks are quite different from CNNs/transformers: they are essentially fully-connected networks optimized for incremental update. The input is not an image but a handcrafted encoding of piece-square relationships (often using "half KP" features – piece on square relative to king positions). The classic NNUE has an extremely large first layer (tens of thousands of inputs feeding into a few hundred neurons) and a few smaller FC layers after [18] [19]. The key is that when a move is made, only a small part of the input changes, so the network's output can be updated efficiently without recomputing everything [19]. In Stockfish's NNUE, for example, the input layer has ~10 million weights, but by summing precomputed contributions for active features, the engine updates evaluations rapidly. The remaining layers (e.g. 256→32, 32→32, 32→1) are lightweight [19].

- **Why consider NNUE?** If your goal is a traditional engine that evaluates millions of nodes per second in an alpha-beta search, a CNN or transformer (which must run from scratch on each position, likely on a GPU) could be a bottleneck. NNUE, on the other hand, runs efficiently on CPU with SIMD instructions and incremental updates, making it feasible to evaluate many positions quickly [20]. Indeed, Stockfish gained ~80-100 Elo by adopting NNUE over handcrafted eval, despite the

slowdown per node [21]. Training an NNUE is a bit different (often a mix of supervised on engine evaluations and reinforcement learning). Given your data (human game positions with outcomes or evaluations), you *could* train a simple 4-layer fully connected network in PyTorch to mimic an NNUE. You'd need to craft input features (e.g. piece-square indicator for each piece type and square, maybe split by side to move's king location). The PyTorch training is doable, though implementing the *efficient incremental logic* outside of training is an additional effort in C++ for integration.
- **Bottom line:** If your priority is **search speed** in a minimax engine, consider NNUE-style (or at least be aware of it for future improvements). If your priority is maximizing evaluation accuracy with a given model size (and maybe using the network in an MCTS-style search with fewer but slower evaluations), then CNNs or transformers are the way to go. Many modern engines use a hybrid approach: e.g. use the NNUE network during deep search and possibly a heavier network at root or for policy guidance.

Hyperparameter Recommendations for Training

Designing the network is half the battle – training it effectively from scratch is the other half. Here we outline recommendations for loss functions, optimizers, learning rate schedules, and other key hyperparameters, tailored to your task:

Loss Function and Output Target

- **Regression vs Classification:** Since your evaluation is a continuous score in [-1, 1], a common approach is to treat this as a regression problem using Mean Squared Error (MSE) loss. MSE is simple and works, but can be sensitive to outliers (positions where the evaluation swings sharply). An alternative is **Huber loss** (smooth L1), which behaves like MSE for small errors but is more robust to outliers – this can stabilize training if a few positions have very wrong scores.
- **Win/Draw/Loss (WDL) Classification:** Another highly effective strategy (used in Leela Chess Zero training) is to reformulate the value prediction as a three-class classification: predict probabilities of White win, draw, or Black win. The network's outputs (after softmax) can be trained with cross-entropy against the actual game result (win/draw/loss). The predicted probability $P(\text{win})$, $P(\text{draw})$, $P(\text{loss})$ can then be converted to a scalar eval (for example, one can take $v = P_w - P_l$, ignoring draws, or $v = P_w + 0.5 P_d - P_l - 0.5 P_d = P_w - P_l$ effectively). This WDL approach often yields better calibrated evaluations. In fact, research showed that a **WDL head outperformed a single scalar MSE head**, giving significant Elo gains (e.g. +33 Elo in one experiment) [22]. If your dataset labels are game results (which are discrete), a WDL classification might be more natural and resilient to the inherent noise (human games can have winning positions that were misplayed). If your labels are engine scores (continuous), WDL is less applicable.
- **Tanh Output:** Using $\tanh$ as the final activation to bound output to [-1,1] is reasonable (AlphaZero did the same for its value head [4]). Just ensure your loss is computed on the output of the $\tanh$ (i.e., comparing network's bounded prediction to the target in [-1,1]). One caution: if many training examples are extreme values (exactly 1 or -1), $\tanh$ can saturate and slow learning; in such cases, sometimes training without $\tanh$ and simply clipping outputs to [-1,1] for use can be tried. But typically, having the network inherently output a bounded value is fine.

Optimizer Choice

- **Adam / AdamW:** For a project like this, **Adam** optimizer is an excellent starting point. Adam is adaptive and tends to converge faster with less tuning than plain SGD. It's widely used in deep learning projects as a “plug-and-play” optimizer [23]. It will adjust the learning rate per parameter, which is helpful given the variety of layers (conv, fc, etc.). Using **AdamW** (Adam with decoupled weight decay) is recommended if you plan to use weight decay (regularization) – it ensures the weight decay is applied correctly. A typical learning rate for Adam in this context might be **1e-3** to start. Given a relatively large dataset, you might even start a bit higher (e.g. 2e-3) but monitor loss closely. If using a smaller batch size, a slightly lower LR (5e-4) could help stability.

- **SGD with Momentum:** The AlphaZero team and many RL-based trainings use **SGD + momentum (0.9)** with large batches [24]. The reason is that SGD can yield better final generalization if tuned well, but it may require more epochs and careful LR scheduling. If you want to squeeze the absolute best from your model and are willing to tune, you could try SGD. For example, an initial learning rate ~0.1 (for scaled inputs) or ~0.01 could work, with momentum 0.9. In practice, given your hardware and data, Adam might reach a good result more quickly, and the difference in final accuracy may not justify the extra effort of tuning SGD. One strategy could be: train with Adam first to get to a plateau, then switch to SGD for fine-tuning if desired (sometimes done in vision tasks to get the last bit of accuracy).
- **Other Optimizers:** There are newer optimizers like **Lion, RAdam, Adafactor** etc., but none have a clear track record in this specific domain that surpasses Adam/SGD. Unless you are experimenting, it’s safest to stick to the well-known ones. One point: if memory is a concern, Adam’s state (which holds 2 extra copies of weights as momentum) will use more memory; in such a case, consider SGD or Adafactor. But with a 3–10M parameter model, this is usually not an issue on a 16GB GPU.

Learning Rate Scheduling

No matter the optimizer, a good learning rate schedule can significantly improve training convergence and final performance: - **Fixed with Step Decay:** A simple and effective schedule is to use a constant learning rate for a while and then **drop it by a factor (e.g. ×0.1)** at certain milestones. For instance, train with LR = 1e-3 until validation error plateaus, then drop to 3e-4, train more, then 1e-4, etc. AlphaZero’s training roughly followed a schedule of this nature (with manually set drop points when progress slowed) [25]. You could plan something like: if after X epochs the validation loss hasn’t improved, reduce LR. - **Cosine Annealing:** Cosine anneal schedules smoothly taper the learning rate over the course of training (optionally with restarts). For example, you might start at 1e-3 and cosine-decay to 1e-5 over 50 epochs. This often leads to a nice converging behavior and can slightly improve generalization by doing very small updates at the end. - **One-Cycle Policy:** With a one-cycle schedule, you increase the LR from a low value to a high value in the first part of training, then decrease it back down in the second half. This “superconvergence” approach can sometimes find a good basin quickly. It’s a bit more experimental, but some practitioners have reported good results even for smaller datasets by using one-cycle (especially with Adam). If you’re unsure, the step decay or cosine are safer bets. - **Warm-up:** If you go with a transformer or a very deep network, consider a short **warm-up** period (e.g. linearly ramp the LR from 0 up to the base LR over the first few thousand iterations). This avoids shocking the network at initialization and is commonly used in transformer training to stabilize the early updates [26]. For a ResNet with BatchNorm, warm-up is usually not necessary but doesn’t hurt.

In summary, a reasonable plan might be: start with LR ~1e-3 (Adam) or ~0.05 (SGD) for epoch 1; if using warm-up, ease into this LR. Train a few epochs, then if using step decay, drop by 10x after, say, 5 epochs, and again after 10 epochs, etc., depending on when progress stalls. Monitor training and validation loss to decide when to reduce.

Batch Size and Training Duration

- **Batch Size:** Your input is fairly small (23×8×8), so memory is dominated by model size and overhead. A Colab T4/P100 (16GB) can likely handle **batch sizes in the few hundreds easily**. If you use mixed precision (FP16), you could even try very large batches (512 or 1024) which might fully utilize the GPU. Large batches can speed up training (through parallelism) and smooth the gradients, but extremely large batches sometimes harm generalization. Since AlphaZero-style training used huge batch sizes (4096) [25] , you might attempt something like 256–512 without worry. Just adjust the learning rate: higher batch often allows a higher LR. For example, if you find 1e-3 works for batch 128, you might try 2e-3 for batch 256 (not strictly linear scaling, but as a rough guide).
- **Epochs:** With 2.2M samples, one epoch is quite large. You likely do not need dozens of epochs. In supervised training on chess positions (like in some research using 1M+ game positions), networks often converge in under 10 epochs of full data. A possible regime: train for ~5 epochs and see where the validation stands. If the model is still improving, you can go 10 or more. Since the data is not expanding (unlike self-play scenarios), be cautious of overfitting after too many epochs. It's crucial to keep a **validation set** (for example, withhold ~5-10% of the data) and monitor the eval loss. Stop (or reduce LR significantly) when you see the validation loss bottom out or start rising while training loss keeps dropping.

Regularization and Other Tips

- **Weight Decay:** Applying weight decay (L2 regularization) is generally beneficial for these networks. A common choice is **1×10^{-4}** for weight decay, which was also used in many AlphaZero replication efforts. If using AdamW, just set `weight_decay=1e-4`. This helps prevent the network weights from growing too large and overfitting.
- **Dropout:** In CNNs with BatchNorm, dropout isn't commonly used (BatchNorm itself provides some regularization and conv nets have spatial redundancy). In a transformer, dropout **is** commonly used (e.g. dropout rate 0.1 on attention and FF layers) to prevent overfitting, especially with not huge data. If you observe overfitting, adding dropout in the fully connected layers of the value head or between residual blocks could help. Keep dropout modest (10-20%) as too high will underfit.
- **Data Augmentation:** For chess, data augmentation can be done in a limited way. One idea: mirror the board left-to-right (swap kingside and queenside) along with adjusting features (e.g. castling rights swap, en passant file mirrored). This effectively doubles your data and is a valid symmetry (chess is symmetric across the vertical midline). Also, if you treat "side to move" properly, you could invert colors of a position (swap White and Black pieces and invert the evaluation sign) to augment data – though since your input already has a side-to-move channel, you'd have to flip that and the eval. These augmentations can help generalization a bit. However, since you said "for now don't think about data improvements," consider this optional. It might be something to try if the model seems to overfit or not generalize well.
- **Gradient Clipping:** If you encounter any instability (e.g. loss jumps, NaNs), consider gradient clipping (e.g. clip norm to 1 or 5). This is more likely necessary with a transformer (where a bad early update can blow up). For a ResNet, usually not needed if LR is reasonable.
- **Mixed Precision Training:** Use PyTorch's automatic mixed precision (`amp`) to train in float16/float32 mixed mode. This will speed up training substantially on T4/P100 and reduce memory usage, allowing larger batches. Just be careful to keep the loss in FP32 to avoid numeric issues, but PyTorch's `GradScaler` handles that. Many modern training setups, including those for large networks, use mixed precision to great effect.

- **Monitoring:** Track both training and validation loss, and possibly secondary metrics. If your labels are game outcomes (thus, effectively classification), track accuracy of predicting winner. If continuous, maybe track correlation between predicted eval and target. Early in training, you should see rapid improvement. If using $\tanh$ output, ensure that the distribution of outputs isn't collapsing (e.g. all zeros); if it is, learning rate might be too high or network too constrained.

Optimizing for Hardware

- On a single GPU like 4060 or Colab's T4, you might find training time to be the main limitation. Use data loaders efficiently (num_workers, etc.) to keep GPU fed. The input is small, so CPU preprocessing is minimal – just ensure the data is loaded from disk efficiently (consider storing the dataset in a binary format for faster read than parsing text).
- If training is still slow, you can utilize Colab's **TPU** (if available) or multi-GPU (if you have that option) to accelerate. But given the scale (a few million samples, moderate model), a single GPU with mixed precision should be okay.
- If you have to choose, prioritize a **smaller model with more training steps** over an extremely large model with fewer steps. The latter can overfit and not reach its potential due to limited training time. It's better to have, say, a 5M parameter network well-trained on the data than a 50M network under-trained.

Summary of Recommendations

In summary, for a strong chess evaluation network under your constraints, a **Residual CNN** is the most straightforward choice – e.g. a ResNet with ~5–10 residual blocks and 64–128 filters, which can be trained on 2M positions from scratch. This approach is backed by AlphaZero's success and should capture the needed patterns [2] [4]. If you are adventurous and have time to experiment, a **transformer-based model** with appropriate positional encoding could yield even better long-range understanding [6], but ensure to regularize and tune it carefully. You can also explore hybrid CNN+Transformer designs for a balance [14]. Keep in mind the end-use: for a minimax engine, evaluation speed is crucial – extremely large networks or complex attention might slow down search, whereas smaller nets or NNUE-style incremental eval can give faster turn-around per node.

On the training side, use **Adam optimizer** initially with a learning rate around 1e-3 (and weight decay ~1e-4), and plan a **learning rate decay schedule** (step or cosine) to ensure convergence. Consider formulating the training target as a classification into win/draw/loss for better gradients [22], especially if your labels come from game outcomes. Use **BatchNorm** (in CNNs) or **dropout** (in transformers) as needed to generalize, and leverage **mixed precision** to speed up training on modern GPUs. With ~2 million examples, a few epochs (on the order of 5–10) with a batch size in the hundreds should be enough to get a strong model – just remember to monitor validation performance to avoid overfitting.

By following these guidelines and iteratively refining, you should be able to train a high-quality evaluation function for your chess engine. Good luck, and happy training!

Sources:

- Silver et al., *Science*, 2018 – AlphaZero's network architecture (8×8 planes, 19-layer ResNet, value head with tanh) [2] [4]

- ChessProgramming Wiki – *AlphaZero* (network details and training regime) [1] [25]
- Monroe & Chalmers, 2024 – *Chessformer* (transformer model outperforming AlphaZero, importance of positional encoding) [6] [7]
- Awruch et al., 2023 – *Representation Matters in AlphaZero* (improved input features and hybrid Conv/Transformer architectures for chess) [22] [14]
- ChessProgramming Wiki – *Stockfish NNUE* (description of NNUE 4-layer structure and incremental update for fast search integration) [20] [17]

---

[1] [2] [3] [4] [5] [25] AlphaZero - Chessprogramming wiki
https://www.chessprogramming.org/AlphaZero

[6] [7] [8] [9] [10] [11] [12] [26] Mastering Chess with a Transformer Model
https://arxiv.org/html/2409.12272v2

[13] [14] [15] [22] Representation Matters for Mastering Chess: Improved Feature Representation in AlphaZero Outperforms Switching to Transformers
https://arxiv.org/html/2304.14918v2

[16] [17] [18] [19] [20] [21] Stockfish NNUE - Chessprogramming wiki
https://www.chessprogramming.org/Stockfish_NNUE

[23] SGD performing better than Adam in Random minority oversampling ...
https://datascience.stackexchange.com/questions/114318/sgd-performing-better-than-adam-in-random-minority-oversampling-i-dont-know-wh

[24] AlphaDDA: strategies for adjusting the playing strength of a fully ...
https://pmc.ncbi.nlm.nih.gov/articles/PMC9575865/