ChatGPT

CNN Model and Training Recommendations for Go Player Style Classification

Model Architecture Design

- **Input Features:** Represent the Go game state as a multi-channel 2D tensor (e.g. 19×19 board with various feature planes). For example, one approach uses **~13 input planes** encoding information like each stone's liberties, empty intersections, and the last 4 moves of the game [1] [2]. This one-hot style encoding from SGF ensures the CNN can exploit spatial patterns and move history on the board.

- **Convolutional Layers:** Use a stack of **convolutional layers** to extract local patterns (shapes of stones, adjacency, etc.) across the board. A moderate-depth CNN (around **4–8 conv layers**) is typically effective. For instance, a simple network for Go used 4 conv layers (128 filters of 5×5 in the first layer, then smaller 3×3 filters) to capture increasingly abstract features [3]. A deeper research model employed **8 convolutional layers**, all with ReLU activations [4]. **Kernel sizes** of 3×3 are common for most layers (as they capture local patterns well), though a slightly larger first-layer kernel (e.g. 5×5) can help capture broader contexts [3]. **Stride** is usually 1 for most conv layers (preserving spatial resolution); some designs downsample by using a stride of 2 in a few layers instead of separate pooling [4]. Always use appropriate **padding** (e.g. pad by 1 for a 3×3 kernel) so that board edges are handled and feature maps don't shrink too quickly [5].

- **Batch Normalization & Activation:** It is recommended to apply **Batch Normalization (BatchNorm)** after conv layers (before activation). BatchNorm stabilizes and accelerates training by normalizing layer inputs. Although some older Go CNNs omitted it, modern practice (e.g. ResNet-style networks) includes BatchNorm to enable deeper nets to train effectively. Use **ReLU** as the activation function (all cited examples use ReLU [4] [3]) since it works well for CNN feature extraction.

- **Dropout Layers:** Include **dropout** primarily to mitigate overfitting, especially in fully-connected or embedding layers. A typical dropout rate is around **0.5** for dense layers [6]. If using dropout in conv layers or mid-network, a smaller rate (e.g. **0.2–0.3**) might be used so as not to drop too many spatial features [7]. In one experiment, a 0.5 dropout on a 64-unit dense layer was used successfully for Go rank prediction [6], whereas another study set a "smaller" dropout rate to preserve important features for move prediction [7]. Adjust the dropout rate based on overfitting observed – use higher dropout if the model is complex relative to the dataset.

- **Residual Connections:** For deeper architectures, consider using **Residual blocks** (skip connections) as in ResNet. Residual connections help training by allowing gradient flow and preventing vanishing issues in very deep networks. For example, Lin & Huang (2024) leveraged skip connections (along with attention) in a Go move predictor CNN to retain key feature information while reducing model size [8]. If your model goes beyond ~5–8 conv layers, incorporating residual connections (each block having a skip path around a couple of conv layers) can significantly improve ease of training and accuracy.

- **Pooling Layers: Pooling** is useful to progressively reduce spatial size and aggregate information, but in Go board data we must balance that with retaining spatial detail. Common designs might use an occasional **MaxPooling** (e.g. 2×2) or use **conv layers with stride 2** to downsample. Notably, one 8-layer Go CNN chose *not* to insert pooling layers between convs, to avoid losing detail when the board is filled with stones; instead they downsampled via convolution stride and padding so edge information is preserved [4]. A reasonable approach is to perform limited pooling (e.g. after every 2–3 conv layers) to reduce the 19×19 feature maps gradually (19→10→5, etc.), or use strided convs for a similar effect.

- **Global Average Pooling vs. Fully-Connected:** For the **final feature aggregation**, a **Global Average Pooling (GAP)** layer is often recommended instead of a large fully-connected layer. GAP computes the average of each feature map, yielding a compact feature vector. This was used in a Go CNN to avoid a fully-connected layer and **reduce parameters** while maintaining spatially aggregated information [9]. Using GAP helps prevent overfitting since it dramatically cuts down the number of weights (no huge dense layer on 361-board features). After GAP, a final small fully-connected layer can produce class logits. In contrast, a traditional approach is to **flatten** the conv feature maps and use one or more **fully-connected (FC) layers**. This can work if the dataset is large enough, but be cautious: flattening a 19×19×F tensor leads to many parameters. For instance, an early experiment flattened the conv output and used a dense layer of 64 units (with dropout) before the output [6] – effective for that task, but a GAP approach would drastically cut those parameters. **Recommendation:** use global average pooling followed by a single FC for classification, unless you have reason to include a deeper classifier.

- **Embedding vs. Output Layer:** The final output depends on your training objective. If you frame player identity recognition as a standard **classification** problem (each player = a class), have the model output a probability for each player via a softmax layer, and train with cross-entropy. If instead you use a **metric learning** approach (comparing playing styles), have the model output an **embedding vector** (e.g. a 64-D or 128-D feature) and *do not* include a softmax layer. In that case you would train with a similarity loss (e.g. triplet loss or contrastive loss) to ensure same-player games map to nearby embeddings. For example, FaceNet's face-recognition approach introduced **triplet loss** to learn an embedding space: the network is trained such that an anchor and positive (same identity) are closer than the anchor and a negative by some margin [10]. In our context, that means games by the *same player* should yield embeddings closer than games by different players. If using triplet/contrastive loss, you'd typically L2-normalize the embedding and use a margin (commonly around **0.2–0.5** in normalized distance). Otherwise, for classification with softmax, the final layer will be of size = number of players and you use a standard cross-entropy loss.

Training Parameter Recommendations

- **Learning Rate and Schedule:** A good starting **learning rate (LR)** for a CNN in PyTorch is on the order of **1e-3**. If using **Adam** optimizer, an initial LR ~0.001 is common (the example Go strength model converged quickly with Adam at default 0.001 [11]). For **SGD** with momentum, you might start higher (e.g. **0.01–0.1**), but then use a decay schedule. In one experiment, an SGD was run at 0.001 with momentum 0.9 [12] (effectively a already-reduced LR for fine-tuning). **Recommended strategy:** start with ~1e-3 and monitor loss; use a learning rate scheduler (e.g. ReduceLROnPlateau or step decay by 0.1 after a certain number of epochs) to adjust. If using a high initial LR (like 0.05–0.1 for SGD), definitely decay it over time (for example, drop by 10× every 10-20 epochs) to ensure convergence. A **cosine annealing** or **cyclical LR** schedule can also be beneficial for smoother convergence.

- **Optimizer Choice:** Both **Adam** and **SGD with momentum** are widely used for CNN training. **Adam** (adaptive moment estimation) is often preferred for quicker convergence and less tuning; it performed well in early Go style experiments (converging in a few epochs) [11]. **SGD + momentum (0.9)** tends to yield slightly better final accuracy when carefully tuned, at the cost of more epochs – for example, a recent Go model used SGD with 0.9 momentum and weight decay to train the network [12]. A good practice is to start with Adam for rapid progress, and if needed, switch to or fine-tune with SGD for final refinement. Ensure to use **momentum ~0.9** if using plain SGD (this helps accelerate training by smoothing gradients). PyTorch's `optim.SGD(..., momentum=0.9)` or `optim.Adam(...)` can be used accordingly. In some cases, **RMSprop** is another alternative (one project started with Adam and also tried RMSprop for a larger network [11]), but Adam generally supersedes it.

- **Batch Size:** Typical **batch sizes** range from 32 up to 128 for training CNNs on Go data, depending on memory. Using larger batches can improve training stability with BatchNorm and utilize GPU better, but you don't want to go so large that you have too few batches (which can hurt generalization). A batch size of **64** is a good middle ground; if memory allows, **128** was used successfully in one Go CNN experiment [12]. In that case, the model (CNN+GCN hybrid) was trained with batch 128 for 10 epochs on a GPU [12]. You may adjust this based on your hardware – if you use BatchNorm, ensure batch size is not too small (< 16) or else consider **GroupNorm/ LayerNorm** as alternatives. For **triplet loss** training, you often want a batch structured to contain multiple examples per player (to form anchor-positive pairs), so you might use smaller batches but stratified sampling. Overall, **32–128** is the recommended range, with 64 as a common choice.

- **Regularization (Weight Decay & Dropout):** It's important to apply **L2 weight decay** to the network weights to prevent overfitting, especially with many parameters. A typical weight decay (L2) is **1e-4** (0.0001) in many CNN setups. Some studies even used a slightly higher decay of **0.1% (0.001)** which is 10×1e-4 [12], but this might be task-specific. We recommend starting with 1e-4 (PyTorch's optimizers have a `weight_decay` parameter for this). Coupled with that, use **dropout** as needed (as discussed in architecture). If your model has a large final dense layer, a dropout of ~0.5 there is prudent [6]. If the model is smaller or you use global pooling, you might use dropout more sparingly (e.g. 0.3) or even not at all if overfitting is not observed. One advanced trick is augmenting the data (random rotations/reflections of the Go board which are symmetry-equivalent) instead of heavy dropout – this leverages game symmetries as a form of regularization [13] [14]. In summary, apply **weight decay ~1e-4**, set **dropout ~0.5** on fully-connected layers (or ~0.2–0.3 on conv layers if used) as a starting point, and adjust based on validation performance.

- **Loss Function:** For **player identity classification**, use a **cross-entropy loss**. This is the standard choice for multi-class classification and is suitable if each training sample (game or position) has a known player label. In practice, after the final softmax, the network is trained to minimize cross-entropy between the predicted probabilities and the true player label. (The Go CNN literature also uses cross-entropy for move prediction with 361 classes [15], which is analogous to our case of players as classes.) Cross-entropy effectively pushes the network to output high probability for the correct player and is minimized when the predicted distribution matches the one-hot ground truth [15]. On the other hand, if pursuing a **metric learning approach**, you would use a **triplet loss** or **contrastive loss**. **Triplet loss** (with a margin α) operates on triplets of samples (anchor, positive, negative): it encourages the distance between anchor and positive (same player) to be smaller than anchor to negative by at least α [16]. A common margin is α ≈ **0.2**. **Contrastive loss** is similar but uses pairs with a binary same/different label. These losses will train the CNN to produce an embedding where games by the same player cluster together.

Keep in mind that training with triplet/contrastive loss can be **trickier** (requiring careful mining of pairs/triplets). If you have many players or want the flexibility to recognize unseen players by similarity, this approach is viable. Otherwise, if the task is simply to classify among a fixed set of known players, cross-entropy with a final softmax is straightforward and usually yields higher accuracy for that closed-set classification [17]. In PyTorch, you can implement triplet loss via `torch.nn.TripletMarginLoss` and cross-entropy via `torch.nn.CrossEntropyLoss`.

Sources: Key architecture and hyperparameter choices are informed by recent Go AI research and projects. For example, Lin *et al.* (2024) developed an 8-layer CNN (with Inception/Attention modules) for move prediction [18] [7], and Xu *et al.* (2023) used an 8-conv CNN with global average pooling (no FC) for Go data [4] [9]. Moudrik's GoStyle project (2016) used a 4-layer CNN plus a dense layer to predict player rank, using 5×5 and 3×3 kernels [3] and found that Adam with LR~0.001 converged in just 2 epochs [11]. Another study on KGS data trained with SGD (LR=0.001, momentum 0.9, weight decay 1e-3) and batch size 128 [12]. We also draw on principles from FaceNet for embedding learning [10]. These references provide a practical basis for the above recommendations.

---

[1] [2] [3] [6] [11] Predicting Go players' strength with Convolutional Neural Nets
http://jmoudrik.github.io/post/2016/01/15/convolutional_neural_net_for_Go_strength_prediction.html

[4] [5] [9] [12] [14] [15] [17] New Hybrid Graph Convolution Neural Network with Applications in Game Strategy
https://www.mdpi.com/2079-9292/12/19/4020

[7] [8] [18] Streamlined Deep Learning Models for Move Prediction in Go-Game
https://www.mdpi.com/2079-9292/13/15/3093?type=check_update&version=1

[10] [16] Triplet Loss: Intro, Implementation, Use Cases
https://www.v7labs.com/blog/triplet-loss

[13] Training Deep Convolutional Neural Networks to Play Go
https://proceedings.mlr.press/v37/clark15.html