# PROBABILISTIC HEURISTICS FOR GRADIENT-FREE HYPERPARAMETER TUNING IN NEURAL NETWORKS

**Christopher Sun**
CS109 Project, Winter 2024

## 1 Introduction

The construction of machine learning and deep learning architectures requires the deliberate and often arbitrary assignment of hyperparameter values. As we explored in class and in Problem Set 6, training a model with different learning rates between a certain range can yield a large contrast in model performance, measured by metrics such as loss and accuracy. In practice, engineers must tune a large number of hyperparameters, each with different ranges, creating a notion of a "hyperparameter space." Let us consider this hyperparameter space: it possesses a unique, intractable topography. If we can somehow optimize this topography and find a hyperparameter configuration that yields its local or absolute minimum, we would be able to numerically solve the hyperparameter tuning problem.

But performance metrics such as loss are not inherently functions of hyperparameters, so we cannot compute analytic gradients to update them. In this project, I introduce a probabilistic heuristic to accomplish gradient-free optimization of the hyperparameter space, constrained to two hyperparameter variables (for ease of visualization and as a proof of concept). I make use of concepts like random variables, probability mass functions, probability density functions, random sampling, bootstrapping, and $p$-values.

Overall, the value provided by my project is an increase in hyperparameter tuning efficiency and a reduction in time and computational cost. Compared to tools such as Grid Search and Random Search, my proposed approach learns to rule out certain hyperparameters configurations and pursue others. This is accomplished by updating our so-called "belief" of where the optimal configuration is based on previously-sampled configurations and their associated losses.

## 2 Baseline Testing

I trained three types of neural networks on three toy data sets representing different classes of data:

- **Artificial Neural Network (ANN):** Tabular data, Breast cancer diagnosis [1]
- **Convolutional Neural Network (CNN):** Image data, Fashion MNIST Classification [2]
- **Recurrent Neural Network (RNN):** Time-series data, Temperature Prediction [3]

I first trained 50 ANNs, 50 CNNs, and 50 RNNs configured with randomly-selected hyperparameters, gathering model performance metrics such as loss and accuracy. This represents the baseline or control of my experiment: using uniform sampling to build a slate of models to test hyperparameter configurations through brute force. Table 1 lists the hyperparameters selected for each type of deep learning model, as well as their sample space.

Table 1: Hyperparameter Specifications

| | **Hyperparameters** | |
|---|---|---|
| **Tabular (ANN)** | Number of hidden units: $2^0$ to $2^6$ | Dropout rate: 0.00 to 1.00 |
| **Image (CNN)** | Number of filters: $2^2$ to $2^6$ | Learning rate: $10^{-4}$ to $10^{-1}$ (log scale) |
| **Time Series (RNN)** | Lookback period: 2 to 120 examples | Number of LSTM units: $2^0$ to $2^7$ |

### 2.1 Observations

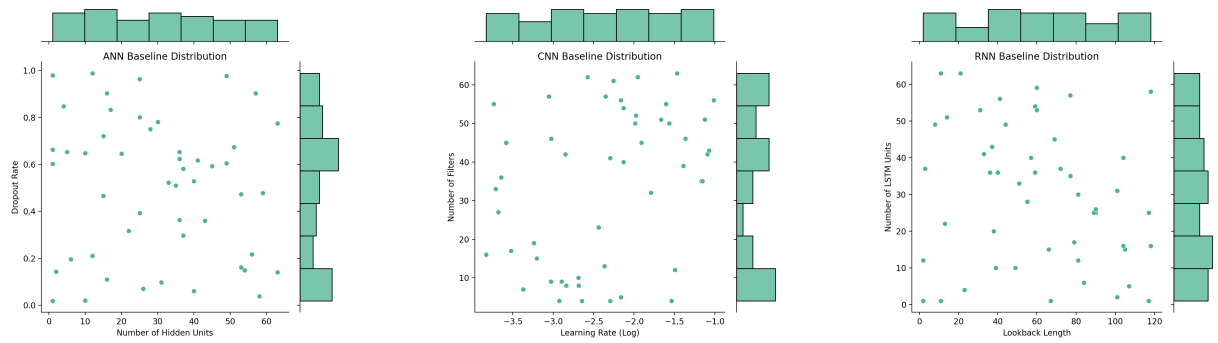



Figure 1: Randomly-sampled hyperparameter configurations for 50 models of each type

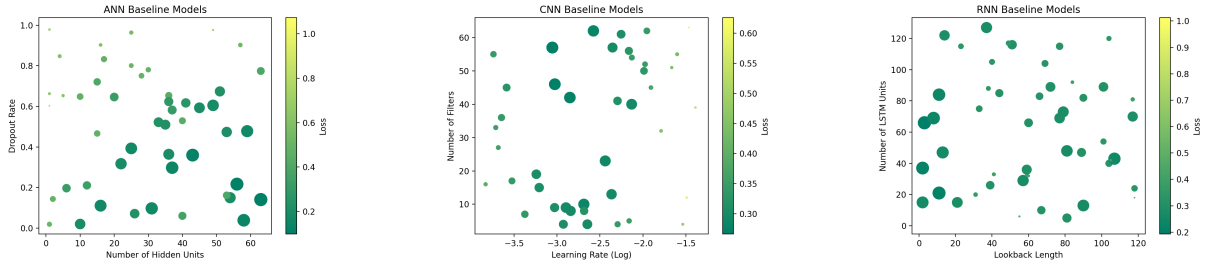



Figure 2: Hyperparameter configurations colored and sized according to model loss

Figure 1 displays the roughly uniform nature of sampling. Figure 2 displays the relationship between a model's hyperparameter configuration and its corresponding loss. Importantly, Figure 2 shows that the hyperparameter space has a distinct topography: some ranges of hyperparameter values are more conducive to lower losses. Though we do not know the precise location of the global minimum loss, *wouldn't it be cool to iteratively narrow down the hyperparameter range this loss dwells in by choosing hyperparameters similar to ones that have previously worked well?* This question serves as the motivation for the tuning heuristic described in the next section.

## 3 Experimental Testing

I implemented a hyperparameter tuning heuristic for the ANNs, CNNs, and RNNs, as follows (a more formal version is in Section 4):

1. Train an initial population of $n_i$ models with randomly-sampled hyperparameter configurations.
2. From the losses of these models, generate a probability mass function that when sampled from will tend to yield the "argmin" configuration that corresponds to an experimentally-verified low cost (i.e. map low costs to high probabilities).
3. Add noise to this "mean value" of hyperparameters by sampling, such as from Gaussian or Uniform with a certain measurement of variance that decreases with more iterations of tuning.
4. Divide $n_i$ by a weighting factor $\beta$ and jump to step 2; this is one "cycle." Repeat until a specified number of cycles $k_{max}$ is reached, or until $n_i = 1$.

My experimental hypothesis was that after several cycles of the proposed tuning heuristic (while ensuring that the cumulative number of models is less than that of the control[1]), there is a statistically significant improvement in

[1]We satisfy this criterion as long as $\sum_{i=0}^{k_{max}} \lfloor n_i \cdot \left(\frac{1}{\beta}\right)^i \rfloor$ is less than the number of randomly sampled configurations tested. By the sum of an infinite geometric series, an upper bound on this sum is $\frac{\beta n_i}{\beta - 1}$, meaning we can constrain $\beta > \frac{x}{x - n_i}$, where $x$ is the number of brute force models, to guarantee that our tuning approach uses less models.

hyperparameter tuning efficiency, which can be quantified by the $p$-value of the difference of mean/max between the two approaches, and the cumulative number of models needed to descend to an ultimate model with minimum loss.

Figure 3 shows the distribution of model losses as the number of tuning cycles increases. Across all three model types, Cycle 0 corresponds to randomly-sampled hyperparameter no different than the baseline testing approach; thus, cycle 0 losses have the highest variance. With the completion of each tuning cycle, model losses decrease, eventually seeming to converge at a minimum loss value. The model that achieved this minimum loss has hypothetically converged at the minimum of the hyperparameter space.

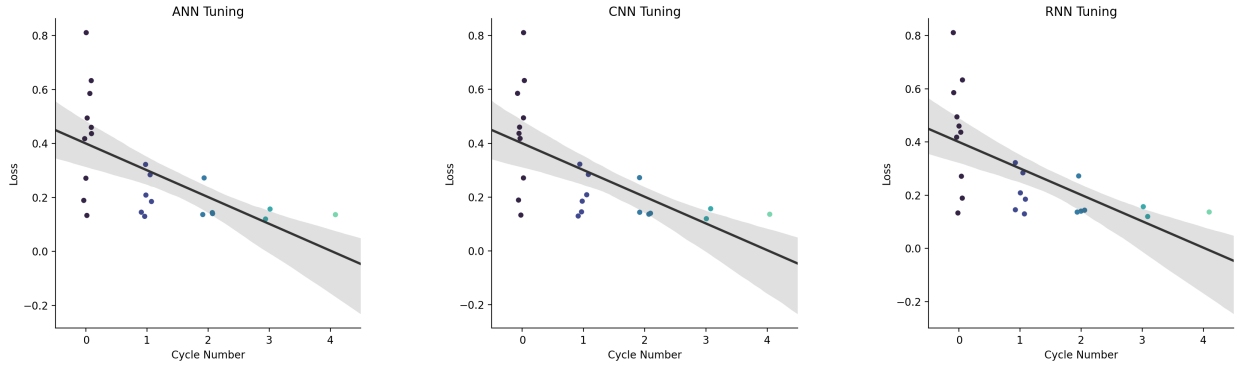
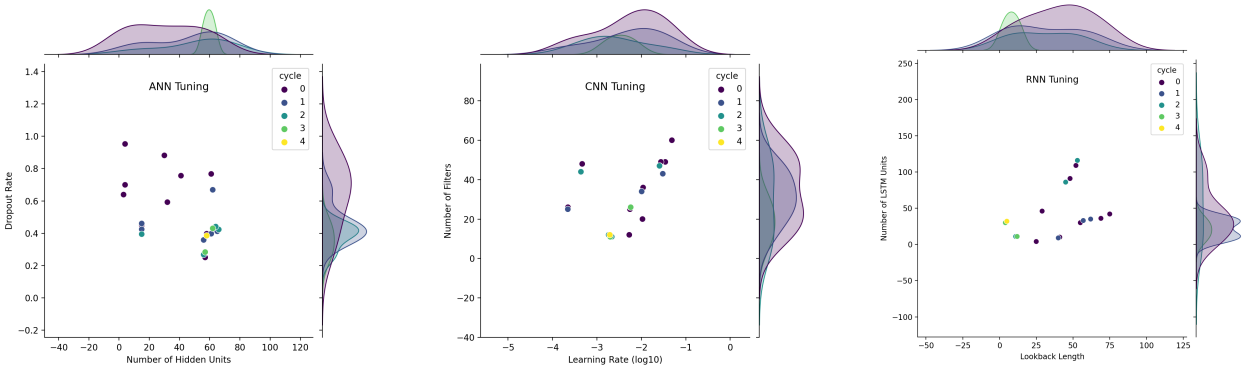



Figure 3: Model losses fitted with linear regression that shows downward trend across tuning cycles

The borders of the plots in Figure 4 display probability density curves that were calculated using kernel density estimation. These distributions represent our belief of where the optimal hyperparameter value is. We can see the effectiveness of the heuristic from the narrowing of these distributions over time.

We can think of the starting distribution as the "prior," the experimental losses as "evidence," and the resulting distribution as the "posterior." The posterior for cycle $N$ becomes the prior for cycle $N + 1$.




(a) $y = x$ (b) $y = 3 \sin x$ (c) $y = 5/x$

Figure 4: Three simple graphs

Finally, to quantify the effectiveness of this hyperparameter tuning approach, I use bootstrapping to calculate the statistical significance of the difference in mean loss and maximum loss between the two populations – baseline and experimental. We would expect to see a very low $p$-value if the mean and maximum loss of the experimental population is significantly lower than that of the baseline; this is what Table 2 confirms. Hence, the tuning heuristic is effectively narrowing down to sub-configurations of hyperparameters that yield progressively lower losses.

Table 2: Significance of Distribution Differences

| | ANN | CNN | RNN |
|---|---|---|---|
| $p$-value (**mean**) | 0.0005 | 0.0594 | 0.1164 |
| $p$-value (**max**) | 0.0002 | 0.0237 | 0.0131 |

# 4 Appendix

**Algorithm 1** Hyperparameter Tuning

**procedure** OPTIMIZE ▷ Driver Method
  configs, metrics ← TRAIN INITIAL POPULATION()
  **for** i < num_cycles **do**
    UPDATE CYCLE()
    configs, metrics ← TUNING STEP(configs, metrics) ▷ parameters are updated in TUNING STEP()
  **end for**
  **return** metrics
**end procedure**

**function** TRAIN INITIAL POPULATION(n=50) ▷ Helper Functions for OPTIMIZE()
  **for** i < n **do**
    hparam_config ← sample from $\mathcal{U}$(hparam_range$_{\text{min}}$, hparam_range$_{\text{max}}$)
    metrics ← train_model(hparam_config)
    Store hparam_config, metrics
  **end for**
  **return** all_hparam_configs, all_metrics
**end function**

**function** UPDATE CYCLE($\beta = 1.5$)
  curr_cycle ← curr_cycle + 1
  weight ← weight / $\beta$
  num_models ← num_models / $\beta$
**end function**

**function** TUNING STEP(configs, metrics)
  losses ← metrics["losses"]
  **for** i < num_models **do**
    idx ← SAMPLE NEW(losses)
    $\mu_{val}$ ← losses[idx]
    hparam_config ← sample from $\mathcal{U}(\mu_{val} \cdot (1 - \text{weight}), \mu_{val} \cdot (1 + \text{weight}))$ or $\mathcal{N}(\mu_{val}, (\text{weight} \cdot \mu_{val})^2)$
    metrics ← train_model(hparam_config)
    Store hparam_config, metrics in new lists
  **end for**
  concatenate(metrics, new_metrics); concatenate(configs, new_configs)
  **return** new_configs, new_metrics
**end function**

**function** SAMPLE NEW(losses, $\lambda = 3$) ▷ Helper Functions for TUNING STEP()
  inverted_losses ← minmax_normalize(losses)
  pmf_numerical ← softmax($\lambda \cdot$ inverted_losses)
  **return** sample index from pmf_numerical
**end function**

# References


[1] Wolberg,William, Mangasarian,Olvi, Street,Nick, and Street,W.. (1995). Breast Cancer Wisconsin (Diagnostic). UCI Machine Learning Repository. https://doi.org/10.24432/C5DW2B.

[2] Xiao, H., Rasul, K., & Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. arXiv preprint arXiv:1708.07747.

[3] Vito,Saverio. (2016). Air Quality. UCI Machine Learning Repository. https://doi.org/10.24432/C59K5F.