A Systematic Approach to Hyperparameter Tuning for Neural Network-Based PDE Solvers

January 31, 2025

This note provides a systematic approach to selecting hyperparameters for training neural networks. [1] Specifically, we consider the context of solving high-dimensional Hamilton-Jacobi-Bellman (HJB) equations in Brownian control problems using deep learning-based methods, e.g. the DeepBSDE and Deep Splitting methods.

[1] It is important to note that particular "engineering tricks" that work for one problem may not work for another. Instead, we hope this note provides a systematic approach to achieving stable training loss and good policy performance.

Overview

Our approach consists of three phases:

- **Exploration**: quick trial and error to choose good hyperparameters
- **Improvement**: identify issues and useful tricks fix them
- **Exploitation**: push to the limit

The key is to *focus on the first two phases*, not the exploitation phase.

Core Metrics

Since we train solvers for control problems, the most important metric is **policy performance**. We also consider the loss curve, focusing on convergence stability and **final loss value**. When the loss does not converge, policy performance tends to be poor.

Start with Minimal Representative Examples

We recommend starting with minimal yet meaningful examples. [2] Once a good configuration is found, move to a slightly more complex version and repeat the process.

[2] The idea is to apply Steps 1 and 2 iteratively on progressively more complex problems. Problems in lower dimensions tend to be more tractable and interpretable. This approach allows for faster iteration cycles and makes it easier to identify and debug issues.

Step 1: Exploration of Hyperparameters

In this step, we perform quick trial and error to identify good configurations. We test **one change at a time** to isolate effects. [3]

[3] Three assumptions are made here:

1. **Generality**: A successful change should improve performance across multiple scenarios.
2. **Monotonicity or Unimodality**: Hyperparameter effects are predictable (e.g., larger networks monotonically improve accuracy up to a point).
3. **No Synergy**: If individual changes (A or B) fail, their combination (A+B) is unlikely to succeed.

The common hyperparameters and their typical values are listed in Table 1. Please remember to add problem-specific hyperparameters, e.g., initial state distribution, diffusion term multiplier (both are commonly used to for getting better coverage of the sample space), and most importantly, the **reference policy** used to generate training data.

Hyperparameters	**Values**
Precision	Typically float64
Neural network architecture	Typically MLP
Number of hidden layers	Typically 2, 3, or 4
Number of nodes per layers	Typically 50 to 200, or larger than problem dimension.
Activation function	Start with ReLU or LeakyReLU
Batch normalization	On / off for input, hidden, output layers. Start with off for all layers.
Gradient clipping	Start with none
Delta clipping	Start with none
Optimizer	Typically Adam
Batch size (training)	Powers of 2, such as 128, 256, etc.
Batch size (validation)	Powers of 2, larger than training batch size
Learning rate scheduler	Start with manual (piecewise constant)
Learning rate (initial)	Typically 10^{-2} or 10^{-3}
Learning rate decay rate	Typically 1/2 or 1/10
Learning rate (minimum)	Typically 10^{-5} or 10^{-6}
Total number of iterations	TBD (manual adjustment)

Table 1: Hyperparameters and their typical values

We have the following remarks based on our experience:

- **Start with testing network architecture**: The first step is to explore the network architecture. We suggest testing things in the following order:
  1. Number of layers
  2. Number of nodes in each layer
  3. Activation function
- **Target for stable loss convergence**: When loss does not converge stably, we suggest trying things in the following order:
  1. Use smaller initial learning rate
  2. Use larger batch size
  3. Try gradient clipping
  4. Try batch normalization
  5. Try delta clipping in loss function
- **Learning rate scheduler**: We suggest starting by manually cutting the learning rate. [4] [5] This helps improve understanding of the problem.

[4] The rule of thumb is to cut the learning rate when the loss curve flattens.

[5] We also suggest waiting more patiently before cutting the learning rate as it gets smaller.

Step 2: Model-Based Validation for Improving Policy

Next, we validate solutions (e.g. the trained policy) against domain knowledge and identify corrective measures. Specifically, we want to:

1. Verify physical plausibility of results,
2. Confirm mathematical consistency with problem structure,
3. Check boundary condition adherence.

Ideally, we want to look from as many angles as possible. [6]

[6] For example, we can look at:

1. **Core metric**: Policy performance (cost/reward) vs. analytical/benchmark solutions
2. **Other visualizations**:
   - Temporal trajectories of states / actions / policy / system dynamics. (E.g., is policy smooth / stable over time?)
   - Surface plots (over states) of value functions, its gradients, and policy. (E.g., are they monotonic as expected?)

If we see important issues that cannot be resolved by revisiting Step 1, then we consider certain "engineering tricks" to fix them. For example, we've found the following useful:

- **Smoothing to prevent vanishing gradients**: In the loss function, terms like $(\cdot)^+$ may cause vanishing gradients. To prevent this, we replace this "ReLU" function with a "ELU" or "LeakyReLU" type function. The smoothed function then gradually converges to the original "ReLU" function as we proceed to a target training step.
- **Shape constraints**: For example, if we know that the output from a network is supposed to be non-negative, we can add a penalty term for negative values to the loss function.

Step 3: Exploitation

This step aims to push performance to its limit for the final 1% to 5% improvement. Methods include early stopping and other techniques.

References