# Smarter modeling over smarter algorithms

Brian Yu

December 14, 2022

## 1 Extended abstract

Reinforcement learning (RL) algorithms suffer from high variance in their environments, further exacerbated in the online RL off-policy case. We fix our reinforcement learning algorithm to be the simple policy gradient algorithm and permute other aspects of the training and modeling pipeline in an attempt to stabilize and improve policy gradient performance. Specifically, we develop a new model that draws heavily from state of the art (SOTA) transformer models, we isolate sources of variance within the policy gradient algorithm and attempt to stabilize them, and we perform ablation studies over sources of bias or variance within the chosen stochastic optimization algorithm.

The most commonly used underlying model is a multi-layer perceptron, a network composed of linear layers interspersed with activations. We can stabilize this model by adding in layernorms which account for feature scaling, softmax-based attention which forces good conditioning of the model, leading to more stability, and improved dense layers. These dense layers are composed of a layernorm, linear layer, ReLU, and a final linear layer. Specifically, the final structure of the model is composed of a dense layer, softmax-based attention, a dense layer, and a final layernorm. All of these components put together create a model that is as capable and expressive as the baseline multi-layer perceptron, but significantly more stable.

Within the policy gradient algorithm, there are several opportunities for variance. First, the q-values are calculated using a single step estimate using the input data, but they are later on normalized within the training batch and also subtracted from baseline values calculated by a neural network with much less variance than our single sample q-values estimate. However, we can further reduce the bias of this algorithm by reducing any overestimation caused by using a single baseline neural network by taking the minimum of a set of output of an ensemble of baseline neural networks.

Finally, we choose to use the popular AdamW stochastic optimization algorithm, which contains two primary sources of bias and variance. First, the decoupled weight decay regularization imposed by the algorithm introduces significant amounts of bias, which may be unproductive for complex environments. Second, the exponential moving average (EMA) values average over many of the previous optimization steps which decreases the bias generally. However, there is a common scenario where this momentum causes the model to "overshoot" its intended set of parameters, which results in a poor set of next step training trajectories, which increases the likelihood that the model derails. We ablate over both of these two sources of bias.

We evaluate and compare all experimental setups on two different environments, a diagnostic and easy environment CartPole and a hard environment HalfCheetah. In summary, we develop and evaluate a more stable model, we evaluate an ensemble baseline, and we ablate over the weight decay and EMA beta values in the AdamW stochastic optimization algorithm.

Generally, we develop a framework for fast and reliable RL pipeline experimentation, which can be found at `https://github.com/bri25yu/homework_fall2022`.

# 2 Introduction

The RL pipeline is very complicated, including components of the training pipeline, the RL algorithm, the underling model, and the stochastic gradient descent algorithm.

## 2.1 Training pipeline

We run our models using an online off-policy training setup. Specifically, we take $T$ update steps, where each update step consists of collecting $N$ trajectories worth of steps in the environment of max length $L$ ($NL$ total steps collected) and performing a single backpropogation step. The model is updated once and only once per update step i.e. there are no tricks in updating a component every other step, etc. Each step consists of a fixed set of data, exactly $NL$ steps collected in the environment. This way, every algorithm is forced to use the same amount of data and compute budget, allowing for a more fair comparison.

**Algorithm 1** Training pipeline

**Require:** $T$ (update steps), $N$ (trajectories collected per update step), $L$ (max length of a single trajectory), $\pi_\theta$ (policy)

**for** $i = 1$ to ... $T$ **do**
  **for** $t = 1$ to ... $NL$ **do**
    Collect next step using $\pi_\theta$ or reset environment if terminal
  **end for**
  $\pi_\theta \leftarrow$ AdamW update$(\pi_\theta, D)$
**end for**

## 2.2 Policy gradient algorithm

There are a variety of online off-policy RL algorithms to choose from. We pick one of the oldest, simplest, and most intuitive algorithms, policy gradient [Sut+99].

The policy gradient algorithm leverages two separate neural networks, a policy network and a baseline network. The policy network is typically parameterized to receive input observations and output a distribution over possible actions in the action space, from which an action can be sampled. The baseline network is parameterized to receive input observations and output an estimate of the "value" of a state. Having a baseline network drastically reduces the variance of the policy gradient's single step update estimation.

The policy gradient's improvement guarantee comes from its calculate of advantages using a set of Q-values and the aforementioned baseline values. These advantages intuitively represent how much better the current action taken in the data is compared to the average action taken at a particular state. Actions that have positive advantages move the policy parameters to pick those actions with higher probability at the step. The formula for Q-values is straightforward, containing the direct reward value and a discounted recursive estimation of the value of the state at the next time step.

The policy loss is backpropogated through the log probabilities calculated for the given actions taken in the data, weighted by their respective advantages. Higher probability actions that are better than average are weighted highly positive and vice versa for the other three cases. The baseline loss uses Q-values as the target and some notion of a distance-related loss function, such as mean-squared-error (MSE) loss.

Finally, each loss is backpropogated to their respective networks using stochastic gradient descent updates.

**Algorithm 2** Policy gradient algorithm single update step

**Require:** $\pi_\theta$ (policy), $\phi$ (baseline)

Calculate action distribution using $\pi_\theta$
Calculate log probabilities using action distribution and actions taken
**for** $t = 1$ to $L$ **do**
    Q-values[t] $\leftarrow$ rewards[t] $+\gamma *$Q-values[t+1]
**end for**
Calculate values using $\phi$ and the observations
Advantages $\leftarrow$ normalize(Q-values - values)
Policy loss $\leftarrow$ - action log probabilities $*$ advantages
Baseline loss $\leftarrow$ DistanceLossFunction(values, Q-values)
$\pi_\theta \leftarrow$ AdamW update($\pi_\theta$, policy loss)
$\phi \leftarrow$ AdamW update($\phi$, baseline loss)

## 2.3 Modeling

State of the art (SOTA) models today primarily consist of four building blocks: layernorm, linear layers without bias, nonlinearities, and dropout. We concern ourselves with the first two: we leave dropout regularization as an unexplored topic and we fix our nonlinearities to use rectified linear units $\text{ReLU}(x) = \max(x, 0)$.

The choice of nonlinearity typically isn't too important as long as it's in this linear unit class that has identity gradient if the value is nonnegative. Intuitively, ReLU suffers from a lack of expressivity, but later structures sandwich nonlinearities between weight layers, allowing negative output, and in combination with scaling weight sizes, the lack of expressivity in ReLU is typically remediated.

### 2.3.1 LayerNorm

The layernorm [BKH16] normalizes by the variance and multiplies each feature by some weight. The weights are initialized to 1.

**Algorithm 3** Layernorm

**Require:** weights $\in \mathbb{R}^d$, inputs $X \in \mathbb{R}^d$

$X \leftarrow X/\text{var}(X)$
$X \leftarrow X *$ weights
return $X$

The layernorm is advantageous in many positions. First, a feature vector is always unit standard deviation, allowing downstream weight matrix multiplication operations to be comparable within a single data point. Second, the weights are an explicit mechanism for scaling, something that is hard to perform over an entire weight matrix. This allows for better conditioning in a variety of scenarios, including softmax-based attention or crossentropy loss.

### 2.3.2 Linear layer

A linear layer without bias simply performs a matrix multiplication. The weights are initialized uniformly randomly according to $\mathcal{U} \sim \text{Uniform}(-a, a), a \propto \frac{1}{\sqrt{d}}$

**Algorithm 4** Linear layer without bias

**Require:** weights $W \in \mathbb{R}^{d_2 \times d_1}$, inputs $X \in \mathbb{R}^{d_1}$

return $WX$

Linear layers are typically used with a bias, but it turns out that a bias is not productive and is not necessary.

### 2.3.3 Putting model structures together

SOTA models typically combine these four basic building blocks into two structures: attention and dense feedforward networks [Raf+19; Bro+20]. These structures are composed of four different building blocks A visualization of these structures is found in 2.3.3.

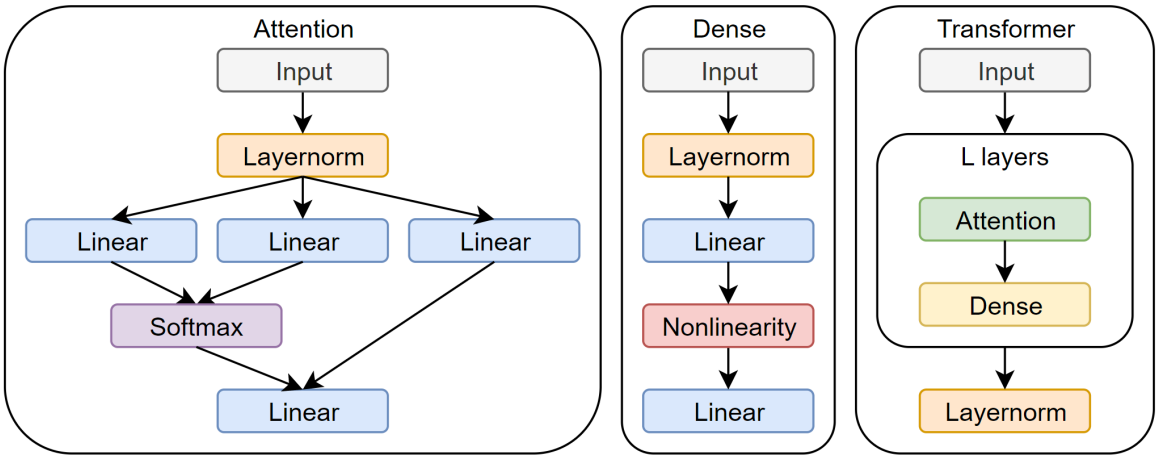


Figure 1: Left: A typical attention module. The linear layers in the middle are the query, key, and value matrices, respectively and the linear layer at the bottom is the output layer. Middle: A typical dense feedforward network. Note that the nonlinearity is sandwiches by two linear layers. Right: A typical transformer that uses both the attention and dense layers

### 2.3.4 Models

The base model is a multi-layer perceptron, a network composed of linear layers interspersed with activations. We can stabilize this model by adding in layernorms which account for feature scaling, softmax-based attention which forces good conditioning of the model, leading to more stability, and improved dense layers. These dense layers are composed of a layernorm, linear layer, ReLU, and a final linear layer. Specifically, the final structure of the model is composed of a dense layer, softmax-based attention, a dense layer, and a final layernorm. All of these components put together create a model that is as capable and expressive as the baseline multi-layer perceptron, but significantly more stable.

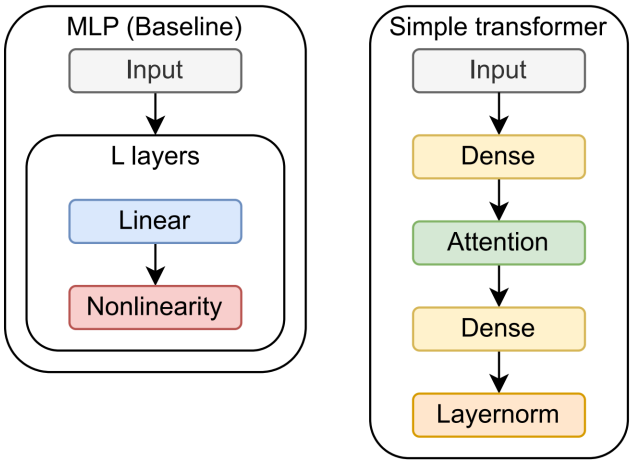


Figure 2: The baseline model and the developed simple transformer model

## 2.4 Optimization

For our stochastic optimization algorithm, we leverage the AdamW [LH17; KB15] optimizer, the most popular optimizer today for performing stochastic gradient descent on deep neural networks. It leverages the ideas of momentum to reduce the variance of updates estimated from minibatches of data and takes steps in weight latent space by adjusting gradient values using a first-order estimate of the hessian.

The AdamW algorithm is summarized below in 5.

**Algorithm 5** AdamW algorithm

**Require:** $\gamma$ (lr), $\beta_1, \beta_2$ (betas), $\theta_{t-1}$ (params), $f(\theta)$ (objective), $\epsilon$ (epsilon), $\lambda$ (weight decay)

1: $m_0 \leftarrow 0$ (first moment)
2: $v_0 \leftarrow 0$ (second moment)
3: **for** $t = 1$ to ... **do**
4: $\quad \theta_t \leftarrow \theta_{t-1} - \gamma\lambda\ \theta_{t-1}$ ▷ weight decay
5: $\quad m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$ ▷ EMA-biased 1st moment estimate
6: $\quad v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$ ▷ EMA-biased 2nd moment estimate
7: $\quad \widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ ▷ bias-corrected 1st moment estimate
8: $\quad \widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ ▷ bias-corrected 2nd moment estimate
9: $\quad \theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ ▷ update
10: **end for**

### 2.4.1 Weight decay

Line 4 imposes an explicit weight decay on the parameters $\theta_t$ proportional to $\theta_{t-1}, \gamma, \lambda$. Intuitively, we certainly need an update proportional to our learning rate $\gamma$ and we should update somewhat on the order of the parameter magnitude. The weight decay $\lambda$ parameter can be interpreted as a percentage of the weight to decay. $\lambda$ is typically 1e-2 or 1% weight decay.

### 2.4.2 Exponential moving average

Lines 5 and 6 calculate an exponential moving average (EMA) of the first and second moments, respectively. This EMA has several purposes. First, it stabilizes training by averaging the moments over steps. This is incredibly important when dealing with overparameterized networks and with noisy data. Second, the EMA surprisingly reduces the bias because averages over recent steps yields better estimations of the true gradient. Productive noise is compounded over time and unproductive noise is averaged out over time.

In practice, the parameters relating to the EMAs are $\beta_1 = 0.9, \beta_2 = 0.999$ for the first and second moment, respectively. We use very high values of $\beta$ in order to retain as much information from past steps as possible, without suffering from the bias reduction from how stale the old moments are. We use a significantly higher value for the second moment estimation because of how unstable it is.

# 3 Experiments

We evaluate and compare all experimental setups on two different environments, a diagnostic and easy environment CartPole and a hard environment HalfCheetah. In summary, we develop and evaluate a more stable model, we evaluate an ensemble baseline, and we ablate over the weight decay and EMA beta values in the AdamW stochastic optimization algorithm.

All of the experiments use the same training pipeline, with different parameters depending on the environment. All experiments use the same policy gradient algorithm and the same stochastic optimization algorithm AdamW. A table of experiment parameters can be found in 1. Shared values may be ablated over later on.

| Experiment parameters | | | |
|---|---|---|---|
| **Parameter** | **Shared** | **CartPole** | **HalfCheetah** |
| Weight decay | 1e-2 | | |
| Eval num trajectories | 2 | | |
| Learning rate | 1e-2 | | |
| Batch size | 1 | | |
| Training steps | | 100 | 1000 |
| Max trajectory length | | 500 | 1000 |
| Env steps seen | | 50k | 1mil |
| Number of seeds | | 3 | 1 |

Table 1: The values for weight decay and eval num trajectories are very typical for today's SOTA experiment setups. The number of training steps is incredibly small for today's standards, which typically use 1mil steps to evaluate models, making this challenge tougher. The learning rate value of 1e-2 is very large compared to today's models which typically have a maximum learning rate of around 1e-4 to 1e-3. The batch size value of 1 is very small compared to today's models which typically have a batch size of 256 to 1024, making this challenge tougher. The formula for number of training env steps seen is the number of steps times the batch size per step times the max trajectory length.

### 3.1 Modeling

We begin by comparing the baseline model with the simple transformer designed in 3.1 named "PolicyGradient" and "SimpleTransformer" respectively. Both have relatively high variance, but the simple transformer model reaches the target reward threshold before the baseline model and stays above it for longer.

Once we introduce an ensembled baseline in addition to using the simple transformer model, we see very low variance all throughout the the runs, with slight variance in the range 40-60 iterations, and absolute convergence starting around 80 iterations. It's clear that using the simple transformer model with the baseline heavily reduces the variance of our algorithm.

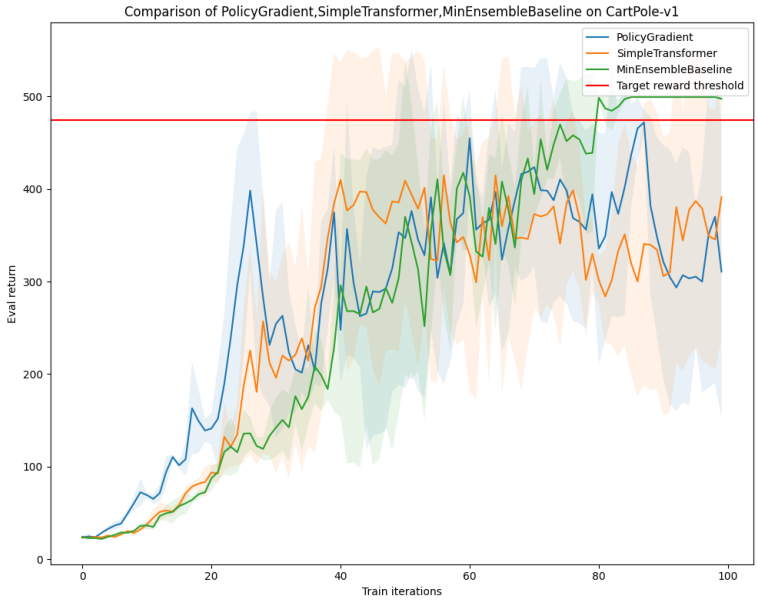


Figure 3: Evaluations on the CartPole environment for the baseline model ("PolicyGradient", blue) and the developed simple transformer model ("SimpleTransformer", orange), and the ensemble baseline model ("MinEnsembleBaseline", green).

On the harder HalfCheetah environment, the baseline policy quickly derails and performs inconveivably poorly, while the models using SimpleTransformer steadily improve. Notice again that the ensembled baseline model slightly outperforms the single baseline estimation model.

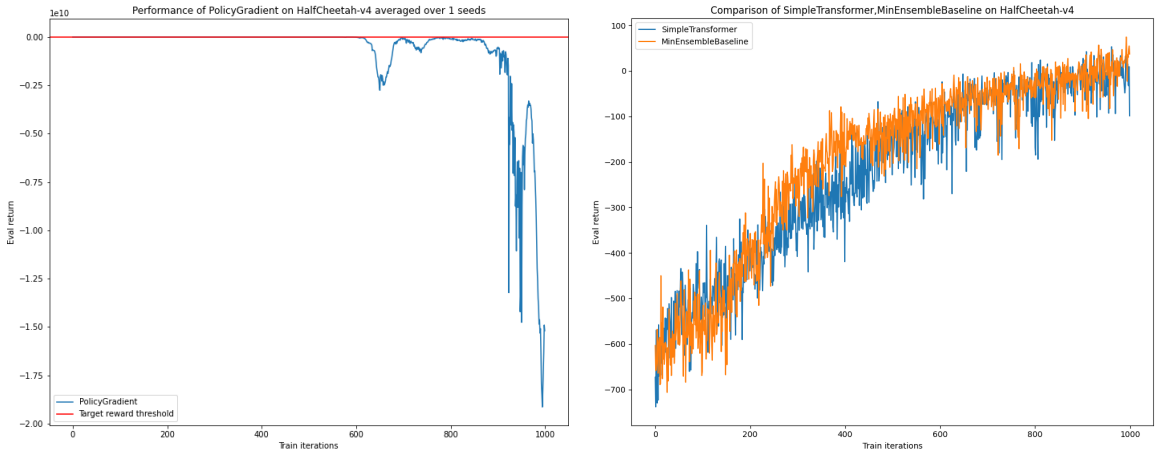


Figure 4: **Left**: Evaluations on the HalfCheetah environment for the baseline model. **Right**: Evaluations on the HalfCheetah environment for the developed simple transformer model ("SimpleTransformer", blue) and the ensemble baseline model ("MinEnsembleBaseline", orange).

## 3.2 Weight decay

Next, we ablate over different values of weight decay. The default weight decay is 1e-2, or 1% of the current weight. We ablate over additional values of 0 (0% weight decay) and 5e-2 (5% weight decay). Results are shown in 3.2.

Clearly, the baseline weight decay value of 1% performs the best, reaching absolute convergence starting at 80 iterations. Having less weight decay such as in the 0% weight decay case causes extreme variance between runs as the model tends to overfit in the absence of weight decay regularization. Having more weight decay beyond the baseline 1% also harms performs, while the variance is about the same as the baseline, 5% weight decay introduces significant enough bias such that the model never correctly converges to the target reward. This is also reflected in the weights over time shown in 3.2.

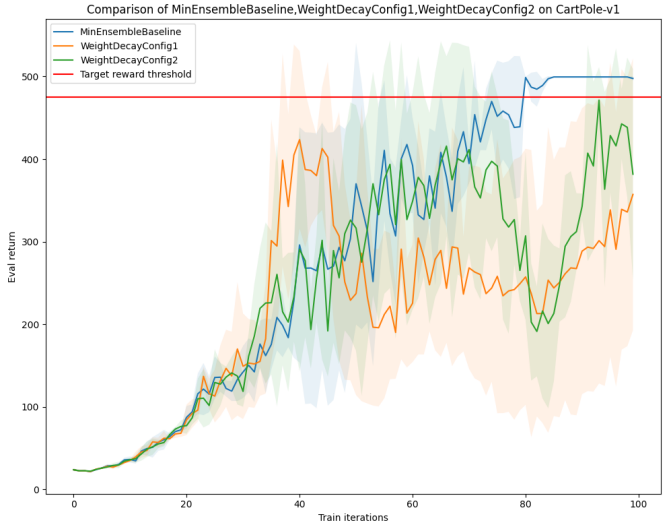


Figure 5: Results for ablations over weight decay values of 0.0 ("WeightDecayConfig1", orange), 1e-2 baseline ("MinEnsembleBaseline", blue), and 5e-2 ("WeightDecayConfig2", green).

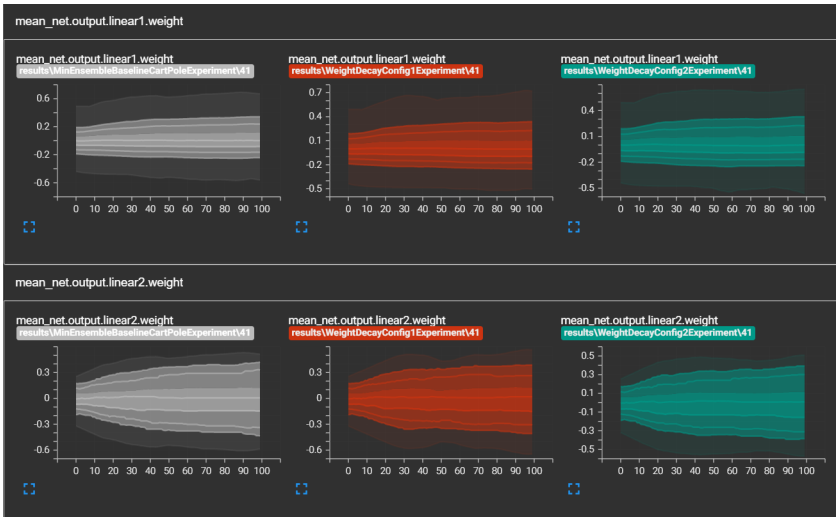


Figure 6: Weights over time for ablations over weight decay values of 0.0 ("WeightDecayConfig1", red, center), 1e-2 baseline ("MinEnsembleBaseline", gray, left), and 5e-2 ("WeightDecayConfig2", green, right).

### 3.3 EMA beta values

We begin by ablating over first order moment estimation $\beta_1$ values of 0.85, 0.9 (baseline), and 0.95. Intuitively, $\beta_1 = 0.9$ means that we keep 90% of the current first order moment estimate and add 10% of the new calculated gradient. Higher values of $\beta_1$ average more gradients from the past together at each update step and lower values average less gradients from the past. This results in higher values of beta having more bias if the model learns quickly, as in the results 3.3 for $\beta_1 = 0.95$. Lower values of $\beta_1$ result in higher variance as the single step moment estimation is given more weight as in the case for $\beta_1 = 0.85$. Neither ablation converges to the target reward threshold.

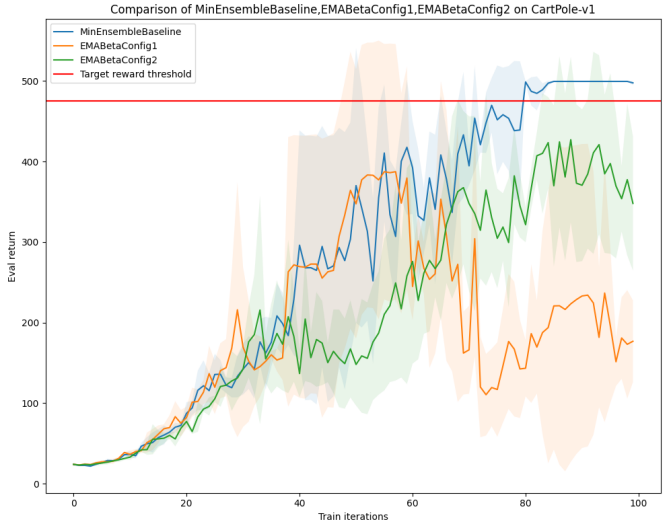


Figure 7: Results for ablations over $\beta_1$ values of 0.85 ("EMABetaConfig1", orange), 0.9 baseline ("MinEnsembleBaseline", blue), and 0.95 ("EMABetaConfig2", green).

This can also be seen in 3.3, where the lower $\beta_1 = 0.85$ weights are much more differentiated (and thus higher variance) than the baseline $\beta_1 = 0.9$, and the $\beta_1 = 0.95$ weights are much less differentiated (and thus lower variance) than the baseline.


Figure 8: Weights over time for ablations over $\beta_1$ values of 0.85 ("EMABetaConfig1", blue, left), 0.9 baseline ("MinEnsembleBaseline", gray, right), and 0.95 ("EMABetaConfig2", pink, center)

We ablate over an additional value for the second order moment estimation $\beta_2$ value of 0.95. Intuitively, lower values of $\beta_2$ destabilize the adaptive steps that AdamW takes in the latent loss space, as reflected by the results shown in 3.3. Using a $\beta_2$ value of 0.95 compared to the baseline value of 0.99 results in incredibly high variance across runs, even if some runs reach the target rewards threshold.

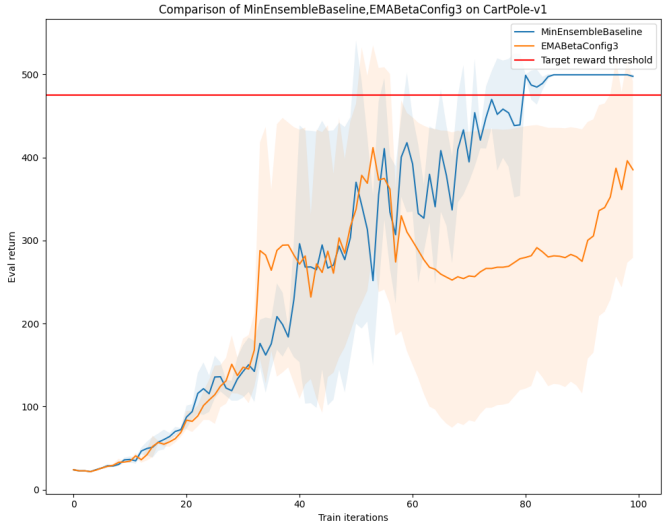


Figure 9: The baseline model and the developed simple transformer model

# 4 Conclusion

We develop and propose a new model that is more resistant to variance, called the Simple Transformer and demonstrate it's efficacy on two environments, an easy CartPole environment and a hard HalfCheetah environment. We perform analysis over various parts of the policy gradient RL algorithm and identify pieces that contribute to high variance. We demonstrate experimentally that these pieces do contribute to the variance. We perform analysis over parts of the AdamW stochastic optimization algorithm, identify components that contribute bias or variance, and experimentally determine their effect by ablating over multiple values.

# References


[Sut+99] Richard S Sutton et al. "Policy gradient methods for reinforcement learning with function approximation". In: *Advances in neural information processing systems* 12 (1999).

[KB15] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: http://arxiv.org/abs/1412.6980.

[BKH16] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. *Layer Normalization*. 2016. DOI: 10.48550/ARXIV.1607.06450. URL: https://arxiv.org/abs/1607.06450.

[LH17] Ilya Loshchilov and Frank Hutter. "Fixing Weight Decay Regularization in Adam". In: *CoRR* abs/1711.05101 (2017). arXiv: 1711.05101. URL: http://arxiv.org/abs/1711.05101.

[Raf+19] Colin Raffel et al. "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer". In: *CoRR* abs/1910.10683 (2019). arXiv: 1910.10683. URL: http://arxiv.org/abs/1910.10683.

[Bro+20] Tom B. Brown et al. "Language Models are Few-Shot Learners". In: *CoRR* abs/2005.14165 (2020). arXiv: 2005.14165. URL: https://arxiv.org/abs/2005.14165.