ChatGPT

Integrating Muon Optimizer into a Neural Physics Transformer Framework

Background: Muon Optimizer and Neural Physics Transformers

Muon Optimizer: Muon (Momentum **Or**thogonalized by **N**ewton-Schulz) is a recently introduced optimization algorithm designed to address limitations of AdamW in training deep networks [1]. Instead of using separate first and second moment estimates like AdamW, Muon maintains only a single momentum buffer (halving memory usage compared to AdamW's two buffers [2]) and updates weight *matrices* via an approximate orthogonalization of the gradient update [3]. In practice, Muon applies Nesterov momentum and then normalizes the resulting gradient *matrix* using a Newton–Schulz iterative method to enforce an approximate orthonormal update direction [3]. This procedure counteracts a known issue in transformer-based networks: gradient updates for large 2D weight matrices often have very high condition numbers (dominated by a few large singular values) [4]. By orthogonalizing the update, Muon effectively "flattens" these dominant directions, boosting the contribution of rarer, smaller singular directions that might be important for learning [4]. Initial results have been promising – for example, on small benchmarks like MNIST/CIFAR, Muon achieves significantly lower loss and higher accuracy than AdamW in the same number of epochs [5]. At large scale, Muon has been shown to expand the performance-efficiency Pareto frontier, retaining better data efficiency than AdamW (especially at large batch sizes) without adding prohibitive overhead [6].

Neural Physics Transformers (NPT): This refers to transformer-based neural network frameworks developed for physical simulation tasks. Recent works aim to create **universal** architectures that can handle a range of physics simulations (fluid flows, particle dynamics, etc.) under a single framework. For instance, *Universal Physics Transformers (UPTs)* introduced a transformer-based neural operator that works for both Eulerian (grid-based) and Lagrangian (particle-based) systems [7]. These NPT models typically use an Encoder–Processor–Decoder design: e.g. encoding physical fields or particles into a latent representation, using transformer self-attention layers to propagate dynamics in latent space, then decoding back to physical space [8] [9]. This unified approach allows one architecture to simulate many phenomena by learning the underlying physics, rather than being domain-specific [7]. NPT frameworks have achieved state-of-the-art accuracy in long-term rollouts for both fluid simulations and particle systems [10]. However, training such models is challenging – they are often large networks trained on complex, noisy simulation data, which can lead to optimization instabilities. Notably, the UPT authors reported that using standard AdamW sometimes caused **sudden loss spikes** (training divergence) when learning complex flows; they had to switch to a newer optimizer (Lion) and higher precision to stabilize training [11]. This underscores the need for a robust optimization strategy in NPT frameworks. Integrating Muon is appealing in this context, as its design directly targets stability and efficient learning in deep transformers.

Strategy for Integrating Muon into an NPT Framework

1. Hybrid Optimizer Approach (Muon + AdamW): The recommended integration strategy is to use Muon for those parameters that benefit most from its matrix-wise optimization, and use AdamW for the rest. In

practice, this means applying Muon updates to all *large weight matrices* in the Neural Physics Transformer – e.g. the weights of attention projection matrices, feed-forward network layers, and any other dense layer weights that are 2D tensors [12]. These matrices constitute the bulk of the model's learning capacity and are exactly where Muon's orthogonalized updates are most effective. For parameters that are **not** naturally treated as matrices, we fall back to AdamW (or an equivalent first-order method). This includes:

- **Embedding layers or lookup tables:** If the NPT uses embedding vectors (e.g. encoding grid cell types or particle attributes via an embedding), their gradients are often sparse (one-hot inputs). Muon's matrix normalization is ill-suited to such sparse, 1-dimensional updates [13]. It's safer to optimize these with AdamW, which handles sparse gradients gracefully.
- **Bias terms and Layer Norm gains:** 1D parameters like biases or scale factors in normalization layers do not have a matrix structure to orthogonalize. They also typically carry much smaller magnitude updates. Keeping these on AdamW avoids any unintended behavior of Muon on essentially vector-shaped data [13].
- **Small matrices or special-case parameters:** Very small weight matrices (e.g. 1×1 convolution kernels or linear layers of tiny dimension) don't meaningfully benefit from Muon's expensive normalization step – their gradients are low-dimensional enough. These can either be left on AdamW or included in Muon without harm. Similarly, any parameter for which the Muon update would be undefined or non-beneficial (such as a scalar coefficient) stays on AdamW.

This **hybrid optimizer** scheme – sometimes dubbed *"MuonW"* – has been used in practice for Muon integration [14]. In fact, recent implementations (e.g. in JAX and PyTorch) allow one to specify parameter groups, making it straightforward: one defines a parameter group for matrices with the Muon optimizer and another group for the rest with AdamW [12]. The training loop then updates all parameters appropriately in one go. This strategy ensures we reap Muon's benefits on the majority of the model (the heavy matrix parameters) while retaining the proven behavior of AdamW on parts that require it. It is a general approach that applies across simulation domains – no matter if your NPT is modeling fluids on a grid or particles in space, it will have large weight matrices internally (from the transformer layers), as well as a few non-matrix parameters; the rule for splitting is the same.

2. Include Weight Decay and Proper Scaling: One critical implementation detail is to preserve **decoupled weight decay** in the Muon updates. In AdamW, weight decay is applied *directly* to the weights (decoupled from the gradient) to avoid distorting the adaptive update [15]. We should do the same with Muon. Recent work found that adding weight decay into Muon's update rule was essential for stability at scale – without it, weights tended to continually grow in norm during training [16]. Thus, the Muon optimizer should be configured to subtract a term λw (where λ is weight decay rate) from the weight at each update, just like AdamW. This maintains generalization benefits of regularization and keeps the model from drifting off due to unchecked weight growth.

Additionally, **per-parameter update scaling** is recommended for Muon in a transformer. Different layers may have different shapes (e.g. a small attention head matrix vs. a very large feed-forward matrix). Muon's orthogonalization normalizes the update *relative to each matrix's own scale*, but we might want to ensure updates across layers have comparable magnitude. The authors of Muon introduced a simple heuristic: scale the orthogonalized update by a factor proportional to the matrix dimension (for example, by $ \sqrt{\text{num_parameters}}$ of that matrix) [17]. The intuition is that larger matrices (with more parameters) can take a smaller step (since their aggregate gradient norm tends to be higher just from having more elements), whereas very small matrices might need a proportionally larger step. In practice, the "best-performing" Muon variant (as identified in follow-up research) incorporated a fixed scaling

constant to match AdamW's update norms. Liu *et al.* (2025) note that multiplying Muon's update by ~0.2 made its root-mean-square update size roughly equal to AdamW's update size [18]. This clever calibration means one can plug Muon in **without retuning the learning rate** – the updates have the same scale as AdamW's on average. We should adopt these improvements: include weight decay in Muon updates and use the recommended scaling adjustments (either via formula or the constant factor from literature) [18]. These tweaks allow Muon to work "out-of-the-box" for large models and were crucial in demonstrating Muon's success on big tasks [19].

3. Minimal Changes to Training Pipeline: Other than splitting the optimizer and adjusting those update rules, integrating Muon doesn't require major changes to the NPT training pipeline. You would initialize the Muon optimizer for the designated parameters with Nesterov momentum (MuON inherently uses Nesterov momentum – this typically yields more stable and slightly faster-converging updates than classic momentum [20]). A typical momentum coefficient is $\beta=0.9$, though some implementations might use $\beta\approx0.95$ for very large models – one can experiment, but in many reports $\beta=0.9$ worked well by default. Learning rate scheduling can remain the same (more on this below). The cost of Muon's Newton-Schulz orthogonalization should be noted: it performs a series of matrix multiplications per update. However, for modern GPUs/TPUs this overhead is relatively small – it scales with the square of matrix dimension, but with large batch sizes the overhead becomes negligible (on the order of <<1% of total training FLOPs in large transformer runs) [21]. For example, one analysis showed Muon added only ~0.5–0.7% FLOP overhead in typical transformer language model training when using 5 Newton-Schulz iterations [21]. In an NPT setting, if the batch size or model size is smaller, overhead might be a few percent higher, but it's generally a good trade-off for the gains in convergence. It's also worth noting Muon's **memory footprint** is lighter than AdamW – by storing one momentum instead of two moments, it cuts optimizer state memory by 50% [2]. This can be important in physics simulations where high-resolution data already pressures GPU memory. In summary, the integration involves configuring a hybrid optimizer (Muon + AdamW), ensuring Muon uses weight decay and scaled updates, and then proceeding with training as usual. The approach is general: it does not depend on domain-specific tricks, so it can be applied consistently whether the NPT is learning fluid dynamics, particle collisions, or any other physical process.

Performance Comparison: Muon vs AdamW

When replacing AdamW with Muon in a Neural Physics Transformer, we expect improvements across several dimensions. Below we compare their **expected** and (where available from research) **observed** performance in terms of training speed, final accuracy, generalization, and training stability.

Training Speed (Convergence Rate and Throughput)

Per-Iteration Cost: Muon's main cost is the Newton-Schulz iterations to orthogonalize gradients. In a naive sense, this makes each training step a bit slower than with AdamW (which only does elementwise operations). However, in practice this overhead is very small for sufficiently large models or batches. The FLOP overhead of Muon vanishes as batch size grows: it scales roughly as O($T \cdot m^2$) per weight matrix (with T the fixed number of NS iterations, e.g. 5, and *m* the matrix dimension) [22]. For a large batch of B training examples, the cost of the forward/backward pass is O($m \cdot B$), which for big B dominates the O(m^2) factor. Concretely, with modern transformer settings, Muon adds on the order of <1% overhead as noted earlier [21]. Thus, **throughput** (steps per second) with Muon is almost the same as with AdamW for a well-parallelized training run. In smaller-scale cases (very small batch or model), one might see a modest slowdown, but it's often insignificant compared to overall training time. It's also

parallelizable – the orthogonalization of each layer's gradients can be done in parallel across layers. In summary, Muon does *slightly* increase per-step computation, but not enough to outweigh its convergence gains.

Convergence in Fewer Steps: Muon typically **converges faster** in terms of number of iterations or epochs required to reach a given level of error. Because it makes more effective use of each batch (more *data-efficient* updates), it can achieve lower loss with the same number of samples. Empirical evidence shows that at any fixed number of training steps, Muon tends to attain a lower training loss than AdamW [23]. This holds true in both small examples and large-scale studies. For instance, in an experiment on language model pretraining, switching to Muon allowed the model to reach the target loss with only ~52% of the training FLOPs that AdamW would need [6]. In other words, Muon can effectively double the "steps per data" yield. In the context of NPT tasks, which can be very data-intensive (requiring many simulation rollouts), this is a huge boon: it implies needing fewer epochs over the dataset to reach a desired accuracy. Faster convergence also means we can potentially shorten the overall training schedule or, alternatively, achieve a better model within the same training duration.

Batch Size Scaling: One of Muon's standout advantages is its ability to maintain performance at **large batch sizes**. With AdamW (and Adam), there is often a *critical batch size* beyond which increasing the batch yields diminishing returns or even degrades convergence (the model becomes less data-efficient per step). Muon has been shown to push this limit much further – it retains strong data efficiency even far beyond the typical critical batch size [6]. This means we can exploit hardware by using very large batches (tens or hundreds of thousands of samples, if available) to speed up training wall-clock time, without suffering the loss curve slow-down that AdamW would exhibit at those scales. In large-scale tests, Muon's iso-loss curves (batches vs time) were consistently below AdamW's, indicating strictly better trade-offs between using more compute and reducing training time [24]. For an NPT framework, which might run on HPC clusters or GPUs with the ability to handle big batches of simulation data, Muon allows near-linear scaling of training speed with batch size. In practical terms, if your AdamW training was limited to batch size 512 before the model's generalization started dropping, you might find you can go to batch 2048 or more with Muon and still converge to the same or better accuracy. This large-batch stability lets you utilize parallel resources more effectively, cutting down time-to-train significantly.

Overall Training Time: Combining the above points – slightly higher per-step cost, but significantly fewer steps needed – Muon can substantially reduce the total training time required to achieve a given accuracy. In many cases, the time saved by converging faster far outweighs the mild per-step overhead. For example, if Muon converges in 60% of the epochs that AdamW needed, and introduces, say, a 5% step overhead, you still come out far ahead (~40% less wall-clock time to target). Even in scenarios where final convergence is similar, Muon tends to reach high accuracy earlier in training, which is beneficial for iterative research (one can get to a good model quicker for analysis or hyper-parameter tuning). Summarizing training speed: **Expected outcome** is that NPT models with Muon train faster to a given error level, especially when leveraging larger batches, compared to the AdamW baseline.

Final Accuracy and Generalization

Final Accuracy: When training to completion (i.e. until the model has essentially converged or the improvements plateau), Muon has shown equal or sometimes better final accuracy than AdamW. There isn't a penalty for using Muon – in theory, both are first-order optimizers that should be able to reach similar minima, but in practice Muon's ability to navigate the loss landscape can land it in a *better* minimum. By not

over-emphasizing the largest gradient directions, Muon may avoid some of the narrow, sharp minima that Adam-type optimizers can fall into. Instead, it tends to find solutions where the weight updates were balanced across directions, which often correspond to flatter minima that generalize well. As a concrete example, a Muon-trained classifier on CIFAR-10 reached about **80.8%** accuracy after 5 epochs, whereas AdamW reached only **71.7%** in that time [5]. Given more epochs, AdamW would improve, but Muon had essentially attained a higher-performing solution much faster. In large-scale runs, researchers observed that at convergence, models trained with Muon were at least as good as their AdamW counterparts, and in some cases slightly ahead in quality [25]. In the context of physics simulation, "accuracy" can mean low prediction error on held-out physical trajectories or hitting certain error thresholds (like below a certain MSE). Using Muon should either match the AdamW-trained model's error or surpass it, particularly in regimes where optimization is the bottleneck. If the NPT model capacity is high enough, both optimizers can eventually fit the training data well; Muon's advantage is more pronounced when the problem is hard to optimize (which is often true for complex physics) – it might find better parameter values that yield lower error on the test set.

Generalization: Generalization for physics models can be a bit nuanced. It often involves testing the model on scenarios not seen in training (e.g., a different fluid speed, a different number of objects, a longer time rollout than trained on). A key observation is that techniques which improve **data efficiency** and provide implicit regularization usually help generalization. Muon incorporates a couple of aspects here: decoupled weight decay (explicit regularization) and normalized updates which act somewhat like *gradient regularization*. By constraining the update to a fixed norm ball in the space of singular values (Schatten norm constraint) [4] [26], Muon performs a form of controlled update step that may avoid overfitting as the model doesn't chase very specific directions too aggressively. Empirically, Muon has been noted to **generalize better with the same data** – for instance, reaching a given loss with fewer data implies the model is extracting more generalizable patterns rather than memorizing noise [6]. In small-scale experiments, Muon's higher accuracy after a fixed number of epochs indicates better generalization to the test set even early in training [5].

For NPTs, generalization is crucial: we want the learned physics model to apply to situations beyond the training distribution. If Muon allows the model to capture the underlying physical laws more efficiently, it could result in stronger generalization. One might expect, for example, an NPT trained with Muon to handle a slightly higher Reynolds number flow or a different particle count better than one trained with AdamW, given the same training data. That said, generalization ultimately also depends on the model architecture and training data diversity. Muon doesn't magically impart physical knowledge, but it can **improve the effective capacity of the model on the training data**, which often yields a model that performs better on unseen data as well (since it found a more "optimal" fit). The inclusion of weight decay in Muon is specifically to improve generalization by controlling complexity [15] – this mirrors AdamW's generalization benefits.

In sum, we expect **comparable or improved generalization** from Muon compared to AdamW. There is no evidence of any generalization drop; on the contrary, in scenarios with limited data, Muon's ability to do more with less data can lead to a model that generalizes from fewer examples (important in scientific contexts where each training sample may come from an expensive simulation or experiment). To maximize generalization, one should still use best practices like validation-based early stopping or learning rate decay – Muon will simply get you to a good generalization regime faster.

Training Stability

Stability of Optimization: Training stability refers to avoiding divergences, wild oscillations in loss, or other training pathologies. Here, Muon has a clear advantage by design. Its orthogonalized gradient updates impose a form of *trust region* on each step – effectively limiting the update to a norm-constrained space (specifically, it ensures the update matrix cannot have arbitrarily large singular values) [4]. This tends to prevent situations where a few gradients components explode and cause a huge weight update. As a result, Muon can safely operate at learning rates or batch sizes that might cause AdamW to become unstable. The literature provides a striking example: in scaling up Muon for very large language models, the authors were able to use Muon with the **same hyperparameters** (LR, etc.) as a tuned AdamW run, and it remained stable, whereas using those hyperparameters directly with AdamW might have required adjustment [18]. In our NPT context, which often involves chaotic systems, this extra robustness is valuable. We already saw the UPT example where AdamW led to sudden loss spikes (the training would blow up unpredictably) and the researchers had to try a different optimizer to cure it [11]. Muon's more conservative and well-conditioned updates should markedly reduce the chance of such events. It provides a smoother training trajectory – users have reported that loss curves under Muon are more monotonic and devoid of the jitter that sometimes appears with AdamW (especially early in training or when using large learning rates).

Large-Batch and Long-Horizon Stability: Muon's stability is especially evident in large-batch training. AdamW's issues beyond the critical batch size can be seen as a form of instability – the optimizer becomes unable to decrease loss effectively per step, sometimes leading to plateau or erratic behavior. Muon, by retaining efficacy at those batch sizes [6], essentially remains stable and efficient where AdamW starts to falter. Moreover, Muon's momentum is Nesterov, meaning it looks ahead by applying momentum before gradient – this generally helps dampen oscillations and can lead to more stable plateaus. For extremely long training runs (which are common in physics simulations, where you might train for many epochs until a very low error), having an optimizer that doesn't drift or require re-tuning mid-way is a big plus. Muon has demonstrated stable convergence even over trillions of token trainings in language tasks [27], suggesting it can handle long schedules without issues.

In some cases, Muon's stability can allow *simplifications* in training. For example, one might reduce reliance on techniques like gradient clipping or adaptive learning rate reduction. If previously you clipped gradients to avoid rare spikes with AdamW, you might find that under Muon those spikes don't occur as often or at all (because Muon inherently attenuates spikes by normalizing the update matrix). That said, one should still monitor training as usual – no optimizer is immune to all problems (e.g., a bug in data or an extremely bad initialization could still cause issues). But generally, we anticipate **fewer training interruptions** (no sudden divergence at high learning rate, no need to restart from earlier checkpoints due to optimizer-induced instability, etc.) when using Muon.

Stability vs. Model Architecture: It's worth noting that Muon complements other stability measures in the NPT architecture. For example, transformers often use *Pre-Norm* (layer normalization before attention and MLP blocks) to stabilize gradient flow. NPT implementations like UPT followed such practices [28], and even introduced special attention normalization (e.g. QK-normalization) to avoid unstable attention scores [29]. These architectural features handle forward-pass stability, whereas Muon handles backward-pass stability. Together, they significantly harden the training process. An anecdotal comparison: UPT initially using AdamW needed a switch to Lion and full precision to overcome instability in one of their benchmarks [11]. If Muon had been available, it's plausible it could have provided the needed stability without changing precision, given that Lion itself is another optimizer designed for more stable convergence. In summary,

Muon greatly improves training stability for NPTs, allowing one to train with confidence at higher learning rates or larger batches, and reducing the likelihood of training failures (divergence). This stable behavior is crucial when training expensive physics models where a blown-up run means a lot of wasted compute – Muon helps mitigate that risk.

Best Practices: Hyperparameters, Scheduling, and Architecture Adjustments

Successfully integrating Muon into an NPT framework also involves tuning certain hyperparameters and possibly making minor adjustments to fully leverage its strengths. Here we outline best practices:

Hyperparameter Tuning: One convenient aspect of Muon is that it was designed to be used with hyperparameters similar to AdamW's with minimal tuning [30]. Thanks to the update normalization (scaling by a constant), the *same* learning rate and weight decay that worked for AdamW are a great starting point for Muon [18]. In many reported cases, researchers kept the learning rate identical when switching to Muon and achieved excellent results without a hyperparameter sweep. We recommend beginning with the known-good hyperparameters from the AdamW-trained NPT (if such exist). For example, if your NPT was typically trained with AdamW at a peak learning rate of 5e-4 and weight decay of 1e-2, use those values for Muon (ensuring Muon's internal scaling is set appropriately). The momentum term for Muon (if exposed) can be left at 0.9 or 0.95; Muon's authors used 0.8–0.9 in some smaller experiments and around 0.95 for very large models, but this generally doesn't require extensive tuning [31].

One thing to double-check is that weight decay is being applied *decoupledly*. Most Muon implementations have followed AdamW's approach (decouple WD from gradients), but it's good to verify because applying weight decay incorrectly (e.g., inside the momentum) could negate some of Muon's stability. Assuming that is correct, the weight decay hyperparameter value itself usually remains the same as you'd use in AdamW – its effect on regularization is unchanged in principle.

If you are attempting to scale your NPT model up significantly (say from millions to tens of millions of parameters, or to a higher-resolution simulation) using Muon, you might consider techniques like **Maximal Update Parameterization (muP)** to transfer hyperparameters [32] [33]. Recent research extended muP (a method to keep optimal hyperparams constant as model width increases) to work with Muon, meaning you can tune on a smaller model and reliably reuse those settings on a larger model [34]. This is an advanced strategy, but it can save a lot of compute in hyperparam searches if you plan to iterate on model sizes. In general, though, for a given model size, Muon does not add much burden in hyperparameter tuning – focus on the same key ones: learning rate and weight decay (and perhaps batch size, as you may want to increase it). The rest (dropout rates, etc.) would remain as per the original model training regime.

Learning Rate Schedule: Use whatever schedule proved effective for training the NPT with AdamW. Commonly, a **warm-up** phase followed by a **cosine decay** or linear decay schedule is employed in physics-informed training. For instance, UPT was trained with a certain number of epochs of warm-up and then cosine annealing [35] [36]. Muon is fully compatible with these schedules. Because Muon often leads to faster initial convergence, one might wonder if the warm-up needs to be shorter – in practice, it's usually fine to keep it the same. The warm-up is more about avoiding instability at the very start (before momentum accumulates); Muon is quite stable even early, but there's no harm in a short warm-up to be safe. The decay schedule (cosine to zero, or to some fraction of initial LR) can also remain. Muon's own

authors used identical schedules (e.g. **linear warmup, cosine decay to 10%** of initial LR) when comparing to AdamW [36]. They explicitly report that the same schedule "works well for both Muon and AdamW" [18]. This implies you do *not* need exotic learning rate tuning for Muon – a big plus for usability.

One small tip: Since Muon may reach low loss faster, if you have a schedule tied to epochs, you might end up effectively training longer than needed. You could monitor validation loss and consider early stopping or more aggressive decay if you notice the model has converged earlier under Muon. But this is optional and depends on how your training pipeline is set up. If using a fixed epoch schedule, Muon will simply end up with a somewhat better model by epoch N compared to AdamW, which is fine.

Batch Size and Gradient Accumulation: As discussed, you can try to **increase the batch size** to leverage Muon's large-batch efficiency. This could involve less gradient accumulation or using more GPUs, etc. If doubling the batch size, one common heuristic is to double the learning rate (linear scaling rule) *if* the optimizer were ideal. However, Muon handles large batches so well that you might not need to change LR at all – it won't degrade like AdamW might. Some practitioners still adjust LR slightly when going very large on batch (to stay in the optimal regime of the compute-time tradeoff), but there's evidence that Muon achieves nearly the same loss even if you don't lower the LR at huge batch sizes [24]. It's a good idea to monitor loss when changing batch size. If you see no dip in performance going from batch 256 to 1024, you can likely keep pushing it up until hardware limits. With NPT training, where one often uses distributed computing for big simulations, this means Muon can help you get almost linear speed-up from using more GPUs – because it preserves the ability to learn with those larger batches. Just be aware of memory: Muon's memory savings helps here, but larger batch obviously uses more memory for activations. If memory is an issue, consider gradient accumulation (which effectively simulates a large batch in multiple smaller forward passes). Muon will accumulate momentum over those just fine.

Gradient Clipping: Many physics-model training setups include gradient clipping (either by global norm or by value) to prevent catastrophic explosions from rare large gradients. With Muon, you might find that gradient clipping triggers less frequently. The orthogonalization step inherently **caps the spectral norm** of the gradient update [4] – this is somewhat analogous to clipping the largest eigenvalues of the gradient matrix. So the worst offenders of unstable gradients are already taken care of in a more granular way. It's still recommended to keep gradient clipping in place initially (e.g., clip norm at 1 or 5, as you did with AdamW) as a safeguard, but don't be surprised if your training log never actually reports any clipping occurred. If you are trying to push performance, you could experiment with raising the clip threshold or removing it under Muon, potentially allowing the optimizer a bit more freedom. The general experience, however, is that Muon is quite stable without heavy clipping – some large-scale training runs with Muon did not use any clipping and encountered no issues, thanks to the structured normalization of updates.

Architecture Considerations: The NPT model architecture itself usually does not need alteration for Muon, but a few points are noteworthy:

- **Normalization Layers:** Continue to use Layer Normalization (or other normalization) as in the original architecture. Muon doesn't replace the need for those. In fact, the combination of normalized inputs/activations (via LayerNorm, etc.) and normalized gradient updates (via Muon) creates a well-balanced training. For example, UPT used a pre-norm transformer architecture (layernorm applied before each sub-layer) [28]; this should be retained. There's no conflict between LayerNorm and Muon – one operates in forward pass, one in backward pass. If anything, having stable activations complements Muon's stable weight updates.

- **Attention Stability:** If your NPT uses attention mechanisms (likely it does), be mindful of any known instability issues there. Researchers have introduced techniques like QK-normalization (normalize the query and key vectors to prevent extremely large attention scores) to improve stability in transformers [29]. Using such a technique in the NPT can further ensure that Muon isn't fighting against exploding gradients from attention. In the Gemma transformer architecture (used in some large-scale experiments), QK-norm was applied and made attention more stable [29]. Muon will then have well-behaved gradients to work with. In summary: leverage known best practices in transformer architecture (pre-norm, attention scaling, etc.) alongside Muon for best results. These are standard in most modern transformer frameworks anyway.

- **Parameter Initialization:** There's no special initialization needed for Muon beyond what the model normally uses. A good initialization (e.g., Xavier/Glorot or similar) is assumed. If anything, Muon's robustness might make the training less sensitive to initialization, but that hasn't been explicitly documented – it's just a reasonable guess since Muon prevents runaway growth initially.

- **Mixing Parameter Types:** As covered, some parameters are on AdamW. This effectively means you are training with two optimizers simultaneously. To avoid confusion, ensure you segregate them properly: e.g., all embeddings and bias parameters assigned to the AdamW instance, all others to Muon. It can help to use clear naming conventions or parameter lists in code to prevent any parameter from accidentally being optimized by both or the wrong one. In terms of learning rate, it's simplest to use the same LR for both optimizers (and the same schedule) – this was done in the Muon+Adam hybrid approach successfully [18]. You could, in theory, tune a separate LR for the AdamW subset if you find (for example) that embedding vectors need a higher or lower LR, but typically this isn't necessary. The idea is that by matching Muon's update magnitude to AdamW's, both groups of parameters can use the same learning rate schedule harmoniously [18].

- **Distributed Training:** If you train NPT across multiple GPUs or nodes, note that Muon's operations (matrix orthogonalizations) need to be done in parallel across devices for distributed weight shards. The Kimi Team that scaled Muon to LLMs implemented a distributed version with a ZeRO-style sharding to maintain efficiency [37]. In our use case, if using PyTorch with DDP or FSDP, one should use a Muon implementation that is compatible (there are open-source implementations emerging that support sharded optimizers). The Hugging Face example offers an FSDP-compatible Muon variant [38]. This ensures that using Muon doesn't introduce undue communication overhead. In fact, since Muon only keeps one buffer, it slightly **reduces** optimizer state communication vs AdamW (which would communicate two buffers in a sharded setup). So distributed training with Muon is not only feasible, it's arguably more communication-efficient than AdamW.

- **Task-Specific Tuning:** While Muon is general, every physical task has its quirks. After integrating Muon, observe how the NPT performs on validation scenarios that test generalization (e.g., a different simulation than training). If you notice any overfitting or underfitting, treat it as you normally would: adjust weight decay, or adjust the training duration. Muon might change the point at which the model starts overfitting (possibly it will overfit *later* since it generalizes better initially). So, for example, you might need to train a few more epochs to fully fit the training data if Muon was being more cautious early on – but usually Muon reaches lower training loss faster, so the opposite is more likely (you might want to stop earlier than you would with AdamW because the model is already as good as it can get).

In summary, applying Muon in an NPT does not require a fundamentally new training recipe – it's largely a drop-in replacement with some parameter grouping. The main points are to use the same learning rate schedule (thanks to Muon's calibrated updates) [18], maintain weight decay, and exploit Muon's strengths by possibly increasing batch size. Continue using all the architectural and training best practices you normally would for physics models (normalization, etc.), as these work in tandem with Muon to produce a stable and efficient training process.

Summary of Recommendations

- **Adopt a Hybrid Muon+AdamW Optimizer:** Utilize Muon for all large **matrix** parameters (transformer weight matrices in the NPT model) and use AdamW for non-matrix parameters (e.g. embedding vectors, bias terms, normalization scales). This ensures Muon is applied where it's most effective while preserving AdamW's behavior for parameters that don't suit Muon [12] [13]. This "MuonW" strategy has been shown to be robust in practice.

- **Keep Hyperparameters Consistent with AdamW:** Start with the same learning rate, momentum, and weight decay values that you would use for AdamW on the given task. Thanks to Muon's update normalization, the *same* learning rate schedule can be used without modification [18]. For example, if a 10^{-3} LR and 0.01 weight decay worked for AdamW, use those for Muon as well. Muon's momentum (Nesterov) can be set to the typical 0.9 (or 0.95 for very deep models); no special tuning is usually needed [30].

- **Use Decoupled Weight Decay in Muon:** Ensure that weight decay is implemented in a decoupled manner (similar to AdamW) when using Muon. This means applying a direct $-\lambda w$ update to weights each step. Including weight decay is crucial for Muon's stability and generalization – without it, models may exhibit growing weight norms [16]. The same weight decay coefficient from AdamW (which aids generalization [15]) should be retained for Muon, as it was identified as a key to scaling Muon successfully [19].

- **Incorporate Muon's Recommended Scaling Tricks:** Use the improved version of Muon that scales updates appropriately for different layer sizes. In practice, this involves multiplying the orthogonalized gradient by a constant factor (about 0.2 as per Liu *et al.* 2025) or by a function of matrix size, so that Muon's update RMS matches that of AdamW [18] [17]. Most recent Muon implementations include this by default. This tweak allows you to seamlessly swap AdamW with Muon without losing training balance, and it avoids having to retune the learning rate.

- **Expect Faster Convergence & Adjust Training Schedule:** With Muon, your NPT model will likely reach a given error level in fewer iterations than with AdamW [23]. Plan for shorter training or higher final performance. For example, you might achieve the desired validation error after 70% of the epochs originally budgeted. Be prepared to either stop early (saving compute) or to attain better accuracy by training to the full schedule. Muon's data efficiency can be especially beneficial if training data is limited or expensive to generate [6].

- **Leverage Larger Batch Sizes:** One clear advantage of Muon is its ability to handle very large batch sizes without loss of effectiveness [6]. If resources permit, increase the batch size to speed up training. You can often maintain the same learning rate when scaling up batch size under Muon (it doesn't suffer from the same critical batch size limitations as AdamW). Larger batches combined

with Muon will yield near-linear speedups in wall-clock time **and** maintain model quality, which is ideal for heavy simulation workloads.

- **Monitor Training Stability (it Should Improve):** You will likely observe a smoother, more stable training curve. Divergences or instabilities that might occur with AdamW (e.g. sudden loss spikes) are expected to diminish with Muon [11]. This means less need for manual intervention. However, continue to monitor metrics like loss and gradient norms during the initial phases of using Muon. In the unlikely event you do see instability, double-check that all non-matrix params were excluded from Muon and that weight decay is correctly applied. In most cases, Muon will *reduce* the need for hacks like gradient clipping or loss scaling, simplifying the training process.

- **Maintain Normal Training Protocols:** No architecture changes are required for Muon, so keep all the effective ingredients of your NPT framework (such as layer normalization, dropout, etc.) as they are. Muon will work in concert with these. If anything, combining Muon with a well-normalized transformer (pre-norm) yields an extremely robust training setup. Continue using the same validation and testing regime to evaluate generalization – you should notice equal or better generalization performance with the Muon-trained model, owing to its more regularized updates [15].

- **Utilize Open-Source Implementations:** Finally, take advantage of the fact that Muon is an emerging optimizer – there are resources available. The Kimi Team has open-sourced a distributed Muon implementation optimized for memory and speed [39], and community projects (e.g. Hugging Face notebooks) provide reference implementations of hybrid Muon+AdamW optimizers [14]. These can be integrated into your training code to ensure correctness. Using a vetted implementation helps avoid pitfalls in the complex orthogonalization step.

By following this integration strategy, you make your Neural Physics Transformer training more efficient and robust. Muon, as the optimizer, will accelerate training (fewer epochs to converge), likely improve the final accuracy and generalization of the model, and provide greater stability especially in large-scale or long-running simulations. The approach is general and has been informed by the latest research on both Muon and physics-informed transformers – applying these best practices will help you get the most out of Muon in any physical simulation domain [18] [6]. The end result should be a faster trained, high-accuracy NPT model that generalizes well across a range of physical scenarios, all achieved with a more streamlined training process.

References: Recent works and findings that underpin these recommendations include Jordan et al. (2024) on the Muon optimizer's design [4] [3], Liu et al. (2025) on scaling Muon to large models with weight decay and normalization tweaks [19] [18], as well as Alkin et al. (2024) on Universal Physics Transformers [7] and the observed training challenges with AdamW in that context [11]. These sources collectively show that Muon can serve as a drop-in enhancement over AdamW for transformer-based physics simulators, yielding improved efficiency and stability without sacrificing generality or requiring domain-specific adjustments. With this integration strategy, one can confidently train Neural Physics Transformers across diverse tasks, harnessing Muon's state-of-the-art optimization performance to push the frontiers of physics simulation accuracy and speed.

---

1 5 14 38 [Tutorial] Understanding and Implementing the Muon Optimizer - Show and Tell - Hugging Face Forums
https://discuss.huggingface.co/t/tutorial-understanding-and-implementing-the-muon-optimizer/167717

2 3 4 13 15 16 17 20 From AdamW to Muon Optimizer. In this article, we will discuss the... | by Astarag Mohapatra | Medium
https://athekunal.medium.com/from-adamw-to-muon-optimizer-cf67d43fb9e9

6 12 18 23 24 29 30 31 32 33 34 36 Practical Efficiency of Muon for Pretraining
https://arxiv.org/html/2505.02222v1

7 [2402.12365] Universal Physics Transformers: A Framework For Efficiently Scaling Neural Operators
https://arxiv.org/abs/2402.12365

8 9 11 28 35 [2402.12365] Universal Physics Transformers
https://ar5iv.labs.arxiv.org/html/2402.12365

10 Learning Physical Simulation with Message Passing Transformer
https://arxiv.org/html/2406.06060v1

19 25 26 27 37 39 [2502.16982] Muon is Scalable for LLM Training
https://ar5iv.labs.arxiv.org/html/2502.16982v1

21 22 Muon: An optimizer for hidden layers in neural networks | Keller Jordan blog
https://kellerjordan.github.io/posts/muon/