Transformer-Based Learned Optimization

Erik Gärtner\textsuperscript{1,2*}Luke Metz\textsuperscript{1}Mykhaylo Andriluka\textsuperscript{1}
C. Daniel Freeman\textsuperscript{1}Cristian Sminchisescu\textsuperscript{1}

\textsuperscript{1}Google Research \textsuperscript{2}Lund University

{em erik.gartner@math.lth.se}
{em \{lmetz,mykhayloa,cdfreeman,sminchisescu\}@google.com}

Abstract

We propose a new approach to learned optimization where we represent the computation of an optimizer’s update step using a neural network. The parameters of the optimizer are then learned by training on a set of optimization tasks with the objective to perform minimization efficiently. Our innovation is a new neural network architecture, Optimus, for the learned optimizer inspired by the classic BFGS algorithm. As in BFGS, we estimate a preconditioning matrix as a sum of rank-one updates but use a Transformer-based neural network to predict these updates jointly with the step length and direction. In contrast to several recent learned optimization-based approaches [24, 27], our formulation allows for conditioning across the dimensions of the parameter space of the target problem while remaining applicable to optimization tasks of variable dimensionality without retraining. We demonstrate the advantages of our approach on a benchmark composed of objective functions traditionally used for the evaluation of optimization algorithms, as well as on the real world-task of physics-based visual reconstruction of articulated 3d human motion.

1. Introduction

This work focuses on a new learning-based optimization methodology. Our approach belongs to the category of learned optimization methods, which represent the update step of an optimizer by means of an expressive function such as a multi-layer perceptron. We then learn the parameters of this function on a set of training optimization tasks. Since the update function of the learned optimizers is estimated from data, it can in principle learn various desirable behaviors such as learning-rate schedules [22] or strategies for the exploration of multiple local minima [23]. This is in contrast to traditional optimizers such as Adam [15], or BFGS [11] in which updates are derived in terms of first-principles. However, as these are general and hard-coded, they may not be able to take advantage of the regularities in the loss functions for specific classes of problems.

Learned optimizers are particularly appealing for applications that require repeatedly solving related optimization tasks. For example, 3d human pose estimation is often formulated as the minimization of a particular loss function [12, 19, 30, 46]. Such approaches estimate the 3d state (e.g. pose and shape) given image observations by repeatedly optimizing the same objective function for many closely related problems, including losses and state contexts. Traditional optimization treats each problem as independent, which is potentially suboptimal as it does not aggregate experience across multiple related optimization runs.

The main contribution of this paper is a novel neural network architecture for learned optimization. Our architecture is inspired by classical BFGS approaches that iteratively estimate the Hessian matrix to precondition the gradient. Similarly to BFGS, our approach iteratively updates the preconditioner using rank-one updates. In contrast to BFGS, we use a transformer-based [40] neural network to generate such updates from features encoding an optimization trajectory. We train the architecture using Persistent Evolution Strategies (PES) introduced in [41]. In contrast to prior work [4, 24, 27], which rely on updates over each target parameter independently (or coupled only via normalization), our approach allows for more complex inter-dimensional dependencies via self-attention while still showing good generalization to different target problem sizes than those used in training. We refer to our learned optimization approach as Optimus in the sequel.

We evaluate Optimus on classical optimization objectives used to benchmark optimization methods in the literature [17, 31, 37] (cf. fig. 1) as well as on a real-world task of physics-based human pose reconstruction. In our experiments, we typically observe that Optimus is able to reach a lower objective value compared to popular off-the-shelf
optimizers while taking fewer iterations to converge. For example, we observe at least a 10x reduction in the number of update steps for half of the classical optimization problems (see fig. 4). To evaluate Optimus in the context of physics-based human motion reconstruction, we apply it in conjunction with DiffPhy, which is a differentiable physics-based human model introduced in [12]. We experimentally demonstrate that Optimus generalizes well across diverse human motions (e.g. from training on walking to testing on dancing), is notably (5x) faster to meta-train compared to prior work [24], leads to reconstructions of better quality compared to BFGS, and is faster in minimizing the loss.

2. Related Work

Learned optimization is an active area of research, and we refer the reader to an excellent tutorial [3] and survey [9] for a comprehensive review of the literature. Our approach is generally inspired by [4, 18, 27, 44] and is most closely related to Adafactor MLP [24]. One of the distinguishing properties of Optimus compared to Adafactor MLP is the ability to couple optimization updates along different dimensions. Arguably coupling of dimensions can be added to Adafactor MLP through additional features such as radial features from [23] that capture pairwise interactions between dimensions. In contrast, the advantage of Optimus over such extensions is that dimension coupling is learned from data and is not limited to be pairwise. Other work incorporates conditioning in some aggregated space. For example, both [44] and [26] introduce hierarchical conditioning mechanisms that operate on individual layers and in a global setting. The Optimus approach can be seen as constructing a learned preconditioner to account for the underlying cost function curvature. The use of meta-learned curvature has been explored in [29], though only in the context of few-shot learning strategies. There exist prior work on using learned optimization in the human pose estimation literature [35] as well as approaches that iteratively refine the solution in an optimization-like fashion based on recurrent neural networks e.g. [8, 50]. The work of [35] is perhaps the most similar to ours in that it genuinely employs a learned optimization in the framing described in sec. 3. However, [35] applies it to the simpler problem of monocular 3d pose estimation. On that task their method converges in as few as four iterations, thus making meta-training based on stochastic gradient descent feasible. By design, the approach of [35] does not generalize to optimization instances with variable (thus different) dimensionality with respect to training, as they employ a single MLP that predicts the entire update vector. Moreover, their work has yet to be evaluated for more complex tasks such as those considered in this paper.

3. Overview and Background

In this paper, we leverage a general approach to learned optimization as introduced in [4, 6, 7, 27], which we review in sec. 3.1. Equipped with this background, we then introduce the details of our new Optimus architecture for learned optimization in sec. 4. Then in sec. 5 we present experimental results comparing Optimus to prior work in learned optimization [27] and against standard off-the-shelf optimizers.

3.1. Learning an Optimizer

Learned optimizers are a particular type of meta-learned system which commonly uses a neural network to param-
eterize a gradient-based step calculation that can then be used to optimize some objective function [4]. To demonstrate this class of models, let us consider an optimization problem $\arg\min_\theta L(x)$. Gradient-based optimization algorithms such as gradient descent (GD) aim to solve the problem by iteratively modifying the parameters $x$ using an update function $U$ which takes gradients of $L$ along the optimization trajectory as input $x_{t+1} = x_t - U(\nabla_{\leq t} L(x_{1:t}))$, where $x_t$ are the parameters at step $t$ and $\nabla_{\leq t} L(x_{1:t}) = \{\nabla L(x_1), \ldots, \nabla L(x_t)\}$. For example, in the case of GD, the update function is simply $U_{\text{gd}}(\nabla_{\leq t} L(x_{1:t})) = -\alpha \nabla L(x_t)$, where $\alpha$ is a learning rate hyperparameter. A learned optimizer is a particular type of update function, which itself is parameterized by a set of meta-parameters $\theta$, and with possibly more features (e.g., the current parameter values). Then, as with GD, it can be iteratively applied to improve the loss.

In this paper, we build on learned optimization as proposed in [24, 27, 41]. In that approach, the update function $U = U(x; \theta)$ is parameterized based on a small multilayer perceptron (MLP) with weights $\theta$, which is applied independently to each dimension of a feature vector $z$. For each parameter, the update function takes a vector of features $z$ as input, including gradient information, as well as additional features such as exponential averages of squared past gradients as done in Adam [15] or RMSProp [38], momentum at multiple timescales [20], as well as factored features inspired by Adafactor [32]. We refer to this approach as Adafactor MLP in the sequel. Training an Adafactor MLP optimizer amounts to minimizing a meta-loss with respect to parameters $\theta$ on a meta-training-set of optimization problem instances. The meta-loss is given by $\sum_i L(x_i)$ where the sum runs over the parameter states of the optimization trajectory. Minimizing the meta-loss is often implemented via truncated backpropagation through unrolled optimization trajectories [4, 21, 43, 44]. As discussed in [25, 27, 41] typical meta-loss surfaces are noisy and direct gradient-based optimization is difficult due to exploding gradients. To address this, we minimize the smoothed version of a meta-loss as in [27] using Adam [15] and adopt Persistent Evolution Strategies (PES) [41] to compensate for bias due to truncated back-propagation [45].

4. Our Approach

Our transformer-based learned optimizer, Optimus, is inspired by the BFGS [11] rank-one approximation approach to estimating the inverse Hessian, which is applied as a pre-conditioning matrix in order to obtain the descent direction. The parameter update is the product of a descent direction ($s^k$) produced by a learned optimizer that operates on each parameter independently, and a learned preconditioner ($B^k$) where $B^k$ is an $N \times N$ matrix which supports conditioning over the entire parameter space. We update $B^k$ with $L$ rank-one updates on each iteration. The full update is thus given by

$$\Delta x^k = B^k s^k,$$

where $\Delta x^k$ is the parameter update at iteration $k$. See fig. 2 for an overview.

**Per-Parameter Learned Descent Direction ($s^k$).** Let us denote a feature vector describing the optimization state of the $n$-th parameter at iteration $k$ as $z^k_n$. As in [24] we predict per-parameter updates using a simple MLP that takes the feature vector $z^k_n$ as input and outputs a log learning rate...
MLP \[24\]. While this preconditioner considerably increases the step quality, its computational cost grows quadratically in the number of parameters.

**Stopping Criterion.** During meta-training, we unroll the optimizer for 50 steps, but at test time, we run Optimus using a stopping criterion based on relative function value decrease, as in classical optimization. We terminate the search if \(f(x^k) > \frac{1}{\lambda} \sum_{i=1}^{N} \beta f(x^{k-i}) + \epsilon\), i.e. if the function value at step \(k\) is greater than the average function value in the previous \(N\) steps. We do not apply this criterion for the first \(N\) steps. We set \(N = 5\) in our experiments.

### 5. Experiments

We evaluate our learned optimization methodology on two tasks. We first present results of a benchmark composed of objective functions typically used for evaluation of optimization methods, and then present results for articulated 3d human motion reconstruction in sec. 5.2.

**Baselines.** We compare the performance of our Optimus optimizer to standard optimization algorithms BFGS \[11\], Adam \[15\], and gradient descent with momentum (GD-M). We independently tune the learning rate of Adam and GD-M for each optimization task given by objective function and input dimensionality using grid-search. To that end we test 100 candidate learning rates between \(10^{-6}\) and 1 and choose the learning rate that results in lowest average objective value after running optimization for 64 random initializations. Finally, we also compare our approach to the state-of-the-art learned optimizer Adafactor MLP \[24, 27\] using the publicly available implementation\(^1\).

#### 5.1. Standard evaluation functions

**Evaluation benchmark.** We define a benchmark composed of 15 objective functions frequently used for evaluation of optimization algorithms. We show a few examples of such functions in fig. 1 and provide a full list in the supplementary material. To define the benchmark we use the catalog of objective functions available at \[37\], focusing on the functions that can be instantiated for any dimensionality of the input. We include both seemingly easy to optimize functions (e.g. “Sphere” function\(^2\)) as well as more challenging functions with multiple local minima (e.g. Ackley function \[11\]) or difficult to find global minima located at the bottom of an elongated valley (e.g. Rosenbrock function \[31\]). We use versions of these functions with input dimensionality of \(2 - 100\) for training, and then evaluate on the dimensions 250, 500 and 1000 to show that optimizer can generalize to different dimensionality of the input. This gives us a test set of 45 objective functions to evaluate on.

To prevent the learned optimizer from memorizing the

\[\alpha_n^k\] and update direction \(d_n^k\) that are combined into a per-parameter update as

\[s_n^k = \lambda_a \exp(\lambda_b \alpha_n^k) d_n^k,\]  

where \(\lambda_a = 0.1\), and \(\lambda_b = 0.1\) are hyperparameters which are constant throughout meta-training. Note that at that stage we **independently** predict the update direction and magnitude for each dimension of the vector \(x\). In particular, the MLP weights are shared across all the dimensions of \(x\). We use a small 4-layer MLP with 128 units per layer at that stage and did not observe improvement when with larger models (see tab. 4). We use the same features \(z_n^k\) as \[24\] and similarly to \[24\] normalize the features to have a second moment of 1. We include the feature list in the supplementary material for completeness.

**Learned Preconditioning (\(B^k\)).** Next, we introduce a mechanism to couple the optimization process of each dimension of \(x\) and enable the optimization algorithm to store information across iterations. Intuitively such coupling should lead to improved optimization trajectories by capturing curvature information, similar to how second-order and quasi-Newton methods improve over first-order methods such as gradient descent. We define these updates as a low-rank update followed by normalization:

\[B^{k+1} = B^k + \sum_{i=1}^L u_i^k (u_i^k)^\top, \quad B^{k+1} = B^{k+1} / \|B^{k+1}\|,\]

where we initialize with \(B^0 = I_{N \times N}\). To predict the \(N\)-dimensional vectors \(u_i^k\) we apply a stack of \(L\) Transformer encoders \[40\] to a set of per-parameter features linearly mapped to \(d = 128\) dimensions. Note that traditionally Transformer architecture has been applied to sequential data, whereas here we use it to aggregate information along the parameter dimensions. We visualize the architecture in fig. 2 for the case \(L = 3\). The \(i\)-th element of \(u_i^k\) is computed by applying a layer-dependent linear mapping \(M_i\) to the \(i\)-th row of the output \(E^{kl}\) of the Transformer encoder at the layer \(l\): \(u_i^k = M_i(E_i^{kl})\).

Note that our formulation of the update equations for \(B^k\) supports several desirable properties. First, it enables coupling between updates of individual parameters through the self-attention of the encoders. Secondly, our formulation does so without making the network specific to the objective function dimensions used during training. This allows us to readily generalize to problems of different dimensionality (see sec. 5.1). Finally, it allows the optimizer to accumulate information across iterations, similarly to how BFGS \[11\] incrementally approximates the inverse Hessian matrix as optimization goes on. Effectively our methodology works by learning a **preconditioning** for the first-order updates estimated in other learned optimizers, such as the Adafactor MLP \[24\]. While this preconditioner considerably increases the step quality, its computational cost grows quadratically in the number of parameters.

\[^1\]https://github.com/google/learned_optimization

\[^2\]\(f_{\text{sphere}}(x) = \sum_{i=1}^{d} x_i^2\), where \(x \in \mathbb{R}^d\)
sition specific information we add random offsets to the objective functions used for training.

Evaluation metrics. We use two types of aggregate metrics in our evaluation to assess both the quality of the minima that were found, as well as the number of iterations an optimizer needed in order to reach the minimum.

We rely on performance profiles \[2,5,10,36\] to compare Optimus to the baseline methods. Let \(P\) be the set of test problems and \(S\) is the set of optimizers tested. To define a performance profile one first introduces a performance measure \(m_{p,s} = \frac{j_{p,s} - f^*_p}{f^*_p - f^{w}_p}\), where \(j_{p,s}\) is best solution of method \(s\) for problem \(p\), \(f^{w}_p\) is the worst solution out of all methods, and \(f^*_p\) is the global minimum. The measure \(m_{p,s}\) is useful because it allows to compare performance of the optimizer across a set of different optimization tasks. The following ratio then compares each optimizer with the best performing one on the problem \(p\):

\[
r_{p,s} = \min\{m_{p,\cdot}\in S\}.
\]

The best solver for each problem has ratio \(r_{p,s} = 1\). Given a threshold \(t\) for each optimizer \(s\) we can now compute the percentage of problems \(\rho_s(t)\) for which the ratio \(r_{p,s} \leq t\):

\[
\rho_s(t) = \frac{1}{|P|} \text{size}\{p \in P : r_{p,s} \leq t\}. \tag{3}
\]

Thus, the performance profile \(\rho_s(t)\) is the proportion of problems a method’s performance ratio \(r_{p,s}\) is within a factor of \(t\) of the best performance ratio. Hence, \(\rho_s(1)\) represents the percentage of tasks for which optimizer \(s\) has best performance (lowest function value) out of the tested methods. We compare the performance profiles of Optimus with our baselines in fig. 3.

Finally, we also measure the relative number of iterations Optimus needed to reach the value of the minima found by another baseline optimizer. We use Adam, BFGS and Adafactor MLP as baselines for such comparison.

Results. We observe that Optimus typically converges to a lower objective value compared to other optimizers (see fig. 1 and supplementary material). We use the performance profile metric \[5\] to aggregate these results across functions. Note that it is not meaningful to directly average the per-function minima since each function is scaled differently and so values of minima are not directly comparable. The performance profile metric tackles this issue by relating minima for each function to its global minima, which
is known for all objective functions in our benchmark. The results are shown fig. 3. We observe that performance of Optimus is higher than other optimizers across all values of performance threshold indicating that on average Optimus gets closer to global minimum of each function compared to other optimizers.

In fig. 4 we show relative number of iterations Optimus needs to reach the same objective value as Adam, BFGS and Adafactor MLP optimizers. The target objective value in this evaluation is defined as average objective value achieved by the corresponding baseline after 100 iterations. The y-axis in fig. 4 corresponds to ratio between number of iterations required by a baseline and number of iterations required by Optimus. Values on y-axis larger than 1 indicate that Optimus required fewer iterations. The x-axis in fig. 4 is a percentile of the tasks in the benchmark. A particular point on the plot then tells us what percentage of the benchmark has a ratio between number of iterations greater or equal than a value on y axis at that point.

For example, we observe that for about 50% of the tasks Optimus requires about $10\times$ fewer iterations than Adam and BFGS and about $5\times$ fewer iterations than Adafactor MLP. In fig. 4 we label the x-axis with names of the objective functions corresponding to each point on the curve comparing Optimus to BFGS. We observe that BFGS excels on simple convex objective functions such as “Sphere” or their noisy versions such as “Griewank”, where it can quickly converge to global minimum. However BFGS fails on more complex functions with multiple local minima such as “Ackley” or functions where minima is located in a flat elongated valley such as “Rosenbrock” or “Dixon-Price”.

**Analysis of update step direction.** To further highlight differences between Optimus and Adafactor MLP we plot the absolute cosine similarity of their step along the steepest descent direction given by $-\nabla x$, and the Newton direction given by $-H^{-1}_k \nabla x$, where $H^{-1}_k$ is the inverse Hessian at point $x$. The results are shown in fig. 5 for optimization of the 2d Rosenbrock function. For clarity we also include the same similarity plot for BFGS. As expected, the direction of BFGS step becomes similar to Newton after a few iterations since the preconditioner in BFGS approximates the inverse Hessian. Note that overall the direction of the Optimus step is much closer to Newton compared to Adafactor MLP and much less similar to steepest descent. This supports the intuition that Optimus’s design extends the learned optimizer with a preconditioner similar to BFGS.

### 5.2. Physics-Based Motion Reconstruction

In principle, learned optimization can be applied to any problem that is solvable by means of local descent, and thus to any physics-based reconstruction that formulates human motion reconstruction as loss minimization [12, 19, 46, 49]. In this paper, we build on the DiffPhy approach of [12], who define a differentiable loss function for physics-based reconstruction. DiffPhy relies on the differentiable implementation of rigid body dynamics in [13] and shape-specific body model based on [47] to define a loss function that measures similarity between simulated motion and observations. The observations are either a set of 3d keyframe poses, when the goal is to reproduce articulated 3d motion in physical simulation, or a sequence of image measurements such as 2d image keypoints or estimated 3d poses in each frame. The physical motion in DiffPhy is parameterized via a control trajectory, which is given by a sequence of quaternions defining a target rotation of each body joint over time. The control trajectory implicitly defines the torques applied to each body joint via PD-control. Please refer to [12] for a more extensive explanation of the loss function. Hence, optimization, in this case, aims to infer the control trajectory that re-creates an articulated motion in physical simulation in a way that is close to the given observations and consistent with constraints (e.g. lack of foot-slace and non-intersection with respect to a ground plane).

**Datasets.** In our human motion experiments, we use the popular Human3.6M articulated pose estimation benchmark [14]. We follow the protocol introduced in [34] to compare to related work for video-based experiments.

For experiments with motion capture inputs, we rely the same subset of 20 validation sequences from Human3.6M as used in [12]. We then additionally evaluate on the dancing sequences from the AIST dataset [39] used in [12] to further evaluate the generalization of our approach across qualitatively different motion types.

**Evaluation metrics.** We report the standard 2d and 3d human pose metrics as well as physics-based metrics. The mean per-joint 3d position error (MPJPE-G), the per-frame translation-aligned error (MPJPE), and the per-frame Procrustes-aligned error (MPJPE-PA) are reported in millimeters. In addition, we measure the 2d keypoint error of the reconstruction (MPJPE-2d) in pixels. Finally, we measure the amount of motion jitter as the total variation in 3d joint acceleration (TV) defined as $\frac{1}{P} \sum_{t\in T} \sum_{k\in K} |\ddot{x}_k - \ddot{x}_k(t)|$, where $\ddot{x}_k$ is the 3d acceleration of joint $k$ at time $t$, as well as the percentage of frames exhibiting foot skate as defined by prior work [12].

**Model training.** We train Optimus and Adafactor MLP on the task of minimizing the DiffPhy loss, which measures similarity between the simulated motion and 3d poses estimated based on [50]. Following an initial grid-search to determine the best learning rate for each model, we train these for 10 days using 500 CPUs to generate training batches of optimization rollouts (128 roll-outs of length 50 per batch).
Figure 6. Left: comparison of validation loss during training of Optimus and AdaFactor MLP. Note how Optimus converges to lower validation loss much faster than the AdaFactor MLP model. Mid: comparison of loss curves during optimizing on “in domain” examples in tab. 1. Right: loss curves during optimizing of “out of domain” examples in tab. 1. Note how Optimus generalizes better than AdaFactor MLP on out of domain data and minimizes the loss faster than BFGS. Shaded area denotes 95% confidence interval.

<table>
<thead>
<tr>
<th>Method</th>
<th>MPJPE-G (in domain)</th>
<th>MPJPE-G (out of domain)</th>
<th># Func. Evals</th>
</tr>
</thead>
<tbody>
<tr>
<td>DiffPhy + BFGS [12]</td>
<td>38.3</td>
<td>24.7</td>
<td>71</td>
</tr>
<tr>
<td>+ AdaFactor MLP [24]</td>
<td>33.0</td>
<td>29.3</td>
<td>50</td>
</tr>
<tr>
<td>+ Optimus (Ours)</td>
<td>24.0</td>
<td>25.0</td>
<td>50</td>
</tr>
</tbody>
</table>

Table 1. Comparison of different approaches to trajectory optimization on motion capture data from our Human3.6M validation set.

Figure 7. Qualitative examples of video reconstructions by Optimus. We train Optimus on the Human3.6M [14] dataset (left). We then evaluate its performance on the dancing sequence from AIST [39] (see tab. 3) and qualitatively verify that Optimus is applicable to in-the-wild internet videos (rightmost images).

We show loss vs. training time curves on the validation set in fig. 6 (left). Note that Optimus generally converges much faster and to a lower loss value compared to competitors. For example, after 48 hours of training, Optimus has essentially converged to a loss of 6.49 whereas AdaFactor MLP requires nearly 240 hours to reach a loss value of 6.55.

Results. As our first experiment in tab. 1 we compare the performance of Optimus to AdaFactor MLP and BFGS using motion capture (mocap) data as input. In fig. 6 (middle and right) we additionally visualize how quickly the loss is reduced by the optimizer at each iteration. Note that iterations of BFGS and of the learned optimizers are not comparable in terms of computational resources because BFGS might evaluate the objective function more than once during line search. To compensate for different iteration costs, in fig. 6 we plot the number of objective function evaluations instead of the optimization iteration along the x-axis. In this experiment we train all models on walking sequences from Human3.6M dataset \(^5\). We refer to walking sequences as “in domain” in fig. 6 and tab. 1. We then assess performance on a validation set of “in domain” and “out of domain” sequences corresponding to motions other than walking. Note that Optimus improves over other approaches in the “in domain” setting and reaches the loss comparable to BFGS at roughly half of loss function evaluations (cf. fig. 6 (middle)). In the “out of domain” setting Optimus reaches nearly the same accuracy compared to BFGS (24.7 vs 25mm.) but again converges much faster. In contrast AdaFactor MLP does not improve over BFGS on the “out of domain” motions and converges to a higher loss (cf. fig. 6 (right)).

In the second experiment, we evaluate performance of our best model DiffPhy+Optimus for video-based human motion reconstruction. Note that Optimus performs well on the dancing sequences from AIST even though it has been trained only on walking data from Human3.6M (150.2 for DiffPhy vs. 149.8 mm MPJPE-G for Optimus) and that Optimus is able to handle mocap and video inputs without retraining. We observe similar results on Human3.6M, where Optimus performs on par with BFGS (138.6 vs. 139.1 MPJPE-G) while requiring roughly half as many function evaluations (88 for DiffPhy vs. 40 for Optimus). We show a few qualitative examples for Optimus results on Human3.6M, AIST and internet videos in fig. 7.

Ablation experiments. We now evaluate how different design choices affect the model performance. Due to high computational load, we train each model for up to 48 hours. The results are shown in tab. 4. We refer to a version of Optimus that does not incrementally update the matrix \(B^k\) in sec. 4 and instead predicts it from scratch at each iteration.

\(^5\)We use “Walking” and “WalkTogether” activities.
Table 2. Quantitative results on Human3.6M [14] comparing our model to prior methods.

<table>
<thead>
<tr>
<th>Model</th>
<th>MPJPE-G</th>
<th>MPJPE</th>
<th>MPJPE-PA</th>
<th>MPJPE-2d</th>
<th>TV</th>
<th>Foot skate</th>
</tr>
</thead>
<tbody>
<tr>
<td>VIBE [16]</td>
<td>207.7</td>
<td>68.6</td>
<td>43.6</td>
<td>16.4</td>
<td>0.32</td>
<td>27.4</td>
</tr>
<tr>
<td>PhysCap [34]</td>
<td>-</td>
<td>97.4</td>
<td>65.1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SimPoE [48]</td>
<td>-</td>
<td>56.7</td>
<td>41.6</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Shimada et al. [33]</td>
<td>-</td>
<td>76.5</td>
<td>58.2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Xie et al. [46]</td>
<td>-</td>
<td>68.1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DiffPhy [12]</td>
<td>139.1</td>
<td>82.1</td>
<td>55.9</td>
<td>13.2</td>
<td>0.21</td>
<td>7.2</td>
</tr>
<tr>
<td>Optimus (Ours)</td>
<td><strong>138.6</strong></td>
<td>82.8</td>
<td>57.0</td>
<td><strong>13.2</strong></td>
<td><strong>0.20</strong></td>
<td><strong>6.5</strong></td>
</tr>
</tbody>
</table>

Table 2. Quantitative results on Human3.6M [14] comparing our model to prior methods.

Table 3. Quantitative results on generalizing to a new dataset. Optimus was trained Human3.6M [14] and is here evaluated on a subset of the dance motion dataset AIST [39].

<table>
<thead>
<tr>
<th>Variant</th>
<th>MPJPE-G</th>
<th>MPJPE</th>
<th>MPJPE-PA</th>
<th>MPJPE-2d</th>
<th>TV</th>
<th>Foot skate</th>
</tr>
</thead>
<tbody>
<tr>
<td>DiffPhy [12]</td>
<td>150.2</td>
<td>105.5</td>
<td>66.0</td>
<td>12.1</td>
<td>0.44</td>
<td>19.6</td>
</tr>
<tr>
<td>Optimus</td>
<td>149.8</td>
<td>104.4</td>
<td>66.4</td>
<td>12.1</td>
<td>0.45</td>
<td>21.5</td>
</tr>
</tbody>
</table>

Table 3. Quantitative results on generalizing to a new dataset. Optimus was trained Human3.6M [14] and is here evaluated on a subset of the dance motion dataset AIST [39].

<table>
<thead>
<tr>
<th>Variant</th>
<th>MPJPE-G</th>
<th>Loss</th>
<th># Params</th>
<th>Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimus</td>
<td>33.1</td>
<td>0.534</td>
<td>832,526</td>
<td>78.9</td>
</tr>
<tr>
<td>Optimus, no state</td>
<td>36.5</td>
<td>0.675</td>
<td>832,526</td>
<td>115.1</td>
</tr>
<tr>
<td>Optimus, no structure</td>
<td>37.3</td>
<td>0.721</td>
<td>832,139</td>
<td>10.7</td>
</tr>
<tr>
<td>Adafactor MLP [24] 512x4</td>
<td>40.0</td>
<td>0.723</td>
<td>811,531</td>
<td>3.7</td>
</tr>
<tr>
<td>Adafactor MLP [24] 128x4</td>
<td>40.6</td>
<td>0.811</td>
<td>22,411</td>
<td>2.6</td>
</tr>
<tr>
<td>Optimus</td>
<td>33.1</td>
<td>0.534</td>
<td>832,526</td>
<td>78.9</td>
</tr>
<tr>
<td>Optimus, 50 iterations</td>
<td>28.5</td>
<td>0.383</td>
<td>832,526</td>
<td>78.9</td>
</tr>
<tr>
<td>Optimus + BasinHopping [42]</td>
<td>25.6</td>
<td>0.352</td>
<td>832,526</td>
<td>78.9</td>
</tr>
</tbody>
</table>

Table 4. Top: ablation of model components on motion capture from our Human3.6M [14] validation set. Bottom: comparison of different test time operating modes.

6. Conclusion

We have introduced a learned optimizer, Optimus, based on an expressive architecture that can capture complex dependency updates in parameter space. Furthermore, we have demonstrated the effectiveness of Optimus for the real-world task of physics-based articulated 3D motion reconstruction as well as on a benchmark of classical optimization problems. While Optimus’s expressive architecture outperforms simpler methods such as Adafactor MLP, the expressiveness comes at an increased computational cost. As a result, Optimus is best suited for tasks where the loss function dominates the computational complexity of optimization (e.g., physics-based reconstruction) but might be less suited for applications where the computation of the loss function is fast (e.g., training neural networks). In future work, we hope to address this limitation by learning factorizations of the estimated prediction matrix.

Ethical Considerations. We aim to improve the realism and quality of human pose reconstruction by including physical constraints. By amortizing the computation through learning from past instances, we hope to reduce the long-term computational demand of these methods. We believe that our physical model’s level of detail (e.g. lack of photorealistic appearance) limits its applications in adverse tasks such as person identification or deepfakes. Furthermore, the model is inclusive in supporting a variety of body shapes and sizes, and their underlying physics.
References


