SKIPPING RECOMPUTATION DURING TRAINING: FROM RECURRENT MODELS TO MODEL PREDICTIVE CONTROL AND BEYOND

Anonymous authors
Paper under double-blind review

ABSTRACT

Sequential model inference can often be accelerated with a binary ”skip or compute” gate. At each step of the sequential computation, the gate determines if an expensive operation is performed, or if it is skipped. The setup is shared across e.g. dynamical systems, diffusion models, sequence classification, and replanning in Model Predictive Control (MPC). During training, however, supervised learning methods still perform the costly updates at every step to obtain gradient signals needed for learning the gate. We address this limitation by showing that the gate can be learned from sparse execution while still receiving dense supervision over the skipped steps, without ever having used the gate to control recomputation. The gate learning signal is derived from the computational graph of the sequence model by combining downstream loss cotangents with the input sensitivity of the expensive operation. This decouples learning of the gate network from the sequential model's recomputation schedule, enabling efficient batch-synchronized training with sparse expensive updates. We demonstrate how this approach can be applied in three distinct scenarios: autonomous dynamics, sequence classification with input-skipping recurrent models and replanning in MPC, with substantial reductions in training time and comparable or improved accuracy.

1 INTRODUCTION

In many sequential learning tasks we can exploit temporal redundancy in the processed signal or sequence, by reusing a computationally expensive object over multiple computational steps, or by applying computation on some of the input steps and ignoring the rest.

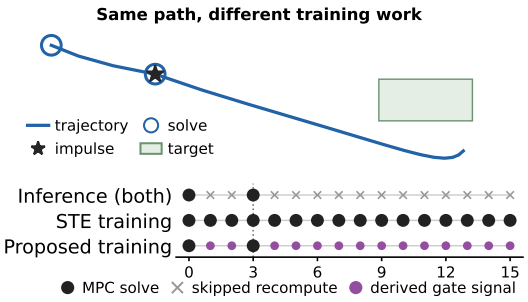

Figure 1: Prior skipping methods based on supervised learning skip costly updates in sequential models, such as re-planning in MPC shown here, during inference using cached versions instead, but still require dense recomputation during training. We address this, showing how we can skip computation also at training time.

This general formulation spans a wide range of problems. For example, in sequence classification all input elements may not be needed to make a decision (Campos et al., 2018; Tao et al., 2019) while in Model Predictive Control (MPC) a plan is expensive to compute but can remain valid over multiple steps (Amos et al., 2018; Bøhn et al., 2021; Conrad & Graichen, 2024). In diffusion-based generation, the most computationally intensive parts of denoising (SDE/ODE) can be cached for some steps while retaining good accuracy(Zhao et al., 2026) and Dynamical system solvers reuse expensive Jacobian/linearizations across multiple steps (Shampine & Reichelt, 1997; Balos et al., 2025; Archut & Corves, 2025). Often the redundancy is exploited by learning a binary ”skip or compute” gate that determines when the costly update needs to be carried out (Campos et al., 2018; Tao et al., 2019; Bøhn et al., 2021; Zhao et al., 2026; Lin, 2026; Dang et al., 2024; Ren & Li, 2026).

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

The previous literature centers on test-time efficiency, paying little attention to the learning phase. The standard approaches (Campos et al., 2018; Lin, 2026) build on the straight-through estimator (STE) (Bengio et al., 2013), where the expensive component is evaluated densely during training to retrieve the gradient. That is, the expensive computation, e.g. complete re-planning in MPC, is still needed for each step, merely for learning the gate. Alternative gate learning solutions have been proposed, but they still retain computational overhead or are limited to specific instances: Bøhn et al. (2021) avoid differentiation through the discrete choice via use of a reinforcement learning formulation, Zhao et al. (2026) learn a gate for inference-time diffusion caching by performing an expensive combinatorial search over refresh schedules, and Tao et al. (2019) assume the backbone is already trained, and devise a post-hoc task-specific proxy learning target for RNNs.

We propose a new approach, deriving a signal for the skipped steps that approximates the loss change that would occur if the cached object were replaced or *refreshed* by the recomputation of a new expensive object. We build this on two complementary aspects, sensitivity and hindsight. Task-loss sensitivity has been used e.g. for improving plans between refreshes in MPC (Palma et al., 2015), to decide for how many steps the current plan is valid (Conrad & Graichen, 2024), and for estimating how a change induced by removing a network component would affect the loss (Molchanov et al., 2017; Lee et al., 2019; Sui et al., 2026). We, instead, exploit sensitivity in the sequential computation, by noting that a refresh and the consecutive cached steps that follow share the same execution of the expensive computation, and therefore also share its local input-output sensitivity. This shared sensitivity lets us predict how refreshing would change the cached object. However, it is not enough alone, but rather we need to integrate hindsight from the gradient of the task loss to indicate whether that change would increase or decrease the loss. By combining the two, we get a first-order Taylor approximation of the loss change from refreshing, which can be used as a dense supervision signal for the skipped steps, without having to perform the expensive computations. This estimate does not depend on the gate itself, enabling use of flexible recomputation schedules.

Contributions: We provide new computationally efficient schema for learning when costly updates can be skipped in dynamic models, and show that it can be learned independently of the computation schedule of the sequential model, enabling efficient batch-synchronized training. We demonstrate the method in three scenarios: dynamic models, recurrent sequence models, and Model-predictive control. For MPC, we provide the first direct supervised learning method for re-planning timing.

2 Problem setting

Let F_θ be a sequentially applied function parameterized by θ, with

$$x_{t+1} = F_\theta(x_t, u_t),$$

where t is the time step, x_t is the propagated state, and u_t is an optional per-step input. The parameter θ is assumed to depend on a context z_t, via a differentiable function $\theta = h(z)$. The distinction between F and h allows us to separately discuss the computational cost of constructing θ and applying F_θ, while distinguishing z_t from x_t allows θ to depend either on the propagated state, a separate context, or both. This setup can be used to describe applications across recurrent neural networks, diffusion models, state-dependent autonomous dynamics, and sequential control methods (Campos et al., 2018; Zhao et al., 2026; Shampine & Reichelt, 1997; Mayne et al., 2000).

Reusing computation Within this general family, our interest is in problems where $\theta = h(z_t)$ is expensive to compute, compared to the computation required for advancing the state with fixed $F_\theta(x_t, u_t)$. For example, in MPC, the planning step can be extremely costly, while following a plan is almost free (Mayne et al., 2000; Bøhn et al., 2021; Ren & Li, 2026). To reduce the total cost, some $k > 1$ consecutive steps can be done using a cached $\theta_c = h(z_c)$, which is reused until the next recomputation, incurring a natural compromise between efficiency and accuracy

$$x_{t+1} = F_{\theta_c}(x_t, u_t), \quad c \leq t < c + k.$$

The central learning task is then determining, dynamically, for how long the cached computation is valid and can be used without substantial reduction in accuracy (Campos et al., 2018; Tao et al., 2019; Bøhn et al., 2021; Dang et al., 2024; Conrad & Graichen, 2024; Ren & Li, 2026). Following this general line of work, we address this by learning a binary gate $g(x, z, u_t)$ that indicates whether

the cached estimate should be recomputed on the current step. We call this *opening the gate* as an easy mnemonic to indicate a new expensive recomputation being performed. The output of the gate is interpreted as probability of whether the expensive recomputation should be performed at the current step. When the gate opens, we call this *refreshing* the relevant values, where we store both the new θ and the input used to generate it to form z_c needed for computing the gate's learning target

$$z_c \leftarrow z_t, \quad \theta_c \leftarrow h(z_t)$$

A broad range of learning algorithms has been applied to the learning of binary gates to provide computational savings during inference (Campos et al., 2018; Tao et al., 2019; Bøhn et al., 2021; Dang et al., 2024; Ren & Li, 2026; Lin, 2026; Zhao et al., 2026). These existing approaches make a trade-off during training: supervised methods require dense evaluation of the expensive computation (Campos et al., 2018; Tao et al., 2019; Lin, 2026), whereas reinforcement learning avoids this requirement but relies on more indirect supervision from rewards rather than task-loss derivatives (Bøhn et al., 2021; Dang et al., 2024; Ren & Li, 2026). Our goal is sparse execution of the expensive computation during training, while retaining dense per-step supervision.

3 METHOD

Our key contribution is a new training signal that provides dense per-step supervision for learning the gate $g(\cdot)$, despite skipping a substantial fraction of the costly recomputations of $\theta = h(z_t)$ during training. The signal efficiently answers "in hindsight, how much would refreshing help or harm?" even at steps where the expensive computation was skipped. The signal is constructed from the computational graphs of the propagator F and the operation h, without involvement of the gate.

Our training signal builds on two distinct components, both intuitively necessary for determining whether refreshing would be beneficial at a given step. Hindsight from the negative loss gradient gives us the direction in which changes to the expensive object would decrease the task loss. Local sensitivity, on the other hand, predicts how refreshing would change that object. If the predicted change is aligned with a loss-decreasing direction, refreshing is beneficial, with the degree of alignment determining its utility. Building on these intuitive concepts, we next show formally that their combination gives a first-order approximation of the real utility of recomputing the expensive object.

3.1 RECOMPUTATION UTILITY

For cached and current states z_c, z_t, define the candidate refresh direction as $\Delta z_t = z_t - z_c$. Let λ_t denote the loss cotangent at the tth use of the cached output θ_c, obtained e.g. via ordinary backpropagation from the task-specific loss.

Define $L_t(z)$ as the task loss obtained when the expensive computation is evaluated with input z at step t. Then $L_t(z_c)$ is an actual measured loss when a cached value z_c was reused at step t. In order to estimate if recomputation would have been useful, we approximate the counterfactual loss $L_t(z_t)$ without computing the entire alternative future rollout.

Multivariate Taylor provides a linear approximation of L_t around z_c evaluated at z_t as

$$L_t(z_t) = L_t(z_c) + \nabla_z L(z_c)^T (z_t - z_c) + O(||z_t - z_c||^2)$$

where second and higher order terms in the displacement are ignored. Chain-rule gives

$$\nabla_z L_t(z_c) = \frac{dL}{d\theta}\frac{d\theta}{dz} = \lambda_t^T J_h(z_c),$$

where $J_h(z_c)$ is the Jacobian of h evaluated at z_c. The first-order estimate of the change in loss at the time of refresh is then

$$L_t(z_t) - L_t(z_c) \approx \lambda_t^T J_h(z_c)(z_t - z_c).$$

This first-order structure with gradient multiplied by a perturbation is closely related to Taylor- and sensitivity-based saliency formulations used in pruning (Molchanov et al., 2017; Lee et al., 2019).

We define recomputation utility, coined RecompUtility, of opening the gate as the negative of the change in loss

$$U_t := -\lambda_t^T J_h(z_c)\Delta z_t \in \mathcal{R}. \tag{1}$$

This signal is agnostic to which task-specific loss and model are used, only requiring that the expensive function h is differentiable.

3.2 Relationship with straight-through estimator

RecompUtility was derived above as the first-order approximation to the loss change of refreshing. Here we show, that in its most basic setting, RecompUtility's scalar value coincides with the straight-through estimator (STE) (Bengio et al., 2013) when the gate is closed, even though STE requires explicit re-computation of $h(\cdot)$ that we can skip (Campos et al., 2018). Consider g as a continuous gate variable controlling $z_t(g) = gz_t + (1-g)z_c$, where $g = 0$ means the gate is closed, and $g = 1$ means z_c was refreshed. Using standard STE, the gate is learned using the gradients

$$\frac{dL_t}{dg} = \frac{dL_t}{d\theta}\frac{d\theta}{dz}\frac{dz}{dg} = \lambda_t^T J_h(z_t(g))(z_t - z_c).$$

When the gate is closed, we hence have

$$\left.\frac{dL_t}{dg}\right|_{g=0} = \lambda_t^T J_h(z_c)\Delta z_t = -U_t.$$

Despite this identity, the methods have a fundamental difference: STE requires executing the expensive operation at every closed step to build the computational graph allowing gradients to pass from the loss to the gate. Furthermore, the two quantities are not equal when the gate is open, but there no sensitivity analysis is needed anyway as the counterfactual expensive computation is available, and utility is simply the agreement of the cotangent with the displacement $\Delta\theta_t = \theta_t - \theta_c$ giving $U_t = -\lambda_t^T \Delta\theta_t \approx L(\theta_c) - L(\theta_t)$.

3.3 Learning the gates with the utility signal

Supervised learning of the gate Each executed trajectory of the sequential model can be used to generate training samples (q_t, U_t) for the gate, where q_t is any cheap context eventually available to the gate during inference. Samples with the utility signal U_t, interpreted as the training label, are then sufficient for supervised learning of the gate network, while the framework is agnostic to the task loss function and even the type of sequence model used. For example, regression or classification losses could be used to learn mappings such as $q_t \to U_t$ or $q_t \to 1[U_t > \alpha]$ respectively. When directly regressing U_t, its decision threshold can be selected post hoc. While the gate inputs q_t can in principle be any arbitrary contextual information, ideally they are computationally cheaply available. This way the computation of the gate itself does not take a significant portion of the model processing time. We will show concrete examples in Section 4.

Decoupled gate learning Since retrieving the training samples does not require that the gate participates in the execution of the trajectory, it allows the gate to be trained either concurrently with the rest of the network, or through a separate trajectory generation and gradient collection pass from a frozen task model, similarly to prior post-hoc gate-learning setups (Tao et al., 2019; Zhao et al., 2026). The task model may therefore be executed under any externally chosen schedule, while the gate is trained to predict utilities collected from those trajectories. This schedule may be chosen for computational efficiency, coverage, or task-model robustness. Decoupled gate learning is particularly useful in batched computation environments. Naive per-step switching can lead to inefficient ragged execution, whereas the external schedule can be chosen such that expensive recomputations are synchronized across each batch. In our experiments, we exploit this property using randomized batch-synchronized schedules.

The described decoupled setup significantly differs from methods whose gate learning is tied to the execution during training. For example, policy-gradient methods such as Proximal Policy Optimization (PPO) learn from trajectories generated by the current policy (Bøhn et al., 2021), and STE-based methods obtain the gate gradient through the forward computational graph, where the gate needs to be executed together with the rest of the network (Campos et al., 2018).

4 EXAMPLE INSTANTIATIONS

The proposed utility signal and the method for learning the gates are generally applicable for the broad family of sequential models characterized in Section 2. Here we explain in more detail how the proposed method can be applied to three distinct scenarios. We start with the conceptually simplest and notation-wise most straightforward scenario of autonomous dynamics with a gate deciding whether to perform an expensive recomputation of a new local propagator, or to reuse the already materialized cached propagator. Then, we discuss a differentiable MPC example, where the gate decides when to perform expensive replanning. Finally, we discuss an extension of our method to a scenario with exogenous inputs, in the form of an RNN with sequence inputs and hidden state propagation. The first two examples use or adapt the first-order utility construction from equation 1 as the learning target, whereas the last one extends the setup with another approximation layer.

4.1 CACHED LOCAL OPERATORS IN DYNAMICS

A standard optimization in numerical Ordinary Differential Equation (ODE) solvers is to reuse expensive Jacobian information across multiple integration steps (Shampine & Reichelt, 1997; Balos et al., 2025). We isolate the same computational pattern into a simplified setting with a sparsely refreshed local state propagator integrating autonomous dynamics.

Assume an ODE $\frac{dx}{dt} = \dot{x} = f(x)$ with a cached local dynamics operator θ_t, where the model starts with an initial condition x_0, and integrates it over discrete observation times $t = 0, ..., T$. Here $z_t = x_t$ so there are no extraneous control, latent, or per-step input signals other than the initial condition, $\theta_t = h(x_t)$ materializes the expensive local state propagator, and $x_{t+1} = F_{\theta_c}(x_t)$ cheaply advances the state within a closed-step interval, given the currently cached θ_c. During integration with the cached local propagator, the evolving state trajectories naturally produce new refresh candidates x_t and associated displacements $\Delta x_t = x_t - x_c$. Here λ_t is the cotangent at the tth reuse of θ_c The corresponding RecompUtility is as before

$$U_t = -\lambda_t^T J_h(x_c)\Delta x_t.$$

Since within a closed-step interval every step reuses the same forward execution $h(x_c)$, its forward residuals and pullback can be shared. The per-step cotangents λ_t can then be pulled back in parallel using the shared pullback, without explicitly materializing a Jacobian, and finally the resulting sensitivities are multiplied with their respective displacement Δx_t. A natural input context for learning the gate includes both the current state, and the state at the last refresh, (x_t, x_c). This allows the gate to learn for example that certain displacements between the current and cached states, or specific areas of the state-space, are such that the task loss would be decreased by a refresh.

4.2 DIFFERENTIABLE MODEL PREDICTIVE CONTROL

Differentiable Model Predictive Control (MPC) often uses slow iterative optimization for the forward pass, while sensitivities can be obtained much faster via implicit differentiation (Amos et al., 2018; Bravo-Palacios et al., 2026), This is a particularly good fit for our method that primarily reduces the number of forward computations. MPC repeatedly solves an optimal control problem from the current state, producing a plan multiple steps into the future, while only applying the first action of the resulting plan (Mayne et al., 2000). It operates on (known or estimated) dynamics $x_{t+1} = f(x_t, a_t)$ with states x_t and per-step actions or forcing a_t. We represent multi-step plans of length K as $\theta = h(x_t) = [a_0^t, ...a_K^t]$, where in a_0^t the subscript denotes the index within current plan, and superscript is the time when the plan was computed. Here, h is the expensive planning operation that solves the optimal control problem for the current state. Computing the plan on every step is expensive and time-consuming, so previous works have studied methods for reusing plans across multiple steps (Palma et al., 2015; Bøhn et al., 2021; Conrad & Graichen, 2024).

Differentiable MPC planners (Amos et al., 2018; Bravo-Palacios et al., 2026) allow backpropagating through the MPC planner. For example, the iterative Linear Quadratic Regulator (iLQR) based planner in Amos et al. (2018) iteratively linearizes nonlinear dynamics around the current guess, optimizes the guess wrt. that linearized solution, and then builds a new trajectory from the new guess, iterated until convergence. Each forward pass requires significant computation and can require dozens of iterations, while the gradients can be retrieved via implicit differentiation, approximately

in the same amount of time as a single iteration (Amos et al., 2018). The differentiability of the planner makes it possible to retrieve the utility signal of Equation 1 for deciding when recomputing the plan is particularly useful.

At time $t = c + k$ with no refresh, the action taken on this step is a_k^c. The counterfactual action whose usefulness we would like to estimate would replace a_k^c with a_0^t, replacing kth action of the cached plan with a new plan's first action. We do not know a_0^t exactly as computing it would require another expensive solve, so we estimate

$$a_0^t \approx a_0^c + J_0(x_t - x_c) \tag{2}$$

where $J_0 = \left.\frac{da_0(x)}{dx}\right|_{x=x_c}$ denotes the block of the Jacobian that describes the first output action's sensitivity to input displacements, evaluated at the cached x_c.

We want to estimate the displacement $\Delta a = a_0^t - a_k^c$ that would happen if the gate opened. This is obtained from the planner sensitivity J_0, by replacing a_0^t with the approximation in Eq. 2 giving

$$\Delta a \approx a_0^c - a_k^c + J_0(x_t - x_c).$$

As before, normal automatic differentiation gives the step gradient at the output $\lambda_t = \frac{dL}{da_k^c}$ via the actually executed action a_k^c. The first-order effect of output changes to the loss is $\lambda_t^T \Delta a$ and therefore RecompUtility can be expressed in output space

$$U_t^{J_0} = -\lambda_t^T [a_0^c - a_k^c + J_0(x_t - x_c)].$$

This output space version is more convenient because the relevance of the actions changes every step even without recomputation. The above is implementation-wise the simplest instantiation of our framework for MPC, as it only uses the derivatives provided by any differentiable planner. However the computation of J_0 requires a number of pullbacks to construct, scaling with dimensionality of the actions. A more efficient version may be extractable from the intermediates the MPC solver already computes internally, at virtually no cost. The derivation of this version is deferred to Appendix A.3.

4.3 Exogenous inputs in RNNs

We next consider the case where only part of the expensive operation's input is cached, while another part continues to change at every step. Skipping RNNs (Campos et al., 2018; Tao et al., 2019) are a natural application: given a sequence as input, use a learned gate to decide when the recurrent hidden-state update can be omitted. For our method it is natural to treat the problem as a cheap gate deciding for every element of the input, whether the RNN state update can be skipped.

Consider a simple RNN with state update skipping, where for skipped steps the state is held $z_t = z_c$, and otherwise it is updated $z_t = h(z_c, u_t)$ with h being an expensive to compute nonlinear recurrent update, z_t the hidden state at step t, and u_t the exogenous inputs (e.g. a sequence of characters or pixels), where we wish to skip computing h on some of the inputs as determined by the skip-gate. In this scenario, due to the state updates being done by the expensive function h itself instead of a separate F_θ, the sensitivity must be computed with respect to a different transition. If the sensitivity for the utility construction were obtained from the computation that generated the current cached output like in previous scenarios, here this would develop the sensitivity based on the $z_{c-} \to z_c$ transition, where z_{c-} refers to the cached state previous to the current one. Meanwhile, a counterfactual refresh happening k steps after c, would use the exogenous input u_{c+k} and hidden state z_c as inputs, updating $z_c \to z_t$, which is one hidden-state transition later. So to obtain a sensitivity around the correct hidden-state z_c we must perform one additional anchor evaluation $h(z_c, u_a)$ where u_a is some representative input from the closed interval. The extra anchor evaluation corrected for the hidden-state update, but the exogenous input still changes at every step. Hence the relevant Jacobian $J_t = \frac{dh}{du}(z_c, u_t)$ changes on every step. Thus it is not possible to exactly recover the first-order approximation of the counterfactual without performing dense forward evaluations. We propose approximating all Jacobians within a closed-step interval by the Jacobian $J_a = \frac{dh}{du}(z_c, u_a)$ for a representative input u_a.

Since the hidden state is fixed within the closed-step interval, the candidate refresh directions have zero hidden-state component, and only the exogenous input changes, leaving $\Delta u_t = u_t - u_a$ as the

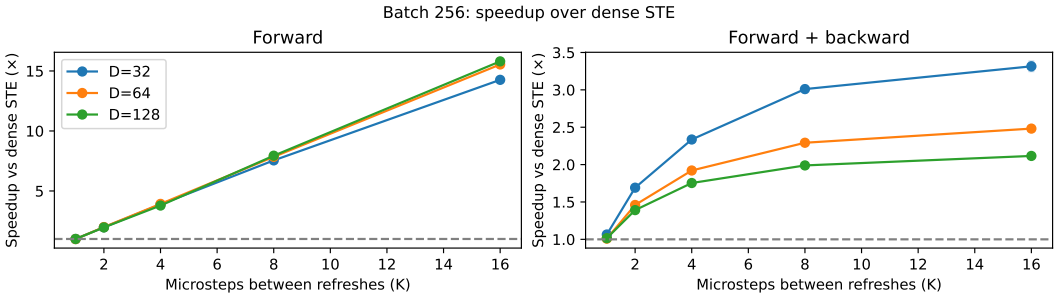

Figure 2: Speed benchmark for the same closed-step gradients as STE up to numerical precision, with the gain increasing with the sparsity of the updates from no benefit at dense evaluation $K = 1$.

only non-zero displacement. Thus the approximated RecompUtility

$$U_t = -\lambda_t^T J_a \Delta u_t$$

is now based on an approximated sensitivity, with the displacement term measuring difference of each input to the anchor input. With this RNN formulation applied sequence classification, if only the last hidden state is emitted and the recurrent state does not change for skipped steps, then the cotangent is identical between adjacent closed steps. A single pullback $\lambda_t^T J_a$ of the shared cotangent is therefore sufficient for an entire closed-step interval, and each intermediate steps' utility is trivially retrieved with an input-space product $U_t = -(\lambda_t^T J_a)\Delta u_t$. Thus an entire closed interval requires one extra forward pass for the anchor, one VJP, and one cheap inner product per skipped step.

5 Experiments

We do three experiments to demonstrate that using the proposed learning signal to learn the gate can also produce substantial training time reductions in practice. First we set up an intentionally minimalistic computational speed benchmark reproducing exact gradients of STE in an idealized autonomous dynamics example following Section 4.1. Second, we experiment the MPC setting, following the derivations in Section 4.2. Here, it is natural to assume the sequence model and its expensive components are pre-existing, so we focus the experiment on gate learning. Third, to further demonstrate the generality of the method, we apply the extension introduced in Section 4.3 to the Sequential MNIST classification problem, measuring both training time and accuracy. Additionally, we use this experiment as an opportunity to demonstrate that the gate learns a better policy than random or periodic refreshing. All experiments are run on a computer with an Intel i7-7700K CPU with 32GB of DDR4 RAM, with a NVIDIA RTX 5070 Ti GPU having 16GB of VRAM.

5.1 Retrieving the equal gradients from RecompUtility and STE

We benchmark the performance of retrieving exactly the same gradients as STE via RecompUtility, on a very computationally heavy task, without performing any actual learning. The setting emulates a realistic workload for an autonomous state-dependent dynamics task following the setup in Section 4.1. The expensive operation in this experiment consists of both computing the Jacobian of nonlinear dynamics, and from that matrix, the "local propagator" via a matrix exponential, as is done in exponential integrators (Hochbruck & Ostermann, 2010; Higham, 2009) $\theta = M = \exp(\Delta t J) \in \mathcal{R}^{D\times D}$, where Δt is a step size in time. The simulated state propagation function used on the skipped steps is a cheap application $x_{t+1} = F_\theta(x_t) = Mx_t$ We refer to each state propagation step with the cached matrix as a microstep. More details in the Appendix A.4. We perform a sweep over schedules of $K \in \{1, 2, 4, 8, 16\}$ microsteps between each refresh, where $K = 1$ corresponds to refreshing on every step. The achieved speedups are in Figure 2, with forward-only timings on the left, and total trajectory plus gradient propagation time on the right.

Results: Figure 2 shows, as expected, our method does not offer any computational advantage if the expensive computation is performed on every step, $K = 1$. For $K > 1$, the forward pass

Table 1: Held-out results on 32 episode specifications per scenario. Training time counts training time and trajectory collection, excluding initial-plan preparation. MPC solves include the initial solve. A/C are correct with no additional solve. B is correct with exactly one additional solve reducing task cost by more than 0.1 relative to the initial-plan-only trajectory.

Method	A	B	C	Total	Task cost ↓	MPC solves ↓	Train decisions	Train min
J_0 (Ours)	32/32	32/32	32/32	96/96	2.3469	1.333	108,288	4.8
Dense STE	32/32	32/32	32/32	96/96	2.3554	1.333	112,800	136.6
PPO	32/32	29/32	20/32	81/96	2.9785	1.458	178,560	39.6
Prediction residual	32/32	0/32	32/32	64/96	3.5254	1.000	N/A	N/A
Initial solve only	32/32	0/32	32/32	64/96	3.5254	1.000	N/A	N/A
Solve every step	N/A	N/A	N/A	N/A	2.3250	48.000	N/A	N/A

speeds up linearly with increasing sparsity. The benefit is generally larger in the forward pass, since our approach completely skips some forward evaluations, while the closed-step utility still requires the corresponding pullbacks. These pullbacks are evaluated in parallel, but as batch size and state dimension increases, the GPU becomes increasingly saturated and parallelization provides less benefit. As forward and backward costs are comparable in this experiment, and once the backward computation fully utilizes the GPU, removing most of the forward work leads naturally to a speedup approaching 2x. The retrieved closed-step gradients agree with STE to floating-point precision in 64-bit arithmetic, while they are not bit-wise exact due to different computational flow.

5.2 Model Predictive Control plan recomputation gate

We apply our method as described in Section 4.2 to a simple MPC replan timing problem on synthetic data. In this experiment the only learned part is the binary gate while the dynamics and the planner could be learned components in a more complex scenario. Our baselines are **Prediction-residual**: measure how much the current observed state x_t differs from what the dynamics model predicted $\bar{x}_*^c$ at the last recomputation, and if their difference exceeds a threshold $x_t - \bar{x}_t^c > \alpha$, trigger a recompute. **Dense STE:** standard supervised learning method that requires recomputation at every step during training. **PPO**: Reinforcement Learning with Proximal Policy Optimization (PPO) resembling but not exactly reproducing the work of Bøhn et al. (2021). **Never recompute:** just uses the initial plan, and **Always recompute:** the prohibitively expensive but maximally accurate baseline. All methods use the same differentiable MPC planner code for their training, based on (Amos et al., 2018).

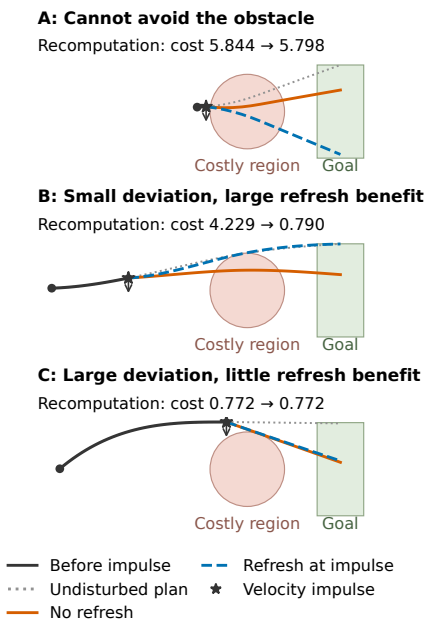

Figure 3: Dataset scenarios.

Dataset design and results The synthetic dataset is visualized in Figure 3 with three scenarios, A/B/C, where the controller's task is to move a point mass with velocity to a goal at minimal cost, while the direct path is obstructed by a circular costly region to be avoided. To ensure there is a need for a recomputation, we add a single disturbance at a randomized time point. See Appendix A.1 for details. The results are reported in Table 1. The proposed method solves the task scenarios with perfect accuracy in significantly lower training time than the trained baselines.

5.3 SEQUENCE MODELS WITH EXOGENOUS OUTPUTS

We test the extension of our method to exogenous inputs based on the approximation introduced in Section 4.3. As the task, we use "Sequential MNIST", where goal is to classify MNIST digits (LeCun et al., 2010) from a sequence of 784 pixels (28*28 flattened to a 1D sequence), presented to the model one pixel at a time. In this scenario, the learned gate is used to decide if the current pixel should be processed by the RNN, or whether the RNN computation is skipped completely. As baselines, we compare to SkipRNN and STE. All models use a 4 layer RNN cell with a 64 dimensional latent state. Further details are in the Appendix A.2, along with an experiment comparing the accuracy of the model under learned, periodic, and random refreshing. The scenario is fundamentally sub-optimal for our method, as the width and depth of an RNN sufficient to solve the task is very small, and so there is limited computational advantage from skipping the cells in comparison to the execution cost of the gate itself. Nevertheless, as shown in Table 2, the proposed method is still almost twice as fast as the other models to train, reaching comparable but slightly lower accuracy. At inference the proposed model only beats the dense model in execution time at very small refresh rates, sacrificing a lot of accuracy. Advantages exist mostly in faster training time with a small cost in accuracy. SkipRNN is clearly the fastest at inference.

Table 2: Sequential MNIST accuracy and computation across three seeds.

Method / setting	Updates (%)	Test acc. (%)	Construction (s)	Inference (ms/image)
Our (GRU inference speed)	12.66 ± 1.24	92.86 ± 1.65	289.7 ± 85.1	9.26 ± 0.33
Our (40% target)	40.08 ± 0.62	97.64 ± 0.39	289.7 ± 85.1	13.69 ± 0.35
SkipRNN	31.87 ± 1.04	98.22 ± 0.19	598.1 ± 37.3	6.58 ± 0.16
Dense STE	36.50 ± 4.58	98.67 ± 0.03	740.3 ± 113.9	13.53 ± 0.74
Dense GRU	100.00 ± 0.00	98.51 ± 0.12	533.5 ± 46.5	9.57 ± 0.15

6 DISCUSSION AND CONCLUSION

The method presented in this paper demonstrates how training time can be significantly decreased by decoupling task model fitting from learning a conditional computation schedule for it, as made possible by our independent learning signal for the gate. **MPC:** We hypothesize that the derivative-based supervision used by the two supervised methods helps them learn more efficiently than PPO's indirect reward-based supervision. In terms of training-time improvement, this is close to the ideal scenario for our method due to the forward-backward asymmetry. **Exactness:** The utility signal is equal to the first-order Taylor approximation of loss decrease in the autonomous and MPC case, while the exogenous input case is another layer of approximation. The extension to exogenous input scenarios, while less exact, produced promising results and leaves room for future work. The sequential MNIST setting where individual pixels are ingested at a time is not realistic, so future work on higher dimensional signals is needed. We expect that for signals that are slowly varying in their information content the proposed extension can be promising. **Schedule decoupling and distribution shift:** Batch-synchronized randomized training makes training efficient, but the labels are collected under a different distribution than where the learned gate and sequence then are applied. In these experiments "off-policy" gate training worked well, but this may not work equally well in more realistic situations.

Limitations and conclusions If the expensive networks' inputs are high-dimensional (such as image or video), the first-order approximation may not be sufficient, as the displacement vector and loss gradient directions may be too noisy as signals. Furthermore, the gate network needs to function in the input-space of the expensive computation, and so for high-dimensional data, designing it to be efficient may not be trivial. As seen in the inference time result of the exogenous input experiment, when the overhead of conditional computation and gate model execution are together as expensive as the skipped network itself, inference time benefits of this approach can be modest or the method may even be slower than the dense model. Still, the training even in this small-scale situation was shown to be significantly faster with our method. As a conclusion, the suggested method is particularly promising for cases where the forward computation is slow, and the sensitivities can be retrieved relatively fast, due to how the method can be used to reduce the number of forward passes during training.

AI USE STATEMENT

AI was **not** used to directly generate any of the text in the main text of the manuscript. AI was used for collecting and generating the experiment details' first draft found in the appendix, while the details were carefully checked and edited for correctness. AI tools were used to help with searching for more prior works. AI was used for partially generating some of the code for the experiments (reviewed by authors). AI was used for suggesting readability and language correctness improvements to the text during drafting.

We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

Code for reproducing the results is included in the supplementary material. Main implementation details are found in the paper.

ACKNOWLEDGMENTS

Omitted from initial submission.

REFERENCES

Brandon Amos, Ivan Jimenez, Jacob Sacks, Byron Boots, and J. Zico Kolter. Differentiable MPC for End-to-end Planning and Control. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

Jan-Lukas Archut and Burkhard Corves. Real-time multibody simulation of vehicle wheel suspensions of different topologies with elastokinematic properties. *Multibody System Dynamics*, 63 (1-2):255–271, February 2025. doi: 10.1007/s11044-024-09984-2.

Cody J Balos, Marcus Day, Lucas Esclapez, Anne M Felden, David J Gardner, Malik Hassanaly, Daniel R Reynolds, Jon S Rood, Jean M Sexton, Nicholas T Wimer, and Carol S Woodward. SUNDIALS time integrators for exascale applications with many independent systems of ordinary differential equations. *The International Journal of High Performance Computing Applications*, 39(1):123–146, January 2025. doi: 10.1177/10943420241280060.

Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation, August 2013. arXiv:1308.3432 [cs].

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Yash Katariya, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/jax-ml/jax`.

Gabriel Bravo-Palacios, Jianghan Zhang, Zachary Pestrikov, Brian Plancher, and Thomas Lew. TurboMPC: Fast, Scalable, and Differentiable Model Predictive Control on the GPU, June 2026. arXiv:2606.24039 [cs.RO].

Eivind Bøhn, Sebastien Gros, Signe Moe, and Tor Arne Johansen. Optimization of the Model Predictive Control Update Interval Using Reinforcement Learning∗. *IFAC-PapersOnLine*, 54 (14):257–262, January 2021.

Victor Campos, Brendan Jou, Xavier Giro-i Nieto, Jordi Torres, and Shih-Fu Chang. Skip RNN: Learning to Skip State Updates in Recurrent Neural Networks. In *International Conference on Learning Representations*, February 2018.

Paulina Conrad and Knut Graichen. A Sensitivity-Based Approach to Self-Triggered Nonlinear Model Predictive Control. *IEEE Access*, 12:153243–153252, 2024.

Fengying Dang, Dong Chen, Jun Chen, and Zhaojian Li. Event-Triggered Model Predictive Control With Deep Reinforcement Learning for Autonomous Driving. *IEEE Transactions on Intelligent Vehicles*, 9(1):459–468, January 2024. ISSN 2379-8904. doi: 10.1109/TIV.2023.3329785.

Nicholas J. Higham. The scaling and squaring method for the matrix exponential revisited. *SIAM Rev.*, 51(4):747–764, 2009. doi: 10.1137/090768539.

Marlis Hochbruck and Alexander Ostermann. Exponential integrators. *Acta Numerica*, 19:209–286, 2010.

Patrick Kidger and Cristian Garcia. Equinox: neural networks in JAX via callable PyTrees and filtered transformations. *Differentiable Programming workshop at Neural Information Processing Systems 2021*, 2021.

Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. 2, 2010. URL `http://yann.lecun.com/exdb/mnist/`.

Namhoon Lee, Thalaiyasingam Ajanthan, and Philip Torr. SNIP: SINGLE-SHOT NETWORK PRUNING BASED ON CONNECTION SENSITIVITY. In *International Conference on Learning Representations*, 2019.

Qingwei Lin. Revisiting Auxiliary Losses for Conditional Depth Routing: An Empirical Study, April 2026. arXiv:2604.17228 [cs.LG] version: 1.

Edward N Lorenz. Predictability: A problem partly solved. In *Proc. Seminar on predictability*, volume 1, pp. 1–18. Reading, 1996.

D. Q. Mayne, J. B. Rawlings, C. V. Rao, and P. O. M. Scokaert. Constrained model predictive control: Stability and optimality. *Automatica*, 36(6):789–814, June 2000.

Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference. In *International Conference on Learning Representations*, 2017.

Vryan Gil Palma, Andrea Suardi, and Eric C. Kerrigan. Sensitivity-based multistep MPC for embedded systems. *IFAC-PapersOnLine*, 48(23):360–365, January 2015.

Rui Ren and Shaoyuan Li. Learning Event-Triggered Model Predictive Control From Data With Hierarchical Reinforcement Learning. *International Journal of Robust and Nonlinear Control*, 36(1):139–154, 2026. ISSN 1099-1239. doi: 10.1002/rnc.70102.

Lawrence F. Shampine and Mark W. Reichelt. The MATLAB ODE Suite. *SIAM Journal on Scientific Computing*, 18(1):1–22, January 1997. doi: 10.1137/S1064827594276424.

Yueyuan Sui, Payal Mohapatra, Doğaç Eldenk, Haodong Yang, Yiting Zhang, Haoyan Zhang, Qi Zhu, and Stephen Xia. Modality-Aware Zero-Shot Pruning and Sparse Attention for Efficient Multimodal Edge Inference, April 2026. arXiv:2604.08971 [cs.LG] version: 1.

Jin Tao, Urmish Thakker, Ganesh Dasika, and Jesse Beu. Skipping RNN State Updates without Retraining the Original Model. In *Proceedings of the 1st Workshop on Machine Learning on Edge in Sensor Systems*, SenSys-ML 2019, pp. 31–36, New York, NY, USA, November 2019. Association for Computing Machinery.

Tong Zhao, Mingkun Lei, Yucheng Han, and Chi Zhang. BAG: Budget-Aware Gating for Diffusion Caching, August 2026. arXiv:2608.09231 [cs.CV] version: 2.

A Appendix

A.1 MPC experiment details

Choosing the best model version As the task cost can generally be reduced with more solves, the best model depends on whether sparse solves or task cost is weighted more. Here we chose the best version of each model based on their total number of correctly solved scenarios based on a separate validation set, while reporting the final results on a third, separate test set. Among evaluated checkpoints attaining the highest validation accuracy, we report the earliest. Training decisions and times refer to the selected model, not the complete experimental search. If the model did not refresh

at all in the A or C scenarios it was counted as a correct decision. If the model refreshed exactly once with a beneficial cost reduction in B, it was counted as a correct decision, irrespective of whether it occurred exactly on the impulse.

Task and planner. The state contains two-dimensional position and velocity. The unit-mass dynamics use timestep 0.05, with $F_x = 0$ and $|F_y| \le 2$. The goal has half-widths $(0.25, 0.5)$ and the costly circle has centre $(-1, 0)$ and radius 0.4. Running cost is $0.05(0.1d_x^2 + 10d_y^2 + 0.1\|F\|^2 + 300d_o^2)$, where d_x, d_y measure distance outside the goal and d_o measures penetration into the circle. The planner additionally uses terminal goal weight 100, excluded from reported task cost. The Amos iLQR solver uses float64, convexified search Hessians, at most 40 iterations and tolerance 10^{-6}. Planning horizons are 17 stages for A and 65 for B/C.

Data. Episodes last 16 steps in A and 64 in B/C. We sample 256 training specifications per scenario by varying initial position, vertical velocity, impulse time and magnitude, with random vertical mirroring. Validation and test each contain 32 independently sampled specifications per scenario. Specifications remain fixed, while changing the refresh schedule changes the resulting trajectory.

Horizontal velocity is fixed by $v_{x,0} = -p_{x,0}/(hT)$ with $h = 0.05$. The velocity impulse is $\Delta v_y = -\widetilde{m}$ in A and C. In B, it is adjusted using the remaining horizontal travel time to the circle's right edge:

$$\Delta v_y = -\frac{\widetilde{m}}{\sqrt{1 + \Delta t_{\text{exit}}^2}}, \qquad \Delta t_{\text{exit}} = \frac{-0.6 - p_{x,0}}{v_{x,0}} - h t_{\text{imp}}.$$

With probability $1/2$, the signs of $p_{y,0}$, $v_{y,0}$ and Δv_y are reversed together.

Table 3: Episode parameters before vertical mirroring. Continuous values are sampled uniformly. Impulse steps are sampled uniformly from the listed integers. The sampled magnitude is adjusted in B as described below.

Parameter	A	B	C
Initial p_x	$[-1.55, -1.45]$	$[-3.1, -2.9]$	$[-3.1, -2.9]$
Initial p_y	$[0.03, 0.07]$	$[-0.025, 0.025]$	$[-0.025, 0.025]$
Initial v_y	$[0.025, 0.075]$	$[0, 0.1]$	$[0.75, 0.85]$
Impulse step t_{imp}	$\{1, 2\}$	$\{14, \dots, 18\}$	$\{35, \dots, 38\}$
Sampled magnitude $\widetilde{m}$	$[0.30, 0.36]$	$[0.27, 0.294]$	$[0.30, 0.36]$
Episode steps T (fixed)	16	64	64

Proposed training. We collect six schedules per training specification and three per validation specification. The reported proposed model uses the first 128 training specifications per scenario. Each batch of eight shares a Bernoulli schedule with probability $p \sim \text{Beta}(20q, 20(1-q))$, where $q = 2/(T-1)$. The initial solve is mandatory. For the reported model, this produces 108,288 training decisions and 103,728 closed-step utility labels. The gate has two tanh layers of width 32. Its 44 inputs are current state, cached state, nominal-state residual and the next 16 two-dimensional actions. We fit utility-weighted BCE with refresh penalty 0.1, Adam learning rate 3×10^{-4} and minibatch size 128.

STE and PPO. STE uses the same gate architecture with hard Bernoulli decisions and a current-action output relaxation. Its initial refresh probability is 0.01, learning rate is 10^{-2} and gradient clipping is 0.5. Each update uses eight episodes per scenario. Closed-step counterfactual solves are batched across time in groups of up to 64. Stable-Baselines3 PPO uses separate two-layer width-64 actor and critic networks. Each iteration collects 5,952 decisions across 32 environments. PPO uses learning rate 3×10^{-4}, ten epochs, minibatches of 288, $\gamma = \lambda = 1$, clipping 0.2, entropy weight 0.01 and value weight 0.5. Both baselines penalize each refresh by 0.1.

Validation and timing. Rollout validation uses eight specifications per scenario every 2,500 proposed updates, 20 STE updates or five PPO iterations. Patience is four checks without an improved task-cost/solve trade-off. The cost and solve tolerances are $(0.001, 0.01)$ for proposed/STE and

$(0.005, 0.05)$ for PPO. The reported J_0 run stopped at 20,000 updates and PPO at 80 iterations under this rule. The reported STE run reached its 100-update budget. Final checkpoint selection uses A/B/C correctness on all 96 validation specifications. Proposed thresholds are $\{0.25, 0.5, 0.75\}$. STE also tests validation-score percentiles $\{90, 95, 98, 99.5\}$, while reported PPO uses deterministic actions. The residual threshold is selected by the same validation A/B/C correctness criterion, yielding a threshold of 1.0. All methods run on CPU. Proposed collection and PPO use four workers, while STE uses three scenario workers. Training times include collection and optimization through the selected checkpoint, excluding initial-plan preparation and rollout evaluation.

A.2 Exogenous input RNN experiment details

The results in Table 4 indicate that the problem can be trivially solved at refresh rates above 20% by any policy, but at all lower refresh rates than 10% the learned gate is consistently slightly better than the baseline policies, indicating learning of a meaningful gate network.

Table 4: Inference-time refresh policies on the same sparsely trained backbone, average ± std over three independently trained backbone and gate seeds

Target updates	Learned gate	Periodic	Random
1%	17.25 ± 0.98%	10.29 ± 0.02%	12.65 ± 1.01%
2%	38.47 ± 3.04%	34.81 ± 0.22%	30.22 ± 0.73%
3%	56.91 ± 4.24%	53.12 ± 0.22%	38.59 ± 0.96%
4%	68.92 ± 3.66%	49.38 ± 9.14%	51.60 ± 0.96%
5%	76.83 ± 3.18%	69.24 ± 2.97%	57.56 ± 2.05%
10%	90.61 ± 1.37%	90.44 ± 0.49%	81.81 ± 1.58%
20%	95.64 ± 0.65%	97.22 ± 0.38%	94.28 ± 0.53%

Architecture. All RNN models are implemented in JAX/Equinox (Bradbury et al., 2018; Kidger & Garcia, 2021). Our SkipRNN reimplementation uses the shared backbone architecture and inputs. We use a four-layer GRU with 64 hidden units per layer and a linear $64 \to 10$ classifier on the final hidden state. A refresh updates all four layers, while a skipped step leaves their states unchanged. Each backbone receives seven inputs: the current pixel, four sinusoidal row/column features, the pixel value at the previous refresh, and the number of pixels since the previous refresh divided by 16.

The proposed gate and the dense-STE gate are affine $75 \to 1$ maps followed by a sigmoid, for a total of 76 parameters. Their inputs are the final hidden state (64), the current and cached pixel values (2), the current and cached positional features (8), and the normalized elapsed count (1). Both gates make Bernoulli refresh decisions. The SkipRNN gate is an affine $64 \to 1$ map followed by a sigmoid (65 parameters), applied to the updated final hidden state. We retain SkipRNN's deterministic accumulator gate, adapting its backbone to the shared architecture and inputs above.

Data and optimization. MNIST images are flattened into sequences of 784 pixels and normalized to $[0, 1]$. We use 55,000 training, 5,000 validation, and 10,000 test images. Backbones are trained with 10-class cross-entropy using Adam with learning rate 10^{-3} and batch size 128. Gradients are clipped to a global norm of 1. We evaluate on the validation set every 430 optimization steps and stop training after three evaluations without improvement, restoring the checkpoint with the best validation performance.

We add 200 dense-update warm-up steps with the gate bypassed to SkipRNN and dense STE, then train jointly with a refresh penalty of 10^{-4} per update.

Proposed two-stage training. During backbone training, each minibatch uses a refresh probability $p \sim \text{Beta}(4, 16)$. A Bernoulli refresh schedule is sampled from p and shared across the minibatch, with the first pixel always refreshed. This gives an average refresh probability of 20%.

After backbone training, we freeze its parameters and collect utility labels at refresh counts $\{39, 78, 157, 314, 627\}$. For each count, we use eight batches of 64 trajectories. The gate is trained on standardized features with a weighted logistic loss. The binary target is $\mathbf{1}[U > \rho]$, with sample

weights proportional to $|U - \rho|$ and $\rho \approx 0.14$ corresponding to the 90th percentile of positive utilities from a calibration run. Because the gate is linear, we optimize it with L-BFGS-B and retain the iterate with the lowest validation loss.

Evaluation. The same fitted gate is used for all refresh budgets. For each target budget, we calibrate only a scalar logit bias on 512 validation images so that the empirical refresh rate matches the requested rate. Test metrics average three Bernoulli realizations for proposed and dense STE. SkipRNN and dense GRU are deterministic. Results are reported as the mean and standard deviation across three training seeds.

A.3 Efficient special case on MPC

While the specification of the utility signal presented in Section 4.2 is conveniently constructable from any differentiable planner, the computation of J_0 still requires a number of pullbacks to construct, according to dimensionality of the actions. However, iLQR solvers such as the one used in (Amos et al., 2018) internally already compute a useful local sensitivity that can be extracted practically at no extra computational cost, completely skipping the need for computing the extra pullbacks.

Let $\bar{x}_k^c$ be the predicted state at the kth step, as predicted by dynamics f when evaluated starting from state x_c. During optimization, iLQR computes as an intermediate result the matrix $G_k^c \approx \frac{da_k}{d\bar{x}_k}$ depicting how a locally optimal action would change if the predicted kth state would change. Note how this is a derivative of kth step action with respect to kth predicted state, instead of the previously discussed sensitivity of the plan's first action to the plan inputs.

To formulate the utility signal, in this derivation we use the difference of the predicted states and actual observed states, $x_t - \bar{x}_k^c$. Multiplying this difference with the derivative G gives the predicted action correction $\Delta a_t = G_k^c(x_t - \bar{x}_k^c)$ which combined with the gradient λ_t^T gives

$$U_t^G = -\lambda_t^T G_k^c (x_t - \bar{x}_k^c). \tag{3}$$

In this formulation, if the observed state follows the trajectory predicted by the cached plan, the utility is exactly zero, possibly removing false positive opening signals.

A.4 Autonomous dynamics example

The implementation of both models is done in JAX (Bradbury et al., 2018), where the standard library implementation performs a standard scaling-and-squaring algorithm (Higham, 2009), which involves conditional branching for the scaling part, a linear solve to retrieve a Padé approximant, and finally dense matrix multiplications for the squaring part. The dataset consists of 128-step trajectories from Lorenz (1996) at different batch sizes (4, 32, 256) and state dimensions $D \in \{32, 64, 128\}$.

The proposed method in this experiment computes a sparse forward pass, and parallel pullbacks for every cotangent within the same skipped segment, in order to efficiently retrieve the RecompUtility signal, As this experiment does not apply a gate network anywhere, it does not show the overhead evaluating a gate network would add at test-time, thus only measuring the training-time part for retrieving gradients for learning the gate efficiently. The baseline STE is implemented with ordinary reverse-mode automatic differentiation with JAX (Bradbury et al., 2018)

The reported figures are built from averages (and standard deviations) calculated after 2 ignored warm-up iterations followed by 8 consecutive repeats for each data point.

A more comprehensive evaluation than in the main text is found in Figure 4, including batch sizes (4, 32, 256).

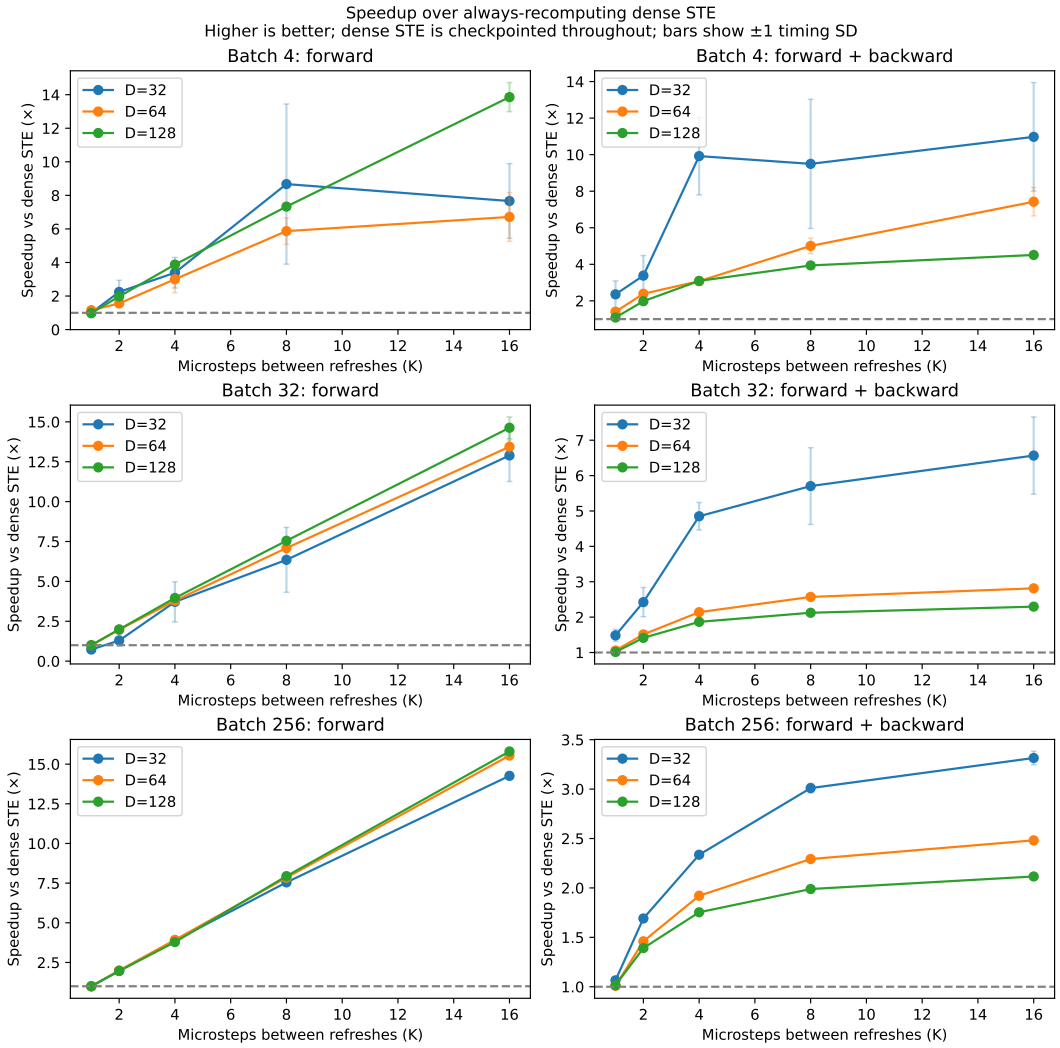

Figure 4: Computational speed benchmark for producing the same closed-step gradients as STE up to numerical precision, with forward and fwd+bwd timings separately on the left and right respectively. As expected, the proposed method does not offer computational advantage when $K = 1$ where the expensive computation is performed on every step. The computational benefit increases with the sparsity of performing the expensive computation.