PoseAgent: Budget-Constrained 6D Object Pose Estimation via Reinforcement Learning

Alexander Krull[1], Eric Brachmann[1], Sebastian Nowozin[2],
Frank Michel[1], Jamie Shotton[2], Carsten Rother[1]
[1] TU Dresden, [2] Microsoft

State-of-the-art computer vision algorithms often achieve efficiency by making discrete choices about which hypotheses to explore next. This allows allocation of computational resources to promising candidates, however, such decisions are non-differentiable. As a result, these algorithms are hard to train in an end-to-end fashion. In this work we propose to learn an efficient algorithm for the task of 6D object pose estimation. Our system optimizes the parameters of an existing state-of-the art pose estimation system using reinforcement learning, where the pose estimation system now becomes the stochastic policy, parametrized by a CNN. Additionally, we present an efficient training algorithm that dramatically reduces computation time. We show empirically that our learned pose estimation procedure makes better use of limited resources and improves upon the state-of-the-art on a challenging dataset. Our approach enables differentiable end-to-end training of complex algorithmic pipelines and learns to make optimal use of a given computational budget.

1. Introduction

Many tasks in computer vision involve *learning a function*, usually learning to predict a desired output label given an input image. Advances in deep learning have led to huge progress in solving such tasks. In particular, convolutional neural networks (CNNs) work well when trained over large training sets using gradient descent methods to minimize the expected loss between the predictions and the ground truth labels.

However, important computer vision systems take the form of *algorithms* rather than being a simple differentiable function: sliding window search, superpixel partioning, particle filters, and classification cascades are examples of algorithms realizing complex non-continuous functions.

The *algorithmic approach* is especially useful in situations where computational budget is limited: an algorithm can dynamically assign its budget to solving different aspects of the problem, for example, to take shortcuts in order to spend computation on more promising solutions at the expense of less promising ones. We would like to *learn the algorithm*. Unfortunately, the hard decisions taken in most algorithmic approaches are non-differentiable, and this means that the structure and parameters of these efficient algorithms cannot be easily learned from data.

Reinforcement learning (RL) [22] offers a possible solution to learning algorithms. We view the algorithm as the *policy* of an RL agent, *i.e.* a description of dynamic sequential behaviour. RL provides a framework to learn the parameters of such behaviour with the goal of maximizing an expected reward, for example, the accuracy of the algorithm output. We apply this perspective on an algorithmic computer vision method. In particular, we address the problem of 6D object pose estimation and use RL to learn the parameters of a deep algorithmic pipeline to provide the best possible accuracy given a limited computational budget.

Object pose estimation is the task of estimating from an image the 3D translation (position) and 3D rotation (orientation) of a specific object relative to its environment. This task is important in many applications such as robotics and augmented reality where the efficient use of a limited computation budget is an important requirement. A particular challenge are small, textureless and partially occluded objects in a cluttered environment (see Fig. 1).

State-of-the-art pose systems such as the system of Krull *et al.* [12] generate a pool of pose hypotheses, then score each hypothesis using a pre-trained CNN. The subset of high-scoring hypotheses get refined and ultimately the highest-scoring hypothesis is returned as the answer. Computationally the refinement step is the most expensive, and there is a trade-off between the number of refinements allowed and the expected quality of the result.

Ideally, one would train such state-of-the-art system end-to-end in order to learn how to use the optimal number

PoseAgent: Budget-Constrained 6D Object Pose Estimation via Reinforcement Learning

Alexander Krull[1], Eric Brachmann[1], Sebastian Nowozin[2],
Frank Michel[1], Jamie Shotton[2], Carsten Rother[1]
[1] TU Dresden, [2] Microsoft

State-of-the-art computer vision algorithms often achieve efficiency by making discrete choices about which hypotheses to explore next. This allows allocation of computational resources to promising candidates, however, such decisions are non-differentiable. As a result, these algorithms are hard to train in an end-to-end fashion. In this work we propose to learn *an efficient algorithm for the task of 6D object pose estimation. Our system optimizes the parameters of an existing state-of-the art pose estimation system using reinforcement learning, where the pose estimation system now becomes the* stochastic policy*, parametrized by a CNN. Additionally, we present an efficient training algorithm that dramatically reduces computation time. We show empirically that our learned pose estimation procedure makes better use of limited resources and improves upon the state-of-the-art on a challenging dataset. Our approach enables differentiable end-to-end training of complex algorithmic pipelines and learns to make optimal use of a given computational budget.*

1. Introduction

Many tasks in computer vision involve *learning a function*, usually learning to predict a desired output label given an input image. Advances in deep learning have led to huge progress in solving such tasks. In particular, convolutional neural networks (CNNs) work well when trained over large training sets using gradient descent methods to minimize the expected loss between the predictions and the ground truth labels.

However, important computer vision systems take the form of *algorithms* rather than being a simple differentiable function: sliding window search, superpixel partioning, particle filters, and classification cascades are examples of algorithms realizing complex non-continuous functions.

The *algorithmic approach* is especially useful in situations where computational budget is limited: an algorithm can dynamically assign its budget to solving different aspects of the problem, for example, to take shortcuts in order to spend computation on more promising solutions at the expense of less promising ones. We would like to *learn the algorithm*. Unfortunately, the hard decisions taken in most algorithmic approaches are non-differentiable, and this means that the structure and parameters of these efficient algorithms cannot be easily learned from data.

Reinforcement learning (RL) [22] offers a possible solution to learning algorithms. We view the algorithm as the *policy* of an RL agent, *i.e.* a description of dynamic sequential behaviour. RL provides a framework to learn the parameters of such behaviour with the goal of maximizing an expected reward, for example, the accuracy of the algorithm output. We apply this perspective on an algorithmic computer vision method. In particular, we address the problem of 6D object pose estimation and use RL to learn the parameters of a deep algorithmic pipeline to provide the best possible accuracy given a limited computational budget.

Object pose estimation is the task of estimating from an image the 3D translation (position) and 3D rotation (orientation) of a specific object relative to its environment. This task is important in many applications such as robotics and augmented reality where the efficient use of a limited computation budget is an important requirement. A particular challenge are small, textureless and partially occluded objects in a cluttered environment (see Fig. 1).

State-of-the-art pose systems such as the system of Krull *et al.* [12] generate a pool of pose hypotheses, then score each hypothesis using a pre-trained CNN. The subset of high-scoring hypotheses get refined and ultimately the highest-scoring hypothesis is returned as the answer. Computationally the refinement step is the most expensive, and there is a trade-off between the number of refinements allowed and the expected quality of the result.

Ideally, one would train such state-of-the-art system end-to-end in order to learn how to use the optimal number of refinements to maximize the expected success of pose estimation. Unfortunately, treating the system as a black box with parameters to optimize is impossible for two reasons: (i) each selection process is non-differentiable with respect to the scoring function; and (ii) the loss used to determine whether an estimated pose is correct is also non-differentiable.

We recast pose estimation as an RL problem in order to overcome these difficulties. We model the pose infer- ence process as an RL agent which we call *PoseAgent*. PoseAgent is granted more flexibility than the original system: it is given a fixed budget of refinement steps, and is allowed to manipulate its hypothesis pool by selecting individual poses for refinement, until the budget is spent. In our PoseAgent model each decision follows a probability distribution over possible actions. This distribution is called the policy and we can differentiate and optimize this continuous policy through the stochastic policy gradient approach [23]. As a result of this stochastic approach the final pose estimate becomes a random variable, and each run of PoseAgent will produce a slightly different result.

This policy gradient approach is very general and does not require differentiability of the used loss function. As a consequence we can directly take the gradient with respect to the expected loss of interest, *i.e.* the number of correctly estimated poses. Training in policy gradient methods can be difficult due to the additional variance of estimated gradients [7, 23], because the additional randomness leads to a bigger variance in the estimated gradients. To overcome this problem we propose an efficient training algorithm that radically reduces the variance during training compared to a naïve technique.

We compare our approach to the state-of-the-art [12] and achieve substantial improvements in accuracy, while using the same or smaller average budget of refinement steps compared to [12]. In summary our contributions are:

- To the best of our knowledge, we are the first to apply a policy gradient approach to the object pose estimation problem.
- Our approach allows the use of a non-differentiable reward function corresponding to the original evaluation criterion.
- We present an efficient training algorithm that dramatically reduces the variance during training.
- We improve significantly upon the best published results on the dataset.

2. Related Work

Below, we first discuss approaches for 6D pose estimation, focusing in particular on object coordinate prediction methods, and then provide a short review of RL methods used in a setting similar to ours.

2.1. Pose Estimation

There is a large variety of approaches for 6D object pose estimation. Traditionally, approaches based on sparse features [14, 15] have been successful, but work well only for textured objects. Other approaches include template-based methods [9, 19], voting schemes [6, 10], and CNN-based direct pose regression [8].

We focus on the line of work called object coordinate regression [3], which provides the basic framework for our approach. Object coordinate regression was originally proposed for human body pose estimation [24] and camera localization [20]. In [3] a random forest provides a dense pixel-wise prediction for 6D object pose prediction. At each pixel, the forest predicts whether and where the pixel is located on the surface of the object. One can then efficiently generate pose hypotheses by sampling a small set of pixels and combining the forest predictions with depth information from an RGB-D camera.

The object coordinate regression methods in [3, 12, 17] score these hypotheses by comparing rendered and observed image patches. While [3, 17] use a simple pixel wise distance function, [12] propose a learned comparison: a CNN compares rendered and observed images and out- puts an energy value representing a parameter of the pos- terior distribution in pose space. Despite their differences in the particular scoring functions, [3, 17, 12] use the same inference technique to arrive at the final pose estimate: they all refine the best hypotheses, re-score them, and output the best one as their final choice. Our PoseAgent approach can be seen as a generalization of this algorithm, in which the agent selects the hypotheses for refinement repeatedly, each time being able to make a more informed choice.

The work of Krull *et al.* [12] is the most closely related to our work. We use a similar CNN construction as Krull *et al*., feeding both rendered and observed image patches into to our CNN. However, we use the output of the CNN as a parameter of the stochastic policy that controls the behaviour of our pose agent. Moreover, while the training process in [12] is seen as learning the posterior distribution, which is then maximized during testing using the fixed inference procedure, our training process instead modifies the behaviour of the agent directly in order to maximize the number of correctly estimated poses.

2.2. Reinforcement Learning in Similar Tasks

RL has traditionally been successful in areas like robotics [21], control [1], advertising, network routing, or playing games. While the application of RL seems natu- ral for such cases where real agents and environments are involved, RL is increasingly being successfully applied in computer vision systems where the interpretation of the system as an agent interacting with an environment is not always so intuitive. While we are to our knowledge the first of refinements to maximize the expected success of pose estimation. Unfortunately, treating the system as a black box with parameters to optimize is impossible for two reasons: (i) each selection process is non-differentiable with respect to the scoring function; and (ii) the loss used to determine whether an estimated pose is correct is also non-differentiable.

We recast pose estimation as an RL problem in order to overcome these difficulties. We model the pose inference process as an RL agent which we call *PoseAgent*. PoseAgent is granted more flexibility than the original system: it is given a fixed budget of refinement steps, and is allowed to manipulate its hypothesis pool by selecting individual poses for refinement, until the budget is spent. In our PoseAgent model each decision follows a probability distribution over possible actions. This distribution is called the policy and we can differentiate and optimize this continuous policy through the stochastic policy gradient approach [23]. As a result of this stochastic approach the final pose estimate becomes a random variable, and each run of PoseAgent will produce a slightly different result.

This policy gradient approach is very general and does not require differentiability of the used loss function. As a consequence we can directly take the gradient with respect to the expected loss of interest, *i.e.* the number of correctly estimated poses. Training in policy gradient methods can be difficult due to the additional variance of estimated gradients [7, 23], because the additional randomness leads to a bigger variance in the estimated gradients. To overcome this problem we propose an efficient training algorithm that radically reduces the variance during training compared to a naïve technique.

We compare our approach to the state-of-the-art [12] and achieve substantial improvements in accuracy, while using the same or smaller average budget of refinement steps compared to [12]. In summary our **contributions** are:

- To the best of our knowledge, we are the first to apply a policy gradient approach to the object pose estimation problem.
- Our approach allows the use of a non-differentiable reward function corresponding to the original evaluation criterion.
- We present an efficient training algorithm that dramatically reduces the variance during training.
- We improve significantly upon the best published results on the dataset.

2. Related Work

Below, we first discuss approaches for 6D pose estimation, focusing in particular on object coordinate prediction methods, and then provide a short review of RL methods used in a setting similar to ours.

2.1. Pose Estimation

There is a large variety of approaches for 6D object pose estimation. Traditionally, approaches based on sparse features [14, 15] have been successful, but work well only for textured objects. Other approaches include template-based methods [9, 19], voting schemes [6, 10], and CNN-based direct pose regression [8].

We focus on the line of work called object coordinate regression [3], which provides the basic framework for our approach. Object coordinate regression was originally proposed for human body pose estimation [24] and camera localization [20]. In [3] a random forest provides a dense pixel-wise prediction for 6D object pose prediction. At each pixel, the forest predicts whether and where the pixel is located on the surface of the object. One can then efficiently generate pose hypotheses by sampling a small set of pixels and combining the forest predictions with depth information from an RGB-D camera.

The object coordinate regression methods in [3, 12, 17] score these hypotheses by comparing rendered and observed image patches. While [3, 17] use a simple pixel wise distance function, [12] propose a learned comparison: a CNN compares rendered and observed images and outputs an energy value representing a parameter of the posterior distribution in pose space. Despite their differences in the particular scoring functions, [3, 17, 12] use the same inference technique to arrive at the final pose estimate: they all refine the best hypotheses, re-score them, and output the best one as their final choice. Our PoseAgent approach can be seen as a generalization of this algorithm, in which the agent selects the hypotheses for refinement repeatedly, each time being able to make a more informed choice.

The work of Krull *et al.* [12] is the most closely related to our work. We use a similar CNN construction as Krull *et al.*, feeding both rendered and observed image patches into to our CNN. However, we use the output of the CNN as a parameter of the stochastic policy that controls the behaviour of our pose agent. Moreover, while the training process in [12] is seen as learning the posterior distribution, which is then maximized during testing using the fixed inference procedure, our training process instead modifies the behaviour of the agent directly in order to maximize the number of correctly estimated poses.

2.2. Reinforcement Learning in Similar Tasks

RL has traditionally been successful in areas like robotics [21], control [1], advertising, network routing, or playing games. While the application of RL seems natural for such cases where real agents and environments are involved, RL is increasingly being successfully applied in computer vision systems where the interpretation of the system as an agent interacting with an environment is not always so intuitive. While we are to our knowledge the first

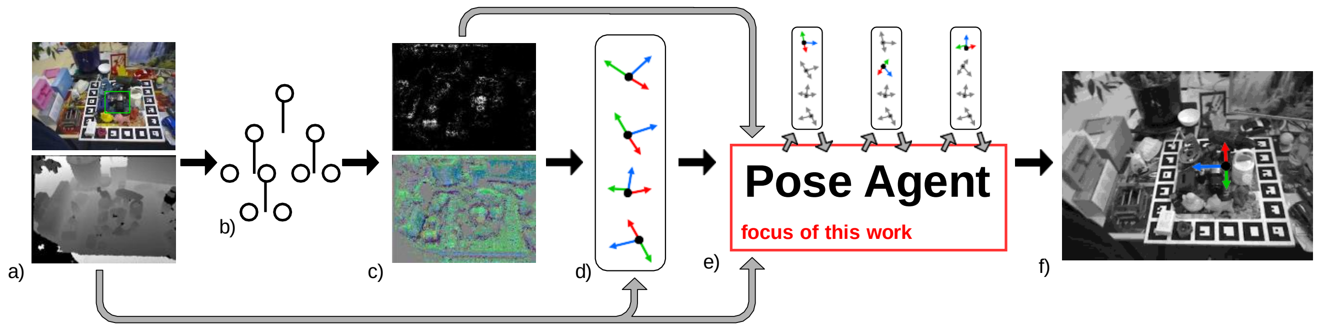

Figure 1. The pose estimation pipeline: a) The input of our system is an RGB-D image. We are interested in the pose of the camera highlighted by the green box. b) Similar to [3], the image is processed by a random forest. c) The forest outputs dense predictions of object probabilities (top) and object coordinates (bottom). The object coordinates are mapped to the RGB cube for visualization. d) We use the predictions together with the depth information to sample a pool of pose hypotheses H^0. e) An RL agent manipulates the hypotheses pool by repeatedly selecting individual hypotheses to refine. This is the focus of this paper. f) The agent outputs a final pose estimate $\tilde{H}$.

to apply RL for 6D object pose estimation, there are several recent papers that apply RL for 2D object detection and recognition [18, 5, 16, 2].

In [18, 5], an agent shifts its area of attention over the image until it makes a final decision. Instead of moving a single 2D area of attention over search space like [18, 5], we work with a pool of multiple 6D pose hypotheses. The agent in [16] focuses its attention by moving a 2D fixation point, though operates on a set of precomputed image regions to gather information and make a final decision. Our agent instead manipulates its hypothesis pool by refining individual hypotheses.

Caicedo *et al.* [5] use Q-learning, in which the CNN predicts the quality of the available state-action pairs. Mnih *et al.* [18] and Mathe *et al.* [16] use a different RL approach based on stochastic policy gradient, in which the behaviour of the agent is directly learned to maximize an expected reward. We follow [18, 5] in using stochastic policy gradient, which allows us to use a non-differentiable reward function, directly corresponding to the final success criterion used during evaluation.

3. Method

In this section, we first define the pose estimation task and briefly review the pose estimation pipeline from [3, 12, 4]. We then continue to describe PoseAgent, our reinforcement learning agent, designed to solve the same problem. Finally, we discuss how to train our agent, introducing our new, efficient training algorithm.

3.1. Pose Estimation Pipeline

We begin by describing the object pose estimation task. Given an RGB-D image x we are interested in localizing a specific, known, rigid object (Fig. 1a). We assume that exactly one object instance is present in the scene. Our goal is to estimate the true pose H^* of the instance, *i.e.* its position in space as well as its orientation. The pose has a total of six degrees of freedom, three for translation and three for rotation. We define the pose as the rigid body transformation that maps a point from the local coordinate system of the object to the coordinate system of the camera.

Our method is based on the work of Krull *et al.* [12]. As in [12], we use an intermediate image representation called object coordinates. By looking at small patches of the RGB-D input image, a random forest (Fig. 1b) provides two predictions for every pixel i. Each tree predicts an object probability $p_i \in [0, 1]$ as well as a set of object coordinates y_i (Fig. 1c). The object probability p_i describes whether the pixel is believed to be part of the object or not. The object coordinates y_i represent the predicted position of the pixel on the surface of the object, *i.e.* its 3D coordinates in the local coordinate system of the object.

Again following [12], we use these forest predictions in a RANSAC-inspired sampling scheme to generate pose hypotheses. We repeatedly sample three pixels from the image according to the object probabilities p_i. By combining the predicted object coordinate y_i with the camera coordinates of the pixels (calculated from the depth channel of the input image), we obtain three 3D-3D correspondences. We calculate a pose hypothesis from these correspondences using the Kabsch algorithm [11]. We sample a fixed number N of hypotheses, which are combined in hypothesis pool $H^0 = (H^0_1 \ldots H^0_N)$ (Fig. 1d). The upper index denotes time steps which we will use later in our algorithm.

Krull *et al.* [12] proposed the following rigid scheme for pose optimization. All hypotheses are scored and the 25 top-scoring hypotheses are refined. Then, the refined hypotheses are scored again, and the best scoring hypothesis is returned as the final pose estimate of the algorithm.

Our paper focuses on improving the process by which the correct pose is found, starting from the same initial hypothesis pool. We propose to use an RL agent (Fig. 1e) to

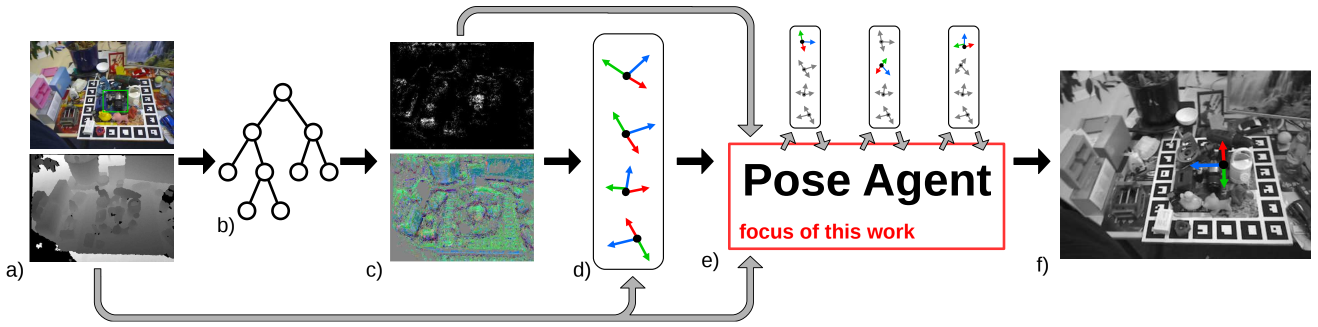

Figure 1. The pose estimation pipeline: a) The input of our system is an RGB-D image. We are interested in the pose of the camera highlighted by the green box. b) Similar to [3], the image is processed by a random forest. c) The forest outputs dense predictions of object probabilities (top) and object coordinates (bottom). The object coordinates are mapped to the RGB cube for visualization. d) We use the predictions together with the depth information to sample a pool of pose hypotheses $\boldsymbol{H}^0$. e) An RL agent manipulates the hypotheses pool by repeatedly selecting individual hypotheses to refine. This is the focus of this paper. f) The agent outputs a final pose estimate $\tilde{H}$.

to apply RL for 6D object pose estimation, there are several recent papers that apply RL for 2D object detection and recognition [18, 5, 16, 2].

In [18, 5], an agent shifts its area of attention over the image until it makes a final decision. Instead of moving a single 2D area of attention over search space like [18, 5], we work with a pool of multiple 6D pose hypotheses. The agent in [16] focuses its attention by moving a 2D fixation point, though operates on a set of precomputed image regions to gather information and make a final decision. Our agent instead manipulates its hypothesis pool by refining individual hypotheses.

Caicedo *et al.* [5] use Q-learning, in which the CNN predicts the quality of the available state-action pairs. Mnih *et al.* [18] and Mathe *et al.* [16] use a different RL approach based on stochastic policy gradient, in which the behaviour of the agent is directly learned to maximize an expected reward. We follow [18, 5] in using stochastic policy gradient, which allows us to use a non-differentiable reward function, directly corresponding to the final success criterion used during evaluation.

3. Method

In this section, we first define the pose estimation task and briefly review the pose estimation pipeline from [3, 12, 4]. We then continue to describe PoseAgent, our reinforcement learning agent, designed to solve the same problem. Finally, we discuss how to train our agent, introducing our new, efficient training algorithm.

3.1. Pose Estimation Pipeline

We begin by describing the object pose estimation task. Given an RGB-D image $\boldsymbol{x}$ we are interested in localizing a specific, known, rigid object (Fig. 1a). We assume that exactly one object instance is present in the scene. Our goal is to estimate the true pose H^* of the instance, *i.e.* its position in space as well as its orientation. The pose has a total of six degrees of freedom, three for translation and three for rotation. We define the pose as the rigid body transformation that maps a point from the local coordinate system of the object to the coordinate system of the camera.

Our method is based on the work of Krull *et al.* [12]. As in [12], we use an intermediate image representation called object coordinates. By looking at small patches of the RGB-D input image, a random forest (Fig. 1b) provides two predictions for every pixel i. Each tree predicts an object probability $p_i \in [0, 1]$ as well as a set of object coordinates $\boldsymbol{y}_i$ (Fig. 1c). The object probability p_i describes whether the pixel is believed to be part of the object or not. The object coordinates $\boldsymbol{y}_i$ represent the predicted position of the pixel on the surface of the object, *i.e.* its 3D coordinates in the local coordinate system of the object.

Again following [12], we use these forest predictions in a RANSAC-inspired sampling scheme to generate pose hypotheses. We repeatedly sample three pixels from the image according to the object probabilities p_i. By combining the predicted object coordinate $\boldsymbol{y}_i$ with the camera coordinates of the pixels (calculated from the depth channel of the input image), we obtain three 3D-3D correspondences. We calculate a pose hypothesis from these correspondences using the Kabsch algorithm [11]. We sample a fixed number N of hypotheses, which are combined in hypothesis pool $\boldsymbol{H}^0 = (H_1^0 \dots H_N^0)$ (Fig. 1d). The upper index denotes time steps which we will use later in our algorithm.

Krull *et al.* [12] proposed the following rigid scheme for pose optimization. All hypotheses are scored and the 25 top-scoring hypotheses are refined. Then, the refined hypotheses are scored again, and the best scoring hypothesis is returned as the final pose estimate of the algorithm.

Our paper focuses on improving the process by which the correct pose is found, starting from the same initial hypothesis pool. We propose to use an RL agent (Fig. 1e) to

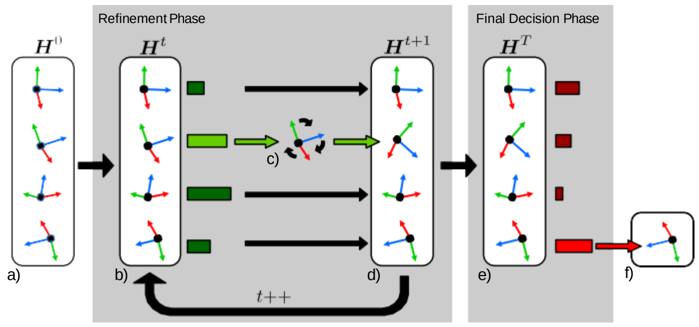

Figure 2. The pose agent inference process: a) The initial pool of hypotheses is sampled and handed to the agent. b) The agent selects a hypotheses $H^t_{a^t}$ by sampling from the policy $\pi(a^t|S^t;\theta)$. c) The selected hypothesis is refined. d) If refinement budget is left, the refinement phase continues. If the budget is exhausted a final selection is made. e) The final selection is made by sampling from the policy $\pi(a^T|S^T;\theta)$. f) The selected pose $H^T_{a^T}$ is output as final pose estimate.

dynamically decide which hypothesis to refine next, in order to make most efficient use of a given computational budget. When its budget is exhausted, the agent selects a final pose estimate (Fig. 1f).

3.2. PoseAgent

We now describe our RL agent, PoseAgent, and how it performs inference. An overview of the process can be found in Fig. 2. The agent operates in two phases: (i) the refinement phase, in which the agent chooses individual hypotheses to undergo the expensive refinement step; and (ii) the final decision phase, in which it has to decide which pose should be selected as final output. In the following, we discuss both phases in detail.

Inference begins with the refinement phase. The pose agent starts with a pool $H^0 = (H^0_1 \dots H^0_N)$ of hypotheses which have been generated as described in Sec. 3.1, and a fixed budget B^0 of possible refinement steps.

At each time step t, the agent chooses one hypothesis index a^t, which we call an action. The chosen hypothesis is refined and the next time step begins. We limit the maximum number of times the agent may choose the same hypothesis for refinement to $\tau_{\max}$. Hence, over time, the pool of actions (resp. hypotheses) the agent may choose for refinement decreases. We denote the set of possible actions $A^t = \{a \in \{1,\dots,N\} | \tau^t_a < \tau_{\max}\}$, where τ^t_a denotes how many times hypothesis a has been refined before time t. Subsequently, the agent modifies the hypothesis pool by refining hypothesis $H^{t+1}_{a^t} = g(H^t_{a^t})$, where $g(\cdot)$ is the refinement function. All other hypotheses remain unchanged $H^{t+1}_a = H^t_a \forall a \neq a^t$.

We perform refinement as follows (see also [12]). We render the object in pose $H^t_{a^t}$. Each pixel within the rendered mask is tested for being an inlier.[1] All inlier pixels are used to re-calculate the pose with the Kabsch algorithm. We repeat this procedure multiple times for the single, chosen hypothesis until the number of inlier pixels stops increasing or until the number m^t of executed refinement steps exceeds a maximum $m_{\max}$. The budget is decreased by the number of refinement steps performed, $B^{t+1} = B^t - m^t$. The agent proceeds choosing refinement actions until $B^t < m_{\max}$, in which case further refinement may exceed the total budget B^0 of refinement steps.

When this point has been reached, the refinement phase terminates, and the agent enters the final decision phase in which the agent chooses a hypothesis as the final output. We denote the final action as $a^T \in \{1\dots N\}$ and the final pose estimate as $\tilde{H} = H^T_{a^T}$. The agent receives a reward of $r = 1$ in case the pose is correct or a negative reward of $r = -1$ otherwise. We use the pose correctness criterion from [9].

In the following, we describe how the agent makes its decisions. During both, the refinement phase and the final decision phase, the agent chooses from the hypothesis pool. We describe the agent behaviour by a probability distribution $\pi(a^t|S^t;\theta)$ referred to as "policy". Given the current state S^t, which contains information about the hypothesis pool and the input image x, the agent selects a hypothesis by drawing a sample from the policy. The vector θ of learnable parameters consists of CNN weights (described in Sec. 3.2.3). We will first give details on the state space S^t before explaining policy $\pi(a^t|S^t;\theta)$.

3.2.1 State Space

We model our state space in a way that allows us to use our new, efficient training algorithm, described in Sec. 3.3.1. We assume that the current state S^t of the hypothesis pool decomposes as $S^t = (s^t_1,\dots s^t_N)$, where s^t_a will be called the state of hypothesis H^t_a. The state of an hypothesis contains the original input image x, the forest prediction z for the image, the pose hypothesis H^t_a, as well as a vector f^t_a of additional context features of the hypothesis (see Sec. 3.2.3). In summary, this gives $s^t_a = (x, z, H^t_a, f^t_a)$.

3.2.2 Policy

Our agent makes its decisions using a softmax policy. The probability of choosing a particular action a^t during the refinement phase is given by

$$\pi(a^t|S^t;\theta) = \frac{\exp(E(s^t_{a^t};\theta))}{\sum_{a\in A^t}\exp(E(s^t_a;\theta))}, \qquad (1)$$

[1] A pixel i is tested for being an inlier for pose H by transforming its predicted object coordinates y_i to camera space using H. If the Euclidean distance between resulting camera coordinates and the observed coordinates at the pixel is below a threshold the pixel is considered an inlier.

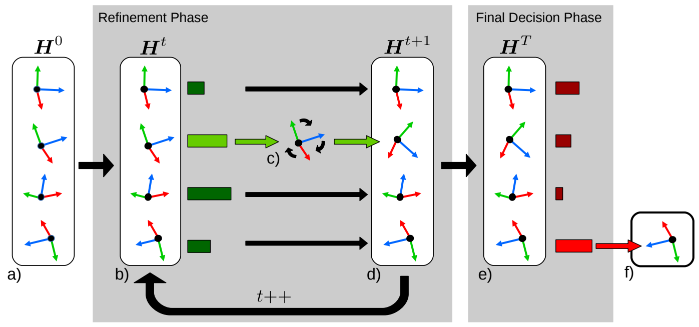

Figure 2. The pose agent inference process: a) The initial pool of hypotheses is sampled and handed to the agent. b) The agent selects a hypotheses $H^t_{a^t}$ by sampling from the policy $\pi\left(a^t|S^t;\boldsymbol{\theta}\right)$. c) The selected hypothesis is refined. d) If refinement budget is left, the refinement phase continues. If the budget is exhausted a final selection is made. e) The final selection is made by sampling from the policy $\pi\left(a^T|S^T;\boldsymbol{\theta}\right)$. f) The selected pose $H^T_{a^T}$ is output as final pose estimate.

dynamically decide which hypothesis to refine next, in order to make most efficient use of a given computational budget. When its budget is exhausted, the agent selects a final pose estimate (Fig. 1f).

3.2. PoseAgent

We now describe our RL agent, PoseAgent, and how it performs inference. An overview of the process can be found in Fig. 2. The agent operates in two phases: (i) the refinement phase, in which the agent chooses individual hypotheses to undergo the expensive refinement step; and (ii) the final decision phase, in which it has to decide which pose should be selected as final output. In the following, we discuss both phases in detail.

Inference begins with the **refinement phase**. The pose agent starts with a pool $\boldsymbol{H}^0 = (H^0_1 \dots H^0_N)$ of hypotheses which have been generated as described in Sec. 3.1, and a fixed budget B^0 of possible refinement steps.

At each time step t, the agent chooses one hypothesis index a^t, which we call an action. The chosen hypothesis is refined and the next time step begins. We limit the maximum number of times the agent may choose the same hypothesis for refinement to $\tau_{\max}$. Hence, over time, the pool of actions (resp. hypotheses) the agent may choose for refinement decreases. We denote the set of possible actions $A^t = \{a \in \{1, \dots, N\} | \tau^t_a < \tau_{\max}\}$, where τ^t_a denotes how many times hypothesis a has been refined before time t. Subsequently, the agent modifies the hypothesis pool by refining hypothesis $H^{t+1}_{a^t} = g(H^t_{a^t})$, where $g(\cdot)$ is the refinement function. All other hypotheses remain unchanged $H^{t+1}_a = H^t_a \; \forall a \neq a^t$.

We perform refinement as follows (see also [12]). We render the object in pose $H^t_{a^t}$. Each pixel within the rendered mask is tested for being an inlier.[1] All inlier pixels are used to re-calculate the pose with the Kabsch algorithm. We repeat this procedure multiple times for the single, chosen hypothesis until the number of inlier pixels stops increasing or until the number m^t of executed refinement steps exceeds a maximum $m_{\max}$. The budget is decreased by the number of refinement steps performed, $B^{t+1} = B^t - m^t$. The agent proceeds choosing refinement actions until $B^t < m_{\max}$, in which case further refinement may exceed the total budget B^0 of refinement steps.

When this point has been reached, the refinement phase terminates, and the agent enters the **final decision phase** in which the agent chooses a hypothesis as the final output. We denote the final action as $a^T \in \{1 \dots N\}$ and the final pose estimate as $\tilde{H} = H^T_{a^T}$. The agent receives a reward of $r = 1$ in case the pose is correct or a negative reward of $r = -1$ otherwise. We use the pose correctness criterion from [9].

In the following, we describe how the agent makes its decisions. During both, the refinement phase and the final decision phase, the agent chooses from the hypothesis pool. We describe the agent behaviour by a probability distribution $\pi\left(a^t|S^t;\boldsymbol{\theta}\right)$ referred to as "policy". Given the current state S^t, which contains information about the hypothesis pool and the input image $\boldsymbol{x}$, the agent selects a hypothesis by drawing a sample from the policy The vector $\boldsymbol{\theta}$ of learnable parameters consists of CNN weights (described in Sec. 3.2.3). We will first give details on the state space S^t before explaining policy $\pi\left(a^t|S^t;\boldsymbol{\theta}\right)$.

3.2.1 State Space

We model our state space in a way that allows us to use our new, efficient training algorithm, described in Sec. 3.3.1. We assume that the current state S^t of the hypothesis pool decomposes as $S^t = (s^t_1, \dots s^t_N)$, where s^t_a will be called the state of hypothesis H^t_a. The state of an hypothesis contains the original input image $\boldsymbol{x}$, the forest prediction $\boldsymbol{z}$ for the image, the pose hypothesis H^t_a, as well as a vector $\boldsymbol{f}^t_a$ of additional context features of the hypothesis (see Sec. 3.2.3). In summary, this gives $s^t_a = (\boldsymbol{x}, \boldsymbol{z}, H^t_a, \boldsymbol{f}^t_a)$.

3.2.2 Policy

Our agent makes its decisions using a softmax policy. The probability of choosing a particular action a^t during the refinement phase is given by

$$\pi\left(a^t|S^t;\boldsymbol{\theta}\right) = \frac{\exp\left(E(s^t_{a^t};\boldsymbol{\theta})\right)}{\sum_{a \in A^t} \exp\left(E(s^t_a;\boldsymbol{\theta})\right)}, \tag{1}$$

[1] A pixel i is tested for being an inlier for pose H by transforming its predicted object coordinates $\boldsymbol{y}_i$ to camera space using H. If the Euclidean distance between resulting camera coordinates and the observed coordinates at the pixel is below a threshold the pixel is considered an inlier.

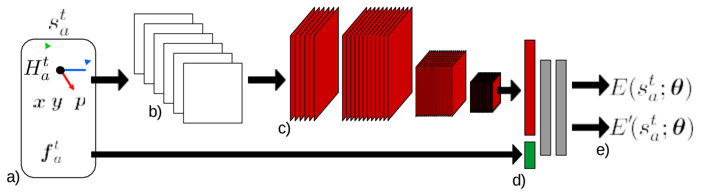

Figure 3. The CNN architecture: a) The system takes a pose hypothesis H_a^t and the additional features f_a^t encoding the context and history of the pose as input. b) We use the hypothesis to render the object and to cut out patches in the observed images. c) The images are processed by multiple convolutional layers. d) We concatinate the output of the convolutional layers with the features f_a^t. The result is fed into multiple fully-connected layers. e) The network predicts two energy values: $E(s_a^t)$ to be used in the refinement phase and $E'(s_a^t)$ to be used in the final decision.

where $E(s_a^t;\theta)$ will be called the energy of the state s_a^t. We will abbreviate it as $E_a^t = E(s_a^t;\theta)$. The energy of a state in the softmax policy is a measure of how desirable it is for the agent to refine the hypothesis. We use the same policy in the final decision phase, but with a different energy $E'(s_a^t;\theta)$ abbreviated by E'^t_a. We use a CNN to predict both energies, E_a^t and E'^t_a. In the next section, we discuss the CNN architecture and how it governs the behaviour of the agent.

3.2.3 CNN Architecture

We give an overview of the CNN architecture used in this work in Fig. 3. As in [12], the CNN compares rendered and observed images. We use the same six input channels as in [12], namely: the rendered depth channel, the observed depth channel, a rendered segmentation channel, the object probability channel, a depth mask, and a single channel holding the difference between object coordinates.

There are however two major differences in our CNN compared to the one used in [12]. Firstly, while Krull *et al.* predict a single energy value of a pose, we jointly predict two separate energy values: one energy E_a^t for the refinement phase and one energy E'^t_a for the final decision phase.

Secondly, we input additional features to the network by concatenating them to the first fully connected layer. The features are: The number of times the hypothesis has already been selected for refinement, The distance the hypothesis has moved during its last refinement and the average distance of the hypothesis before refinement to all other hypotheses in the original pool.

Our CNN consists of the following layers: 128 kernels of size $6\times3\times3$, 256 kernels of size $128\times3\times3$, a 2×2 *max-pooling* layer, 512 kernels of size $256\times3\times3$, a max-pooling operation over the remaining size of the image, finally 3 fully connected layers. The features f_a^t are concatenated to the first fully connected layer, as shown in Fig. 3. Each layer, except the last is followed by a *tanh* operation.

3.3. Policy Gradient Training

We will now discuss the training procedure for our PoseAgent. First, we will give a general introduction of policy gradient training, and then apply the approach to the PoseAgent. Finally, we will introduce an efficient algorithm that greatly reduces variance during training and makes training feasible.

The goal of the training is the maximization of the expected reward $\mathrm{E}[r]$. This expected value depends on the environment as well as on the policy of our agent. In stochastic policy gradient methods one attempts to approximate the gradient with respect to the policy parameters θ. Note that since we are dealing with the expected value it becomes possible calculate derivatives, even if the reward function itself is non differentiable. By making use of the equality $\frac{\partial}{\partial\theta_j}p(x;\theta_j) = p(x;\theta_j)\frac{\partial}{\partial\theta_j}\ln p(x;\theta_j)$, we can write the derivative of the expected reward with respect to each parameter θ_j in θ as

$$\frac{\partial}{\partial\theta_j}\mathrm{E}[r] = \mathrm{E}\left[r\frac{\partial}{\partial\theta_j}\ln p(s^{1:T}, a^{1:T};\theta)\right], \qquad (2)$$

where $p(s^{1:T}, a^{1:T};\theta)$ is the probability of a particular sequence of states $s^{1:T} = (s^1 \dots s^T)$ and actions $a^{1:T} = (a^1 \dots a^T)$ to occur.

Because of the Markov property of the environment, it is possible to decompose the probability and rewrite it as

$$\frac{\partial}{\partial\theta_j}\mathrm{E}[r] = \mathrm{E}\left[r \sum_{t=0}^{T}\frac{\partial}{\partial\theta_j}\ln\pi\left(a^t|S^t;\theta\right)\right]. \qquad (3)$$

Following the REINFORCE algorithm [25], we approximate Eq. 3 using sampled sequences $(s_k^{1:T_k}, a_k^{1:T_k})$, generated by running the agent, as described in Sec. 3.2, on training images,

$$\frac{\partial}{\partial\theta_j}\mathrm{E}[r] \approx \frac{1}{M}\sum_{k=1}^{M} r_k \sum_{t=0}^{T_k}\frac{\partial}{\partial\theta_j}\ln\pi\left(a_k^t|S_k^t;\theta\right), \qquad (4)$$

where T_k is the number of steps in the sequence and r_k is the reward achieved in the sequence.

3.3.1 Efficient Gradient Calculation

We will now introduce an algorithm (Alg. 1), to dramatically reduce the variance of estimated gradients, by allowing us to use a higher number of sequences M, in a given time. The basic idea is to make use of the special decomposable structure of the state space and our policy. The advantage of our algorithm compared to the naïve implementation is illustrated in Fig. 4.

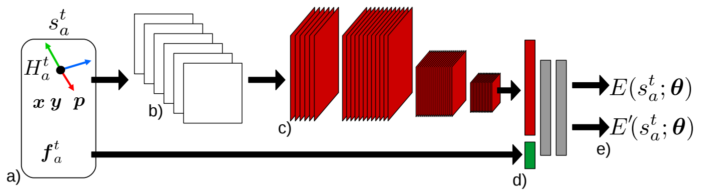
s_a^t
H_a^t
$x\ y\ p$
f_a^t
a)
b)
c)
d)
e)
$E(s_a^t;\boldsymbol{\theta})$
$E'(s_a^t;\boldsymbol{\theta})$

Figure 3. The CNN architecture: a) The system takes a pose hypothesis H_a^t and the additional features $\boldsymbol{f}_a^t$ encoding the context and history of the pose as input. b) We use the hypothesis to render the object and to cut out patches in the observed images. c) The images are processed by multiple convolutional layers. d) We concatinate the output of the convolutional layers with the features $\boldsymbol{f}_a^t$. The result is fed into multiple fully-connected layers. e) The network predicts two energy values: $E(s_a^t)$ to be used in the refinement phase and $E'(s_a^t)$ to be used in the final decision.

where $E(s_a^t;\boldsymbol{\theta})$ will be called the energy of the state s_a^t. We will abbreviate it as $E_a^t = E(s_a^t;\boldsymbol{\theta})$. The energy of a state in the softmax policy is a measure of how desirable it is for the agent to refine the hypothesis. We use the same policy in the final decision phase, but with a different energy $E'(s_a^t;\boldsymbol{\theta})$ abbreviated by ${E'}_a^t$. We use a CNN to predict both energies, E_a^t and ${E'}_a^t$. In the next section, we discuss the CNN architecture and how it governs the behaviour of the agent.

3.2.3 CNN Architecture

We give an overview of the CNN architecture used in this work in Fig. 3. As in [12], the CNN compares rendered and observed images. We use the same six input channels as in [12], namely: the rendered depth channel, the observed depth channel, a rendered segmentation channel, the object probability channel, a depth mask, and a single channel holding the difference between object coordinates.

There are however two major differences in our CNN compared to the one used in [12]. Firstly, while Krull *et al.* predict a single energy value of a pose, we jointly predict two separate energy values: one energy E_a^t for the refinement phase and one energy $E_a'^t$ for the final decision phase.

Secondly, we input additional features to the network by concatenating them to the first fully connected layer. The features are: The number of times the hypothesis has already been selected for refinement, The distance the hypothesis has moved during its last refinement and the average distance of the hypothesis before refinement to all other hypotheses in the original pool.

Our CNN consists of the following layers: 128 kernels of size $6\times3\times3$, 256 kernels of size $128\times3\times3$, a 2×2 *max-pooling* layer, 512 kernels of size $256\times3\times3$, a max-pooling operation over the remaining size of the image, finally 3 fully connected layers. The features $\boldsymbol{f}_a^t$ are concatenated to the first fully connected layer, as shown in Fig. 3. Each layer, except the last is followed by a *tanh* operation.

3.3. Policy Gradient Training

We will now discuss the training procedure for our PoseAgent. First, we will give a general introduction of policy gradient training, and then apply the approach to the PoseAgent. Finally, we will introduce an efficient algorithm that greatly reduces variance during training and makes training feasible.

The goal of the training is the maximization of the expected reward $\mathbb{E}[r]$. This expected value depends on the environment as well as on the policy of our agent. In stochastic policy gradient methods one attempts to approximate the gradient with respect to the policy parameters $\boldsymbol{\theta}$. Note that since we are dealing with the expected value it becomes possible calculate derivatives, even if the reward function itself is non differentiable. By making use of the equality $\frac{\partial}{\partial\theta_j}p(x;\theta_j) = p(x;\theta_j)\frac{\partial}{\partial\theta_j}\ln p(x;\theta_j)$, we can write the derivative of the expected reward with respect to each parameter θ_j in $\boldsymbol{\theta}$ as

$$\frac{\partial}{\partial\theta_j}\mathbb{E}[r] = \mathbb{E}\left[r\frac{\partial}{\partial\theta_j}\ln p(s^{1:T},a^{1:T};\boldsymbol{\theta})\right], \qquad (2)$$

where $p(s^{1:T},a^{1:T};\boldsymbol{\theta})$ is the probability of a particular sequence of states $s^{1:T} = (s^1\dots s^T)$ and actions $a^{1:T} = (a^1\dots a^T)$ to occur.

Because of the Markov property of the environment, it is possible to decompose the probability and rewrite it as

$$\frac{\partial}{\partial\theta_j}\mathbb{E}[r] = \mathbb{E}\left[r\sum_{t=0}^{T}\frac{\partial}{\partial\theta_j}\ln\pi\left(a^t|S^t;\boldsymbol{\theta}\right)\right]. \qquad (3)$$

Following the REINFORCE algorithm [25], we approximate Eq. 3 using sampled sequences $(S_k^{1:T_k}, a_k^{1:T_k})$, generated by running the agent, as described in Sec. 3.2, on training images,

$$\frac{\partial}{\partial\theta_j}\mathbb{E}[r] \approx \frac{1}{M}\sum_{k=1}^{M}r_k\sum_{t=0}^{T_k}\frac{\partial}{\partial\theta_j}\ln\pi\left(a_k^t|S_k^t;\boldsymbol{\theta}\right), \qquad (4)$$

where T_k is the number of steps in the sequence and r_k is the reward achieved in the sequence.

3.3.1 Efficient Gradient Calculation

We will now introduce an algorithm (Alg. 1), to dramatically reduce the variance of estimated gradients, by allowing us to use a higher number of sequences M, in a given time. The basic idea is to make use of the special decomposable structure of the state space and our policy. The advantage of our algorithm compared to the naïve implementation is illustrated in Fig. 4.

Starting from a hypothesis pool $H^0 = (H_1^0 \dots H_N^0)$, only a finite number of different hypothesis states $s_a^\tau | a \in \{1, \dots N\}, \tau \in \{0, \dots \tau_{\max}\}$ can occur during a run of our PoseAgent. Here, $s_a^\tau = (H_a^\tau, f_a^\tau)$ shall denote the state of hypothesis a after it has been refined τ times.

The algorithm pre-computes all possibly occurring hypothesis states s_a^τ, and predicts all corresponding energy values E_a^τ[2] in advance using the CNN.

While this comes with some computational expense, it allows us to rapidly sample large numbers of sequences without having to re-evaluate the energy function.

To illustrate why this is possible, let us now reconsider the calculation of the derivatives in Eq. 4. Using the chain rule, we can write them as

$$\frac{\partial}{\partial \theta_j} \ln \pi\left(a^t | S^t; \theta\right) = \sum_{a \in A^t} \frac{\partial E_a^t}{\partial \theta_j} \frac{\partial}{\partial E_a^t} \ln \pi\left(a^t | S^t; \theta\right), \tag{5}$$

where

$$\frac{\partial}{\partial E_a^t} \ln \pi\left(a^t | S^t; \theta\right) = \begin{cases} 1 - \pi(a^t | S^t; \theta) & \text{if } a = a^t \\ -\pi(a^t | S^t; \theta) & \text{else} \end{cases}. \tag{6}$$

We can now rearrange Eq. 4 as sum over possible hypothesis states

$$\sum_{\tau=0}^{\tau_{\max}} \sum_{a=1}^{N} \frac{\partial E_a^\tau}{\partial \theta_j} \underbrace{\frac{1}{M} \sum_{k=1}^{M} \sum_{t=1}^{T_k} \mathbb{1}(\tau_{a,k}^t = \tau) \frac{\partial}{\partial E_a^\tau} \ln \pi\left(a_k^t | S_k^t; \theta\right) r_k}_{D(a,\tau)}. \tag{7}$$

Here, $\mathbb{1}(\tau_{a,k}^t = \tau)$ is the indicator function. It has the value 1 only when the hypothesis a at time t in sequence k has been selected for refinement exactly $\tau_{a,k}^t = \tau$ times. It has the value 0 in any other case.

Our algorithm works by first calculating the inner sums in Eq. 7 and storing the results in the entries $D(a, \tau)$ of a table D. We compute these sums with a single iteration over all sequences k and all time steps t. The accumulation of these values is computationally cheap, because it does not not require any rendering or involvement of the CNN.

This structure allows us to increase the number of sampled sequences M without much computational cost. The algorithm can process an arbitrary amount of sequences using only a single back propagation pass of the CNN for each possible hypothesis state s_a^t. In a naïve implementation, the number of required forward-backward passes would increase linearly with the number of sampled sequences.

Let us look at the algorithm in detail. It consists of three parts:

Initialization Phase: We generate the original hypothesis pool as described in 3.1. Then, we refine all hypotheses $\tau_{\max}$ times and predict the energy values E_a^τ for all of them using the CNN.

Sampling Phase: We sample sequences $(s_k^{1:T}, a_k^{1:T})$ as described in Sec. 3.2, using the precomputed energies. We observe the reward r_k for each sequence. Then, we calculate for each time t, each selected hypothesis a_k^t and each possible hypothesis a the derivative $\frac{\partial}{\partial E_a^\tau} \ln \pi(a_k^t | S_k^t; \theta)\, r_k$ using Eq. 6. We accumulate the results in the corresponding table entries $D(a, \tau_{a,k}^t)$. This corresponds to the inner sums in Eq. 7.

Gradient Update Phase: We once more process each of the hypothesis states s_a^τ with the CNN and use standard back propagation to calculate $\frac{\partial E_a^\tau}{\partial \theta_j}$. We multiply the results with $D(a, \tau)$ and accumulate them up in another table G to obtain the final gradients. This corresponds to the outer sums in Eq. 7.

```
Initialization Phase:
    generate hypothesis pool $H^0$;
    refine each hypothesis $\tau_{\max}$ times;
    calculate and store $E_a^\tau$;
    initialize table entries $D(a,\tau) \leftarrow 0$ and $G(j) \leftarrow 0$
Sampling Phase:
    for $k = 1 : M$ do
        sample path $(s_k^{1:T_k}, a_k^{1:T_k})$ using $E_a^\tau$;
        receive reward $r_k$;
        for $t = 1 : T_k, a = 1 : N$ do
            $D(a, \tau_{a,k}) \leftarrow D(a, \tau_{a,k}) + \frac{\partial}{\partial E_a^\tau} \ln \pi\left(a_k^t | S_k^t; \theta\right) r_k$
        end
    end
Gradient Update Phase:
    for $\tau = 0 : \tau_{\max}$, $a = 1 : N$ do
        calculate $\frac{\partial E_a^\tau}{\partial \theta_j}$ via back propagation;
        for all CNN parameters $j$ do
            $G(j) \leftarrow G(j) + \frac{\partial E_a^\tau}{\partial \theta_j} \frac{1}{m} D(a, \tau)$;
        end
    end
    Output: $G(j) \approx \frac{\partial}{\partial \theta_j} \mathbb{E}[r]$;
```

Algorithm 1: Efficient Gradient Calculation

4. Experiments

In the following we will describe the experiments to compare our method to the baseline system from [12]. Our experiments confirm, that our learned inference procedure is able to use its budget in a more efficient way. It outperforms the [12], while using on average a smaller number refinement steps.

Additionally we will describe an experiment regarding the efficiency of our training algorithm compared to a naïve implementation of the REINFORCE algorithm. We find, that our algorithm can dramatically reduce the gradient vari-

[2] To improve readability, we will not differentiate between E_a^τ and E'^τ_a in this section.

Starting from a hypothesis pool $\boldsymbol{H}^0 = (H_1^0 \dots H_N^0)$, only a finite number of different hypothesis states $s_a^\tau | a \in \{1, \dots N\}, \tau \in \{0, \dots \tau_{\max}\}$ can occur during a run of our PoseAgent. Here, $s_a^\tau = (H_a^\tau, \boldsymbol{f}_a^\tau)$ shall denote the state of hypothesis a after it has been refined τ times.

The algorithm pre-computes all possibly occurring hypothesis states s_a^τ, and predicts all corresponding energy values E_a^τ[2] in advance using the CNN.

While this comes with some computational expense, it allows us to rapidly sample large numbers of sequences without having to re-evaluate the energy function.

To illustrate why this is possible, let us now reconsider the calculation of the derivatives in Eq. 4. Using the chain rule, we can write them as

$$\frac{\partial}{\partial \theta_j} \ln \pi \left(a^t | S^t; \boldsymbol{\theta}\right) = \sum_{a \in A^t} \frac{\partial E_a^t}{\partial \theta_j} \frac{\partial}{\partial E_a^t} \ln \pi \left(a^t | S^t; \boldsymbol{\theta}\right), \tag{5}$$

where

$$\frac{\partial}{\partial E_a^t} \ln \pi \left(a^t | S^t; \boldsymbol{\theta}\right) = \begin{cases} 1 - \pi \left(a^t | S^t; \boldsymbol{\theta}\right) & \text{if } a = a^t \\ -\pi \left(a^t | S^t; \boldsymbol{\theta}\right) & \text{else} \end{cases}. \tag{6}$$

We can now rearrange Eq. 4 as sum over possible hypothesis states

$$\sum_{\tau=0}^{\tau_{\max}} \sum_{a=1}^{N} \frac{\partial E_a^\tau}{\partial \theta_j} \frac{1}{M} \underbrace{\sum_{k=1}^{M} \sum_{t=1}^{T_k} \mathbb{1}(\tau_{a,k}^t = \tau) \frac{\partial}{\partial E_a^\tau} \ln \pi \left(a_k^t | S_k^t; \boldsymbol{\theta}\right) r_k}_{D(a,\tau)}. \tag{7}$$

Here, $\mathbb{1}(\tau_{a,k}^t = \tau)$ is the indicator function. It has the value 1 only when the hypothesis a at time t in sequence k has been selected for refinement exactly $\tau_{a,k}^t = \tau$ times. It has the value 0 in any other case.

Our algorithm works by first calculating the inner sums in Eq. 7 and storing the results in the entries $D(a, \tau)$ of a table D. We compute these sums with a single iteration over all sequences k and all time steps t. The accumulation of these values is computationally cheap, because it does not not require any rendering or involvement of the CNN.

This structure allows us to increase the number of sampled sequences M without much computational cost. The algorithm can process an arbitrary amount of sequences using only a single back propagation pass of the CNN for each possible hypothesis state s_a^t. In a naïve implementation, the number of required forward-backward passes would increase linearly with the number of sampled sequences.

Let us look at the algorithm in detail. It consists of three parts:

Initialization Phase: We generate the original hypothesis pool as described in 3.1. Then, we refine all hypotheses $\tau_{\max}$ times and predict the energy values E_a^τ for all of them using the CNN.

Sampling Phase: We sample sequences $(s_k^{1:T}, a_k^{1:T})$ as described in Sec. 3.2, using the precomputed energies. We observe the reward r_k for each sequence. Then, we calculate for each time t, each selected hypothesis a_k^t and each possible hypothesis a the derivative $\frac{\partial}{\partial E_a^\tau} \ln \pi \left(a_k^t | S_k^t; \boldsymbol{\theta}\right) r_k$ using Eq. 6. We accumulate the results in the corresponding table entries $D(a, \tau_{a,k}^t)$. This corresponds to the inner sums in Eq. 7.

Gradient Update Phase: We once more process each of the hypothesis states s_a^τ with the CNN and use standard back propagation to calculate $\frac{\partial E_a^\tau}{\partial \theta_j}$. We multiply the results with $D(a, \tau)$ and accumulate them up in another table G to obtain the final gradients. This corresponds to the outer sums in Eq. 7.

```
Initialization Phase:
    generate hypothesis pool $\boldsymbol{H}^0$;
    refine each hypothesis $\tau_{\max}$ times;
    calculate and store $E_a^\tau$;
    initialize table entries $D(a,\tau) \leftarrow 0$ and $G(j) \leftarrow 0$
Sampling Phase:
    for $k = 1 : M$ do
        sample path $(s_k^{1:T_k}, a_k^{1:T_k})$ using $E_a^\tau$ ;
        receive reward $r_k$;
        for $t = 1 : T_k, a = 1 : N$ do
            $D(a, \tau_{a,k}) \leftarrow D(a, \tau_{a,k}) + \frac{\partial}{\partial E_a^\tau} \ln \pi \left(a_k^t | S_k^t; \boldsymbol{\theta}\right) r_k$
        end
    end
Gradient Update Phase:
    for $\tau = 0 : \tau_{\max}$ ,  $a = 1 : N$ do
        calculate $\frac{\partial E_a^\tau}{\partial \theta_j}$ via back propagation;
        for all CNN parameters $j$ do
            $G(j) \leftarrow G(j) + \frac{\partial E_a^\tau}{\partial \theta_j} \frac{1}{m} D(a,\tau)$;
        end
    end
    Output: $G(j) \approx \frac{\partial}{\partial \theta_j} \mathbb{E}[r]$;
```

Algorithm 1: Efficient Gradient Calculation

4. Experiments

In the following we will describe the experiments to compare our method to the baseline system from [12]. Our experiments confirm, that our learned inference procedure is able to use its budget in a more efficient way. It outperforms the [12], while using on average a smaller number refinement steps.

Additionally we will describe an experiment regarding the efficiency of our training algorithm compared to a naïve implementation of the REINFORCE algorithm. We find, that our algorithm can dramatically reduce the gradient vari-

[2] To improve readability, we will not differentiate between E_a^τ and $E_a'^\tau$ in this section.

ance during training.

We conducted our experiments on the dataset introduced in [13]. It features six RGB-D sequences of hand held sometimes strongly occluded objects.

4.1. Training and Validation Procedure

We train our system on the *samurai 1* sequence and, as [12] omit the first 400 frames to achieve a higher percentage of occluded images.

We train our system with two different parameter settings: Using a hypothesis pool size of $N = 210$, which is the setting used in [12], and a larger pool size of $N = 420$.

To determine the adequate size of the budget B^0 of refinement steps, we ran the system from [12] on our validation set and determined the average number of refinement steps it used. We set our budget during training to the resulting number $B^0 = 77$.

During training we allow a hypothesis to be chosen $\tau_{\max} = 3$ times for refinement. We set the maximum number of refinement steps per iteration to $m_{\max} = 10$.

Using stochastic gradient descent, we go through our training images in random order and run Algorithm 1 to approximate the gradient. We sample $M = 50k$ sequences for every image. An additional $50k$ sequences are used to estimates the average reward for the image, which is then subtracted from the reward to further reduce variance [23, 18]. We perform a parameter update after every image. Starting with an initial learning rate of $\lambda^0 = 25 \cdot 10^{-4}$, we reduce it according to $\lambda^l = \lambda^0/(1 + l\upsilon)$, with $\upsilon = 0.01$. We use a fixed momentum of 0.9.

We skip images in which none of the hypotheses from the pool lead to a correct pose after being refined $\tau_{\max}$ times, and in which more than 10% of the hypotheses from the pool lead to a correct pose. Such images contribute little, because they are impossibly or to easily solved.

We run the training procedure for 96 hours on an Intel E5-2450 2.10GHz with Nvidia Tesla K20x GPU and save a snapshot every 50 training images. To avoid over-fitting, we test these saved snapshots on our validation set and choose the model with the highest accuracy. In order to reduce the computational time during validation, we considered only images in which the object was at least 5% occluded[3].

4.2. Additional Baselines

Two demonstrate the advantage of dynamically distributing a given computational budget, we implemented two cut-down versions of PoseAgent, which serve as additional baselines. The baseline method abbreviated as *RandRef* randomly selects a hypothesis to refine at every iteration. When the budget is exhausted, it chooses the hypothesis with the best predicted final selection energy $E'(s^t_a;\theta)$.

[3]according to the definition from [12]

The baseline *BestRef* directly picks the hypothesis with the best $E(s^t_a;\theta)$, refines it until the budget is exhausted and outputs it as final decision. We used the best performing settings when running the baselines: ($\tau_{max} = 6$, $m_{max} = 5$) for pool size N = 210 and ($\tau_{max} = 7$, $m_{max} = 4$) for pool size N = 420.

4.3. Testing Conditions

We compared both versions of our model, using $N = 210$ and $N = 420$, against the system of [12] using the corresponding pool size. In all experiments with the baseline method [12], we use the identical CNN with the original weights trained by Krull *et al.* in [12] on the *samurai 2* sequence. This is the network that [12] reports the best results for.

Apart from the pool size, we used the identical testing conditions as in [12], including the same random forest originally trained in [3]. To classify a pose in correct or false we use the same point-distance-based criterion used in [12]. A pose is considered correct, when the average distance between the vertices of the 3D model in the ground truth pose and the evaluated pose is below a threshold.

While the number of refinement steps in our setting is restricted, the method of [12] does not provide any guarantees on how many refinement steps are used. To ensure a fair comparison, we first ran the method from [12] and recorded the average number of refinement steps that it required on each test sequence. When running our method, we set the budget for each sequence to this recorded value, making sure that PoseAgent could never use more refinement steps than [12]. The total average number of refinement steps required by both methods can be found Tabs. 1 and 2.

We evaluate our method using different parameters for $\tau_{\max}$ and $m_{\max}$, so that $\tau_{\max} \cdot m_{\max} \approx 30$. Meaning that a each pose can have an approximate maximum of 30 refinement steps. A higher value of $\tau_{\max}$ (and lower value of $m_{\max}$) means that PoseAgent can make more fine grained decisions on where to spend its budget. We use the following combinations for the two values ($\tau_{\max}{=}3, m_{\max} = 10$), ($\tau_{\max} = 5$, $m_{\max} = 6$), ($\tau_{\max} = 6$, $m_{\max} = 5$), ($\tau_{\max} = 7$, $m_{\max} = 4$).

4.4. Results

The results of our experiments can be seen in Tabs. 1 and 2. PoseAgent is able to improve the best published results on the dataset by a total of 10.56% (comparing 60.06% from Tab. 1 with 70.62% from Tab. 2). When we compare our method to [12] working on the same hypothesis pool size we are still able to outperform it. With the original pool size of $N = 210$ by 2.12% and the increased pool size of $N = 420$ by 2.59%.

Note, that the budget is set in a way, that is extremely ance during training.

We conducted our experiments on the dataset introduced in [13]. It features six RGB-D sequences of hand held sometimes strongly occluded objects.

4.1. Training and Validation Procedure

We train our system on the *samurai 1* sequence and, as [12] omit the first 400 frames to achieve a higher percentage of occluded images.

We train our system with two different parameter settings: Using a hypothesis pool size of $N = 210$, which is the setting used in [12], and a larger pool size of $N = 420$.

To determine the adequate size of the budget B^0 of refinement steps, we ran the system from [12] on our validation set and determined the average number of refinement steps it used. We set our budget during training to the resulting number $B^0 = 77$.

During training we allow a hypothesis to be chosen $\tau_{\max} = 3$ times for refinement. We set the maximum number of refinement steps per iteration to $m_{\max} = 10$.

Using stochastic gradient descent, we go through our training images in random order and run Algorithm 1 to approximate the gradient. We sample $M = 50k$ sequences for every image. An additional $50k$ sequences are used to estimates the average reward for the image, which is then subtracted from the reward to further reduce variance [23, 18]. We perform a parameter update after every image. Starting with an initial learning rate of $\lambda^0 = 25 \cdot 10^{-4}$, we reduce it according to $\lambda^l = \lambda^0/(1 + l\nu)$, with $\nu = 0.01$. We use a fixed momentum of 0.9.

We skip images in which none of the hypotheses from the pool lead to a correct pose after being refined $\tau_{\max}$ times, and in which more than 10% of the hypotheses from the pool lead to a correct pose. Such images contribute little, because they are impossibly or to easily solved.

We run the training procedure for 96 hours on an Intel E5-2450 2.10GHz with Nvidia Tesla K20x GPU and save a snapshot every 50 training images. To avoid over-fitting, we test these saved snapshots on our validation set and choose the model with the highest accuracy. In order to reduce the computational time during validation, we considered only images in which the object was at least 5% occluded[3].

[3]according to the definition from [12]

4.2. Additional Baselines

Two demonstrate the advantage of dynamically distributing a given computational budget, we implemented two cut-down versions of PoseAgent, which serve as additional baselines. The baseline method abbreviated as *RandRef* randomly selects a hypothesis to refine at every iteration. When the budget is exhausted, it chooses the hypothesis with the best predicted final selection energy $E'(s_a^t; \boldsymbol{\theta})$.

The baseline *BestRef* directly picks the hypothesis with the best $E(s_a^t; \boldsymbol{\theta})$, refines it until the budget is exhausted and outputs it as final decision. We used the best performing settings when running the baselines: (τ_{max} = 6, m_{max} = 5) for pool size N = 210 and (τ_{max} = 7, m_{max} = 4) for pool size N = 420.

4.3. Testing Conditions

We compared both versions of our model, using $N = 210$ and $N = 420$, against the system of [12] using the corresponding pool size. In all experiments with the baseline method [12], we use the identical CNN with the original weights trained by Krull *et al.* in [12] on the *samurai 2* sequence. This is the network that [12] reports the best results for.

Apart from the pool size, we used the identical testing conditions as in [12], including the same random forest originally trained in [3]. To classify a pose in correct or false we use the same point-distance-based criterion used in [12]. A pose is considered correct, when the average distance between the vertices of the 3D model in the ground truth pose and the evaluated pose is below a threshold.

While the number of refinement steps in our setting is restricted, the method of [12] does not provide any guarantees on how many refinement steps are used. To ensure a fair comparison, we first ran the method from [12] and recorded the average number of refinement steps that it required on each test sequence. When running our method, we set the budget for each sequence to this recorded value, making sure that PoseAgent could never use more refinement steps than [12]. The total average number of refinement steps required by both methods can be found Tabs. 1 and 2.

We evaluate our method using different parameters for $\tau_{\max}$ and $m_{\max}$, so that $\tau_{\max} \cdot m_{\max} \approx 30$. Meaning that a each pose can have an approximate maximum of 30 refinement steps. A higher value of $\tau_{\max}$ (and lower value of $m_{\max}$) means that PoseAgent can make more fine grained decisions on where to spend its budget. We use the following combinations for the two values ($\tau_{\max}$=3,$m_{\max} = 10$), ($\tau_{\max} = 5, m_{\max} = 6$), ($\tau_{\max} = 6, m_{\max} = 5$), ($\tau_{\max} = 7, m_{\max} = 4$).

4.4. Results

The results of our experiments can be seen in Tabs. 1 and 2. PoseAgent is able to improve the best published results on the dataset by a total of **10.56%** (comparing 60.06% from Tab. 1 with 70.62% from Tab. 2). When we compare our method to [12] working on the same hypothesis pool size we are still able to outperform it. With the original pool size of $N = 210$ by **2.12%** and the increased pool size of $N = 420$ by **2.59%**.

Note, that the budget is set in a way, that is extremely

		Ours					
	Krull *et al.*	$\tau_{\max}$=3	$\tau_{\max}$=5	$\tau_{\max}$=6	$\tau_{\max}$=7	RandRef	BestRef
Cat 2	63.05	67.81	68.61	71.52	68.74	45.17	49.27
Samurai 2	60.30	51.66	53.32	54.82	51.66	31.73	34.88
Toolbox 2	52.96	52.07	60.06	54.44	59.76	35.50	32.84
Total	60.06	58.94	61.47	62.18	60.88	38.47	40.88
Avg. ref. steps	68.71	62.94	65.91	66.60	67.20		

Table 1. Percent correct poses using a hypothesis pool of $N = 210$

		Ours					
	Krull *et al.*	$\tau_{\max}$=3	$\tau_{\max}$=5	$\tau_{\max}$=6	$\tau_{\max}$=7	RandRef	BestRef
Cat 2	72.98	74.70	74.97	76.29	78.01	54.17	56.03
Samurai 2	66.45	59.30	59.63	58.80	61.13	39.87	40.86
Toolbox 2	64.79	65.38	68.93	71.60	71.01	52.07	53.85
Total	68.03	67.37	68.32	69.14	70.62	48.67	50.21
Avg. ref. steps	71.12	65.00	68.04	68.66	69.39		

Table 2. Percent correct poses using a hypothesis pool of $N = 420$

restrictive, ensuring that PoseAgent can never use more refinement steps than [12] uses on average.

In both settings ($N = 210$ and $N = 420$) there ap- pears to be a trend, that an increase of $\tau_{\max}$, which cor- responds to a more fine grained control of PoseAgent, leads to an improvement in accuracy. The only exception here is $\tau_{\max} = 7$ in the $N = 210$ setting. It should be noted that PoseAgent was trained with a different setting of $\tau_{\max} = 3$ and was able to generalize to the different settings used during testing.

We measured the average run time of the method (using CPU rendering) to be between 17 (Samurai 2) and 34 (Cat 2) seconds per image on an Intel E5-2450 2.10GHz with NVidia Tesla K20x GPU using a hypothesis pool of $N = 420$.

4.5. Efficiency of the Training Algorithm

In order to investigate the efficiency of out training algorithm compared to a na¨ıve implementation of the REINFORCE algorithm, we conducted the following experiment: We ran our training algorithm as well as the na¨ıve implementation up to 100 times on a single training image without updating the network.

To estimate the variance of the gradient, we calculated the standard deviation of 1000 randomly selected elements from the resulting gradient vector of the CNN and averaged them. We recorded the required computation time to process the image on an Intel E5-2450 2.10GHz with Nvidia Tesla K20x GPU.

The process was repeated for $M = 5$, $M = 50$, $M = 500$, $M = 5000$ and $M = 50000$ sequences in case of the efficient algorithm. In case of the na¨ıve implementation we used $M = 1$, $M = 2$, $M = 3$ and $M = 4$ sequences. To keep the computation time in reasonable limits we used

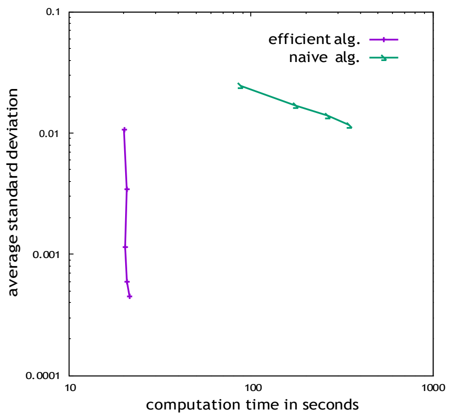

Figure 4. Observed gradient variance during training as a function of time: Our method is able process dramatically more sequences with almost no increase in computation time compared to a na¨ıve implementation of the REINFORCE algorithm. The result is a drastically reduced gradient variance

a reduced setting with a hypothesis pool size of $N = 21$ for both methods. As can be seen in Fig. 4 our algorithm allows us to reduce variance greatly with almost no increase in computation time.

5. Conclusion

We have demonstrated a method learn the algorithmic inference procedure in a pose estimation system using a policy gradient method. Our system learns to make efficient use of a given budget and is able to outperform the original system, while using on average less computational resources. We have presented an efficient algorithm for the gradient approximation during training. The algorithm is able to sharply reduce gradient variance, without a significant increase in computation time.

We see multiple interesting future directions of research in the context of our system. (i) One could investigate a soft version of PoseAgent, which is not working with a fixed budget, but can instead decide what is the appropriate time to stop. In such systems the used computational budget can be part of the reward function. (ii) The sequential structure of the current system does not allow simple parallelization, but a PoseAgent that learns to do inference making use of multiple computational cores could be conceived.

Acknowledgements. This work was supported by: European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 647769); The computations were performed on an HPC Cluster at the Center for Information Services and High Performance Computing (ZIH) at TU Dresden. We thank Joachim Staib for his help with CPU rendering on the Cluster.

		Ours					
	Krull *et al.*	$\tau_{\max}$=3	$\tau_{\max}$=5	$\tau_{\max}$=6	$\tau_{\max}$=7	RandRef	BestRef
Cat 2	63.05	67.81	68.61	**71.52**	68.74	45.17	49.27
Samurai 2	**60.30**	51.66	53.32	54.82	51.66	31.73	34.88
Toolbox 2	52.96	52.07	**60.06**	54.44	59.76	35.50	32.84
Total	60.06	58.94	61.47	**62.18**	60.88	38.47	40.88
Avg. ref. steps	68.71	62.94	65.91	66.60	67.20		

Table 1. Percent correct poses using a hypothesis pool of $N = 210$

		Ours					
	Krull *et al.*	$\tau_{\max}$=3	$\tau_{\max}$=5	$\tau_{\max}$=6	$\tau_{\max}$=7	RandRef	BestRef
Cat 2	72.98	74.70	74.97	76.29	**78.01**	54.17	56.03
Samurai 2	**66.45**	59.30	59.63	58.80	61.13	39.87	40.86
Toolbox 2	64.79	65.38	68.93	**71.60**	71.01	52.07	53.85
Total	68.03	67.37	68.32	69.14	**70.62**	48.67	50.21
Avg. ref. steps	71.12	65.00	68.04	68.66	69.39		

Table 2. Percent correct poses using a hypothesis pool of $N = 420$

restrictive, ensuring that PoseAgent can never use more refinement steps than [12] uses on average.

In both settings ($N = 210$ and $N = 420$) there appears to be a trend, that an increase of $\tau_{\max}$, which corresponds to a more fine grained control of PoseAgent, leads to an improvement in accuracy. The only exception here is $\tau_{\max} = 7$ in the $N = 210$ setting. It should be noted that PoseAgent was trained with a different setting of $\tau_{\max} = 3$ and was able to generalize to the different settings used during testing.

We measured the average run time of the method (using CPU rendering) to be between 17 (Samurai 2) and 34 (Cat 2) seconds per image on an Intel E5-2450 2.10GHz with NVidia Tesla K20x GPU using a hypothesis pool of $N = 420$.

4.5. Efficiency of the Training Algorithm

In order to investigate the efficiency of out training algorithm compared to a naïve implementation of the REINFORCE algorithm, we conducted the following experiment:

We ran our training algorithm as well as the naïve implementation up to 100 times on a single training image without updating the network.

To estimate the variance of the gradient, we calculated the standard deviation of 1000 randomly selected elements from the resulting gradient vector of the CNN and averaged them. We recorded the required computation time to process the image on an Intel E5-2450 2.10GHz with Nvidia Tesla K20x GPU.

The process was repeated for $M = 5$, $M = 50$, $M = 500$, $M = 5000$ and $M = 50000$ sequences in case of the efficient algorithm. In case of the naïve implementation we used $M = 1$, $M = 2$, $M = 3$ and $M = 4$ sequences. To keep the computation time in reasonable limits we used

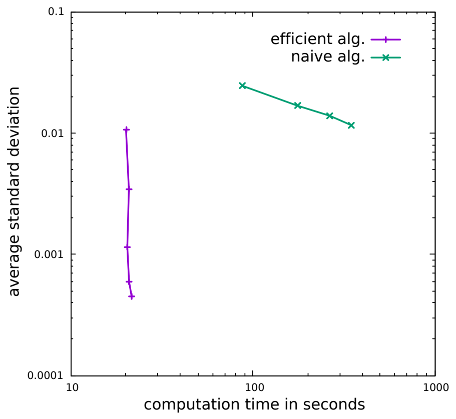

Figure 4. Observed gradient variance during training as a function of time: Our method is able process dramatically more sequences with almost no increase in computation time compared to a naïve implementation of the REINFORCE algorithm. The result is a drastically reduced gradient variance

a reduced setting with a hypothesis pool size of $N = 21$ for both methods. As can be seen in Fig. 4 our algorithm allows us to reduce variance greatly with almost no increase in computation time.

5. Conclusion

We have demonstrated a method learn the algorithmic inference procedure in a pose estimation system using a policy gradient method. Our system learns to make efficient use of a given budget and is able to outperform the original system, while using on average less computational resources. We have presented an efficient algorithm for the gradient approximation during training. The algorithm is able to sharply reduce gradient variance, without a significant increase in computation time.

We see multiple interesting future directions of research in the context of our system. (i) One could investigate a soft version of PoseAgent, which is not working with a fixed budget, but can instead decide what is the appropriate time to stop. In such systems the used computational budget can be part of the reward function. (ii) The sequential structure of the current system does not allow simple parallelization, but a PoseAgent that learns to do inference making use of multiple computational cores could be conceived.

Acknowledgements. This work was supported by: European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 647769); The computations were performed on an HPC Cluster at the Center for Information Services and High Performance Computing (ZIH) at TU Dresden. We thank Joachim Staib for his help with CPU rendering on the Cluster.