ATML Report - PonderNet

Jacopo Di Ventura
jacopo.di.ventura@usi.ch

Jury Andrea D'Onofrio
jury.donofrio@usi.ch

Matteo Martinoli
matteo.martinoli@usi.ch

Abstract

The goal of this project is to implement PonderNet, an innovative algorithm based on the research of Andrea Banino, Jan Balaguer, and Charles Blundell from DeepMind [1]. PonderNet dynamically adjusts the number of computational steps based on the complexity of the given problem. In simpler terms, PonderNet is able to learn the number of computational steps that allow an effective balance between prediction accuracy, computational cost, and generalization.

1 Introduction

Due to the increasing complexity of modern machine learning tasks, algorithms that dynamically adjust computational efforts while maintaining an effective balance of accuracy and efficiency have become an active field of research. This adaptability allows machine learning models to optimize their performance in real-time. Andrea Banino, Jan Balaguer, and Charles Blundell at DeepMind have addressed this challenge creating a new groundbreaking architecture: PonderNet. This innovative solution introduces a contemporary learning method, in which the algorithm calculates the optimal computational steps for the given task.

This paper aims to provide an analysis of the original architecture with the goal of replicating and expanding the findings of the research conducted by Andrea Banino, Jan Balaguer, and Charles Blundell.

As an initial step, to confirm the flexibility of PonderNet across different datasets, we have decided to implement the PonderNet architecture on an instance of the MNIST dataset based on this repository GitHub [2]. This allowed us to familiarize with the architecture and acquire a better understanding of the training process. Subsequently, we explored the task of replicating the innovative architecture on the parity task, as outlined in the original paper. By expanding our implementation, we aimed to increase our understanding of the architecture and evaluate its performance.

2 Related works

PonderNet builds on the previous ideas of Adaptive Computation Time (ACT; Graves, 2017) [3], Adaptive Early Exit Networks (Bolukbase et al., 2017) [4] and reinforcement learning (Williams, 1992) [5].

ACT is a technique specifically for recurrent neural networks that dynamically adjusts the computation steps for each input. However, it displays sensitivity to hyper-parameters tuning causing a trade-off between accuracy and computation cost. Moreover, this method involves biased gradient estimation because the gradient for the computational cost can only propagate backward through the final computational step.

In Adaptive Early Exit Networks, during evaluation, the forward pass of a deep neural network is halted if there is a high likelihood that the portion of the network employed so far has already accurately predicted the desired outcome.

The last method applies reinforcement learning to Recurrent Neural Networks (Chung et al., 2016; Banino et al., 2020) [6]. However, this approach showed high variance in gradient estimations as it required large batch sizes to train.

3 Methodology

In this section, our objective is to provide an overview by extracting and clarifying the fundamental components and design principles inherent in the architecture of the original paper.

3.1 Problem setting

We consider a supervised setup, where the goal is to learn a function $f : x \to y$ which maps input values x to the corresponding output values y. The solution intends to rewrite the architecture of neural networks by combining a new forward pass with a customized loss function.

3.2 Step recurrence and halting process

The PonderNet architecture consists of a step function, denoted as s, and an appropriate initial state, h_0. This step function is flexible, allowing it to be any neural network depending on the specific task to solve. The construction of the step function is outlined as follows:

$$\hat{y}_n, h_{n+1}, \lambda_n = s(x, h_n) \tag{1}$$

The main output of the function s are $\hat{y}_n$ and λ_n.

- The first output ($\hat{y}_n$) is the network's prediction conditioned on the dynamic number of steps $n \in 1, ..., N$.
- The second one (λ_n) regards the probability of halting at step n.

In order to decide whether to halt or not, it is useful to introduce a Markov process with a Bernoulli random variable Λ_n. The Markov process has two states:

1. Continue $\to \Lambda_n = 0$
2. Halt $\to \Lambda_n = 1$ (terminal state)

The transition probability of the Markov process is set as:

$$P(\Lambda_n = 1 | \Lambda_{n-1} = 0) = \lambda_n \qquad \forall 1 \leq n \leq N \tag{2}$$

This represents the conditional probability of entering state "halt" at step n conditioned that there has been no previous halting. We can estimate the unconditioned probability that the halting occurs within steps 0 to N, where N is the maximum number of steps.

The probability distribution p_n can then be derived as a generalization of the geometric distribution:

$$p_n = \lambda_n \prod_{j=1}^{n-1} (1 - \lambda_j) \tag{3}$$

This distribution probability is valid if we integrate over an infinite number of possible computation steps ($N \to \infty$).

The prediction $\hat{y}$ generated by PonderNet is sampled from a random variable $\hat{Y}$, following the probability distribution $P(\hat{Y} = y_n) = p_n$. In simpler terms, PonderNet's prediction corresponds to the prediction made at the specific step n at which the algorithm halts.

3.3 Training loss

The total loss function is a combination of two essential components: the reconstruction L_{rec} and the regularization term L_{reg}.

$$L = L_{rec} + L_{reg} \tag{4}$$

3.3.1 Reconstruction term

$$L_{rec} = \sum_{n=1}^{N} p_n \mathcal{L}(y, \hat{y}_n) \tag{5}$$

It represents the expected reconstruction loss across halting steps, reflecting the model's performance over multiple prediction stages. $\mathcal{L}$ here is a predefined metric such as mean squared error or cross-entropy, capturing the dissimilarity between the predicted and target values. In the context of the MNIST implementation, we chose cross-entropy, a suitable metric for classification problems. On the other hand, for the parity task, we selected binary cross-entropy with logits, aligning with the nature of the problem. The loss function $\mathcal{L}$ is scaled by p_n which represents the probability of halting at step n.

3.3.2 Regularization term

$$L_{reg} = \beta KL(p_n || p_G(\lambda_p)) \tag{6}$$

Kullback-Leibler (KL) divergence between the distribution of halting probabilities p_n and a reference geometric distribution, scaled by β. The geometric distribution defined as $P(S = s) = (1 - \lambda_p)^{s-1} \cdot \lambda_p$ is characterized by the hyper-parameter λ_p. This distribution biases the network toward an expected prior number of steps while stimulating exploration by encouraging non-zero probabilities for all possible number of steps. The hyper-paremeter β controls the influence of the regularization loss function on the loss function.

3.4 Evaluation sampling

During the inference, the network decides whether to continue or halt at each step by sampling from the halting Bernoulli random variable $\Lambda_n \sim B(p = \lambda_n)$. The iteration continues until the "halt" outcome is reached, indicating the final prediction as the output $y = y_n$. If the maximum number of steps N is reached without encountering the "halt" result, the network is automatically stopped, and the prediction is set to $y = y_N$.

4 Implementation

Our exploration started with the MNIST dataset, which is widely used for image classification tasks because of its straightforwardness and accessibility. This decision was made due to the well-defined nature of MNIST, which allows us to comprehend and implement more advanced architectures, including PonderNet. Then, having successfully accomplished this initial task, we moved to the Parity task as explained by Graves (2017)[4].

4.1 MNIST

Before delving into more complex tasks, we opted to implement PonderNet on a classifier trained with the MNIST dataset. This allowed us to enhance our comprehension of the task at hand and facilitated the development of a robust codebase, much of which was subsequently reused for the more intricate parity task.

The main challenge of this intermediate work was to develop an architecture that uses both a Convolutional Neural Network (CNN) and a Multi-Layer Perceptron (MLP) to dynamically identify the number of processing steps needed for an accurate and precise classification. CNN serves as the principal feature extractor, while the MLP is the dynamic decision-maker. It receives the features as input from the CNN and utilizes them to compute the ideal number of processing steps required.

To complete this task, we referred to the implementation presented in the GitHub repository and re-implemented particular components of the pipeline. Since this was not our main focus, we decided to simplify the training process by computing a few epochs. Nevertheless, we were able to obtain a meaningful evaluation.

4.1.1 Datasets

As already anticipated, we used the MNIST dataset to train, validate, and evaluate our model's performance. The training set is composed of 55,000 samples, and to spot overfitting during the training, we also created a

validation set composed of 5,000 samples. To evaluate our trained model's performance on unseen data, we have created a test set of 10,000 samples. All three datasets are pre-processed by standardized transformations. Additionally, the data are partitioned in batches of `BATCH_SIZE` by using DataLoader objects.

4.1.2 Hyper-parameters

The hyper-parameters we have set to control our model are:

- `N_CLASSES` = 10
- `N_OUT_CNN` = 32
- `N_OUT_MLP` = 32
- `N_HIDDEN_MLP` = 32
- `MAX_STEPS` = 15
- `BATCH_SIZE` = 64
- `BETA` = 0.05
- `LEARNING_RATE` = 0.0003

All hyper-parameter values are chosen by us. With the only exception being the number of classes, which is a result of the chosen dataset. λ_p is left out on purpose.

4.1.3 Experimental setup

We have run our experiments on a MacBook Pro 16' with ARM architecture and the M1 Max CPU. The code for this task can be found in the Jupyter Notebook named `pondernet_mnist.ipynb` in the following GitHub repository.

4.1.4 Computational requirements

Our experiments were conducted on a MacBook Pro 16' with ARM architecture, featuring the M1 Max CPU. We decided to tackle this challenge using only the CPU, as the computational time remained reasonable. Specifically, training a model with the given hyper-parameters for each λ_p value in the range of [0.1, 0.9] and a step size of 0.1 required approximately an hour and 15 minutes in total.

4.2 Parity

The parity task is a binary classification problem that focuses on identifying the parity (whether it is even or odd) of a given set of values. This task is usually applied to sequences of binary digits or numerical values, where the goal is to assign a binary label based on the evenness or oddness of the total count of specific values within the sequence. In the context of our implementation, we utilize a vector of random digits $[-1, 1]$ to implement the parity task, where the labels are assigned based on the parity of the count of $1's$.

We have chosen to approach this task using a Gated Recurrent Unit (GRU) cell for its efficiency in capturing sequential dependencies. The GRU's selective update and reset mechanisms make it well-suited for retaining relevant context in tasks requiring memory of past inputs. Since we have already implemented the pipeline for the MNIST task previously, we have used the same forward step with few modifications. One of them was implementing the clip gradient norm to stabilize the optimization procedure and prevent the occurrence of excessively large gradient values that may lead to numerical instability. It takes around 3.5 hours to train efficiently.

4.2.1 Datasets

In order to create our datasets for training, validating, and testing, we have created a specific function to complete this task. The observations are stored in a tensor of 1,000,000 elements for the training set, 500,000 elements for the validation set, and 1,000,000 elements for the test set. Note that the parity task input comprises a vector consisting of $0s$, $1s$, and $-1s$. The output indicates the parity of $1s$ present; 1 if there is an odd quantity and 0 otherwise. The input is created by setting a random number of elements (from 1 to `VECTOR_LEN`) of a zero vector to a either -1 or 1. Additionally, the data are partitioned into batches of `BATCH_SIZE`.

4.2.2 Hyper-parameters

The hyper-parameters we have set to control our model are:

- `N_HIDDEN` = 64
- `MAX_STEPS` = 20
- `VECTOR_LEN` = 8
- `BATCH_SIZE` = 128
- `BETA` = 0.01
- `LEARNING_RATE` = 0.0003
- `EPOCHS` = 100

We have followed the values of the hyper-parameters utilized in the original paper with some modifications due to computational constraints. To reduce the computational load, we constrained the training epochs to 100 and compressed the vector length to 8, as opposed to the 64 elements specified in the original paper.

The hyper-parameter β was set to 0.01, following the hyper-parameter presented in the original paper for the parity task. This value makes the regularization part of the loss significantly smaller than the reconstruction, therefore the influence of the regularization loss is limited in this experiment.

4.2.3 Experimental setup

Also for the parity task, we have decided to run our experiments on local machines. We have tested on MacBook Pro 16' with ARM architecture and the M1 Max CPU. You can find the code in the Jupyter Notebook `pondernet_parity.ipynb` in the following GitHub repository.

4.2.4 Computational requirements

As previously said, our experiments were carried out on local machines; namely a MacBook Pro 16' with ARM architecture and an M1 Max CPU. Since the computational time was reasonable, we have decided to test the code utilizing only the CPU of our local machines. It takes roughly 3.5 hours to address the parity task for each lambda value. Therefore, to complete all lambda values in the range of [0.1, 0.9] with a step size of 0.1, a total of around 35 hours is necessary.

5 Results

As outlined in the previous sections, our training focused on addressing the image classification on the MNIST dataset and the parity task. In both cases we trained a model for all lambda values in the range of [0.1, 0.9] with a step size of 0.1. The results of these training sessions are visually represented through a series of plots, which are discussed in the following pages. Note that we deliberately chose a subset of visualizations to capture the general trends in the distribution, regardless of the specific lambda values.

5.1 Beta's influence on halting step distribution

An interesting analysis of the model's behavior revolves around its response to the geometric distribution. Specifically, in the context of image classification on the MNIST dataset, the plots in Figures 1, 3, and 5 illustrate the impact of the geometric distribution introduced in the regularization loss over the distribution of halting step. Notably, with a β value of 0.05, it becomes evident that the distribution exhibits a moderate bias towards the proposed reference geometric distribution.

On the other hand, the visualizations for the parity task, as shown in Figures 2, 4 and 6, indicate that - in this case - the geometric distribution did not have significant influence during training. The scaling factor β for L_{reg} plays a crucial role in determining the extent to which the geometric distribution influences the halting probability for each step. Our results suggest the need for a higher β which would make L_{reg} weigh more on the total loss pushing the halting step distribution closer to the geometric distribution. However, the value of β cannot be set too high as the model should be able to break free of the suggestion made by the

geometric distribution if results are sub-optimal. This emphasises the need for a balanced and task-specific tuning of hyper-parameters for optimal model performance. The significance of the β factor can be further understood by examining the training reconstruction and regularization losses, as illustrated in Figure 7. Here, we observe that the variance of L_{reg} is minimal, indicating its negligible impact over L_{rec}, and therefore over the total loss.

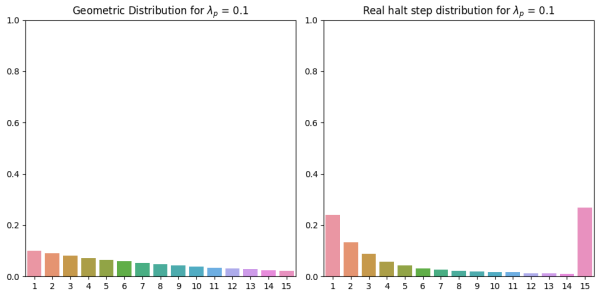

Figure 1: Step distributions for $\lambda_p = 0.1$ (MNIST)

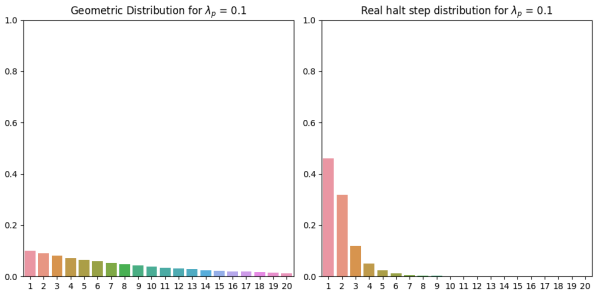

Figure 2: Step distributions for $\lambda_p = 0.1$ (Parity)

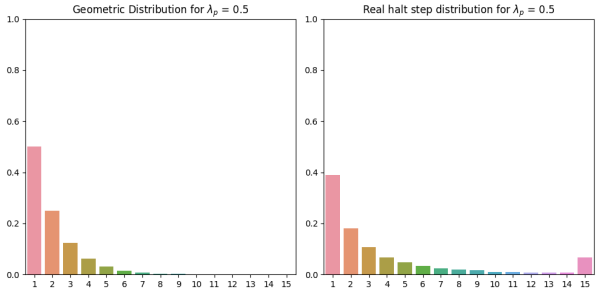

Figure 3: Step distributions for $\lambda_p = 0.5$ (MNIST)

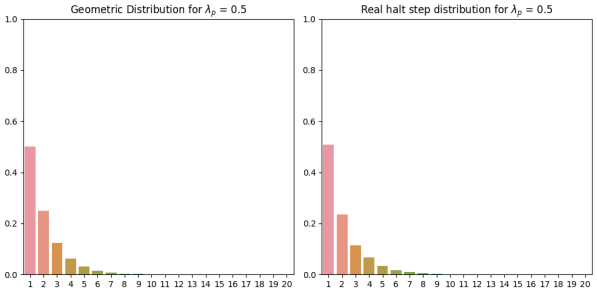

Figure 4: Step distributions for $\lambda_p = 0.5$ (Parity)

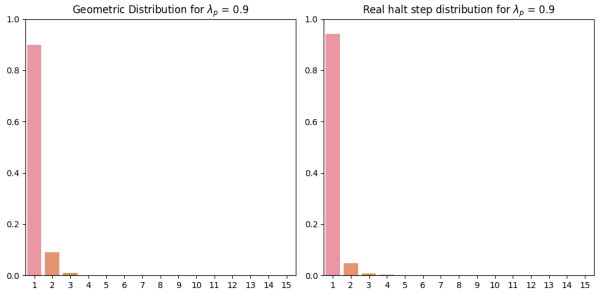

Figure 5: Step distributions for $\lambda_p = 0.9$ (MNIST)

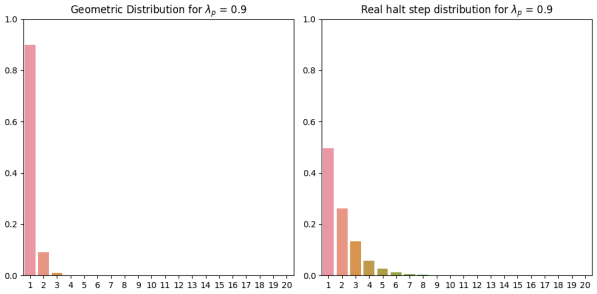

Figure 6: Step distributions for $\lambda_p = 0.9$ (Parity)

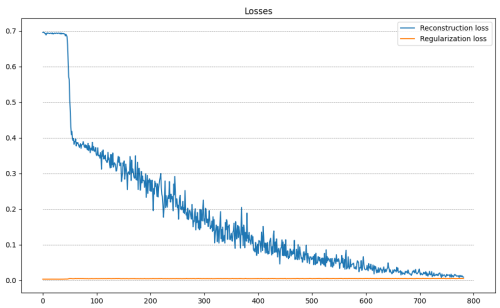

Figure 7: Losses during training with $\lambda_p = 0.3$

5.2 Number of computational steps

We tested the performance of our model by comparing the number of computational steps performed in two scenarios: one with dynamic computation of the halting step using PonderNet, and the other without. To assess the reduction in computational steps with our implementation of PonderNet, we conducted two tests for both MNIST classification and the parity task. The first test involved halting at the step determined dynamically by PonderNet, while the second test simulated a conventional network without PonderNet by halting at the `MAX_STEP`.

In Figures 8 and 9, we can see that the model configured with $\lambda_p = 0.3$, successfully halted after 114,523 steps using PonderNet, contrasting with the 150,000 steps required in a regular network to complete the task. The difference in computational steps is even more pronounced in the parity task, as illustrated in Figures 10 and 11. In this case, rather than executing the full 20,000,000 steps, the process stopped after only 1,753,922 steps. Despite the reduction in the number of computational steps, the overall accuracies of the models are not compromised, demonstrating the ability of the models to achieve great performance while efficiently managing computational resources.

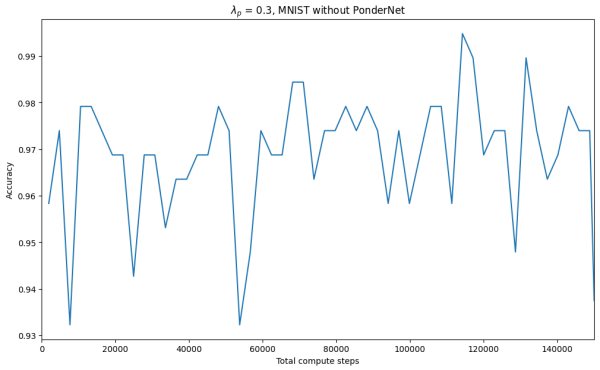

Figure 8: Compute steps without PonderNet (MNIST)

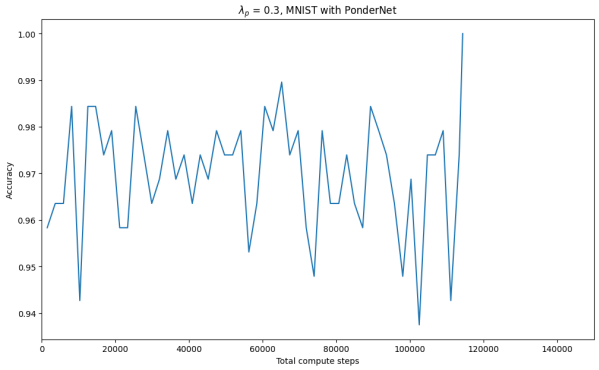

Figure 9: Compute steps with PonderNet (MNIST)

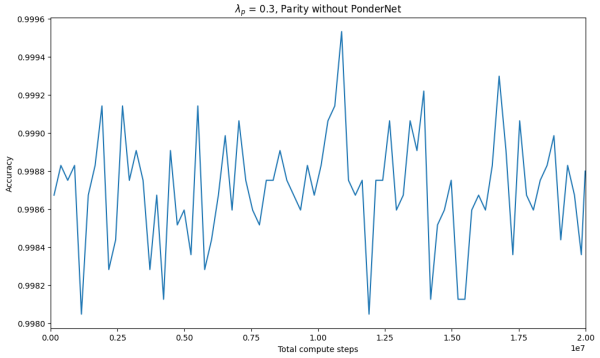

Figure 10: Compute steps without PonderNet (Parity)

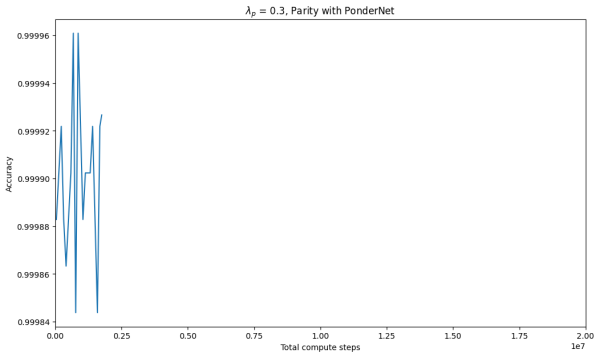

Figure 11: Compute steps without PonderNet (Parity)

Figure 12 shows that, despite some variance in mean accuracy at higher λ_p values, there are no instances where PonderNet failed to resolve the classification task with the MNIST dataset.

6 Discussion and conclusion

In this study, we implemented and evaluated PonderNet, which is a new method that dynamically computes the number of steps needed based on the input. We concentrated our efforts in the analysis of the effect of the regularization loss and evaluating the efficiency of computational tasks with PonderNet.

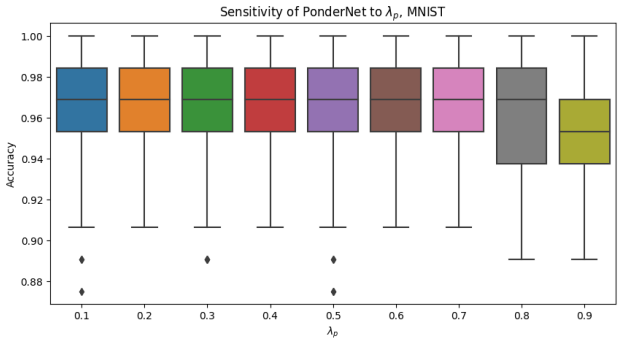

Figure 12: Sensitivity of PonderNet to λ_p on MNIST

Our analysis on the effect of β on the distribution of halting gave interesting results. For example, when classifying MNIST, a β value of 0.05 showed a noticeable bias towards the proposed geometric distribution. Conversely, the effect of the geometric distribution was less evident in the parity task, highlighting the importance of careful tuning.

Additionally, our implementation of PonderNet has evidenced its effectiveness in minimizing the required computational steps for both MNIST classification and the parity task. PonderNet has demonstrated its ability to improve computational efficiency without compromising model accuracy.

In conclusion, our research adds to the understanding of the PonderNet approach, clarifying how hyper-parameters, particularly the scaling factor β, impact on model behaviour. Our implementation of PonderNet reinforces the claims of the original paper showing comparable performances with a noticeable reduction in computational steps, opening up possibilities for further investigation and use in different areas.

References

[1] Andrea Banino, Jan Balaguer, and Charles Blundell. Pondernet: Learning to ponder, 2021.

[2] conradkun. Pondernet: complexity in mnist.

[3] Tolga Bolukbasi, Joseph Wang, Ofer Dekel, and Venkatesh Saligrama. Adaptive neural networks for efficient inference, 2017.

[4] Alex Graves. Adaptive computation time for recurrent neural networks, 2017.

[5] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.

[6] Junyoung Chung, Sungjin Ahn, and Yoshua Bengio. Hierarchical multiscale recurrent neural networks, 2017.