PonderNet does wonders: Using PonderNet to determine the optimal amount of computational iterations

Alex Labro
11872470
University of Amsterdam

Guilly Kolkman
11822465
University of Amsterdam

Joris Hijstek
11876980
University of Amsterdam

Abstract

Some of the most important problems of modern computing research include efficiency. It is not always feasible to allocate more computing power to a problem due to the ever-increasing costs of modern architectures. PonderNet tries to optimize the computation-performance trade-off of supervised architectures that apply a recurrent function to obtain predictions by regulating the amount of times this function is applied to each input. In this paper, we analyze the effectiveness of PonderNet on a convolutional neural network (CNN) with three different datasets: MNIST, Fashion MNIST and CIFAR-10.

1 Introduction

Computational complexity has been one of the most important subjects in computer science ever since the conception of the very concept of computers. While computation is cheap and more abundant by the year, there is still a rising demand for solutions to the ever-increasing complexity of state-of-the-art models.

Usually, machine learning researchers have to manually find a balance between computing time and performance, which is a time-consuming and labour-intensive task sometimes referred to as *pondering*. PonderNet[1] is an architecture that allows an architecture to ponder automatically, that is, decide when to 'halt' the recurrent function from being applied to the current prediction again. For example, when considering the MNIST digit dataset, PonderNet might decide to use fewer steps when classifying the digit six, as it has distinctive features, but might use more steps when classifying the digits one and seven, as they have fewer distinctive features.

In contrast to traditional machine learning tasks that try to train a model to map an input to a prediction, recurrent architectures instead attempt to learn a recursively applicable function from a given input and a *given desired output*. This function is then repeatedly applied to attempt to reconstruct the desired output as closely as possible. The amount of iterations can be statically set as a parameter by researchers, but could also be estimated during training. PonderNet is a prime candidate for performing this estimation.

In this paper, we will examine the effectiveness of PonderNet in several experiments to see if it can be applied in machine learning contexts to effectively decrease computational cost without losing signicant performance. We expect PonderNet to slightly decrease in overall accuracy, while significantly decreasing the average amount of necessary steps.

2 Related work

2.1 Precursors to PonderNet

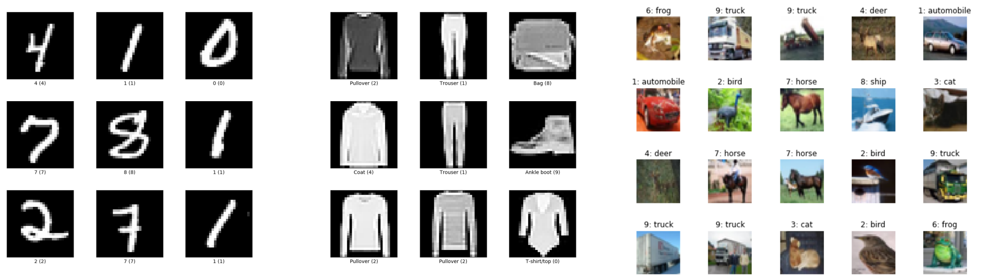
PonderNet is not the first architecture to attempt to optimize the computational power used for different inputs. For example, the *Adaptive Computation Time (ACT)*[3] was proposed, which attempted to automatically learn to scale the required computation time via a scalar halting probability. The problem with this approach is that it is not stable, and is reliant on the choice of a single hyperparameter that determines the trade-off between computational cost and performance.
With REINFORCE, Williams[4] attempts to use a discrete latent variable to adjust the number of computation steps.
Yu et al.[5] and Campos et al.[2] explored skipping certain elements in an input sequence all together, leading to similar performance but increased time efficiency.

2.2 Datasets

In this paper, three different datasets are used to test the effectiveness of PonderNet. All datasets are split into training, validation and test sets using a 80/10/10 split for train, test and validation set respectively.

1. **MNIST**
   MNIST is a dataset which consists of 28x28 grayscale images of handwritten single digits as shown in figure1a. It contains 60.000 samples divided over 9 classes, written by humans.

2. **Fashion MNIST**
   Fashion MNIST is a dataset containing 70.000 28x28 grayscale images of 10 classes of fashion items as shown in figure 1b: tops, trousers, pullovers, dressed, coats, sandals, shirts, sneakers, bags, and ankle boots.
3. **CIFAR-10**
   The CIFAR-10 dataset (*Canadian Institute For Advanced Research*) consists of 50.000 32x32 colour images of one of ten mutually exclusive classes, visible in figure 1b: airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks.



(a) MNIST (b) Fashion MNIST (c) CIFAR-10

Figure 1: Samples of the MNIST, FMNIST and CIFAR10 datasets

3 Approach

3.1 PonderNet

The PonderNet architecture is used to decide when to halt the current computation step. In order to function, the PonderNet architecture requires any neural network where the ponder step can be applied recurrently. Where the step function requires the form $\hat{y}_n, h_{n+1}, \lambda_n = S(x, h_n)$ as well as an initial state h_0. X is represents the original input, h represents the hidden state that is propagated through the different steps and y is the output of the current step. Lastly λ_n is the probability of halting at the current step. Thus, this means that the step function takes the original input and the latest hidden state. It produces a new prediction, the updated hidden state and the probability of halting the next step. A biased coin will be flipped with probability λ, which decides if it will propagate further. The value of λ_n determines the optimal amount of iterations. The authors define a Bernoulli random variable in order to represent the halting process:

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

Which represents the probability of 'halting' at iteration n out of N. This also allows for only the maximum number of pondering steps. When the last step is reached λ is forced to be 1, thus it will halt.

PonderNet still requires training, which is done by optimizing the loss, as visible in equation 2. This loss is divided into two elements: the reconstruction loss and the regularization loss. The trade-off between the losses is regulated by the hyperparameter β. First, the reconstruction loss calculates the loss of the output from that step, after which it is then weighted by the probability that it stopped on that step. For this paper, the cross-entropy loss was used to classify the output. Secondly, the regularization loss has a bias that controls how the halting mechanic should behave. It attempts to minimize the KL-divergence, which depends on the halting distribution generated by PonderNet and the prior geometric distribution with the hyperparameter λ_p. Two things happen with this regularization term. Firstly, the expected number of pondering steps is biased towards $1/\lambda_p$. Furthermore, it allows for the model to stop at every step since halting has a positive probability on each step.

$$L = \underbrace{\sum_{n=1}^{N} p_n \mathcal{L}(y, \hat{y}_n)}_{L_{Rec}} + \underbrace{\beta KL(p_n || p_G(\lambda_p))}_{L_{Reg}} \tag{2}$$

3.2 Our contribution

Contrary to the original paper, where a large RNN was used to evaluate the trade-off between performance and computation, we use a rudimentary CNN to consider PonderNet's trade-off. This architecture was chosen to support the claim from the original paper that PonderNet can be applied to any architecture. Along with trade-off evaluation on a small dataset, we also contribute to the body of literature with a verification of PonderNet's performance compared to the same model without PonderNet. Our GitHub repository can be found in AppendixA

3.3 Architecture

The architecture chosen for this paper consists of three sections: a CNN section, MLP section and an output section. This specific architecture was chosen to be an architecture that simple as possible, while still getting positive results on the simple datasets like MNIST and FMNIST. The aim of this network is to create an embedding using the CNN section, get an output using the MLP section and judge that output using the lambda layer.

The CNN section consists of two convolutional layers with filter sizes 10 and 20 respectively, both with a kernel size of 5. The second convolutional layer goes into a dropout layer with a dropout of 0.2, followed by a max pooling layer and a ReLU. The first layer is identical to the second, except it does not contain a dropout layer. The CNN section ends with a linear layer with an output size of 50.
The MLP section is a 2-layer fully connected neural network. It consists of one linear layer of size 64, followed by a ReLU which is followed by a dropout layer with dropout 0.2 which is followed by a linear layer of size 50 which is followed by a ReLU.
Finally, the output section consists of two seperate layers: one linear layer of size 50 called the lambda layer and one linear layer called the output layer. The lambda layer output is a float that contains a halting probability, and the output of the output layer is a class prediction.
The embedding obtained by the CNN is combined with the old hidden state h, which then flows into the MLP section that creates a new h. h then flows into the output section, where the output layer calculates a class prediction $\hat{y}$ and the lambda layer calculates a halting probability λ.

3.4 Training

The architecture was trained for 40 epochs on the three datasets separately with an Adam optimizer, a learning rate starting at 0.001 which is multiplied by 0.1 if there is no significant loss change for 2 epochs and a batch size of 64. Due to the limited timeframe, we were not able to conduct a thorough hyperparameter search, so hyperparameters were chosen by experience and trial and error.

4 Experiments

To verify PonderNet's versatility, several experiments were performed. To keep things simple in the limited timeframe that was available, a relatively small two-layer CNN-architecture was used as our model, as previously mentioned. In all experiments, the model was trained using the PonderNet loss function or, in the case of no PonderNet being used, using the cross-entropy loss. Thus, two models were created (Vanilla model and Ponder model) which were compared to see the effect of PonderNet. The maximum amount of iterations that the models could perform was 20, which results in the fact that the Vanilla models all run for 20 iterations.

4.1 Experiment 1: Datasets

To verify if PonderNet is able to perform similarly to the Vanilla model, three models were trained on the MNIST, FMNIST and CIFAR10 datasets respectively, and were consequently prompted to perform a classification task. Their validation accuracy was monitored for both the Vanilla and Ponder models, as well as the average halting step for the Ponder model.
We expect that the model will be able to perform classification similar to the Vanilla model while needing fewer computational steps, although it is likely that CIFAR10 will require a higher amount of steps than MNIST and FMNIST.

4.2 Experiment 2: Image transformation

To verify if the model computationally benefits from the pondering technique, the model's performance must be tested on a task with a higher difficulty. Therefore, two models were trained on MNIST and FMNIST respectively. The training samples in these datasets however, were rotated clockwise by 22 degrees. Then, during testing, the testing samples were also randomly rotated clockwise, where rotations were chosen from the set $\{22, 45, 67, 90\}$.
The rotations act as a artificial method of improving the difficulty, so the expectation is that the model will halt at a higher step than in the non-rotated dataset.

5 Results

In this section we cover the results from the aforementioned experiments.

5.1 Results of experiment 1

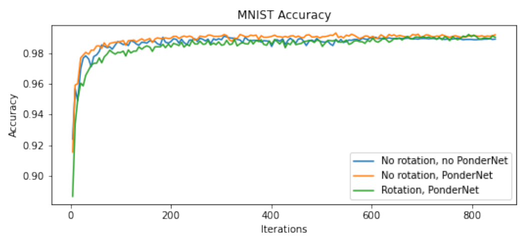
In figure 3a we can see that the accuracy difference between the three methods with MNIST are nearly identical, staying at around ±0.98 accuracy. This can also be seen in figures 3b and 4a.
The loss at the end of training, however, does differ for the three methods with MNIST. The loss with no rotation and no Pondernet is around 0.13 while the loss for the tests with Pondernet are lower, as shown in figure 3c. Meanwhile, the loss has the same curvature for MNIST. However, FMNIST with rotation and with Pondernet has a slower loss decrease in comparison to the other two methods as shown in figure 3d. One likely reason for this phenomenon is that the input size for FMNIST is larger than that of MNIST.

5.2 Results of experiment 2

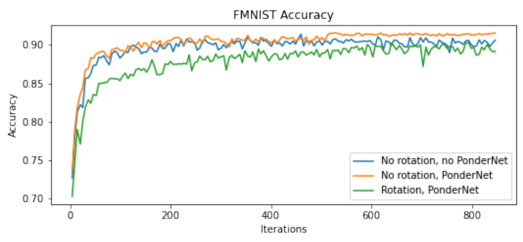
Another part of the experiment was to see the mean decrease in steps used for inference. The default number of steps, for networks without PonderNet, was 20. In contrast, when PonderNet was used, the steps at the end of training were 6 for MNIST, 12 for FMNIST and 18 for CIFAR10 respectively3e3f4c. The behavior of MNIST and FMNIST is as expected from the original paper[1]. In contrast the amount of steps for CIFAR10 increased over time, starting with 8 but then gradually going up to 18 steps at the end of training. Nevertheless, the accuracy of CIFAR10 did increase over time. Thus, one plausible conclusion would be that to increase the accuracy the computational power of 18 steps was necessary to improve classification.

	PonderLoss accuracy	Non-PonderLoss accuracy	Avg steps non-rotated	Avg steps rotated
MNIST	0.989	0.986	5.6	6.3
FMNIST	0.914	0.902	12	9.6
CIFAR10	0.621	0.6113	18	-

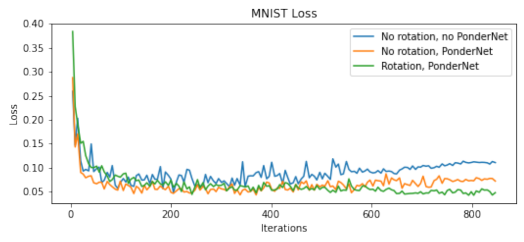
Figure 2: Table displaying the average validation accuracy for models trained with PonderLoss and models trained with Non-Ponderloss (cross-entropy loss). The table also displays the average amount of iterations that are deemed necessary by pondernet to get a sufficient result.



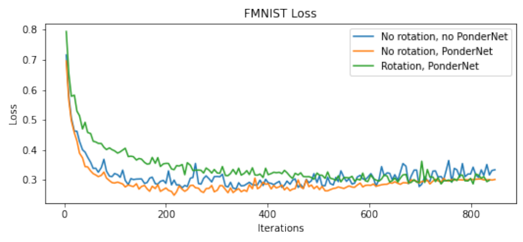
(a) The val accuracy graph of the MNIST dataset



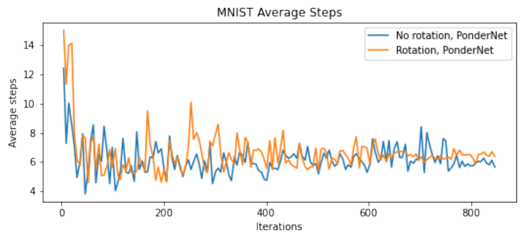
(b) The val accuracy graph of the FMNIST dataset



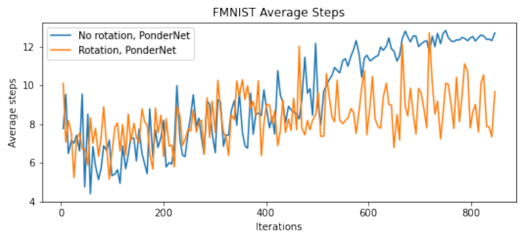
(c) The loss graph of the MNIST dataset



(d) The loss graph of the FMNIST dataset

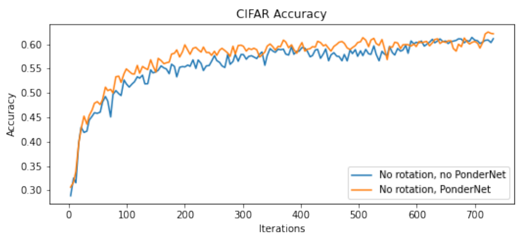


(e) The average step graph of the MNIST dataset

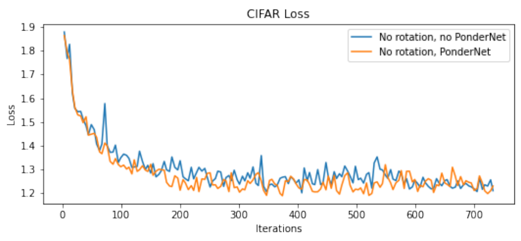


(f) The average step graph of the FMNIST dataset

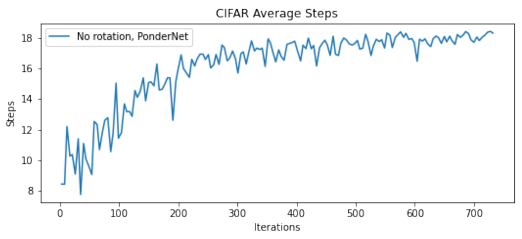
Figure 3: Results over MNIST and FMNIST. Accuracy, loss and average amount of steps



(a) The accuracy graph of the CIFAR10 dataset



(b) The loss graph of the CIFAR10 dataset



(c) The average step graph of the CIFAR10 dataset

Figure 4: Results over the CIFAR10 dataset. Accuracy, loss and amount of steps

6 Conclusion

In this paper, we compared three distinct datasets: MNIST, FMNIST, and CIFAR10. These datasets contained images with corresponding labels, and were classified using three different setups: No PonderNet and no rotation, PonderNet and no rotation, and PonderNet and rotation. The difference was achieved by altering the loss function of the network to not use the Ponder loss, and by rotating the images to be classified. Looking at the results, we observe that the average steps decrease significantly from the default of 20, which, when combined with the accuracy staying within a small margin of the non-PonderNet models. We also observe a slight decrease with the CIFAR10 dataset with a mean steps of 18 instead of the default 20.

An unexpected observation, however, is that the average amount of steps for FMNIST with rotation is lower than that of the model without rotation.

Overall, it seems that PonderNet is able to decrease the average amount of steps required to reach the same margin of accuracy as a non-Pondernet model.

7 Discussion

One of the most notable results in this report is the average steps of the FMNIST dataset. Contrary to our hypothesis, there seems to be a higher demand for more iterations when *not* applying rotation as opposed to applying it. One reason as to why this might be the case could be that the CIFAR10 dataset is easier to classify when rotated, leading to a decrease in the average amount of steps.

An improvement could be made by using cross-validation to verify the findings of this paper. We could then further support our conclusion that PonderNet is able to decrease the amount of steps while keeping the accuracy close to the original.

References

[1] A. Banino, J. Balaguer, and C. Blundell. Pondernet: Learning to ponder. *arXiv preprint arXiv:2107.05407*, 2021.

[2] V. Campos, B. Jou, X. Giró-i Nieto, J. Torres, and S.-F. Chang. Skip rnn: Learning to skip state updates in recurrent neural networks. *arXiv preprint arXiv:1708.06834*, 2017.

[3] A. Graves. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*, 2016.

[4] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

[5] A. W. Yu, H. Lee, and Q. V. Le. Learning to skim text. *arXiv preprint arXiv:1704.06877*, 2017.

A Appendix

https://github.com/GuillyK/DL2-pondernet