

UNIVERSITAT POLITÈCNICA DE CATALUNYA

TREBALL FINAL DEL GRAU EN ENGINYERIA FÍSICA

Reproducing and analyzing Adaptive Computation Time in PyTorch and TensorFlow

Author:
Daniel FOJO

Supervisors:
Víctor CAMPOS
and Xavier GIRÓ

Universitat Politècnica de Catalunya
Barcelona Supercomputing Center

January, 2018

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Abstract

TelecomBCN

Signal Theory and Communications Department

Engineering Physics

Reproducing and analyzing Adaptive Computation Time in PyTorch and TensorFlow

by Daniel FOJO

The complexity of solving a problem can differ greatly to the complexity of posing that problem. Building a Neural Network capable of dynamically adapting to the complexity of the inputs would be a great feat for the machine learning community. One of the most promising approaches is Adaptive Computation Time for Recurrent Neural Network (ACT) (Graves, 2016). In this thesis, we implement ACT in two of the most used deep learning frameworks, PyTorch and TensorFlow. Both are open source and publicly available. We use this implementations to evaluate the capability of ACT to learn algorithms from examples. We compare ACT with a proposed baseline where each input data sample of the sequence is read a fixed amount of times, learned as a hyperparameter during training. Surprisingly, we do not observe any benefit from ACT when compared with this baseline solution, which opens new and unexpected directions for future research.

Acknowledgements

First, I would like to acknowledge my advisor Xavier Giró, for helping me with this work, inviting me into his team, and showing me the world of research. I would also like to acknowledge Víctor Campos for his help with this work and for helping me get in the world of machine learning. I am very much looking forward to keep working on research with both of them after this work.

Contents

Abstract	iii
Acknowledgements	v
1 Introduction	1
2 Theoretical Background	3
2.1 Machine Learning Basics	3
2.1.1 The task T	3
2.1.2 The Performance Measure, P	4
2.1.3 The Experience, E	4
2.1.4 Capacity, Overfitting and Underfitting	4
2.1.5 The No Free Lunch Theorem	5
2.2 Gradient-Based Optimization	6
2.2.1 The loss function	6
2.2.2 Gradient descent	6
2.2.3 Stochastic Gradient Descent	7
2.2.4 Momentum	7
2.3 Deep Learning Basics	7
2.3.1 Feedforward Neural Networks	8
Backpropagation	8
2.3.2 Recurrent Neural Networks	9
Long Short-Term Memory	9
3 Related Work	11
4 ACT model	13
4.1 ACT model	13
4.2 Limiting Computation Time	15
4.3 Error gradients	16
5 Implementing ACT	17
5.1 PyTorch	17
5.1.1 <i>HogWild!</i>	18
5.2 TensorFlow	18
6 Experiments	19
6.1 A new baseline: Repeated inputs	19
6.2 Tasks	19
6.2.1 Parity	20
6.2.2 Addition	20
6.3 Training Parameters	21
6.4 Results	21
6.4.1 Parity	22

6.4.2 Addition	23
7 Conclusions and Future Work	27
Bibliography	29

List of Figures

2.1	Underfitting and overfitting	5
2.2	Momentum	8
2.3	Diagram of an LSTM cell	10
3.1	S-ACT for ResNets	11
3.2	LSTM-Jump	12
4.1	RNN	13
4.2	ACT	15
6.1	Repeated inputs	19
6.2	Parity task example	20
6.3	Addition task example	21
6.4	Parity task with ACT	22
6.5	Parity task repeated inputs	23
6.6	Addition task with ACT	24
6.7	Addition task with repeated inputs	25
6.8	Ponder distribution for the Addition task	26

List of Tables

6.1	Performance of a simple RNN, an RNN with ACT and an RNN with repetitions in the Parity task. Here we can see that our baseline with repetitions can outperform ACT in this task.	23
6.2	Performance of a simple LSTM, an LSTM with ACT and an LSTM with repetitions in the Addition task. Here we can see that our baselines with repetitions outperforms ACT when choosing the right amount of repetitions.	25

List of Abbreviations

SGD	S tochastic G radient D escent
MLP	M ulti L ayer P erceptron
NN	N eural N etwork
RNN	R ecurrent N eural N etwork
LSTM	L ong S hort T erm M emory
ACT	A daptive C omputation T ime
CPU	C entral P rocessing U nit
GPU	G raphics P rocessing U nit

Chapter 1

Introduction

The amount of time required to pose a problem and the amount of computation required to solve it are notoriously unrelated. A clear example of this is the Traveling Salesman Problem: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?", which is believed to be unsolvable in polynomial time (Cook, 2012). In general, we expect the time necessary to solve a problem to increase with the complexity of the task. However, in general, machine learning algorithms do not allocate resources based on the complexity of the task.

Recurrent Neural Networks (RNNs) are a type of Neural Networks that have a *state* or *memory*, and can naturally handle sequences of input samples of any length. However, a vanilla RNN will treat all input samples and states the same way, independently of their complexity or relevance for the machine learning problem to solve. For example, a classical task solved by RNNs is character prediction for language modeling: given a sequence of characters, the following character must be predicted. This task clearly requires different amount of computation because, for example, predicting the character that will come after **elephan** is easy, it will most likely be just **t**. On the other hand, predicting which character will come after **elephant.** is much harder, and may require a more complex model. Dealing with a task of irregular complexity with a static model is prone to problems: designing a model for the simple cases will inevitably cause errors at the challenging inputs, while using a high capacity model to be able to deal with the complex cases will result in an inefficient use of resources when treating the simple states. A much better solution would be to use machine learning models that may adjust their computational cost according to the data that is being processed.

One of the most popular approaches to adjust the complexity of the model to the data is *Adaptive Computation Time (ACT)* for RNNs (Graves, 2016). This model is able to increase computation time by feeding each input sample to the network more than once. For example, in the former case of predicting the character after **elephant.**, the dot character might be fed multiple times to the RNN to allow the model more iterations before taking a decision. The basic assumption of ACT is that it will learn how many times each input should be processed, giving more or less computation time to each step as needed.

This bachelor thesis provides a detailed analysis of ACT and sheds some light about its potentials and limitations. Our work can be summarized with the following contributions:

1. implementation of ACT in the two most popular software frameworks for deep learning, TensorFlow and PyTorch,

2. design and implementation of a novel baseline capable of solving the same tasks as ACT, based on a fixed amount of repetitions set at training time as a new hyperparameter,
3. comparison of ACT with the proposed baseline which, surprisingly, indicates that ACT does not bring any gain and opening this way unexpected directions for research that we plan to address in the future.

Further than the ACT algorithm itself, we also want to raise awareness about current discussions about reproducibility in machine learning. This thesis has devoted a very important amount of resources in implementing and reproducing the results from (Graves, 2016), as its authors limited their publication to a preprint on arXiv, with no accompanying source code. This has been a drawback when reproducing the published results, and it is seen with concern by some part of the community. For example, ICLR2018 (one of the most important machine learning conferences) will host the Reproducibility Challenge, a workshop on reproducing important deep learning papers¹. In order to help the community in this sense, we have published our implementation in the PyTorch and TensorFlow deep learning frameworks, and both implementations are open and publicly available at <https://imatge-upc.github.io/danifojo-2017-tfg/>.

¹<http://www.cs.mcgill.ca/~jpineau/ICLR2018-ReproducibilityChallenge.html>

Chapter 2

Theoretical Background

2.1 Machine Learning Basics

Deep learning is a specific kind of machine learning. To understand deep learning well, one must also have a solid understanding of machine learning. Machine learning is a form of applied statistics with an increased emphasis on the use of computers to statistically estimate complicated functions and a decreased emphasis on proving confidence intervals around these functions.

Tom Mitchell (Mitchell, 1997) describes a machine learning task as follows: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P , if its performance at tasks in T , as measured by P , improves with experience E ."

2.1.1 The task T

In this definition of "task", the process of learning itself is not the task. For example, if we want a program to learn to distinguish dogs and cats, then distinguishing dogs and cats is the task.

Machine learning tasks are usually described in terms of how the system should process an example. An example is a collection of features from some object or event that we want the algorithm to process. We typically represent an example as a vector $x \in \mathbb{R}^n$ where each entry x_i of the vector is a feature. For example, the features of an image are usually the values of the pixels in the image. Some typical examples of tasks are:

- **Classification:** Classification tasks ask the computer program to specify to which category does one input vector belong. Formally, this is equivalent to approximate a function $f : \mathbb{R}^n \mapsto \{1, 2, \dots, k\}$.
- **Regression:** In this type of task, the computer program is asked to predict a numerical value given some input. This consists in approximating a function $f : \mathbb{R}^n \mapsto \mathbb{R}$.
- **Transcription:** In this type of task, the machine learning system is asked to observe an unstructured representation of some kind of data and transcribe the information into text. For example, in optical character recognition (OCR), the computer program is shown an image containing text and is asked to return this text in the form of a sequence of characters.
- **Denoising:** In this type of task, the machine learning algorithm is given an input a corrupted example $\tilde{x} \in \mathbb{R}^n$ obtained by an unknown corruption process from a clean example $x \in \mathbb{R}^n$.

2.1.2 The Performance Measure, P

To evaluate how good our machine learning algorithm is, we must design quantitative measures of its performance. For tasks such as classification, we often measure the accuracy of the model, which is just the proportion of examples for which the algorithm gives the correct output. Similarly, the error rate is the proportion of examples for which the algorithm gives an incorrect output.

2.1.3 The Experience, E

Machine learning algorithms can be understood as being allowed to experience an entire dataset. A dataset is a collection of many examples.

Generally, machine learning algorithms can be categorized as supervised or unsupervised

- **Unsupervised:** The algorithm experiences a dataset containing features and has to learn some structure or pattern from it.
- **Supervised:** The algorithm experiences a dataset containing features, but each example is also associated with a label or target. We will generally refer to the examples as \mathbf{x} and to the targets as \mathbf{y} . When we talk about a specific example and target, we will use superindexes: $\mathbf{x}^{(i)}$ and $\mathbf{y}^{(i)}$.

Throughout this work, we will focus mainly on supervised learning, i.e. each example $\mathbf{x}^{(i)}$ has a corresponding target $\mathbf{y}^{(i)}$.

Some machine learning algorithms do not just experience a dataset. For example, reinforcement learning (Williams, 1992) algorithms interact with an environment, so there is a feedback loop between the algorithm and its experiences. An example of reinforcement learning algorithm is AlphaGo Zero (Silver et al., 2017), an algorithm that learned to play the game of go from scratch better than any human.

2.1.4 Capacity, Overfitting and Underfitting

The central challenge of machine learning is that our algorithm must perform well on unseen examples. This is called generalization. Generally, we do not have access to the set of all possible inputs, and for this reason, our network will only learn from a specific subset which will be an estimation of the complete set. Note that this problem comes from the fact that usually we can only train with a subset of the possible input data, which is an estimate of the real distribution. This is not always the case. For example, when working with synthetic tasks (as the ones in Chapter 6), we sometimes have access to the real distribution, because we are artificially generating it. This means that generalization will not be a problem, since our data will not be biased to a subset.

Typically, when training a machine learning model, we compute some error measure over our training set, called training error. This is simply an optimization problem. In machine learning, we want the test error, computed over a test set which the algorithm has never seen to be low as well. The test error will always be greater or equal than the training error.

The factors to determine how well a machine learning algorithm will perform are its ability to make the training error small, and to make the gap between the training error and the test (generalization gap) error small. These two factors correspond to underfitting (when the training error is too high) and overfitting (when the generalization gap is too high). Whether a model is going to underfit or overfit depends on

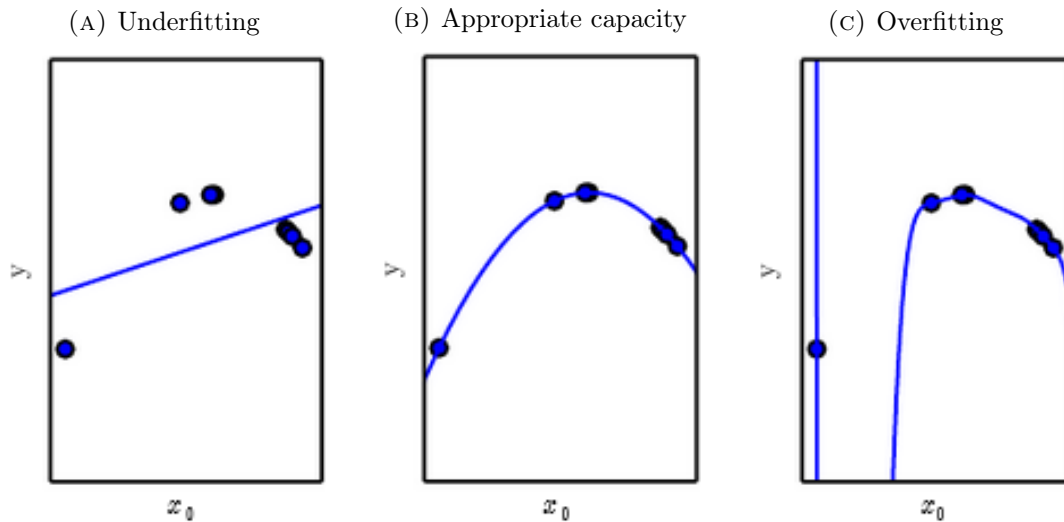


FIGURE 2.1: Here we have three examples of a regression. The training data was generated synthetically, by randomly sampling x values and choosing y deterministically by evaluating a quadratic function. (A) A linear function cannot capture the curvature and underfits the data. (B) A quadratic function fits to the data and generalizes well to unseen points. It does not suffer from overfitting or underfitting. (C) A polynomial of degree 9 overfits the data and does not generalize well to unseen points (Goodfellow, Bengio, and Courville, 2016).

its capacity, which corresponds to the capability of a model to approximate a wide variety of functions. If the model capacity is not enough, it will not be able to properly fit the training set. If the model capacity is too high, it will overfit the training set and the generalization gap will be large.

Regularization is a technique to avoid overfitting, and thus reduce the generalization gap. For example, in figure 2.1, we could "penalize" our algorithm for having large parameters. This would reduce the algorithm capacity (it would increase the training error), but it would help avoid overfitting (decreasing the generalization gap). There are many ways to regularize, and which is the correct way depends on the task we want our algorithm to solve.

2.1.5 The No Free Lunch Theorem

The no free lunch theorem (Wolpert, 1996) says that, averaged over all possible data, every classification algorithm has the same error rate when classifying previously unobserved points. This means that no machine learning algorithm is better than any other in all possible tasks.

Fortunately, this results only holds when we average over all data-generating distributions. This means that if we make assumptions about the distributions that we will find in the real world, we can design an algorithm that does well in these distributions.

The no free lunch theorem implies that we must design our algorithm to work well only on a single task. This is done by assuming hypothesis into the algorithm. For example, a linear regression will work well if the outputs come from a linear function applied to the inputs, but will not work if the data is of the form $y = \sin(x)$.

2.2 Gradient-Based Optimization

Many machine learning algorithms and nearly all deep learning algorithms use Stochastic Gradient Descent (SGD), which is an extension from the classical Gradient Descent.

2.2.1 The loss function

Generally, for an algorithm to learn a task it must optimize some objective function or loss function. In general, this function will be a measure of the error, and we will want to minimize it with respect to the parameters of our algorithm. Some of the most common loss functions are:

- **Mean Squared Error:** usually used in regressions. For each example, the loss is:

$$L(x, y) = \sum_{i=1}^n (x_i - y_i)^2 \quad (2.1)$$

- **Binary Cross Entropy:** usually used in binary classification. Here, y can be either 1 or 0, and x is the probability of that input of belonging to class 1 (usually obtained by applying a sigmoid function). For each example, the loss is:

$$L(x, y) = y \log(x) + (1 - y) \log(1 - x) \quad (2.2)$$

- **Categorical Cross Entropy:** usually used in multiclass classification. Here, y is a one-hot encoding (a vector with all zeros except a 1 in the position it encodes), and x are the probabilities of belonging to each class of that input (usually obtained by applying a softmax function). For each example, the loss is:

$$L(x, y) = \sum_{i=1}^n y_i \log(x_i) \quad (2.3)$$

There are many more loss functions, e.g. L_1 , Noise Contrastive Estimation (Gutmann and Hyvärinen, 2010) or triplet loss (Schroff, Kalenichenko, and Philbin, 2015). The choice of an appropriate loss function is fundamental for the machine learning algorithm to be able to learn.

2.2.2 Gradient descent

It is well known that the gradient of a function gives the direction of maximum ascend, so, for small enough ϵ , $f(\mathbf{x} - \epsilon \nabla f(\mathbf{x}))$ will be smaller than $f(\mathbf{x})$. We can iterate this process, which will reduce $f(\mathbf{x})$ at each step. This technique is called gradient descent (Cauchy, 1847).

In machine learning, the ϵ parameter is called learning rate, as it symbolizes the speed at which the algorithm learns. Choosing the right learning rate is crucial, since a learning rate too low will make the algorithm learn too slowly, and a learning rate too high will make the gradient descent diverge and not find a minimum.

If L is the loss function that we want to minimize with respect to some parameters θ , and \mathbf{x} and \mathbf{y} are the examples with their corresponding labels, and \mathbf{f} is the function that we are approximating, the update rule will be:

$$\theta_{t+1} \leftarrow \theta_t - \epsilon \nabla_{\theta} L(\mathbf{f}(\mathbf{x}; \theta_t), \mathbf{y}) \quad (2.4)$$

2.2.3 Stochastic Gradient Descent

A problem in machine learning is that large datasets give better results, but they are also computationally expensive.

Usually, the loss function in a machine learning algorithm decomposes in the sum of some per-example loss function:

$$L_{\text{total}}(\mathbf{f}(\mathbf{x}; \boldsymbol{\theta}), \mathbf{y}) = \frac{1}{m} \sum_{i=1}^m L(\mathbf{f}(\mathbf{x}^{(i)}; \boldsymbol{\theta}), \mathbf{y}^{(i)}) \quad (2.5)$$

If the training examples grows to billions, the cost to make a single gradient step becomes too large.

The idea behind SGD is that the gradient is just an expected value of the largest descent direction, and it can be estimated with a small set of examples. On each step we can sample a minibatch of examples $\mathbb{B} = \{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(m')}\}$ from the whole dataset, and estimate $\nabla f(x)$ using only these examples. The idea is that m' is fixed even though the size of the dataset can grow.

$$\nabla_{\boldsymbol{\theta}} L_{\text{total}}(\mathbf{f}(\mathbf{x}; \boldsymbol{\theta}), \mathbf{y}) \approx \nabla_{\boldsymbol{\theta}} \frac{1}{m'} \sum_{i=1}^{m'} L(\mathbf{f}(\mathbf{x}^{(i)}; \boldsymbol{\theta}), \mathbf{y}^{(i)}) \quad (2.6)$$

2.2.4 Momentum

Even though SGD works well, it can sometimes be slow. The method of momentum is designed to accelerate learning, especially in the face of high curvature, small but consistent gradients or noisy gradients.

The momentum algorithm introduces a variable \mathbf{v} that plays the role of velocity. The name momentum derives from the physical analogy. The new update rule is given by:

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_{\boldsymbol{\theta}} \frac{1}{m'} \sum_{i=1}^{m'} L(\mathbf{f}(\mathbf{x}^{(i)}; \boldsymbol{\theta}), \mathbf{y}^{(i)}) \quad (2.7)$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v} \quad (2.8)$$

Here, α determines how quickly the contribution from previous gradients decays.

There are many other more sophisticated optimizers based on SGD apart from momentum. Some of the more popular ones are RMSprop (Tieleman and Hinton, 2012) and Adam (which we will use for our experiments) (Kingma and Ba, 2014)

2.3 Deep Learning Basics

Deep learning is the family of machine learning algorithms that use Neural Networks to approximate the target function, and learn using some form of gradient descent via backpropagation. Neural Networks are called networks because they are made by composing many non-linear functions: $f(x) = f^{(m)}(\dots f^{(2)}(f^{(1)}(f^{(0)}(x))))$. This allows the network to capture non-linearities, as opposed to simple linear regressions. Each of the functions is called layer, and the amount of layers is called the depth of the network. Usually, each layer of the network is vector-valued. The dimension of each layer is called width. Another way of thinking about these layers is to think of them as many units, each of them representing a vector-to-scalar function.

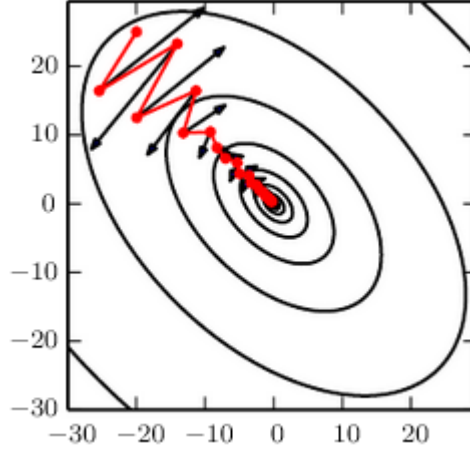


FIGURE 2.2: Here we can see that the steps (in red) that the momentum takes go in the direction of the minimum, while the gradient (in black) points in other directions (Goodfellow, Bengio, and Courville, 2016).

2.3.1 Feedforward Neural Networks

Feedforward Neural Networks, also called Multilayer Perceptrons (MLP) are the most classical example of Deep Learning. Their goal is to approximate a function $f(x)$. They are called feedforward because information (x) only goes forward in the network. A neural network in which there are feedback connections is called Recurrent Neural Network.

In the case of Feedforward Neural Networks, each of the layers is composed of an affine transformation $\phi(x) = Wx + b$ and a non-linear function, such as a Gaussian ($h(x) = \exp(-x^2)$) or a sigmoid ($\sigma(x) = \frac{1}{1+e^{-x}}$). The non-linear function is called the activation. Then, each layer is of the form:

$$f^{(i)}(x) = h\left(W^{(i)}x + b^{(i)}\right) \quad (2.9)$$

The activation functions allow the network to capture non-linearities. Without the non-linearities we would have a composition of linear functions, which is basically a single linear function. Here, the learnable parameters are the weights of the affine transformation at each layer.

Backpropagation

Gradient-based learning in deep learning requires calculating the gradients for quite complicated functions. Numerical approximations for the gradient are slow and can give numerical errors. The main method for calculating the gradients is called backpropagation, and it's based on recursively applying the chain rule from calculus.

For example, given a network in which: $x = f(w)$, $y = g(x)$, $z = h(y)$, to compute $\frac{\partial z}{\partial w}$ we would calculate:

$$\frac{\partial z}{\partial w} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} \frac{\partial x}{\partial w} \quad (2.10)$$

$$= h'(y)g'(x)f'(w) \quad (2.11)$$

$$= h'(g(f(w)))g'(f(w))f'(w) \quad (2.12)$$

The reason because this algorithm is called back propagation is because it is similar to propagation an input through a network, but backwards and with the derivatives of the activations in the right places.

2.3.2 Recurrent Neural Networks

Recurrent Neural Networks or RNN (Rumelhart, Hinton, and Williams, 1986), are a family of networks specialized in processing sequential data $(x^{(1)}, x^{(2)}, \dots, x^{(T)})$. Most RNN can also process sequences of variable length. We will refer to RNN acting on a sequence of vectors x^t with t going from 1 to T .

To understand RNN, consider a classical dynamical system driven by an external signal $x^{(t)}$, in which the state at each timestep is $s^{(t)}$:

$$s^{(t)} = f\left(s^{(t-1)}, x^{(t)}; \theta\right) \quad (2.13)$$

RNN generally take the same form as 2.13, where the state contains information for the past sequence. The state is generally referred as the hidden state.

The most simple example of an RNN would be:

$$s^{(t)} = h\left(W_x x^{(t)} + W_s s^{(t-1)} + b\right) \quad (2.14)$$

A typical choice for the activation function is $h(x) = \tanh(x)$.

A very well known problem with RNNs is vanishing or exploding gradients (Bengio, Simard, and Frasconi, 1994) when calculating them through very long sequences in time, since the gradient at each time step depends on the prior one. A great improvement over the typical RNN that helps with these problems are Long Short-Term Memory RNNs.

Long Short-Term Memory

The most effective sequence models used in Deep Learning are called gated RNNs. One of the most important gated RNNs is Long Short-Term Memory or LSTM.

The equations for the LSTM gates are:

The forget gate $f^{(t)}$:

$$f^{(t)} = \sigma\left(W_s^f h^{(t-1)} + W_x^f x^{(t)} + b^f\right) \quad (2.15)$$

The external input gate $g^{(t)}$:

$$g^{(t)} = \sigma\left(W_s^g h^{(t-1)} + W_x^g x^{(t)} + b^g\right) \quad (2.16)$$

The output gate $q^{(t)}$:

$$q^{(t)} = \sigma\left(W_s^q h^{(t-1)} + W_x^q x^{(t)} + b^q\right) \quad (2.17)$$

With these gates, the state unit $s^{(t)}$ depends on the last output $h^{(t-1)}$ and the current external input $x^{(t)}$, which are controlled by the forget gate $f^{(t)}$ and the external input gate $g^{(t)}$:

$$s^{(t)} = f^{(t)} s^{(t-1)} + g^{(t)} \sigma \left(W_s h^{(t-1)} + W_x x^{(t)} + b \right) \quad (2.18)$$

The output $h^{(t)}$ is also controlled by the the output gate $q^{(t)}$:

$$h^{(t)} = \tanh \left(s^{(t)} \right) q^{(t)} \quad (2.19)$$

Note that it is also common to denote the state unit $s^{(t)}$ as $c^{(t)}$.

This formulation allows the gradients to propagate through time without vanishing or exploding, thanks to the gated self-loop of 2.18. A block diagram is shown in 2.3

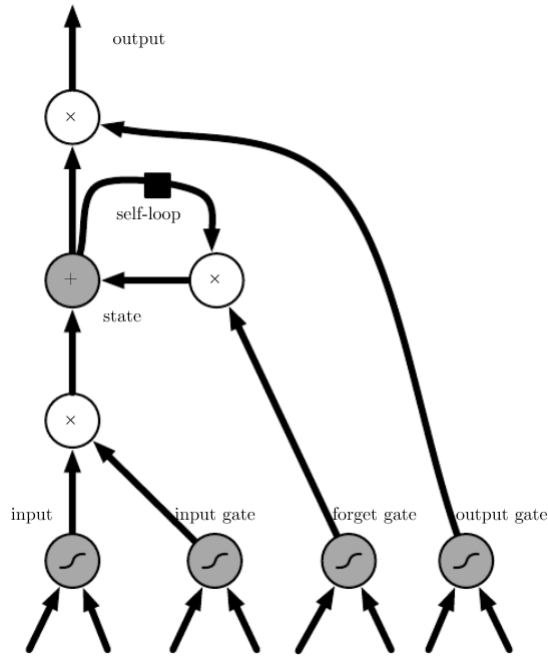


FIGURE 2.3: Block diagram of an LSTM cell. The black square indicates a delay of a single step. (Goodfellow, Bengio, and Courville, 2016)

Our work will focus mainly on RNNs, and 2 of the 3 experiments will use LSTMs.

Chapter 3

Related Work

ACT is not the only deep learning architecture that can dynamically adapt computation to the inputs. Bengio, 2013 presents conditional computation, which allows to increase model capacity without a proportional increase in computational cost by evaluating only certain computation paths for each input. Denil et al., 2013 demonstrates that there is significant redundancy in the weights of several deep learning models. This can be exploited by only using a part of the network for each inputs. An example of this are Adaptive Early Exit Networks (Bolkubasi et al., 2017), which adapt to the inputs by allowing them to exit the network prematurely if no more computation is needed. Bengio, Léonard, and Courville, 2013 explore the idea of having stochastic neurons with hard non-linearities for conditional computation.

Another approach to conditional computation is the use of REINFORCE (Williams, 1992) to adjust the number of computation steps using discrete latent variables. For example, choosing the amount of patches in an image to process (Li et al., 2017), or dropping a subset of neurons in a fully connected layer (Bengio et al., 2015).

For the case of RNNs, an example of conditional computation are LSTMs whose number of layers depends on the input data, and each layer decides whether or not to activate the next one (Chung, Ahn, and Bengio, 2016). Some works do time-dependent computation in RNNs by only updating a fraction of the hidden state depending on the input (Jernite et al., 2016). There is also Spatially-ACT for Residual Networks (Figurnov et al., 2016), which uses halting units like ACT, but unlike ACT, they are not used to look at an input more than once, they are used to decide when to stop propagating the input through a residual block (He et al., 2015), so each input goes through an adaptive amount of layers.

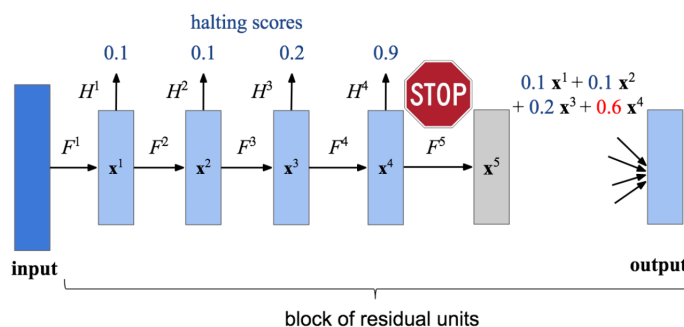


FIGURE 3.1: Example of a block of residual units from Spatially Adaptive Computation Time for Residual Networks. The network propagates the input until the cumulative *halting score* reaches 1, and then it stops. Taken from (Figurnov et al., 2016).

Skipping samples of a sequence in an RNN can be seen as a form of conditional computation. For each sample, the network adaptively decides whether it requires computation or not. With the goal of decreasing computation, LSTM-jump (Yu, Lee, and Le, 2017) decides how many steps to "jump" between RNN updates by using REINFORCE. More recently, without the need of reinforcement learning, Skip RNN (Campos et al., 2017) decides how many samples to "skip" depending only on the input and the hidden state at each time step.

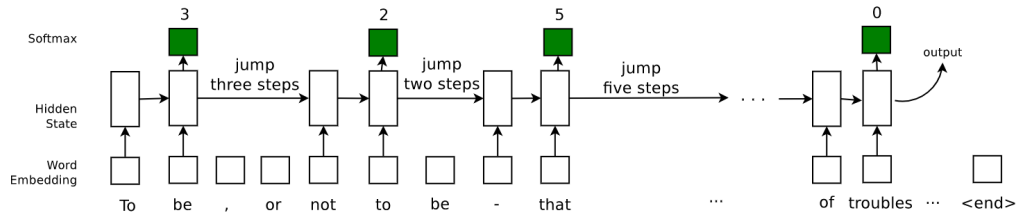


FIGURE 3.2: Example of LSTM-Jump processing a text document. At each timestep, a softmax classification decides how many steps to "jump". Taken from (Yu, Lee, and Le, 2017).

Chapter 4

ACT model

This Chapter provides an overview of the Adaptive Computation Time (ACT) model proposed in (Graves, 2016). This thesis is structured around this proposal by Dr. Alex Graves, one of the most influential researchers in the field of machine learning nowadays. Dr. Graves is currently a research scientist at Google DeepMind, one of the leading research centers in artificial intelligence. He previously obtained his PhD under the supervision of Dr. Jürgen Schmidhuber at IDSIA, where he introduced connectionist temporal classification (CTC) for LSTMs, a key contribution that allows speech recognition on mobile devices. Before joining Google DeepMind, Dr. Graves was a postdoc at the University of Toronto under the supervision of Dr. Geoffrey Hinton. Both Dr. Schmidhuber and Dr. Hinton are considered two of the fathers of deep learning.

4.1 ACT model

Consider a simple RNN with input sequence $x = (x_1, x_2, \dots, x_T)$, hidden states $s = (s_1, s_2, \dots, s_T)$ and outputs $y = (y_1, y_2, \dots, y_T)$. Its equations would be:

$$s_t = \mathcal{S}(s_{t-1}, x_t) \quad (4.1)$$

$$y_t = W_s s_t + b_s \quad (4.2)$$

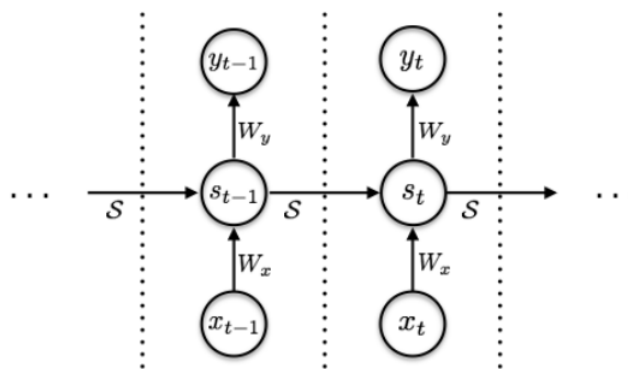


FIGURE 4.1: Computation graph of a simple RNN unrolled over two time steps. (Graves, 2016).

ACT modifies these equations to allow the network to process each input a variable number of times. The modified equations are:

$$s_t^n = \begin{cases} \mathcal{S}(s_{t-1}, x_t^1) & \text{if } n = 1 \\ \mathcal{S}(s_t^{n-1}, x_t^n) & \text{if } n > 1 \end{cases} \quad (4.3)$$

$$y_t^n = W_s s_t^n + b_s \quad (4.4)$$

where $x_t^n = (\delta_{1,n}, x_t)$ is the original input augmented with a binary flag at the beginning of the input that indicates whether it's the first time the network sees the input. Note that the same state function is used for state transitions (for repeated inputs or otherwise), and the weights and bias are also shared.

To determine how many times each input is repeated, an additional halting unit is added:

$$h_t^n = \sigma(W_h s_t^n + b_h) \quad (4.5)$$

Then, the amount of computation steps $N(t)$ is defined as:

$$N(t) = \min \left\{ n' : \sum_{k=1}^{n'} h_t^k > 1 - \epsilon \right\} \quad (4.6)$$

where ϵ is a small constant whose purpose is to allow the network to do only one computation step if needed, since the output from a sigmoid is always smaller than 1. In our experiments, we set $\epsilon = 0.01$.

From the activation of the halting units h_t^n , we define the halting probabilities as:

$$p_t^n = \begin{cases} h_t^n & \text{if } n < N(t) \\ \mathcal{R}(t) & \text{if } n = N(t) \end{cases} \quad (4.7)$$

Where the **residual** $\mathcal{R}(t)$ is defined as:

$$\mathcal{R}(t) = 1 - \sum_{k=1}^{N(t)-1} h_t^k \quad (4.8)$$

With this definition, the halting probabilities add up to 1, so they are a valid probability distribution.

From here, we define the state and the output of our network as:

$$s_t = \sum_{k=1}^{N(t)} p_t^k s_t^k \quad (4.9)$$

$$y_t = \sum_{k=1}^{N(t)} p_t^k y_t^k \quad (4.10)$$

Note that 4.4 and 4.10 can be rewritten as:

$$y_t = W_y s_t + b_y \quad (4.11)$$

which we will use in our implementation.

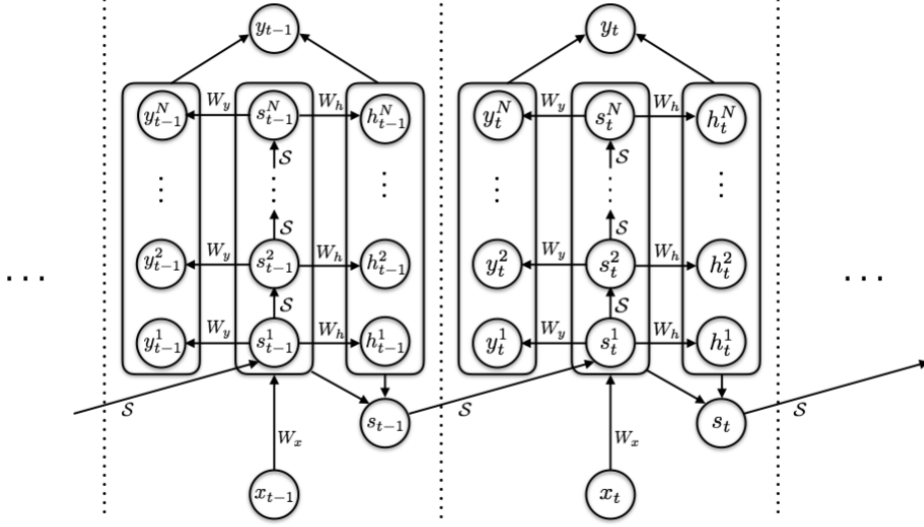


FIGURE 4.2: Computation graph of ACT unrolled over two time steps (Graves, 2016).

4.2 Limiting Computation Time

Without limiting the computation steps the network can take, it could look at each sample indefinitely. To avoid this, we add a ponder cost term to the loss. We define the ponder at each time step as:

$$\rho_t = N(t) + \mathcal{R}(t) \quad (4.12)$$

And the total ponder cost for a sequence \mathbf{x} as:

$$\mathcal{P}(\mathbf{x}) = \frac{1}{N(t)} \sum_{k=1}^{N(t)} \rho_t \quad (4.13)$$

Then, we modify the loss function of the network as:

$$\hat{\mathcal{L}}(\mathbf{x}, \mathbf{y}) = \mathcal{L}(\mathbf{x}, \mathbf{y}) + \tau \mathcal{P}(\mathbf{x}) \quad (4.14)$$

where τ is the a time penalty hyperparameter (a non-learnable parameter that has to be manually set).

The authors of ACT claim that this term minimizes the amount of pondering at each step, although this statement is not explicitly explored in their work. We observed that, since $N(t)$ is constant almost everywhere, we are just minimizing $R(t)$, which in equation 4.8 we see that it has a constant part (1), and the negative sum of the halting probabilities. This means that minimizing $\mathcal{P}(\mathbf{x})$ is equivalent to maximizing the halting probabilities, which makes the network stop earlier.

The network may process each sample for an arbitrarily large number of times at the beginning of training due to the random initialization of the weights. It is possible to limit such behavior by imposing an upper bound on $N(t)$:

$$N(t) = \min \left\{ M, \min \left\{ n' : \sum_{k=1}^{n'} h_t^k > 1 - \epsilon \right\} \right\} \quad (4.15)$$

here, M is the maximum computation for each step.

4.3 Error gradients

The ponder cost $\mathcal{P}(\mathbf{x})$ is not differentiable everywhere. The term $N(t)$ of each step is not differentiable whenever the halting probabilities change such that $N(t)$ changes. But, since $N(t)$ is constant *almost everywhere* (in the mathematical sense), this is just ignored.

Then, the gradients of the ponder costs with respect the halting activations h_t^n are just:

$$\frac{\partial \mathcal{P}(\mathbf{x})}{\partial h_t^n} = \begin{cases} -1 & \text{if } n < N(t) \\ 0 & \text{if } n = N(t) \end{cases} \quad (4.16)$$

The halting activations h_t^n only influence the original loss $\mathcal{L}(\mathbf{x}, \mathbf{y})$ via their effect on the halting probabilities p_t^n :

$$\frac{\partial \mathcal{L}(\mathbf{x}, \mathbf{y})}{\partial h_t^n} = \sum_{k=1}^{N(t)} \frac{\partial \mathcal{L}(\mathbf{x}, \mathbf{y})}{\partial p_t^k} \frac{\partial p_t^k}{\partial h_t^n} \quad (4.17)$$

Since the halting probabilities only influence \mathcal{L} in their states and outputs from equations 4.9 and 4.10 it follows:

$$\frac{\partial \mathcal{L}(\mathbf{x}, \mathbf{y})}{\partial p_t^n} = \frac{\partial \mathcal{L}(\mathbf{x}, \mathbf{y})}{\partial y_t} y_t^n + \frac{\partial \mathcal{L}(\mathbf{x}, \mathbf{y})}{\partial s_t} s_t^n \quad (4.18)$$

while, from equations 4.7 and 4.8 we have:

$$\frac{\partial p_t^i}{\partial h_t^i} = \begin{cases} \delta_{i,j} & \text{if } i, j < N(t) \\ -1 & \text{if } i = N(t), j < N(t) \\ 0 & \text{if } j = N(t) \end{cases} \quad (4.19)$$

Combining equations 4.16, 4.18 and 4.19 we have:

$$\frac{\partial \hat{\mathcal{L}}(\mathbf{x}, \mathbf{y})}{\partial h_t^n} = \frac{\partial \mathcal{L}(\mathbf{x}, \mathbf{y})}{\partial y_t} (y_t^n - y_t^{N(t)}) + \frac{\partial \mathcal{L}(\mathbf{x}, \mathbf{y})}{\partial s_t} (s_t^n - s_t^{N(t)}) + (1 - \delta_{n, N(t)}) \tau \quad (4.20)$$

note that this equation becomes $\frac{\partial \hat{\mathcal{L}}(\mathbf{x}, \mathbf{y})}{\partial h_t^n} = 0$ for $n = N(t)$.

Here we have seen that the network can be differentiated as usual, via backpropagation through time.

Chapter 5

Implementing ACT

Due to the lack of an official repository with code, we had to write our own implementation of ACT. We used Python programming language, which has many frameworks specialized in Deep Learning (e.g. Theano, Caffe, TensorFlow and PyTorch). The main problem with implementing Adaptive Computation Time is its *adaptive* feature. Most Deep Learning frameworks are static (also called lazy execution). This means that a computation graph is firstly defined, then it is executed, and it is difficult to modify this graph during execution. This is why our first decision was to use PyTorch, which is a new dynamic Deep Learning framework. After testing our PyTorch implementation, we run into problems with its training speed, so we ended up implementing it also on TensorFlow, which we used to run our experiments.

We ran our experiments in both CPU and GPU. The main advantage of GPUs is that they allow to speed up computation by doing many calculation at the same time in parallel. Our source code is open and publicly available at <https://imatge-upc.github.io/danifojo-2017-tfg/>.

5.1 PyTorch

PyTorch¹ is a Deep Learning framework that, at the time of this work, was one of the only frameworks that generates the computation graph dynamically (also called eager execution). This means that having an adaptive architecture is much easier and natural. On the other hand, the main drawback of PyTorch is that it is quite recent (as of the time of this work, it was in beta 0.2 version), and many parts of the documentation were missing.

We started with a very naive implementation using CPU, which was too slow, so we decided to implement it for GPUs. For a GPU to work well, it has to be able to run many operations in parallel, so they generally work in batches. This means that they do the same operations for many different input sequences at the same time. This works well with the SGD, since the loss is only computed for a batch of inputs before making a step, as we explained in section 2.2.3.

The main problem with this approach is that we are working with a dynamic network, so each element of the batch may need a different amount of computation. We solved this by making the computation for all the elements of the batch, but keeping track of which elements had already stopped and only update the ones that continued. This was done by multiplying the outputs with a binary mask made of 0s in the position of the inputs that had stop and 1s for the inputs that needed more computation.

Even though using GPUs accelerated computation, this was still too slow. We thought that the main reason for this could be the fact that we were doing many

¹www.pytorch.org

small operations in CPU (native Python operations like the `while` loop for the ACT layer) interchanged with the GPU tensor operations. This created a lot of overhead, which slowed down our computation.

Our next approach was to return using CPUs, but implementing parallel computation using the *HogWild!* algorithm. This allows to train the network asynchronously using multiple CPU threads, which could speed up the training without adding the overhead of making operations in GPU.

5.1.1 *HogWild!*

To increase the speed of the training, we used the *HogWild!* algorithm (Niu et al., 2011). This algorithm allows the network to do multiple SGD steps in parallel asynchronously. The main idea of *HogWild!* is to have multiple processes, all of them doing SGD steps asynchronously with batches of different data, with shared memory storing the model parameters. This allows to speed up computation in multi-core machines, by having each of these processes in a different core.

We used 16 threads to train our networks. This was the faster approach, but it was still slow (it would take up to 20 days to 40 to train a single task). We suspected that the overhead for native Python operations and PyTorch operations was still present, and was slowing the network too much. We also found that we were not the only ones with this issue, as many people have reported similar problems when implementing custom RNN in PyTorch on the official PyTorch forums and other Machine Learning communities.^{2 3 4}

5.2 TensorFlow

Since asynchronous training was still too slow, we decided to change the framework to TensorFlow⁵, a deep learning framework developed by Google. TensorFlow is a static framework, but it is a much more mature than PyTorch (we used version 1.4 for this work). Since TensorFlow is static it does not have the problem of the overhead we had with PyTorch, because all the computation graph is sent at once to the GPU.

Even though getting our implementation to work was more difficult, the training speed was much faster than the one from our PyTorch implementation. We went back to using GPUs, with batch computation. TensorFlow was finally fast enough, and training processes that took 40 days in PyTorch, were reduced to only one day in TensorFlow.

²<https://discuss.pytorch.org/t/how-to-speed-up-for-loop-in-customized-rnn/1012>

³<https://discuss.pytorch.org/t/gpu-slower-than-cpu-on-a-simple-rnn-test-code/1306>

⁴https://www.reddit.com/r/MachineLearning/comments/5weeom/discussion_why_are_rnnss_so_slow_in_pytorch/

⁵www.tensorflow.org

Chapter 6

Experiments

This chapter includes the experimentation details obtained thanks to the implementation described in Chapter 5 of the ACT model presented in Chapter 4. We propose a new baseline that is compared with ACT in three tasks solvable with an RNN.

6.1 A new baseline: Repeated inputs

One of the limitations of the original ACT paper (Graves, 2016) is its lack of proper baselines. In that work, ACT is only compared to a basic RNN or LSTM. We consider that these baselines should be improved because they read each input sample only once, while ACT can read each input sample more times. A more appropriate baseline would be an RNN that reads the same number of input samples as ACT. To do this, we propose a new baseline configuration where each input sample is processed a fixed amount of iterations. The number of times that each input is repeated is a new hyperparameter, which we refer as ρ .

We also added the same binary flag as in the original ACT (equation 4.3) paper to the the inputs to allow the network to differentiate sequences with more that one equal input and our artificially repeated input.

We took as output from our network the hidden state from the last repetition of each sample. An example of an input sequence and the modified sequence with repetitions is shown in 6.1.

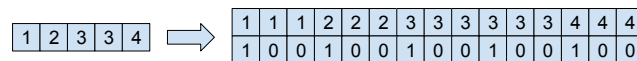


FIGURE 6.1: An input sequence (left) and corresponding modification with 3 repetitions (right). In the modified sequence, the first row consists of the inputs repeated 3 times each and the second row the binary flags that indicate to the network whether it is seeing a new input (with a 1) or the same repeated input (with a 0).

6.2 Tasks

Our implementation of ACT and the proposed baseline have been assessed and compared when used to solve the same tasks as in the original ACT paper (Graves, 2016). This section describes these tasks, while Section 6.4 provides the results of the experiments.

6.2.1 Parity

The first task is not actually a sequence modeling task because the network has to find the parity (or XOR) of a vector presented all at once, in a single timestep. This task could be solved with a simple feedforward network, but it is also addressable with a recurrent architecture like ACT or our baseline of repeated inputs.

The input vectors has 64 elements, of which a random number from 1 to 64 is set to 1 or -1 and the rest are set to 0. The corresponding target is 1 if there is an odd number of ones, and 0 if there is an even number of ones. Each training sequence consists of a single input vector and a single target, which is a 1 or a 0.

The implemented network is single-layer RNN with 128 tanh units, and a single sigmoidal output unit. The loss function is binary cross-entropy and the batch size is 128. The maximum ponder was set to 100 for this task. An example input and target are shown in Figure 6.2.

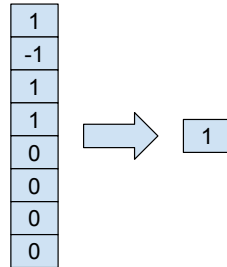
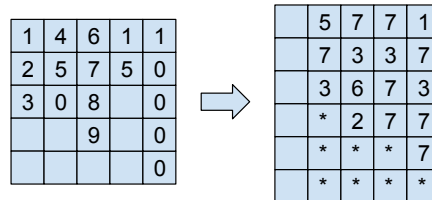


FIGURE 6.2: An example of the input sequence and target sequence for the parity task. Here, the first 4 elements of the vector are 1's and -1's and the rest are set to zero. The network has to be able to realize there are an odd number of ones and output a 1 accordingly.

6.2.2 Addition

The addition task aims at summing the values of a sequence of five numbers, each number represented by D digits, where D is drawn randomly from 1 to 5. Each number is coded by a concatenation of D one-hot encodings of its composing digits, being each digit value randomly chosen between 0 and 9. The length of these one-hot encodings is 10, one position for each possible digit. In case that D is smaller than five, the representation of the number is completed with zero vectors, so that the total length of the representation per number is 50. The required output is the cumulative sum of all inputs up to the current one, represented as a set of six simultaneous classifications (for the 6 possible digits in the result of the sum). There is no target for the first vector in the sequence, as no sums have yet been calculated. Because the previous sum must be carried over by the network, this task again requires the internal state of the network to remain coherent. Each classification is modeled by a size 11 softmax, where the first 10 classes are the digits and the 11th is a special marker used to indicate that the number is complete. An example input and target are shown in Figure 6.3.

The network was a single-layer LSTM with 512 hidden cells. The loss function was the sum of the categorical cross-entropy of all 6 digits at each time-step (except the first one) and the batch size was 32. The maximum ponder M was set to 20 for this task, as (Graves, 2016) reports that some networks had very high ponder times early in training.



1	4	6	1	1
2	5	7	5	0
3	0	8		0
		9		0
				0

	5	7	7	1
	7	3	3	7
	3	6	7	3
	*	2	7	7
	*	*	*	7
	*	*	*	*

FIGURE 6.3: An example of the input sequence and target sequence for the addition task. Each column in the left represents an input, and each column on the right represents the expected output for that input. The special marker * indicates that the number has ended. The first input has not an output target. The second input has as output target the sum of the first and the second inputs ($123+450=573$). The third input has as output target the cumulative sum of the first three inputs ($123+450+6789=7377$) and so on for the fourth and fifth input.

6.3 Training Parameters

There exist some minor differences in the parameters used to optimize the model with respect to the ones reported in (Graves, 2016). Our experiments are run with Adam (Kingma and Ba, 2014) optimizer and learning rate of 10^{-3} . The Adam optimizer has also 3 extra parameters, which we fixed to $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\varepsilon = 10^{-8}$. Instead of using a learning rate of 10^{-4} as proposed in (Graves, 2016), we were able to achieve better results and a faster convergence by increasing it to 10^{-3} . Thanks to the increase in learning rate, we found that running the experiments for 2 millions steps was enough for the models to converge (instead of 16 millions in the original paper). For the experiments with ACT, the value of ϵ from 4.6 was fixed to 0.01.

6.4 Results

This section presents the quantitative results of the ACT model and the repetition-based baseline for the considered tasks. The presented plots were smoothed with a Savitzky Golay filter (Savitzky and Golay, 1964) with a window of size 71 and cubic polynomial. The images for the plots were generated with the Python library Matplotlib. We will measure the accuracy ($\text{accuracy} = \frac{\text{correct examples}}{\text{total examples}}$) and the average ponder cost, which is calculated by adding the residual plus the amount of computation steps for each input, (equation 4.13). This means that we can know the amount of computation steps that the network is doing by just looking at the integer part of the ponder e.g. a ponder of 3.7 means that 3 computation steps are being done. In the

provided tables, we considered a task as being solved when the accuracy reached 98% after being filtered (to avoid variance).

6.4.1 Parity

The accuracy and ponder cost of ACT for the parity task are represented in Figure 6.4, for different values of the time penalty τ . We can see that the value of τ directly influences the value of the ponder, and thus the amount of times the network looks at the input. We can see that with smaller values of τ , the ponder gets larger and the accuracy reaches 100% much faster.

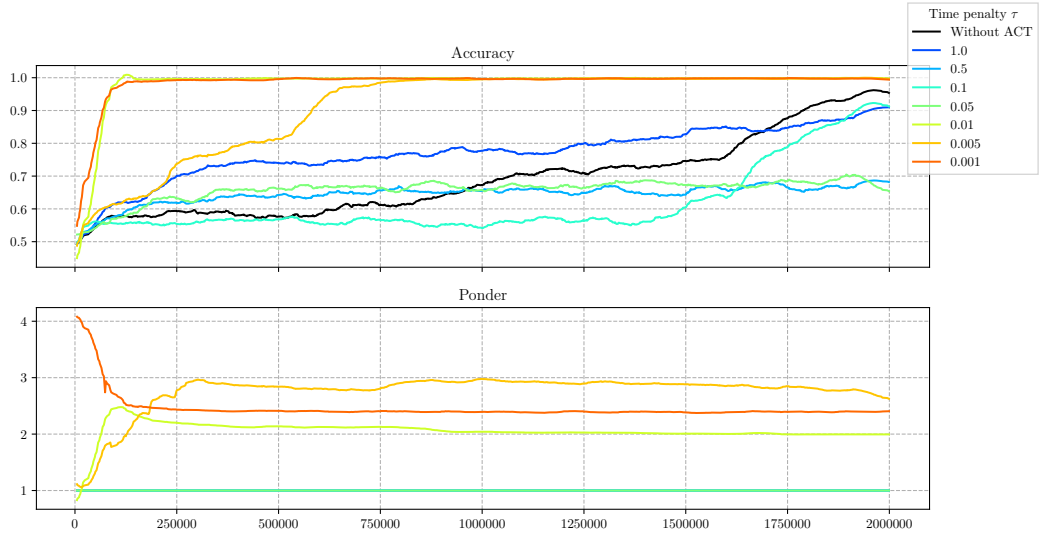


FIGURE 6.4: Accuracy and Ponder plot for the Parity task with ACT. Here we can see how if the ponder reaches 2 (so some inputs are being repeated) the accuracy increases much faster. We can also see that the value of the time penalty τ affects how much the ponder can grow, as expected. It should be noted that an accuracy of 0.5 comes just from random guessing, since we are trying to predict 0s or 1s.

Figure 6.5 shows the parity task with the proposed baseline solution based on repetitions. Surprisingly, the results of setting the amount of repetitions manually are very similar to those of ACT. We also see that, when doing too many repetitions, the accuracy decreases. We also observe that when doing too many repetitions, the network becomes unstable: even though it starts learning, the accuracy decreases in the late stages. We believe that this might be caused by exploding gradients, because the sequence gets too long. Table 6.2 shows a comparison between a simple RNN, ACT and our new baselines. We surprisingly see that our baselines with repeated inputs performs even better than ACT when choosing the right amount of repetitions.

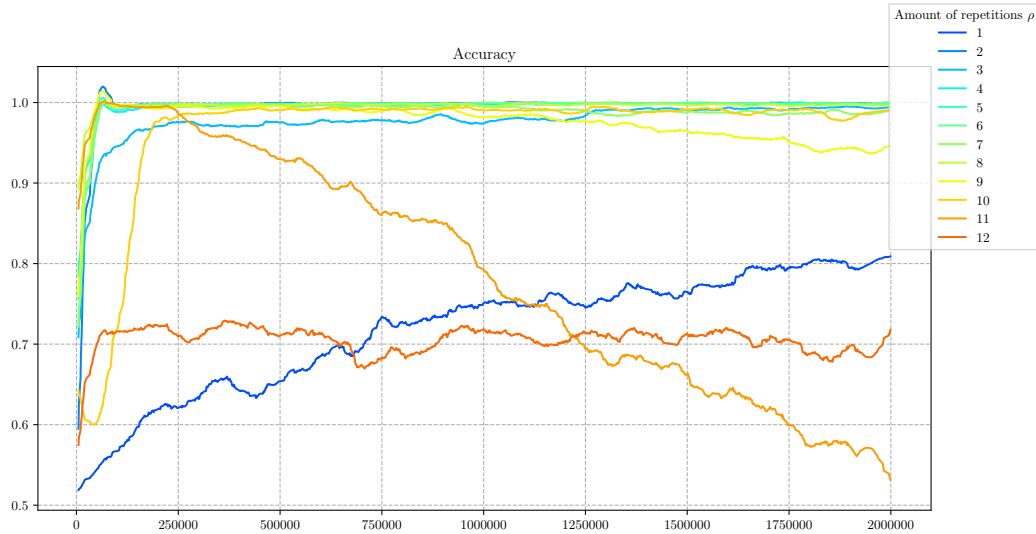


FIGURE 6.5: Accuracy plot for the Parity task with repeated inputs. We see that with more repetitions we achieve a faster growth in accuracy, but there is a point (at about 8 repetitions) where we start seeing instabilities. It should be noted that an accuracy of 0.5 comes just from random guessing, since we are trying to predict 0's and 1's.

Model	Task solved	Updates until solved	Mean repetitions
RNN	No	-	1
ACT-RNN, $\tau = 10^{-1}$	No	-	1.000
ACT-RNN, $\tau = 10^{-2}$	Yes	53000	1.805
ACT-RNN, $\tau = 5 \cdot 10^{-3}$	Yes	356000	2.027
ACT-RNN, $\tau = 10^{-3}$	Yes	55000	2.044
Repeat-RNN, $\rho = 2$	Yes	22000	2
Repeat-RNN, $\rho = 3$	Yes	49000	3
Repeat-RNN, $\rho = 5$	Yes	27000	5
Repeat-RNN, $\rho = 8$	Yes	26000	8

TABLE 6.1: Performance of a simple RNN, an RNN with ACT and an RNN with repetitions in the Parity task. Here we can see that our baseline with repetitions can outperform ACT in this task.

6.4.2 Addition

The results obtained with the Addition task are similar to those from the Parity task. Figure 6.6 confirms that the value of τ directly influences the ponder, but in this case we also see that the choice of this value is not trivial. For very large values, the network does not repeat inputs and no improvement is measured. For small values, the ponder cost and the amount of repetitions grow too much and we see the same phenomenon on instability that we saw in the Parity task with repeated inputs. We suspect again that this is due to dealing with sequences that are too long.

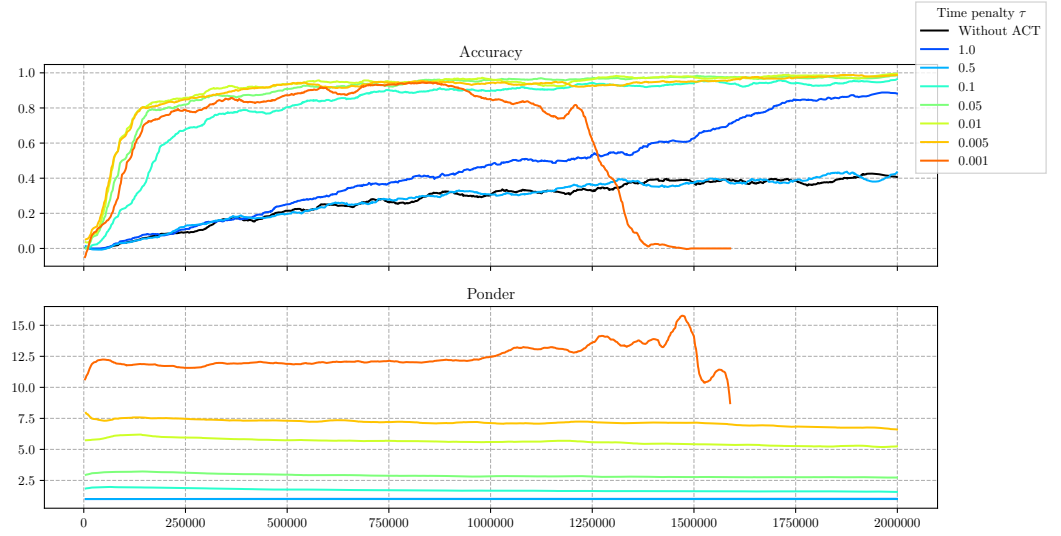


FIGURE 6.6: Accuracy and Ponder plot for the Addition task with ACT. We see that adding ACT to the network and choosing the right value for the time penalty τ allows it to learn to solve the task, going from about 40% to 100%. We also see instabilities for values of τ too small, which give values of ponder (and amount of repetitions) too large.

Figure 6.7 shows how the more repetitions of the baseline model, the faster the model converges. There is also a limit: with too many repetitions we see the same problem as before, the networks shows instabilities with excessively long sequences. We also see again that we can get a performance as good or even better than ACT by just repeating inputs. Table 6.2 shows a comparison between our baseline and ACT. We can see again a surprising better performance from repeating the inputs than from ACT.

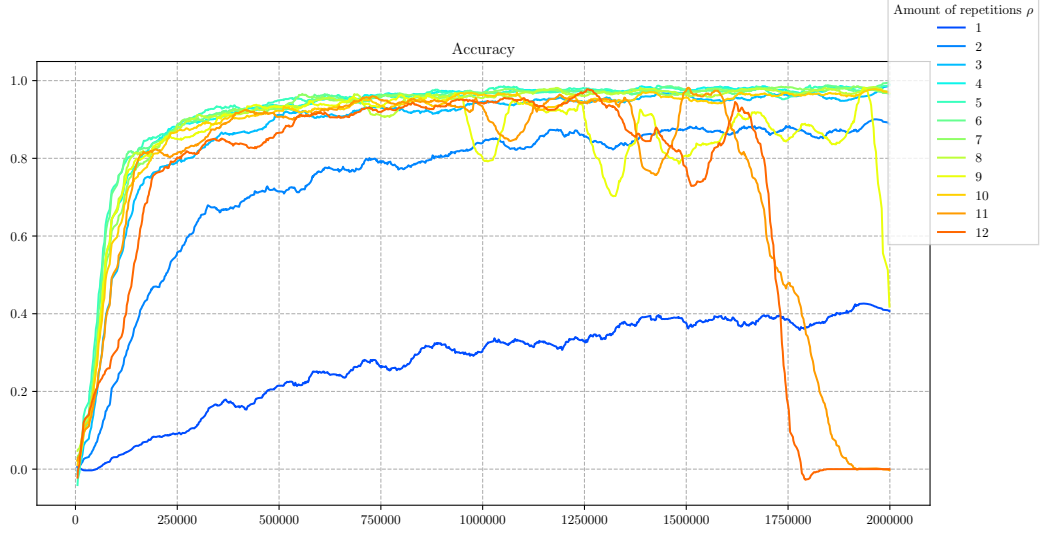


FIGURE 6.7: Accuracy plot for the Addition task with repeated inputs. We see that repeating the inputs in the sequence greatly improves the learning speed and the maximum accuracy achieved (going from about 40% to 100%). We also see instabilities when doing 9 repetitions or more.

Model	Task solved	Updates until solved	Mean repetitions
LSTM	No	-	1
ACT-LSTM, $\tau = 10^{-1}$	No	-	1.012
ACT-LSTM, $\tau = 10^{-2}$	Yes	899000	5.079
ACT-LSTM, $\tau = 5 \cdot 10^{-3}$	Yes	988000	6.736
ACT-LSTM, $\tau = 10^{-3}$	No	-	11.907
Repeat-LSTM, $\rho = 2$	No	-	2
Repeat-LSTM, $\rho = 3$	Yes	997000	3
Repeat-LSTM, $\rho = 5$	Yes	514000	5
Repeat-LSTM, $\rho = 8$	Yes	576000	8

TABLE 6.2: Performance of a simple LSTM, an LSTM with ACT and an LSTM with repetitions in the Addition task. Here we can see that our baselines with repetitions outperforms ACT when choosing the right amount of repetitions.

We were also curious to measure the distribution of the ponder throughout the sequence. Figure 6.8 shows a boxplot of the distribution of ponder for $\tau = 0.01$ after reaching accuracy of 98%. We see that the networks ponders much less the first input than the rest, which decrease at each timestep. This could be caused by the lack of a hidden state at the beginning of the sequence (in the first timestep a the we used a hidden state initialized to all zeros). Another possible explanation for this could be the lack of a target for the first input, which means that the network might not need to ponder this input as much.

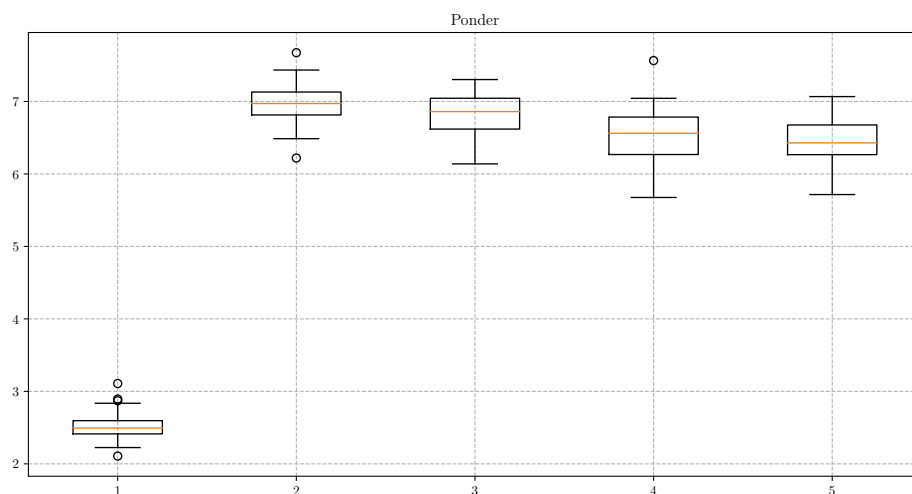


FIGURE 6.8: Boxplot of the distribution of the ponder throughout the sequence for the Addition task. We observe that the first element of the sequence has a ponder value much smaller than the rest, so the networks looks at it much less (2 times in this case) than the following ones.

Chapter 7

Conclusions and Future Work

In this work we have reproduced Adaptive Computation Time (ACT) for RNNs (Graves, 2016), a well known paper from a well regarded deep learning researcher at Google Deepmind. The model has been implemented from scratch with two of the most popular deep learning frameworks, and the source code has been released under a free software license for other researchers to use.

We have tested this architecture in two different tasks, tweaked some hyperparameters from the original paper which improved overall performance, and tested a new baseline that we considered fairer. With this new baseline, we found some very interesting and surprising results, showing that by just repeating inputs we can improve the performance of an RNN and speed up the training time. Repeating inputs requires fixing a new hyperparameter, the amount of repetitions, which is task dependent. This tuning can be compared to the hyperparameter that must be fixed in ACT (the time penalty τ), which is also task dependent and arguably much less intuitive. It should be noted that an equal amount of repetitions between ACT and our new baseline does not mean an equal amount of computation, since ACT requires many more extra calculations to determine the amount of repetitions and generate the output and ponder loss term.

As a result of this research work, we have identified different aspects from ACT that could be improved. Firstly, the ponder loss term is not differentiable and, even though this should not be a problem, it is not intuitive that this term minimizes the amount of times the networks ponders each input. For this reason, setting the required hyperparameter τ is not intuitive either. Another shortcoming is the fact that the output of an ACT layer is made of a weighted average of the output at each step. This is not common in neural networks, where the output usually comes just from the last time step. Moreover, the weights of this weighted average are the halting probability, which is also not natural.

Another possible future work is to think of variable computation from another perspective. Because of the surprisingly good performance of repeating the inputs of a sequence, we could try to repeat the inputs of a sequence, and then skip them adaptively, using the Skip RNN model (Campos et al., 2017) developed at UPC/BSC. We believe that this could be a way of achieving a good-performing adaptive RNN.

Bibliography

- Bengio, E. et al. (2015). “Conditional Computation in Neural Networks for faster models”. In: *ArXiv e-prints*. arXiv: [1511.06297 \[cs.LG\]](#).
- Bengio, Y. (2013). “Deep Learning of Representations: Looking Forward”. In: *ArXiv e-prints*. arXiv: [1305.0445 \[cs.LG\]](#).
- Bengio, Y., N. Léonard, and A. Courville (2013). “Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation”. In: *ArXiv e-prints*. arXiv: [1308.3432 \[cs.LG\]](#).
- Bengio, Y., P. Simard, and P. Frasconi (1994). “Learning long-term dependencies with gradient descent is difficult”. In: *IEEE Transactions on Neural Networks* 5.2, pp. 157–166. ISSN: 1045-9227. DOI: [10.1109/72.279181](#).
- Bolukbasi, Tolga et al. (2017). “Adaptive Neural Networks for Efficient Inference”. In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. International Convention Centre, Sydney, Australia: PMLR, pp. 527–536. URL: <http://proceedings.mlr.press/v70/bolukbasi17a.html>.
- Campos, V. et al. (2017). “Skip RNN: Learning to Skip State Updates in Recurrent Neural Networks”. In: *ArXiv e-prints*. arXiv: [1708.06834 \[cs.AI\]](#).
- Cauchy, Augustin (1847). “Méthode générale pour la résolution des systemes d’équations simultanées”. In:
- Chung, J., S. Ahn, and Y. Bengio (2016). “Hierarchical Multiscale Recurrent Neural Networks”. In: *ArXiv e-prints*. arXiv: [1609.01704 \[cs.LG\]](#).
- Cook, William J. (2012). *In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation*. Princeton University Press. ISBN: 9780691152707. URL: <http://www.jstor.org/stable/j.ctt7t8kc>.
- Denil, M. et al. (2013). “Predicting Parameters in Deep Learning”. In: *ArXiv e-prints*. arXiv: [1306.0543 \[cs.LG\]](#).
- Figurnov, M. et al. (2016). “Spatially Adaptive Computation Time for Residual Networks”. In: *ArXiv e-prints*. arXiv: [1612.02297 \[cs.CV\]](#).
- Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. <http://www.deeplearningbook.org>. MIT Press.
- Graves, A. (2016). “Adaptive Computation Time for Recurrent Neural Networks”. In: *ArXiv e-prints*. arXiv: [1603.08983](#).
- Gutmann, Michael and Aapo Hyvärinen (2010). “Noise-contrastive estimation: A new estimation principle for unnormalized statistical models”. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterton. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, pp. 297–304. URL: <http://proceedings.mlr.press/v9/gutmann10a.html>.
- He, K. et al. (2015). “Deep Residual Learning for Image Recognition”. In: *ArXiv e-prints*. arXiv: [1512.03385 \[cs.CV\]](#).
- Jernite, Y. et al. (2016). “Variable Computation in Recurrent Neural Networks”. In: *ArXiv e-prints*. arXiv: [1611.06188 \[stat.ML\]](#).

- Kingma, D. P. and J. Ba (2014). “Adam: A Method for Stochastic Optimization”. In: *ArXiv e-prints*. arXiv: [1412.6980 \[cs.LG\]](#).
- LeCun, Y. (1989). “Generalization and Network Design Strategies”. In: *Connectionism in Perspective*. Ed. by R. Pfeifer et al. an extended version was published as a technical report of the University of Toronto. Zurich, Switzerland: Elsevier.
- Li, Z. et al. (2017). “Dynamic Computational Time for Visual Attention”. In: *ArXiv e-prints*. arXiv: [1703.10332 \[cs.CV\]](#).
- Mitchell, Tom (1997). *Machine Learning*.
- Niu, F. et al. (2011). “HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent”. In: *ArXiv e-prints*. arXiv: [1106.5730 \[math.OC\]](#).
- Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams (1986). “Learning representations by back-propagating errors”. In: *Nature* 323, 533 EP –. URL: <http://dx.doi.org/10.1038/323533a0>.
- Savitzky, Abraham. and M. J. E. Golay (1964). “Smoothing and Differentiation of Data by Simplified Least Squares Procedures.” In: *Analytical Chemistry* 36.8, pp. 1627–1639. DOI: [10.1021/ac60214a047](#). eprint: <http://dx.doi.org/10.1021/ac60214a047>. URL: <http://dx.doi.org/10.1021/ac60214a047>.
- Schroff, F., D. Kalenichenko, and J. Philbin (2015). “FaceNet: A Unified Embedding for Face Recognition and Clustering”. In: *ArXiv e-prints*. arXiv: [1503.03832 \[cs.CV\]](#).
- Silver, David et al. (2017). “Mastering the game of Go without human knowledge”. In: *Nature* 550.7676, pp. 354–359. ISSN: 0028-0836. DOI: [10.1038/nature24270](#). URL: <http://dx.doi.org/10.1038/nature24270>.
- Tieleman, T. and G. Hinton (2012). *Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude*. COURSERA: Neural Networks for Machine Learning.
- Williams, Ronald J. (1992). “Simple statistical gradient-following algorithms for connectionist reinforcement learning”. In: *Machine Learning* 8.3, pp. 229–256. ISSN: 1573-0565. DOI: [10.1007/BF00992696](#). URL: <https://doi.org/10.1007/BF00992696>.
- Wolpert, David H. (1996). “The Lack of A Priori Distinctions Between Learning Algorithms”. In: *Neural Computation* 8.7, pp. 1341–1390. DOI: [10.1162/neco.1996.8.7.1341](#). eprint: <https://doi.org/10.1162/neco.1996.8.7.1341>. URL: <https://doi.org/10.1162/neco.1996.8.7.1341>.
- Xu, K. et al. (2015). “Show, Attend and Tell: Neural Image Caption Generation with Visual Attention”. In: *ArXiv e-prints*. arXiv: [1502.03044 \[cs.LG\]](#).
- Yu, A. W., H. Lee, and Q. V. Le (2017). “Learning to Skim Text”. In: *ArXiv e-prints*. arXiv: [1704.06877 \[cs.CL\]](#).
- Zhou, Y. T. et al. (1988). “Image restoration using a neural network”. In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 36.7, pp. 1141–1151. ISSN: 0096-3518. DOI: [10.1109/29.1641](#).