UNIVERSITY OF AMSTERDAM

MSc Artificial Intelligence
Master Thesis

Stochastic approach to Adaptive Computation Time in (Deep) Neural Networks

by

Jörg Sander

(10881530)

supervised by Patrick Putzky

4th May, 2017

1 Introduction

Anticipating how much mental effort is needed to solve a problem is a notoriously difficult task for us humans although experience and intuition can give us some guidance especially if the task has strong resemblance with previous encountered challenges. The best approach is often to iteratively adjust our approximation of the time and effort needed based on the new insights we gain and transfer the acquired knowledge to other tasks.

Machine learning algorithms generally *suffer* from the same lack of foresight and mostly miss the ability to dynamically adjust the number of computational steps to the complexity of the task at hand. More specifically the architecture of feedforward neural networks consists of an a priori fixed number of layer-to-layer transformation (network *depths*) and in recurrent neural networks (RNN) additionally the fixed sequence length determines the number of computational steps. Evidently it would be beneficiary if e.g. the number of times computations are passed through the same network layer would become a function of the input received so far.

In this work we describe a model that is able to learn the number of optimization steps needed before emitting a *sufficient* accurate result. In addition we investigate whether such a model can be trained in significantly less time than our baseline model.

The rest of this document is structured as follows. After giving an overview of the related work in section 2 we explicitly state the objectives of this research in section 3. The details of our approach are outlined in section 4 and section 5 specifies the conducted experiments. Finally we give a coarse-grained overview of the high-level planning in section 6.

2 Related work

The recent work of Alex Graves [6] marks the cognitive origin of the project described here. The study outlines an approach of *Adaptive Computational Time* (ACT) applied to Recurrent Neural Networks (RNNs) that learn how many computational steps to take between receiving an input and emitting an output. Inspired by the *push* and *pop* operations of neural stacks [7] the model adds parallel (to the recurrent states) layers of sigmoidal halting units, which are computed at each iteration. The cumulative probability of the halting units determines the moment when computation stops.

A very similar approach has been proposed in [8]. The work extends the basic Elman RNN unit [3] with the ability to decide at each time step how much computation it requires to perform based on the current hidden state and input.

The work of [5] extends ACT to spatially adaptive computational time (SACT) for Residual Networks. Their model incorporates attention into Residual Networks by learning a deterministic policy that stops computation in a spatial position as soon as the features become *good enough*.

A stochastic approach to ACT has been applied to scene understanding with recurrent networks [4]. This study develops a probabilistic inference framework that learns to choose the appropriate number of inference steps. The model attends to scene elements and processes them one at a time, deciding autonomously to how many objects in the scene it attends to.

A related line of work uses approaches from reinforcement learning to increase the computational efficiency of (deep) neural networks. In [11] the underlying idea is that machine learning models are often used at test-time subject to constraints and trade-offs not present at training-time. The authors propose a model that learns to change behavior at test time with reinforcement learning by adaptively constructing computational graphs from sub-modules on a per-input basis. The recent work of [10] is particularly close in spirit to the work we have presented here. The authors propose a dynamic computational time model to adaptively adjust computational time during inference for a recurrent visual attention (RAM) model. Using reinforcement learning the model learns the optimal attention and *stopping* policy which is modelled by separate parameters of the recurrent neural network.

Our work is also related to the recent surge of interest in using neural networks to learn optimization procedures, applying a range of innovative meta-learning techniques ([9], [1], [2]). In particular we take the RNN optimizer (model) described in [1] as our baseline model. The approach taken in this study focuses on learning how to utilize gradient observations over time, of the functions to be optimized (e.g. a loss function of an image classification network), in order to achieve fast learning of the underlying model.

Another recent study by [2] uses a meta-learning approach for training RNNs to perform black-box global optimization. Inspired by the Bayesian optimization framework the authors replace the expectation in the loss function of [1] with the *expected posterior improvement* which encourages an exploratory behavior into the meta learning model. The work is interesting for us because the model is compared to Bayesian optimization and focusses on the issues of computation speed, horizon length (i.e. number of iterative optimization steps during training), and exploration-exploitation trade-offs. Their research reveals that the RNN optimizer is faster and

tends to achieve better performance within the horizon for which it is trained. It however underperforms against Bayesian optimization for much longer horizons as it has not learned to explore for longer horizons.

3 Objectives of research

The research to be conducted pursues the following two objectives. First, we extend the work of [1] in such a way that the optimizer is endowed with the disposition to decide when to stop the iterative optimization procedure. Following the work of Graves [6] we denote this feature as *adaptive computation time* hereafter referred to as ACT.

Second, we intend to show that the enhanced optimizer learns to adjust the *size* of its optimization steps if it is forced to do so during training (induced through a prior distribution over the total number of computational steps to take). In other words, the ACT model should achieve a qualitative, satisfactory optimization result (*to be defined)* in less optimization steps (aka training time) than the baseline model presented in [1]. We will abbreviate the baseline model as *meta optimizer* hereafter.

4 Approach

We will implement the baseline model described in [1] as a recurrent neural network (RNN) m parameterized by ϕ. The update steps $\mathbf{g}_t$ of the optimizer will be the output of the RNN. Given a distribution over functions f the expected loss of the model for a discrete number of computational steps T will be equal to

$$\mathbb{E}[\mathrm{L}_{meta}(\phi)] = \mathbb{E}_{p(f)}\Big[\sum_{t=1}^{T} f(\boldsymbol{\theta}_t)\Big] \text{ where } \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \mathbf{g}_t \ , \tag{1}$$

where f is the function to be optimized (hereafter referred to as *optimizee*) which is parameterized by $\boldsymbol{\theta}$. The input of the RNN are the gradients of f denoted by $\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_t)$ (we will use the shorthand notation $\nabla_t f(\boldsymbol{\theta}_t)$ hereafter). The model can be formalized as follows

$$\begin{bmatrix} \mathbf{g}_t \\ \mathbf{h}_{t+1} \end{bmatrix} = m(\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_t), \mathbf{h}_t, \phi) \ , \tag{2}$$

where $\mathbf{h}$ denotes the hidden state calculated by the RNN which is passed as input to the next time step $t+1$ and can be interpreted as a compressed state of the current and previous time steps (note that $\mathbf{h}_0$ will be initialized with $\mathbf{0}$).

The so called ACT model is an extension of the meta model and additionally learns to approximate a discrete posterior distribution over time steps $q(\mathbf{t}|\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_t))$ that specifies the probability that computation stops at step t. The posterior distribution will be used to dynamically determine how many computational steps to take before the algorithm submits an output (the exact way still has to be developed).

Our approach is inspired by the work of [12] in which the authors develop an algorithm to translate the problem of solving a Markov Decision Problem (MDP) into a problem of likelihood maximization.

We are defining a probabilistic model in which $\mathbf{x}$ denotes the observed, continuous random variable (e.g. the gradients of the function f we want to optimize) and $\mathbf{t}$ (time steps) a discrete latent random variable. The joint distribution $p_\phi(\mathbf{x}, \mathbf{t})$ is parameterized by a set of parameters ϕ (e.g. modelled by an RNN). Our goal is to maximize the marginal likelihood function $p_\phi(\mathbf{x})$ but we instead maximize the complete-data likelihood function $p_\phi(\mathbf{x}, \mathbf{t})$ because this is significantly easier. We introduce the distribution $q_\phi(\mathbf{t}|\mathbf{x})$ which is an approximation of the true prior distribution $p_\phi(\mathbf{t}|\mathbf{x})$ which we assume belongs to the family of geometric distributions. Making use of the following decomposition

$$\log p_\phi(\mathbf{x}) = \mathcal{L}(\phi, \mathbf{x}) + D_{KL}\big(q_\phi(\mathbf{t}|\mathbf{x}) \,||\, p_\phi(\mathbf{t}|\mathbf{x})\big) \ , \tag{3}$$

where $\mathcal{L}$ denotes the *estimated lower bound* and D_{KL} the Kullback-Leibler divergence. In the following we will omit the parameterization notation by ϕ to prevent cluttering. Instead of minimizing the KL-divergence we can maximize the lower bound which decomposes into

$$\begin{aligned} \log p(\mathbf{x}) \geq \mathcal{L}(\phi, \mathbf{x}) &= \mathbb{E}_{q(\mathbf{t}|\mathbf{x})}\Big[\log p(\mathbf{x}, \mathbf{t}) - \log q(\mathbf{t}|\mathbf{x})\Big] = \mathbb{E}_{q(\mathbf{t}|\mathbf{x})}\Big[\log p(\mathbf{x}|\mathbf{t})\Big] - D_{KL}\big(q(\mathbf{t}|\mathbf{x}) \,||\, p(\mathbf{t})\big) \\ &\geq \sum_{t=0}^{\infty} q(t|\mathbf{x})\Big(\log p(\mathbf{x}|t) - \log \frac{q(t|\mathbf{x})}{p(t)}\Big) \ . \end{aligned} \tag{4}$$

We can always decompose $q(t|\mathbf{x})$ as

$$q(t|\mathbf{x}) = \sum_{T=0}^{\infty} q(t|\mathbf{x}, T)p(T) \ , \tag{5}$$

where again T denotes some finite time horizon (the details of the distributions $q(t|\mathbf{x}, T)$ and $p(T)$ are given in section 4.1). Replacing equation 5 in 4 results in

$$\log p(\mathbf{x}) \geq \sum_{t=0}^{\infty} \sum_{T=0}^{\infty} q(t|\mathbf{x}, T)p(T)\Big(\log p(\mathbf{x}|t, T) - \log \frac{q(t|\mathbf{x}, T)p(T)}{p(t|T)p(T)} \Big). \tag{6}$$

Pulling the summation over T to the front results in

$$\begin{aligned} \log p(\mathbf{x}) &\geq \sum_{T=0}^{\infty} p(T) \sum_{t=0}^{T} q(t|T)\Big(\log p(\mathbf{x}|t, T) - \log \frac{q(t|\mathbf{x}, T)}{p(t|T)} \Big) \\ &\geq \sum_{T=0}^{\infty} p(T) \ \Big[\mathbb{E}_{q(\mathbf{t}|\mathbf{x},T)} \left[\log p(\mathbf{x}|\mathbf{t}, T) \right] - D_{KL}\big(q(\mathbf{t}|\mathbf{x}, T) \ || \ p(\mathbf{t}|T)\big) \Big] \\ &\geq \mathbb{E}_{p(T)} \Big[\mathbb{E}_{q(\mathbf{t}|\mathbf{x},T)} \left[\log p(\mathbf{x}|\mathbf{t}, T) \right] - D_{KL}\big(q(\mathbf{t}|\mathbf{x}, T) \ || \ p(\mathbf{t}|T)\big) \Big]. \end{aligned} \tag{7}$$

Using this result we can formulate the lower bound on the log-likelihood of the ACT model as follows:

$$\mathcal{L}_{act}(\phi, \mathbf{x}) = \mathbb{E}\Big[\sum_{t=0}^{T} q(t|\mathbf{x}, T) \ \log p(\mathbf{x}|t, T) - D_{KL}\big(q(t|\mathbf{x}, T) \ || \ p(t|T)\big) \Big] \ . \tag{8}$$

In comparison to the meta model which outputs the update steps g_t our model will additionally emit a differential Δq_t logit that will be used in the update rule of the q_t logit

$$q_t = q_{t-1} + \Delta q_t \quad \text{where} \ \ q_0 = 0 \ . \tag{9}$$

During training the final probabilities $q(\mathbf{t}|\mathbf{x}, T)$ over the horizon T will be calculated by means of the Softmax function and the q_t logits computed at each time step t. At inference time the probabilities need to be computed at each time step t up to horizon $T = t$.

$$q(t|\mathbf{x}, T) = \frac{\exp\left(q_t\right)}{\sum_{t'=0}^{T} \exp(q_{t'})} \ . \tag{10}$$

The ACT model will be implemented as an RNN and can be formalized as follows

$$\begin{bmatrix} \mathbf{g}_t \\ \Delta q_t \\ \mathbf{h}_t \end{bmatrix} = m'(\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_t), \mathbf{h}_{t-1}, \phi, \lambda, \mathbf{w}_q) \ \text{ where } \ q_t = q_{t-1} + \Delta q_t \text{ and } \Delta q_t = \lambda \ \tanh(\mathbf{w}_q \mathbf{h}_t) \tag{11}$$

where $\Theta = \{\lambda, \phi, \mathbf{w}_q\}$ is the set of learnable parameters.

4.1 Training the ACT model

The important extension of our model in comparison to the meta-model is the approximation of the true prior probability distribution over time steps $p(\mathbf{t})$ that specifies the probability that computation stops at step t. Please note that this distribution is task-dependent. In order to be able to learn this distribution the decomposition in equation 5 of the approximated distribution $q(\mathbf{t}|\mathbf{x})$ is essential for training the model. We make the assumption that the true distribution $p(\mathbf{t})$ belongs to the family of geometric distributions. The approximated distributions during training $q(\mathbf{t}|\mathbf{x}, T)$ mainly differs from the true distribution in the fact that the former only has finite support (whereas the true distribution has infinite support).

During training we need to be able to sample the horizon T from $p(T)$ in equation 5 for the optimizee (e.g. in case the optimizee is a 2D quadratic function f we sample T for each function the model optimizes). In addition in order to compute $\mathcal{L}_{act}$ as specified in equation 8 we are required to compute the prior probabilities of the true distribution $p(\mathbf{t}|T)$.

Hence we need to find a decomposition of $p(t)$ into

$$p(\mathbf{t}) = \sum_{T=0}^{\infty} p(\mathbf{t}|T) \ p(T) \tag{12}$$

such that

$$p(t) = p(1-p)^t = pq^t \quad \text{where } p, q \in (0, 1] \tag{13}$$

A possible solution for $p(T)$ and $p(t|T)$ is given by (refer to appendix for proof)

$$\begin{aligned} p(T) &= \widetilde{q}^T(1-\widetilde{q}^{T+1})(1-\widetilde{q}^2) \\ p(t|T) &= \frac{\widetilde{p}\,\widetilde{q}^t}{1-\widetilde{q}^{T+1}} \end{aligned} \tag{14}$$

with $\widetilde{q} = \sqrt{q}$ and $\widetilde{p} = 1 - \sqrt{q}$.

5 Experiments

As previously mentioned, we will use the meta model described in [1] as our baseline model in order to evaluate the performance (or *behavior*) of the newly developed ACT model. We intend to conduct the following experiments: (1) training the optimizer models on synthetic, two dimensional quadratic functions; (2) train the optimizers to learn optimize a small neural network on the MNIST dataset; (3) evaluate the performance of the trainable optimizers when optimizing a convolutional neural network for image classification on the CIFAR-10 dataset. Due to the *dynamic nature* of the model we will be using *PyTorch*[1] to implement both models.

5.1 2D Quadratic functions

In this experiment we train both optimizers on a simple class of synthetic 2-dimensional quadratic functions. In particular we consider minimizing functions of the form

$$f(\boldsymbol{\theta}) = \frac{(\theta_1 - w_1)^2}{w_3^2} + \frac{(\theta_2 - w_2)^2}{w_3^2} + w_4$$

with $\{w_i\}_1^3 > 0$ and initialized according to $\mathcal{N}(0, 1)$.

The ACT optimizer to be trained uses the following loss function

$$\mathcal{L}_{act}(\phi) = \mathbb{E}\Big[\sum_{t=1}^{T} q(\mathbf{t}|\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_t), T)\; f(\boldsymbol{\theta}_t) - D_{KL}\big(q(\mathbf{t}|\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_t), T)||p(\mathbf{t}|T)\big)\Big]\,. \tag{15}$$

This formalization still contains the *non-probabilistic* term $f(\boldsymbol{\theta}_t)$, an issue that needs to be resolved.

5.2 2D Linear regression with two parameters

In this experiment the optimizer models will learn to find the parameters $\boldsymbol{\theta}$ of two dimensional linear regression models in the form

$$f(\mathbf{x}, \boldsymbol{\theta}^*) = \theta_1^* x_1^2 + \theta_2^* x_2 + \epsilon\,, \tag{16}$$

where $\mathbf{x}, \boldsymbol{\theta} \in \mathbb{R}^2$ and $\boldsymbol{\theta}^*$ denotes the true parameters to be determined by the optimizer models. For each regression model to be optimized we sample the parameters $\boldsymbol{\theta}^*$ from a Gaussian distribution $\mathcal{N}(0, \alpha^{-1})$ and add some fixed noise ϵ to each function value where $\epsilon \sim \mathcal{N}(0, \beta^{-1})$.

The meta model will minimize the expected negative log-likelihood for a discrete number of computational steps T given a distribution over functions $p(f)$

$$\begin{aligned} \mathbb{E}[L_{meta}(\phi, \mathbf{x})] &= -\,\mathbb{E}_{p(f)}\Big[\sum_{t=1}^{T} \log p(f(\mathbf{x}, \boldsymbol{\theta}_t)|\mathbf{x}, \boldsymbol{\theta}_t, \beta^{-1})\Big] \\ &= -\,\mathbb{E}_{p(f)}\Big[\sum_{t=1}^{T} -\frac{\beta}{2}\sum_{n=1}^{N}\Big\{f(\mathbf{x}^{(n)}, \boldsymbol{\theta}_t) - f(\mathbf{x}^{(n)}, \boldsymbol{\theta}^*)\Big\}^2\Big]\,, \end{aligned} \tag{17}$$

where we omitted all terms from the log-likelihood that do not depend on $\mathbf{x}$ and $\boldsymbol{\theta}_t$ and where N denotes the number of data points (we used $N = 100$) that will be sampled for each function to be optimized from $f(\mathbf{x}, \boldsymbol{\theta}^*)$ in order to be able to determine the loss.

[1] `http://pytorch.org/`

As previously mentioned the ACT model will optimize the expected lower bound on the log-likelihood $\log p(\mathbf{x})$. Equation 8 can be easily adjusted by replacing the log-likelihood term $\log p(\mathbf{x}|t,T)$ with the log-likelihood specified above, which results in

$$\mathcal{L}_{act}(\phi,\mathbf{x}) = \mathbb{E}\Big[\sum_{t=1}^{T} q(t|\mathbf{x},T)\ \Big(-\frac{\beta}{2}\sum_{n=1}^{N}\Big\{f(\mathbf{x}^{(n)},\boldsymbol{\theta}_t) - f(\mathbf{x}^{(n)},\boldsymbol{\theta}^*)\Big\}^2 + \frac{N}{2}\log\beta - \frac{N}{2}\log(2\pi)\Big) - D_{KL}\big(q(t|\mathbf{x},T)\ ||\ p(t|T)\big)\Big]\ . \tag{18}$$

6 Planning

Please note that *iterative loops* will be necessary for most of the activities pointed out below.

1. February was used for literature study;
2. March was used to start a proof of concept;
3. April, evaluate proof of concept, write introduction, related work & approach section for final paper;
4. May, run experiments for generalized linear regression functions, record results, prepare MNST experiment;
5. June, 04–06–2017 to 18–06–2017 vacation, run MNIST experiments, record results;
6. July, 19–07–2017 to 23–07–2017 vacation, prepare CIFAR-10 experiments, write on final paper;
7. August, 1 week vacation, run CIFAR-10 experiments, record results, write on final paper;
8. September, write on final paper, wrap up last results, may be re-do short experiment;
9. October, intend to graduate.

References

[1] Andrychowicz, M., Denil, M., Gomez, S., Hoffman, M. W., Pfau, D., Schaul, T., Shillingford, B., and de Freitas, N. Learning to learn by gradient descent by gradient descent. *arXiv:1606.04474 [cs]* (June 2016). arXiv: 1606.04474.

[2] Chen, Y., Hoffman, M. W., Colmenarejo, S. G., Denil, M., Lillicrap, T. P., and de Freitas, N. Learning to Learn for Global Optimization of Black Box Functions. *arXiv:1611.03824 [cs, stat]* (Nov. 2016). arXiv: 1611.03824.

[3] Elman, J. L. Finding structure in time. *Cognitive science 14*, 2 (1990), 179–211.

[4] Eslami, S. M. A., Heess, N., Weber, T., Tassa, Y., Szepesvari, D., Kavukcuoglu, K., and Hinton, G. E. Attend, Infer, Repeat: Fast Scene Understanding with Generative Models. *arXiv:1603.08575 [cs]* (Mar. 2016). arXiv: 1603.08575.

[5] Figurnov, M., Collins, M. D., Zhu, Y., Zhang, L., Huang, J., Vetrov, D., and Salakhutdinov, R. Spatially adaptive computation time for residual networks. *arXiv preprint arXiv:1612.02297* (2016).

[6] Graves, A. Adaptive Computation Time for Recurrent Neural Networks. *arXiv:1603.08983 [cs]* (Mar. 2016). arXiv: 1603.08983.

[7] Grefenstette, E., Hermann, K. M., Suleyman, M., and Blunsom, P. Learning to transduce with unbounded memory. *CoRR abs/1506.02516* (2015).

[8] Jernite, Y., Grave, E., Joulin, A., and Mikolov, T. Variable Computation in Recurrent Neural Networks. *arXiv:1611.06188 [cs, stat]* (Nov. 2016). arXiv: 1611.06188.

[9] Li, K., and Malik, J. Learning to Optimize. *arXiv:1606.01885 [cs, math, stat]* (June 2016). arXiv: 1606.01885.

[10] Li, Z., Yang, Y., Liu, X., Wen, S., and Xu, W. Dynamic Computational Time for Visual Attention. *arXiv:1703.10332 [cs]* (Mar. 2017). arXiv: 1703.10332.

[11] Odena, A., Lawson, D., and Olah, C. Changing model behavior at test-time using reinforcement learning. *arXiv preprint arXiv:1702.07780* (2017).

[12] Toussaint, M., and Storkey, A. Probabilistic Inference for Solving Discrete and Continuous State Markov Decision Processes. In *Proceedings of the 23rd International Conference on Machine Learning* (New York, NY, USA, 2006), ICML '06, ACM, pp. 945–952.