Adaptive Context Modeling for Arithmetic Coding Using Perceptrons

Lucas S. Lopes, Student Member, IEEE, Philip A. Chou, Fellow, IEEE, and Ricardo L. de Queiroz, Fellow, IEEE

Abstract—Arithmetic coding is used in most media compression methods. Context modeling is usually done through frequency counting and look-up tables (LUTs). For long-memory signals, probability modeling with large context sizes is often infeasible. Recently, neural networks have been used to model probabilities of large contexts in order to drive arithmetic coders. These neural networks have been trained offline. We introduce an online method for training a perceptron-based context-adaptive arithmetic coder on-the-fly, called adaptive perceptron coding, which continuously learns the context probabilities and quickly converges to the signal statistics. We test adaptive perceptron coding over a binary image database, with results always exceeding the performance of LUT-based methods for large context sizes and of recurrent neural networks. We also compare the method to a version requiring offline training, which leads to equally satisfactory results.

Index Terms—Adaptive arithmetic coding, neural context modeling.

I. INTRODUCTION

A RITHMETIC coding (AC) [1], [2] is widely used in media compression standards as well as other commercial applications and research. In AC, good estimates of next-symbol probabilities are essential for good compression, and those probabilities often depend on previous symbols (the context). Every pattern of symbols in the context is mapped to a probability estimate, which feeds the AC for achieving compression. The context model is the rule that maps the context patterns to probability estimates. Context modeling is of vital importance for the performance of AC [3]. Binary AC is the most commonly used form, even for m-ary and infinite-alphabet symbols, which may be expressed as a sequence of binary choices. Binarization simplifies the problem of context modeling.

Adaptive context models are context models that adapt their probabilities after every symbol, based on the history of symbols encountered. Adaptivity allows the context models to tailor themselves to the underlying information sources, regardless of their probabilities. This gives adaptive context models a universality that is not available with pre-trained context models. Thus context-adaptive ACs are very desirable when the statistics of the information source may not be fully known in advance, as is common in practice.

Context modeling is often carried out by counting frequencies of occurrence of the output symbols, in a data set, in each context. These frequencies are equivalent to conditional probabilities. A look-up table (LUT) is commonly used to store the frequencies. For a context of, say, $N$ binary symbols, the LUT stores the frequencies of occurrence of the output symbols for each of $2^N$ possible contexts. A context model based on a LUT can easily be made adaptive by incrementing the appropriate frequency count after each output symbol is processed. The frequency counts constitute a maximum likelihood estimate (MLE) of the probabilities and a LUT would be all that is needed as the context model for driving AC. However, as $N$ increases, such tables may become impractical. Even for a moderate context size $N$, the context may be diluted, i.e., many context patterns may not be observed in the past data set, despite their probabilities being required to code future data.

Nevertheless, context-adaptive binary AC (CABAC) is the entropy coding method behind most image and video coders such as JBIG [4], JBIG-2 [5], JPEG extended [6], JPEG 2000 [7], H.264/AVC [8], H.265/HEVC [9], and H.266/VVC [10]. Recent video coders such as AVC, HEVC and VVC all use the acronym CABAC for their own flavors of context modeling, although their details differ. All chop $m$-ary symbols such as motion vectors and quantized coefficients into bits and use frequency counting in LUTs to model contexts for adaptive binary AC. All use a collection of context models to code the binary symbols in different states. The state machines vary between CABAC implementations. The states can be considered a form of merging LUT entries to reduce memory and to avoid context dilution.

In this letter, we make use of the universal approximation capabilities of multi-layer perceptrons (MLPs) to replace LUTs for the case of large context sizes, when LUTs may become impractically large or too diluted. Other neural network (NN) architectures besides MLPs could be used for this purpose as well. Indeed, other NN architectures have been used for context modeling to drive AC in various settings (e.g., [11], [12], [13], [14], [15], [16], [17], [18], [19]). However, most of these networks have been pre-trained, that is, trained offline on representative datasets [11], [12], [13], [14], [15], [16], [17], [18]. In [18], it was hinted that the off-line methods they proposed could be...
adapted by blocking the data and continuously retraining the model. In [19], a hybrid on-off-line method, DZIP, was proposed specifically for sequential data such as text. Outside of the data compression literature, in the context of language modeling, [20] and [21] proposed methods to continuously adapt recurrent neural network (RNN) pre-trained weights during evaluation.

We propose an online, or adaptive, method like the methods in the aforementioned LUT and CABAC implementations, which requires no previous training. Unlike the previous works, our method is sample-adaptive, efficient and converges rapidly. Thus it has the potential for being a drop-in replacement for adaptive LUTs in CABAC engines. We call this method adaptive perceptron coding (APC). While we could have used other NN architectures besides perceptrons, we find that MLPs are flexible, not application-specific, and work well with online training.\(^1\)

II. PERCEPTRON-BASED CONTEXT MODELING

Assume we want to encode a binary signal \(y_k\). Let such binary signal depend stochastically on a tuple of binary or non-binary symbols \(X_k = (X_{k1}, \ldots, X_{kN})\) according to the conditional probability distribution \(P(y_k | X_k)\). The idea is to encode \(y_k\) with an ideal arithmetic coder driven by probability \(q_k\) if \(y_k = 1\) and \((1-q_k)\) if \(y_k = 0\), where \(q_k\) is an estimate close to \(P(y_k = 1 | X_k)\). We know that the expected number of bits needed to encode \(y_k\) achieves its minimum, the conditional entropy \(H(P|Q) = H(P) + D(P||Q)\), which is the conditional entropy (here denoted \(H(P)\)) plus the Kullback-Leibler divergence (KLD) \(D(P||Q)\) between the true conditional distribution \(P\) and the model \(Q\) [22]. Thus we would like a model \(Q\) that minimizes the cross-entropy \(H(P||Q)\) or equivalently minimizes the KLD \(D(P||Q)\).

Assume that \(q_k = Q_{01}(y_k = 1 | X_k)\), where \(Q_\theta\) is a conditional distribution in a parametric family of conditional distributions indexed by parameter vector \(\theta\). For example, assume in particular that

\[
q_k = \sigma(s_k), \quad \sigma(x) = \frac{1}{1 + e^{-x}},
\]

and

\[
s_k = \theta_0 + \sum_{i=1}^N \theta_i X_{ki},
\]

where \(\theta = (\theta_0, \ldots, \theta_N)\). With our ideal arithmetic coder, encoding \(y_k\) using \(q_k\) yields \(-\log_2 (q_k)\) bits if \(y_k = 1\) and \(-\log_2 (1-q_k)\) bits if \(y_k = 0\). The total number of bits spent to encode a sequence \(\{y_k\}\) is thus

\[
R = - \sum_k (y_k \log_2 (q_k) + (1-y_k) \log_2 (1-q_k)).
\]

We, then, want to find the parameter vector \(\theta\) that minimizes \(R\). One can show that

\[
\frac{\partial R}{\partial \theta_n} = - \frac{1}{\ln 2} \sum_k X_{kn} (y_k - \sigma(s_k)),
\]

which can be used to construct gradients for optimization as:

\[

\nabla \theta = \left[ \frac{\partial R}{\partial \theta_0}, \frac{\partial R}{\partial \theta_1}, \ldots, \frac{\partial R}{\partial \theta_N} \right] = - \frac{1}{\ln 2} \begin{bmatrix}
\sum_k (y_k - \sigma(s_k)) \\
\sum_k X_{k1} (y_k - \sigma(s_k)) \\
\vdots \\
\sum_k X_{kN} (y_k - \sigma(s_k))
\end{bmatrix}.
\]

At this point, the reader familiar with neural networks may identify this process as one of training a network to classify a pattern vector \(X_k\) as Class 0 or Class 1, where the ground truth class label is \(y_k\) [23], [24]. (Here, specifically, a single-layer perceptron, sigmoid activation function, and a cross-entropy loss function are used.) Thus, minimizing the length of a code for a binary sequence \(\{y_k\}\) given a context sequence \(\{X_k\}\) is equivalent to minimizing the cross-entropy loss of a binary classification problem whose objective is to classify each context. (This equivalence holds as well in the \(m\)-ary case.) In the classification problem, the use of the cross-entropy is heuristic: the cross-entropy loss is a differentiable proxy for the classification error. In the coding problem, the cross-entropy is the natural measure of bit rate.

III. ADAPTIVE PERCEPTRON CODING

Inspired by the least mean squares (LMS) algorithm and adaptive filtering [25], we propose to continuously update the neural network on-the-fly by using previously coded samples. (A related idea was proposed in [21].) Let \(f\) denote the function being approximated by the (single or multi-layer) perceptron network and let \(f_k\) be the approximation of such function at instant \(k\). The perceptron network estimated probability at instant \(k\) is

\[
q_k = \sigma(f_k(X_k)).
\]

At each step, we seek to minimize the code length to encode \(y_k\):

\[
R_k = -y_k \log_2 (q_k) - (1-y_k) \log_2 (1-q_k).
\]

A given parameter \(\alpha\) of the network at instant \(k\), \(\alpha(k)\), is updated with the gradient descent according to

\[
\alpha(k+1) = \alpha(k) - \lambda \frac{\partial R_k}{\partial \alpha(k)},
\]

where \(\lambda\) is a constant, the adaptation step in adaptive filters, or the learning rate in neural networks. This is equivalent to training the neural network with stochastic gradient descent (SGD) with a batch size of 1, learning rate \(\lambda\), and a cross-entropy loss function.

We use a slightly modified version of Xavier initialization [26]. Let \(h_l\) denote the number of nodes in the \(l\)'th hidden layer of a MLP with \(L\) hidden layers. By extension, let \(h_0 = N\) be the number of inputs and \(h_{L+1} = 1\) be the number of outputs.
Based on the fact that the more diverse the initial values are in a layer the better [24], instead of initializing the weights or biases of the \( t \)th layer by sampling a uniform distribution in the interval \((-1/\sqrt{h_{t-1}}, 1/\sqrt{h_{t-1}})\), we take their values from a random permutation of equally spaced values in this interval.

We need to set the same initial weights for both encoder and decoder. For that, they must share the random number generation seeds.

### IV. Experimental Results

We are proposing a CABAC and, in order to test it, we need suitable binary test data. We also need to define an MLP configuration and performance references for comparison.

For binary test data, it is desirable to use a long-memory binary signal with a well-organized context support, such as binary images, instead of breaking unstructured \( m \)-ary data into binary channels. Hence, we created a dataset for our tests made of binary scanned images, from two volumes of the *IEEE Signal Processing Letters*. The images in the 27th volume were reserved for offline training and validation while the 28th volume was used for coding tests and online training. Pages were digitized at 93 dpi to 768 \( \times \) 1024 pixels, being converted to grayscale and then to binary using a 40% threshold.

With \( N \) as the context size, we created contexts using the closest \( N \) pixels within a 2D causal neighborhood. For the network we used a 2-hidden-layer MLP with 64\( N \) units in the first hidden layer and 32\( N \) units in the second hidden layer, a rectified linear unit (ReLU) activation in the hidden neurons, and sigmoid activation function in the last (output) layer. Such a network has 2112 parameters in the network.

In terms of CABAC alternatives for comparisons, H.26X CABAC flavors are not references for many reasons. They were meant for \( m \)-ary data and different data types would yield different performances. The states in H.26X CABAC form bins which are AC coded with adaptive frequency counting, effectively forming a subset of the full LUT method wherein contexts are merged. Furthermore, H.26X CABAC variants are too specific and not readily extensible to other binary coders. On the other hand, if the training set would be representative of the test set, the LUT method, for smaller \( N \), would yield MLE of the probability and would be extremely competitive.

Our expectation of outperforming LUT-based method relies on large-enough \( N \) wherein LUT contexts may become diluted. We tested the LUT whenever \( N < 27 \).

There is also JBIG [4], [27] which is a specialized binary image coder. JBIG also uses CABAC, but with a variable-geometry context neighborhood and with a floating context pixel that can be far-located to capture halftone periodicity. With more than 10 pixel contexts, thousands of concurrent QM coders are run in JBIG along with multiresolution prediction. JBIG is very efficient and our expectation of outperforming JBIG solely relies in the brute force of largely increasing context sizes.

We refer to CABAC using MLP-based pre-trained context modeling as pre-trained perceptron coding (PPC). We, then, refer to our proposed method in Sec. III as adaptive perceptron coding (APC). Similarly, we refer to the pre-trained LUT-based method as PLUT and its online-frequency-counting version as ALUT. We have tested our APC against JBIG, ALUT, PLUT, and PPC.

Before we go into the APC tests, in order to establish references, we tested the non-adaptive approaches (batch or offline training). We used 10 pages for training and 5 pages for validation, while tests were conducted over 10 pages of a different volume. All pages were randomly selected without replacement. The MLPs were trained over 300 epochs, using a learning rate of \( \lambda = 10^{-5} \), stochastic gradient descent, and a batch size of 2048. After training, we selected the model with lowest average code length on the validation data for each context size \( N \). Table I(a) shows results for different context sizes, where \( N = 0 \) indicates static AC. The results bring us no surprises. The trained PPC works very well. On the training set, PPC follows PLUT well until the latter becomes impractical, after which PPC outperforms. On the test set, PPC outperforms PLUT for \( N > 10 \), since context dilution becomes a factor. It is hard to compete with JBIG for lower \( N \), but PPC performance surpasses that of JBIG for larger \( N \).

As for the APC tests, Fig. 1 illustrates its performance on a random page of the test set. It shows the cumulative average bit-rate for APC at different learning rates (\( \lambda \)) as the page of nearly 787 K samples is scanned and encoded. The first pixel has no priors from where to infer probabilities and is encoded with 1 b. The borders of a page in the test set are all white, so that all methods quickly learn to encode whites and the rate drops to nearly 0. As the page progresses, and the encoder encounters text and graphics, it slowly builds context models and stabilizes after 200 K pixels or so. Fig. 1 also shows results for ALUT, wherein both the APC and ALUT can use only past samples to estimate probabilities. Because of the small sample sizes during learning, we biased the ALUT with 1 frequency count for each context, in order to improve convergence. For \( N = 10 \) or so, contexts in LUT-based methods are not too diluted and the results in Fig. 1(a) indicate that ALUT has similar performance to the APC. However, for the case \( N = 26 \), in Fig. 1(b) we can see that the APC has the best performance. In both cases, \( \lambda = 0.01 \) leads to the best performance. Table II shows the final page coding rate for different values of \( N \) and \( \lambda \). Indeed, it seems that \( \lambda = 0.01 \)

| \( N \) | \( \lambda \) | APC | ALUT |
|------|------|-----|-----|
| 0 | 0.255 | 0.228 | 0.223 |
| 2 | 0.156 | 0.156 | 0.156 | 0.151 | 0.155 |
| 4 | 0.144 | 0.144 | 0.149 | 0.149 | 0.143 | 0.146 |
| 10 | 0.134 | 0.134 | 0.119 | 0.140 | 0.141 | 0.124 |
| 26 | 0.098 | 0.092 | 0.109 | 0.132 | 0.122 | 0.132 |
| 67 | 0.065 | 0.083 | 0.117 |
| 170 | 0.040 | 0.077 | 0.121 |
Fig. 1. Cumulative rate (bits/symbol) for coding a random page from the test set with APC and online ALUT, for (a) $N = 10$ and (b) $N = 26$.

TABLE II

| $N$ | 0.1 | 0.01 | 0.001 | 0.0001 |
|-----|-----|------|-------|--------|
| 4   | 0.149 | 0.141 | 0.143 | 0.157 |
| 10  | 0.158 | 0.157 | 0.141 | 0.155 |
| 26  | 0.176 | 0.122 | 0.130 | 0.144 |
| 67  | 0.208 | 0.116 | 0.125 | 0.138 |

leads to best results for a single test page. We, then, fixed that value of $\lambda$ and repeated the tests for 10 test pages and the average results are shown in Table I(b). In it, larger contexts were tested and compared to the ALUT whenever possible. If one compares the results in Table I(b) to those in Table I(a), one notes that, for the same test set, even without any previous training, APC outperforms ALUT, PLUT and JBIG. APC is outperformed only by the pre-trained PPC.

Another approach worth comparing uses an adaptive RNN (ARNN), that is, RNN trained on-the-fly [20], [21]. A related method is [19]. We tested both flavors of ARNN in [20], [21] and chose to present the results using the method in [21], which consistently outperformed the method suggested in [20]. In that approach, a RNN continuously accumulates context information from previous samples, such that its context size ($N$) is unbounded. Specifically, we trained a RNN on-the-fly with two GRU layers with 650 hidden units each, SGD, batch size of 64, and truncated backpropagation through time.

Finally, we tested encoding multiple pages for $N = 26$ and the results are shown in Fig. 2. The ARNN results are also presented as a reference. Note that the cumulative rate keeps decreasing as the APC keeps learning context patterns. Table III shows results for encoding the same 5-page article in the test set, comparing APC, PPC, ALUT, PLUT and JBIG, for different $N$.

We also included the results for ARNN for completeness. One can clearly see that APC is highly effective. For the computation, we used Pytorch, a GeForce GTX 1080 GPU, and an Intel Core i7-8700 CPU. Encoding one page with APC took 10–12 min for $N \leq 26$ (compared to 2 min LUT-building context-adaptation). Offline training took 2 – 3 days for $N = 170$ and less than a day for $N \leq 67$. We over-estimated the number of parameters of the MLPs to guarantee good approximations. It is likely that other network architectures with fewer parameters would lead to similar results in less time. Processing time could also be reduced by increasing the batch size.

V. CONCLUSION

We propose an adaptive method for non-linear context modeling for AC using MLP (APC), which does not require pre-training and is shown to be competitive with its pre-trained counterpart (PPC), and advantageous against alternatives (PLUT, ALUT, JBIG), in the coding of binary sources such as binary images. The MLP is trained online as the input data is being made available, i.e., on-the-fly. Tests are conducted using binary images, as a source of binary data, which show that the proposed APC method may outperform LUT-based methods (pre-trained or online) as well as the JBIG standard, which is a known binary-document-specific coder also based on CABAC. While LUT-based context modeling can model discrete probability distributions with zero approximation error, MLP context modeling offers a better bias-variance trade-off and is useful for long-memory low-complexity signals (like images). We believe that MLPs well-represent the class of NN architectures that perform well for adaptive context modeling. However, it is possible that other architectures, topologies, or hyperparameters may be more effective. Further, we need to look into issues such as model complexity as a function of data length and source memory. Complexity vs. efficiency trade-off for perceptron coding is an interesting topic that will be discussed elsewhere.
REFERENCES

[1] J. Rissanen and G. Langdon, “Arithmetic coding,” *IBM J. Res. Develop.*, vol. 23, no. 2, pp. 149–162, 1979.

[2] I. H. Witten, R. M. Neal, and J. G. Cleary, “Arithmetic coding for data compression,” *Commun. ACM*, vol. 30, no. 6, pp. 520–540, Jun. 1987.

[3] W. Pearlman and A. Said, *Digital Signal Compression: Principles and Practice*. Cambridge, Cambridgeshire, U.K.: Cambridge Univ. Press, 2011.

[4] Information Technology - Coded Representation of Picture and Audio Information - Progressive Bi-level Image Compression, ITU-T Recommendation T.82, International Standard ISO/IEC 11544, 1993.

[5] P. Howard, F. Kossentini, B. Martins, S. Forchhammer, and W. Rucklidge, “The emerging JBIG2 standard,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 8, no. 7, pp. 838–848, Nov. 1998.

[6] W. B. Pennebaker and J. L. Mitchell, *JPEG: Still Image Compression Standard*. New York, NY, USA: Van Nostrand Reinhold, 1993.

[7] M. Goyal, K. Tatwawadi, S. Chandak, and I. Ochoa, “DZip: Improved general-purpose loss less compression based on novel neural network modeling,” in *Proc. Data Compression Conf.*, 2021, pp. 153–162.

[8] T. Mikolov, M. Karafiát, L. Burget, J. Černocký, and S. Khudanpur, “Recurrent neural network based language model,” in *Proc. 35th Int. Conf. Mach. Learn.*, 2018, pp. 2766–2775.

[9] M. Haykin, *Adaptive Filter Theory*. Upper Saddle River, NJ, USA: Prentice Hall, 2002.

[10] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *Proc. 13th Int. Conf. Artif. Intell. Statist.*, Sardinia, Italy: Chia Laguna Resort, 2010, pp. 249–256.

[11] K. Sayood, *Introduction to Data Compression*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.