Neural Network Online-Pruning: Accelerating Weighted Sum Calculation by Early Stopping

Julian Lorenz
Chair of Data Processing
Technical University of Munich
Munich, Germany
julian.lorenz@tum.de

***Abstract*—In this paper I propose a new method to shorten the weighted sum computation at each neuron in a neural network without requiring retraining. I sort the weighted sum computation order by the magnitude of the weights. If the activation function shows converging behavior, I stop the weighted sum computation early after it has passed a predetermined stopping threshold. I show how to find the stopping thresholds by statistical analysis of the weighted sum computation in a network. I also provide an experimental analysis on how the online-pruning method performs in comparison to the normal feed-forward computation. Using my approach, the MAC operations in the tested network can be reduced by 14.1%. This results in a speed improvement of 5.1% while achieving an average R2 score of 99.09%.**

***Index Terms*—neural networks, efficient, weighted sum, activation function**

I. Introduction

Efficient neural network inference is an important research area next to the proper neural network design and training. Less computational effort means that manufactures can use more affordable computing devices, they can deploy larger, more capable networks or it saves on energy. Less energy consumption leads to longer battery life in smartphones or to lower operating costs.

There are many existing ways in which engineers can improve the efficiency of neural networks. A very comprehensive summary of ways to run neural networks more efficiently has been done by Sze et al in [1]. The paper also provides performance metrics which can be used to measure a neural networks efficiency. The authors list many approaches such as using specialized memory architectures or reduced precision. While there is specialized hardware for neural networks such as FPGAs and GPUs, typically they do not come at low cost. In this paper, I focus on CPU based neural network inference and algorithmic optimization that reduce the total number of operations performed. This allows the neural networks to run on generic, low-cost, low-energy microcontrollers.

The most well known method is called ”pruning” and is typically applied offline after training and before inference. During pruning researchers identify and remove weights or neurons with low impact. Often the networks are retrained to adjust to the pruned structure. There are countless examples for neural network pruning. The most prominent pruning method is ”Optimal Brain Damage” by Le Cun et al [2]. A more recent method is proposed by Kamma et al in [3].

Because pruning happens in preprocessing, it treats all input samples the same way. I however want to find a technique that prunes weights dynamically depending on the input vector. One approach for example would be to use layers of increasing complexity depending on the difficulty of the current sample as Wang et al do it in [4]. Such approaches however completely alter the structure of the network and can not be applied to already trained networks. I on the other side search for a technique that can be applied to any already trained network. For example, Albericio et al propose a technique that skip zero-valued neurons during the weighted sum calculation [5] dynamically. They also prune the network online by setting neuron outputs to 0 if their activation is below a certain threshold. Another very dynamic neural network accelerator is called ”SnaPEA” by Akhlaghi et al in [6]. SnaPEA concentrates on networks with rectified linear unit (ReLU) activation functions with only positive inputs. The ReLU function returns 0 for all negative inputs. They sort the weights in descending order. If the weighted sum intermediate result (WSIR) becomes negative, the weighted sum will remain negative and the computation can be stopped early. The authors also propose to earlier predict that the weighted sum will be negative by using speculative thresholds while the weighted sum is still positive. The method proposed in my paper shares the same core idea. My approach however is not limited to positive inputs and can be applied to any activation function which can be approximated by a constant if the input exceeds a convergence threshold.

I propose an online-pruning technique that aims to stop the weighted sum computation early by exploiting the activation function behavior. I have observed that I can sort the order of the previous neurons such that during the computation the weighted sum tends early to values where I can consider the activation function as constant (i.e. convergence zone). This allows us to stop the weighted sum computation early after only considering a few, most significant previous neurons.

In this paper I will explain the idea of online-pruning with the example of a generic activation function with an upper bound threshold. Then I will show how to find the right stopping thresholds for the method using statistical analysis of the neural networks. Later, I will evaluate the impact of online-pruning using a neural network for drone control as

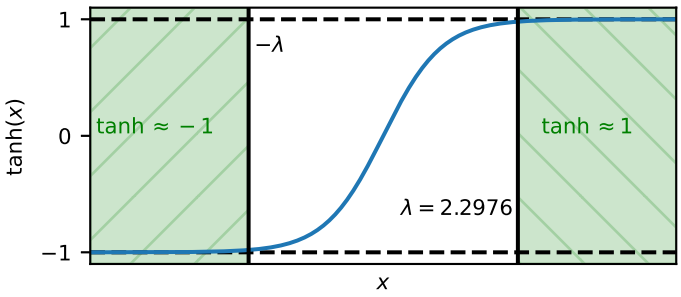

Fig. 1. tanh approximation

an example and compare the results with the normal network feed-forward operation.

II. Terminology

I use the following terminology to describe neural networks. I focus solely on feed-forward neural network structures with only fully connected layers. A network has L layers. Each layer l has got N_l neurons. The neurons are connected by the weights w_{ij}^l, where i is the index of the neuron in the l-th layer and j is the neuron's index in the $(l-1)$-th layer. Each neuron also has got a bias b_i^l, which initializes the weighted sum computation. The value x_i^l is the weighted sum at a neuron i in the l-th layer, which I define as followed:

$$x_i^l = b_i^l + \sum_{j=0}^{N_{l-1}} w_{ij}^l \sigma_{l-1}(x_j^{l-1}) \tag{1}$$

The weighted sum consists out of many so called Multiply-And-Accumulate (MAC) operations. One MAC is defined as the product of the previous neuron's activation and it's corresponding weight (Multiply) which is then added to a temporary sum (Accumulate). The output of a neuron i is the result of the activation function applied to the weighted sum: $\sigma_l(x_i^l)$. Neurons of one layer share the same activation function. Different layers may have different activation functions.
In the method that I propose I will make use of converging activation functions, meaning that they converge to a constant for $x \to \infty$ and/or for $x \to -\infty$. I will also write about "converging weighted sums" for simplicity. A weighted sum does not necessarily converge, but with the term I mean that the activation function that is applied to the weighted sum has exceeded a convergence threshold λ. An example of an activation function can be seen in Fig.1. If the weighted sum x has exceeded the convergence threshold λ, we approximate the activation function as a constant. I call the weighted sum "converged", if $|x| > \lambda$.

III. Method

At each neuron in a neural network we calculate a weighted sum of all previous neurons activation and individual weights. After that, an activation function is applied to the weighted sum. In this paper I propose a method to decrease the computation effort for the weighted sum by reducing the total amount of MAC operations. I take advantage of the activation functions which often show converging behavior, when I sort the order of summing weights in an appropriate way. Precisely, if x exceeds a predetermined threshold λ, I can assign to the activation function $\sigma(x)$ a constant value a. Note that in that case x is the result of the weighted sum at the neuron. A generic formulation can be found in (2).

$$\sigma'(x) = \begin{cases} a \text{ , if } x > \lambda \\ \sigma(x) \text{ , otherwise} \end{cases} \tag{2}$$

The activation function could also have a lower-bound threshold, such that $x < \lambda$ to assign a constant a. An example is the ReLU in (7). The activation function can also be bounded by an upper and a lower threshold. An example is the tangens hyperbolicus which can be approximated as in (8). During the computation of the weighted sum x, I observe that often the value x stays beyond the threshold after it has exceeded it once if the order of the weights has been sorted in a specific way. This hints us that I can stop the computation of the weighted sum early under the condition that the weighted sum intermediate result (WSIR) exceeds a stopping threshold α. Instead of the explicit, general equation of the weighted sum as in (1) I propose an iterative formulation in algorithm 1. The WSIR after the k-th step is denoted by $x_i^l(k)$. Note that $\alpha \neq \lambda$. Just because the weighted sum x_i is

Algorithm 1 Weighted sum with early stopping

$x_i^l(0) \leftarrow b_i^l$
$k \leftarrow 0$
while $k <= N_{l-1}$ and $x_i^l < \alpha$ **do**
  $j \leftarrow \text{order}(k)$
  $x_i^l(k+1) \leftarrow x_i^l(k) + w_{ij}^{l-1} \sigma_{l-1}(x_j^{l-1})$
  $k \leftarrow k+1$
end while
$x_i^l \leftarrow x_i^l(k)$

greater than λ during its computation, does not mean that it will stay beyond the threshold after considering all inputs. I need to compensate those cases, which first seem to converge but then end up non-converged by introducing the stopping threshold α. Later in this paper, I will refer to such cases as "false friends".
After approximating the weighted sum, I can then use the activation function's approximations (refer to (2), (7), (8)) to compute the output of the neuron.

To be able to run the algorithm 1, we need to perform two steps in preprocessing: Determine the order of the weighted sum computation order(k) and after that find the thresholds α. I describe the generic procedure for a network with an activation function with an upper bound threshold as in (2) in the following two subsections.

A. Sort the weights

I need to sort the order of the previous neurons during the weighted sum computation, such that two goals are full-filled:

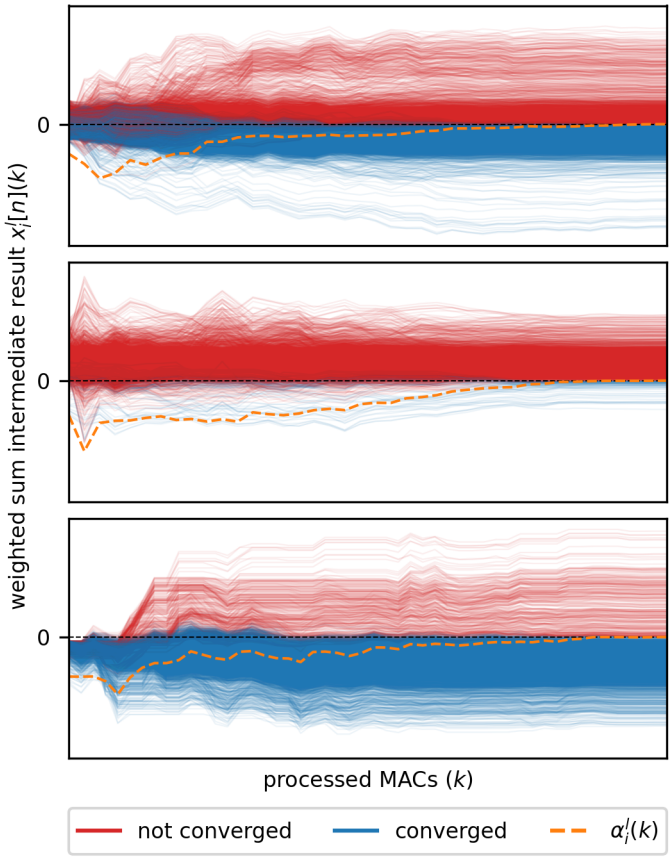

Fig. 2. Three examples of weighted sum computation processes at one neuron. The figure shows 5000 samples. Everything below the orange-dashed threshold line is considered as converged and can be stopped early.

1) The weighted sum converges quickly.
2) A converged weighted sum remains in the convergence interval.

It works very well to use the absolute value of the weights $|w_{ij}^l|$ to a neuron i in layer l as the sorting criteria. We then sort the computation order $j = \text{order}(k)$ in descending order. k indicates how many MACs have been performed already. j is the index of the previous neuron that shall be processed next. In Fig.2 we can see, that both of the goals are met. The WSIRs change a lot after the first MAC operations, but then remain almost static for the remaining part of the weighted sum computation.

B. Learning the thresholds

While early stopping of the weighted sum and sorting the weights is relatively straight-forward, the biggest difficulty in the online-pruning method is to find the right stopping thresholds α. The stopping threshold α must not be confused with the threshold λ. After λ, I consider the activation function as converged. We define a value for λ before trying to find the thresholds. A more detailed explanation can be found in subsection IV-A and IV-B. Variable α is the threshold after which I stop the computation of the weighted sum early. In Fig.2 both thresholds are shown. Because a ReLU activation function is used, $\lambda = 0$. The orange-dashed line shows the stopping threshold α.

In a first step, I run the neural network again many times using the determined processing order and record all WSIRs for each neuron and each run. I store the n-th weighted sum computation sample for the i-th neuron in the l-th layer as a function over all WSIRs k as $x_i^l[n](k)$. This is then the k-th WSIR during the n-th computation.

I then assign our weighted sum computation samples $x_i^l[n]$ to the following groups:

- CONVERGED NEURONS $\mathbb{C}$: Weighted sum computation samples that converged: $x_i^l[n](N_{l-1}) > \lambda$
- FALSE FRIENDS $\mathbb{F}$: Weighted sum computation samples that converged, but then left the convergence zone again: $x_i^l[n](o) > \lambda$ and $x_i^l[n](N_{l-1}) < \lambda$ and $o < N_{l-1}$
- OTHERS $\mathbb{O}$: Neurons that do not converge: $x_i^l[n](k) < \lambda$ for all k

I then determine the stopping threshold $\alpha_i^l(k)$ as a function for each neuron i in each layer $1 < l \leq L$ over all steps k. The first layer is excluded, as the input layer does not have an activation function. I choose $\alpha_i^l(k)$ such that the false-stop probability $P(x_i^l[n](k) > \alpha_i^l(k) \ \forall \ x_i^l[n](k) \in \mathbb{F})$ is minimized. In other words: The probability to stop early even though the current weighted sum computation is a false friend shall be minimized. The trivial solution would be to choose

$$\alpha_i^l(k) = \max\left(\lambda \ , \ \max_{n \in \mathbb{F}} x_i^l[n](k)\right) \ . \tag{3}$$

In (3) I assign to $\alpha_i^l(k)$ the largest value of all false friend WSIRs $x_i^l[n](k)$. If this value is lower than λ (not converged), I choose $\alpha_i^l(k) = \lambda$. This way I ensure that at least from our observations $\alpha_i^l(k)$ is always larger or equal to the largest false friend WSIR at the k-th step. While (3) is the safest way to lower the false-stop probability, it is not the best solution when it comes to MAC savings. I suggest to trade a higher false-stop probability for better MAC savings. For example we could fix the false-stop probability to a constant of choice:

$$P(x_i^l[n](k) > \alpha_i^l(k) \ \forall \ x_i^l[n](k) \in \mathbb{F}) = p = 0.001 \tag{4}$$

To find the value of $\alpha_i^l(k)$ for which (4) is true, I need to approximate the distributions of false friend WSIRs $f(x_i^l(k))$ using a histogram. The stopping threshold $\alpha_i^l(k)$ is then the $(1-p)$-th quantile of the empirical distribution $f(x_i^l(k))$ for the generic example with an upper bound threshold λ. For activation functions with a lower bound threshold such as the ReLU, I choose the p-th quantile of the distribution as shown in Fig.3. The p-th quantile of the empirical distribution $f(x_i^l(k))$ will be denoted as $z_p^{f(x_i^l(k))}$. The right value for p needs to be chosen experimentally with the goal of finding a tradeoff between accuracy and processing speed (i.e. MAC savings).

The stopping thresholds $\alpha_i^l(k)$ for each neuron at all steps k can be determined using algorithm 2. First, I initialize all thresholds for all neurons to ∞. Then, I assign the value λ to the thresholds of each neuron that has converged at least

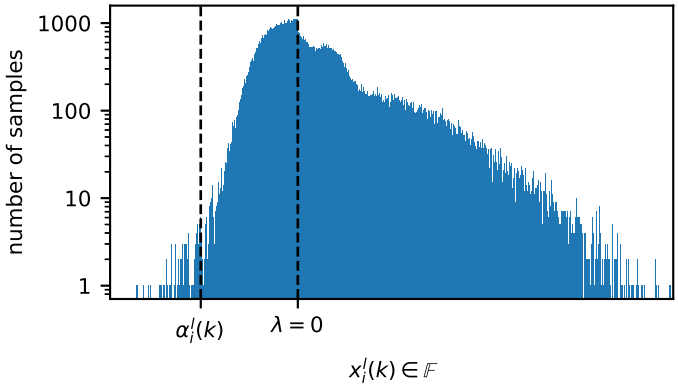

Fig. 3. Distribution of false friend WSIRs at neuron i at step k with a ReLU activation function. Stopping threshold $\alpha_i^l(k)$ is set to the 0.001-quantile of the histogram. Total number of false friend samples in histogram is 91548.

Algorithm 2 Determining the threshold (General formulation)

for all x_i^l **do**
  $\alpha_i^l(k) \leftarrow \infty \ \forall \ k$
end for
for all $x_i^l[n] \in \mathbb{C}$ **do**
  $\alpha_i^l(k) \leftarrow \lambda \ \forall \ k$
end for
$\alpha_i^l(k) \leftarrow \max\left[\alpha_i^l(k), z_{1-p}^{f(x_i^l(k))}\right] \ \forall \ k$

once. After the last step, $\alpha_i^l(k)$ is either ∞ if the weighted sum never converged, or either of λ or the $(1-p)$-th quantile of the empirical false friend WSIR distribution $z_{1-p}^{f(x_i^l(k))}$, depending on which one is larger.

C. Speed tradeoff

The modified weighted sum computation technique described in algorithm 1 will take more processing time on most computing architectures. This is due to the additional check whether the weighted sum has exceeded the threshold. On an amd64 CPU, one iteration of the loop in algorithm 1 requires about 16% more time than one iteration of the conventional weighted sum computation. This implies that I increased the processing time for weighted sums that do not converge, if I applied the online-pruning method to all neurons. The online-pruning method should only be applied to neurons where the average MAC savings outweigh the increased processing time. To do so, I first need to determine the MAC time ratio (MTR) between the normal MAC operation and the online-pruning MAC operation:

$$\text{MTR} = \frac{\text{time per iteration during normal WS}}{\text{time per iteration during pruning WS}} \tag{5}$$

On my hardware the MTR is 0.87. I then determine the average computed MAC operations for each neuron. Using the average computed MAC operations we can compute the MAC count ratio (MCR) for each neuron as followed:

$$\text{MCR} = \frac{\text{avg. processed MACs at neuron}}{\text{number neurons in previous layer}} \tag{6}$$

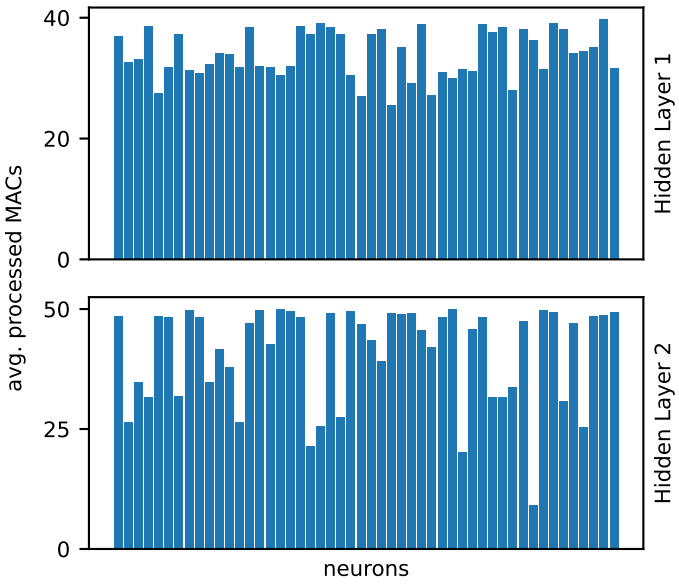

Fig. 4. Avg. number of processed MACs in the hidden layers for each neuron.

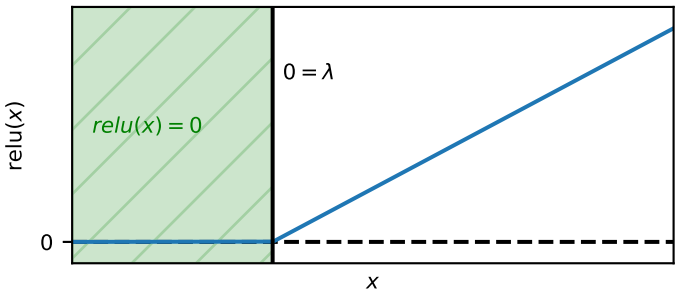

Fig. 5. ReLU activation function

I then only apply the online-pruning method to neurons where the MCR is less than the MTR to maximize the processing time savings.

Fig.4 shows the average number of processed MAC operations at each neuron in a trained online-pruning network. Considering the speed tradeoff and an MTR of 0.87, only 50 out of 100 neurons are eligible for online-pruning.

IV. Implementation

After having explained the general method of online-pruning, I now apply the concept to the activation functions rectified linear unit (ReLU) and tangens hyperbolicus (tanh). The concept can be extended to other activation functions, as long as they can be approximated similar to (2).

A. Rectified Linear Unit

The ReLU activation function does not require an approximation, but I can use the original definition:

$$\sigma_{\text{ReLU}}(x) = \begin{cases} 0 \ , \text{ if } x < 0 = \lambda \\ x \ , \text{ otherwise} \end{cases} \tag{7}$$

The ReLU function has got a lower threshold $\lambda = 0$ after which I always return a constant 0.

During preprocessing, I sort the order of previous neurons during weighted sum computation for all neurons as described in III-A. After that I perform statistical analysis as described in subsection III-B. The weighted sum computation samples $x_i^l[n]$ will then be grouped as followed:

- CONVERGED NEURONS $\mathbb{C}$: Weighted sum computation samples that converged: $x_i^l[n](N_{l-1}) < \lambda = 0$
- FALSE FRIENDS $\mathbb{F}$: Weighted sum computation samples that converged, but then left the convergence zone again: $x_i^l[n](o) < \lambda = 0$ and $x_i^l[n](N_{l-1}) > \lambda = 0$ with $o < N_{l-1}$
- OTHERS $\mathbb{O}$: Weighted sum computation samples that do not converge: $x_i^l[n](k) > \lambda$ for all k

The stopping threshold can then be determined using algorithm 3.

Algorithm 3 Determining the thresholds (ReLU)

for all x_i^l **do**
  $\alpha_i^l(k) \leftarrow -\infty \ \forall \ k$
end for
for all $x_i^l[n] \in \mathbb{C}$ **do**
  $\alpha_i^l(k) \leftarrow 0 \ \forall \ k$
end for
$\alpha_i^l(k) \leftarrow \min \left[\alpha_i^l(k), z_p^{f(x_i^l(k))}\right] \ \forall \ k$

B. Tangens Hyperbolicus

Approximating functions by constants at their extremes is not a new concept. For example, Kundu et al propose a fast approximation of the tangens hyperbolicus function by dividing the function into five segments in [7]. They use a linear segment for $|x|$ close to 0, two constant segments for $|x| \to \infty$, and a lookup table for the remaining parts. In my work, I will use an approximation with only the two constant segments as followed:

$$\tanh'(x) = \begin{cases} -1 \text{ , if } x < \lambda_1 \\ +1 \text{ , if } x > \lambda_2 \\ \tanh(x) \text{ , otherwise} \end{cases} \tag{8}$$

This activation function approximation has got two thresholds: a lower threshold λ_1 and an upper threshold λ_2. I suggest to use the same magnitude for both thresholds, due to the point-symmetry of the $\tanh$ function:

$$\lambda = |\lambda_1| = |\lambda_2| \tag{9}$$

When choosing a value for λ I need to consider two different effects. If I choose a large value, the error in our approximation remains low, because $\lim_{x\to\infty} |\tanh(x)| = 1$. On the other side, it becomes more unlikely that the weighted sum exceeds the threshold during computation. That effect is more likely when I choose a lower value for λ. Lowering the value increases the errors that I introduce to our network through the approximation. In my example, I choose to accept that a 0.98 will be considered a 1. Now, I need to find λ by using:

$$0.98 = \tanh \lambda = 1 - \frac{2}{e^{2\lambda} + 1} \tag{10}$$

Reformulation yields:

$$\lambda = 0.5 * \ln\left(\frac{2}{1 - \tanh \lambda} - 1\right) = 2.2976 \tag{11}$$

I now sort the order of previous neurons during weighted sum computation for all neurons as described in III-A. After that I perform statistical analysis as described in subsection III-B. Because I deal with two thresholds this time, I need to sort the weighted sum computation samples into the following groups:

- LOWER CONVERGED NEURONS $\mathbb{C}_1$: Weighted sum computation samples that converged to lower bound: $x_i^l[n](N_{l-1}) < \lambda_1$
- UPPER CONVERGED NEURONS $\mathbb{C}_2$: Weighted sum computation samples that converged to upper bound: $x_i^l[n](N_{l-1}) > \lambda_2$
- FALSE FRIENDS $\mathbb{F}$: Weighted sum computation samples that converged, but then left the convergence zone again: $|x_i^l[n](o)| > \lambda$ and $|x_i^l[n](N_{l-1})| < \lambda$ with $o < N_{l-1}$
- OTHERS $\mathbb{O}$: Weighted sum computation samples that do not converge: $|x_i^l[n](k)| < \lambda$ for all k

I also need to determine two threshold functions for early stopping: $\alpha_i^l(k)$ for the lower bound and $\beta_i^l(k)$ for the upper bound. Algorithm 4 shows how to determine them.

Algorithm 4 Determining the thresholds ($\tanh$)

for all x_i^l **do**
  $\alpha_i^l(k) \leftarrow -\infty \ \forall \ k$
  $\beta_i^l(k) \leftarrow +\infty \ \forall \ k$
end for
for all $x_i^l[n] \in \mathbb{C}_1$ **do**
  $\alpha_i^l(k) \leftarrow \lambda_1 \ \forall \ k$
end for
for all $x_i^l[n] \in \mathbb{C}_2$ **do**
  $\beta_i^l(k) \leftarrow \lambda_2 \ \forall \ k$
end for
$\alpha_i^l(k) \leftarrow \min \left[\alpha_i^l(k), z_{p/2}^{f(x_i^l(k))}\right] \ \forall \ k$
$\beta_i^l(k) \leftarrow \max \left[\beta_i^l(k), z_{1-p/2}^{f(x_i^l(k))}\right] \ \forall \ k$

V. EXPERIMENTS

I apply the online-pruning method proposed in this paper to a neural network that was trained to control a light-weight quadrocopter drone. The network has only fully connected layers. The input vector has the length of 40 neurons followed by two hidden layers with 50 neurons each. The output vector has 4 values corresponding to the PWM settings of the drones motors. The two hidden layers have a ReLU activation function. The output neurons have identity activation functions ($\sigma(x) = x$) which are not eligible for online-pruning. I trained the stopping thresholds using a training data set

TABLE I
ONLINE-PRUNING PERFORMANCE ANALYSIS

	general mode			selective mode		
$\alpha_i^l(k)$	FSR	MSR	S	FSR	MSR	S
0.01-th quantile	2.33	22.33	8.44	1.29	19.70	10.3
0.005-th quantile	1.36	19.88	5.19	0.67	17.41	8.0
0.001-th quantile	0.35	16.02	2.12	0.16	14.10	5.1
0.0001-th quantile	0.05	12.79	-1.93	0.02	10.55	2.0
$\min(f(x_i^l(k)))$	0.007	10.57	-3.89	0.002	7.89	1.4

TABLE II
ONLINE-PRUNING ERROR ANALYSIS

$\alpha_i^l(k)$	mean	99-percentile	max	R2[%]
0.01-th quantile	0.1138	2.11	6.26	90.93
0.005-th quantile	0.0671	1.48	5.62	95.10
0.001-th quantile	0.0185	0.51	4.47	99.09
0.0001-th quantile	0.0016	0.01	1.07	99.97
$\min(f(x_i^l(k)))$	0.0003	0.00	1.31	99.99

with $N_T = 500000$ samples with different settings for p. I then validated the online-pruning networks using a different validation data set with $N_V = 50000$ samples.

A. Performance analysis

For each different stopping threshold setting p I measured:

- False-Stop Ratio (FSR)[%] - The average percentage of false-stops in the network across all validation samples
- Mac-Savings Ratio (MSR)[%]- The average percentage of performed MAC operations w.r.t. to standard inference across all validation samples
- Speed-up (S)[%] - The average speed increase across all validation samples. $S = 100\% * (1 - t_p/t_s)$, with t_p being the execution time for the online-pruning inference and t_s being the execution time for the standard inference

I measured all parameters first by applying online-pruning to all neurons (general mode) and then by using online-pruning only for neurons which have a smaller MCR than MTR (selective mode, refer to III-C).

The results in table I show that the selective mode is faster for all networks, even though it saves less MACs in total. The table also shows that the selective mode has less false-stops in the network than in the general mode. We can also see the impact of parameter p. For larger values of the false-stop probability more MACs are cut off and the speed improvement is better in total.

B. Error analysis

I also measured the error between the n-th output vector of the standard feed-forward operation $v_s[n]$ and the n-th output vector of the online-pruning feed-forward operation $v_p[n]$, with $1 < n <= N_V$. I use the max-norm of the error vector:

$$e[n] = |v_s[n] - v_p[n]|_\infty \tag{12}$$

I compute the mean, the 99-th percentile and the maximum of the distribution of all e. I also provide the average R2 score of the data set across the 4 output dimensions. The results can be seen in table II. Due to the increased number of false-stops, the error increases for for larger values of p. All of the online-pruning networks were also tested in a flight-simulation. All networks flew well, except for the one where I used the 0.01-th quantile of the false friend WSIR distribution as the stopping threshold. The results show that the goals processing time and accuracy are contradicting each other and that a good compromise needs to be found.

VI. CONCLUSION

In this paper, I proposed a new method to decrease the total number of performed MAC operations in a neural networks feed-forward operation. I have demonstrated that we can reorder the MAC operations of one weighted sum computation such that the weighted sum shows converging behavior. I also showed how to choose the stopping thresholds such that I can account for so called false friend weighted sum computation samples. I achieved the best result by choosing the 0.001-th quantile of the false friend WSIR distribution for the stopping thresholds α. This configuration saved 14.1% of the MAC operations and about 5% computation time while maintaining a reasonably low error. The processing speed improvement is comparatively low due to the increased effort of comparing the weighted sum against a threshold. The method could perform better on hardware architectures that have a better MTR. The network has got a MSR of 16.0% in the general mode, which is the theoretical maximum speedup for hardware architectures with an MTR of 1. The cost of online-pruning method is an increased memory consumption, because I need to store the thresholds and the processing order for each neuron. We can expect the required memory to be tripled.

REFERENCES

[1] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[2] Y. LeCun, J. Denker, and S. Solla, "Optimal brain damage," in *Advances in Neural Information Processing Systems*, D. Touretzky, Ed., vol. 2. Morgan-Kaufmann, 1990.

[3] K. Kamma, Y. Isoda, S. Inoue, and T. Wada, "Neural behavior-based approach for neural network pruning," *IEICE Transactions on Information and Systems*, vol. E103.D, no. 5, pp. 1135–1143, 2020.

[4] Q. Wang, K. Wang, Q. Li, Z. Yang, G. Jin, and H. Wang, "Mbnn: A multi-branch neural network capable of utilizing industrial sample unbalance for fast inference," *IEEE Sensors Journal*, vol. 21, no. 2, pp. 1809–1819, 2021.

[5] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," *SIGARCH Comput. Archit. News*, vol. 44, no. 3, p. 1–13, jun 2016. [Online]. Available: https://doi-org.eaccess.ub.tum.de/10.1145/3007787.3001138

[6] V. Akhlaghi, A. Yazdanbakhsh, K. Samadi, R. Gupta, and H. Esmaeilzadeh, "Snapea: Predictive early activation for reducing computation in deep convolutional neural networks," 06 2018, pp. 662–673.

[7] A. Kundu, S. Srinivasan, E. C. Qin, D. D. Kalamkar, N. K. Mellempudi, D. Das, K. Banerjee, B. Kaul, and P. Dubey, "K-tanh: Hardware efficient activations for deep learning," *CoRR*, vol. abs/1909.07729, 2019. [Online]. Available: http://arxiv.org/abs/1909.07729