TENSORFLOW IMPLEMENTATION OF SPARSEMAX

Andreas Madsen (s123598), Frederik Wolgast Rørbech (s123956), Marco Dal Farra Kristensen (s152630)

Technical University Of Denmark

ABSTRACT

This report investigates if the "sparsemax" transformation suggested in Martins et al. [1] can be a suitable and practical replacement for the softmax transformation. This is done by looking at both its predictive abilities, but also how it can be implemented in practice. The implementation is done in Tensorflow [2]; In particular a parallel GPU implementation of sparsemax is constructed and its computational performance is compared to softmax. The sparsemax transformation is applied to classification problems and in an attention model for a RNN. It is shown that sparsmax has a similar classification performance as softmax for classification problems and greatly improves the attention model performance.

***Index Terms*—** neural networks, regression analysis, quadratic programming, cost function

1. INTRODUCTION

The softmax transformation is a well known component in classic statistics such as multinomial logistic regression. The transformation is also used in neural networks both in multi-class classification and more recently in attention mechanisms.

The softmax transformation is very popular, likely because of its simplicity. It is easy to evaluate, differentiate and has a simple convex loss function associated to it.

However, while softmax is very simple, its simplistic design also has drawbacks. It is impossible for its output to be sparse. This can be a big issue in cases where one would expect or want a sparse distribution.

A simple way of obtaining sparse probabilities, is to optimize a threshold value and then truncate all the softmax values below this threshold value to zero. This would be easy to implement but doesn't fit well with the cross entropy function that uses $\log(\mathrm{softmax}(\mathbf{z})_i)$ as it is not defined for $\log(0)$.

A recent alternative to softmax is the sparsemax transformation [1]. This has been shown to have many properties in common with the softmax transformation. It also has an associated convex loss function. This can make it an attractive alternative to multinomial logistic regression, especially when the target is multi-labelled.

Another suitable application of sparsemax is in attention mechanisms. When a standard Encoder-Decoder RNN is applied to a sentence the last state of the encoder has to retain all the information. For longer sentences this requires a huge state vector which is hard to learn. Attention mechanisms let's the decoder selectively look at different parts of the sentence based on the encoding vectors and decoding produced so far. This reduces the needed size of the encoding state vector. This attention mechanism is essentially a weighted mean, where the weights traditionally are calculated using a sparsemax transformation. However in many applications such as natural language processing it could make sense for these weights to be sparse, such that the attention becomes sparse.

In this paper we will show if:

- Sparsemax is useful as a multinomial logistic regression.
- Sparsemax is useful in attention models and if the attention becomes sparse.
- Sparsemax is computationally equivalent to softmax.

2. METHODS

The softmax transformation is defined as:

$$\mathrm{softmax}_i(\mathbf{z}) := \frac{\exp(z_i)}{\sum_j \exp(z_j)} \tag{1}$$

This transformation maps from $z \in \mathbb{R}^K$ to a probability distribution $\mathbb{P}^K := \{\mathbf{p} \in \mathbb{R}^K, \mathbf{1}^T\mathbf{p} = 1, \mathbf{p} \geq \mathbf{0}\}$. $\mathbf{1}^T\mathbf{p} = 1$ is ensured by dividing by the sum, likewise $\mathbf{p} \geq \mathbf{0}$ is ensured by the exponential function. However it is precisely the exponential function that prevents the softmax probability distribution from becoming sparse.

To allow sparsity, Martins et al. [1] suggest using the following optimization problem instead:

$$\mathrm{sparsemax}(\mathbf{z}) := \underset{\mathbf{p} \in \mathbb{P}^K}{\mathrm{argmin}} \, ||\mathbf{p} - \mathbf{z}||_2^2 \tag{2}$$

This can be written as a constrained quadratic program:

$$\begin{aligned} \min_{\mathbf{p}} \ & \mathbf{p}^T\mathbf{p} - 2\mathbf{z}^T\mathbf{p} \\ \text{s.t. } & \mathbf{1}^T\mathbf{p} = 1 \\ & \mathbf{p} \geq \mathbf{0} \end{aligned} \tag{3}$$

2.1. Sparsemax

Problem (3) is the constrained quadratic program known as the "continuous quadratic knapsack problem" or a "singly linearly constrained quadratic program". Solving it has been studied for a few decades. A breakthrough happened in 1980 where Helgason et al. [3] proposed an $\mathcal{O}(K\log(K))$ algorithm for solving it.

Algorithm 1 Calculate sparsemax probability distribution from logits $\mathbf{z}$.

```
function SPARSEMAX($\mathbf{z}$)
    SORT($\mathbf{z}$) as $z_{(1)} \geq \cdots \geq z_{(K)}$
    $k(\mathbf{z}) \leftarrow \max\left\{k \in [1..K] \mid 1 + kz_{(k)} > \sum_{j\leq k} z_{(j)}\right\}$
    $\tau(\mathbf{z}) \leftarrow \frac{\left(\sum_{j\leq k(\mathbf{z})} z_{(j)}\right)-1}{k(\mathbf{z})}$
    return $[\max(z_1 - \tau(\mathbf{z}), 0), \cdots, \max(z_K - \tau(\mathbf{z}), 0)]$
```

Other solution methods has been developed later. [4] contains a short overview of these. A popular approach is based on median search, these algorithms have a linear time complexity. The first algorithm using median search is [5], for a general overview of more recent algorithms see [6]. Another category is the variable fixing methods which avoid sorting and searching for the median, by finding the solution to one variable in each iteration. These algorithms have quadratic complexity, but performs much better in practice [7].

Unfortunately all of these methods were developed with serial computation in mind. As a consequence they do not easily apply to a GPU implementation, which has become the mainstream computational device for neural networks. For this reason the original method [3] is used with only minor modifications for parallelism as described in Algorithm 2.

Algorithm 2 Parallel sparemax.

```
function SPARSEMAX($\mathbf{z}$)
    $\mathbf{s} \leftarrow$ SORT($\mathbf{z}$)                    ▷ For example bitonic sort
    $\mathbf{c} \leftarrow$ CUMSUM($\mathbf{s}$)                  ▷ Parallel prefix sum
    $\mathbf{b} \leftarrow$ MAP($\lambda(k) := 1 + ks_k > c_k,\ k \in [1..K]$)
    $k(\mathbf{z}) \leftarrow$ SUM($\mathbf{b}$)                  ▷ Parallel reduce
    $\tau(\mathbf{z}) \leftarrow \frac{c_{k(\mathbf{z})}-1}{k(\mathbf{z})}$
    return MAP($\lambda(k) := \max(z_k - \tau(\mathbf{z}), 0),\ k \in [1..K]$)
```

Assuming K processors Algorithm 2 has time complexity $\mathcal{O}(\log(K))$, which is the same as softmax.

In neural network optimization where batches or mini-batches are used, one can also parallelize over the observations.

The sparsemax transformation is mostly differentiable, resulting in a well-defined Jacobian for the transformation. In Tensorflow [2] the chain rule is used for automatic differentiation. For example if the logits $\mathbf{z}$ is a function of some weight matrix $\mathbf{W}$, then the derivative is:

$$\frac{\text{sparsemax}(\mathbf{z})}{\partial \mathbf{W}} = \frac{\text{sparsemax}(\mathbf{z})}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{W}} \tag{4}$$

Thus in Tensorflow it is the "Jacobian times a vector" operation that needs to be defined. From [1] this is:

$$\mathbf{J}_{\text{sparsemax}}(\mathbf{z}) \cdot \mathbf{v} = \mathbf{s} \odot (\mathbf{v} - \hat{v}\mathbf{1}), \ \text{where } \hat{v} := \frac{\mathbf{s}^T\mathbf{v}}{||\mathbf{s}||} \tag{5}$$

2.2. Sparsemax Loss

The sparsemax loss function is derived by defining the gradient of the sparsemax loss to be similar to the gradient of the softmax loss (entropy loss):

$$\nabla_{\mathbf{z}} \mathcal{L}_{\text{sparsemax}}(\mathbf{z}; \mathbf{q}) = -\mathbf{q} + \text{sparsemax}(\mathbf{z}) \tag{6}$$

where $\mathbf{q} \in \mathbb{P}^K$ is the target probability distribution.

From the gradient (6) one can easily create the "Gradient times a scalar" operation that tensorflow needs.

$$\nabla_{\mathbf{z}} \mathcal{L}_{\text{sparsemax}}(\mathbf{z}; \mathbf{q}) \cdot v = (-\mathbf{q} + \text{sparsemax}(\mathbf{z}))v \tag{7}$$

From (6) the sparsemax loss (its primitive function) can then be derived [1]:

$$\mathcal{L}_{\text{sparsemax}}(\mathbf{z}; \mathbf{q}) = \frac{1}{2} \sum_{j \in S(\mathbf{z})} (z_j^2 - \tau^2(\mathbf{z})) + \frac{1}{2}||\mathbf{q}||^2 - \mathbf{q}^T\mathbf{z} \tag{8}$$

The loss function (8) dependency on $\tau(\mathbf{z})$ is not ideal, as it is inconvenient to recalculate it or alternatively store it in memory. Fortunately, because the sum is over the support $S(\mathbf{z})$ one can reformulate it to only be dependent on $\mathbf{z}$, $\mathbf{p} = \text{sparsemax}(\mathbf{z})$ and $\mathbf{q}$:

$$\mathcal{L}_{\text{sparsemax}}(\mathbf{z}; \mathbf{q}) = \frac{1}{2} \sum_{j \in S(\mathbf{z})} p_j(2z_j - p_j) + \frac{1}{2}||\mathbf{q}||_2^2 - \mathbf{q}^T\mathbf{z} \tag{9}$$

Finally the sparsemax loss (9) can be changed slightly, to be a simple map and reduce which is ideal for parallelization:

$$\mathcal{L}_{\text{sparsemax}}(\mathbf{z}; \mathbf{q}) = \mathbf{1}^T \left(\mathbf{s} \odot \mathbf{p} \odot \left(\mathbf{z} - \tfrac{1}{2}\mathbf{p}\right) + \mathbf{q} \odot \left(\tfrac{1}{2}\mathbf{q} - \mathbf{z}\right)\right) \tag{10}$$

2.3. Sparsemax Regression

We define sparsemax regression as the equivalent to a multivariate logistic regression, but where the activation function at the output layer is sparsemax rather than softmax and with a L2-regularization on the weights and biases. Thus the minimization problem becomes

$$\min_{\mathbf{W},\mathbf{b}} \frac{\lambda}{2}(||\mathbf{W}||_F^2 + ||\mathbf{b}||_F^2) + \frac{1}{N}\sum_{i=1}^{n} \mathcal{L}(\mathbf{W}\mathbf{x}_i + \mathbf{b}; y_i) \quad (11)$$

Where $\mathcal{L} = \mathcal{L}_{\text{sparsemax}}$ as defined in (8).

2.4. Sparsemax Attention

An attention mechanism implements the ability to look at certain part of the input sequence in the encoding, in order to produce a prediction in the decoding. This is often a desired feature in seq2seq models. For these tasks the classical attention approach is to use a softmax activation function to compute the attentions $\boldsymbol{\alpha}_i$. This means that $\boldsymbol{\alpha}_i = 0$ never occurs, which implies that the decoder does pay some attention to the entire sequence. However, if one uses the sparsemax activation function, $\boldsymbol{\alpha}_i$ will be sparse and thus the model should only look at the relevant subset of the sequence.

The model used here is a naive simplification of the one presented in the Bahdanau et al., 2014 model [8].

$$\begin{aligned}
&\text{encoding:} \\
&\quad \mathbf{h}_t = \text{GRU}(\overrightarrow{\mathbf{E}}\mathbf{x}_t, \mathbf{h}_{t-1}) \\
&\text{attention:} \\
&\quad e_{it} = \mathbf{v}^T \tanh(\mathbf{W}\mathbf{s}_{i-1} + \mathbf{U}\mathbf{h}_t) \\
&\quad \boldsymbol{\alpha}_i = \text{sparsemax}(\mathbf{e}_i) \qquad (12) \\
&\quad \mathbf{c}_i = \textstyle\sum_{t=1}^{T} \alpha_{it}\mathbf{h}_t \\
&\text{decoder:} \\
&\quad \mathbf{s}_i = \text{GRU}(\mathbf{c}_i, \mathbf{s}_{i-1}) \\
&\quad \mathbf{y}_i = \text{softmax}(\overleftarrow{\mathbf{E}}\mathbf{s}_i)
\end{aligned}$$

In this attention model (12), $\overrightarrow{\mathbf{E}}$ and $\overleftarrow{\mathbf{E}}$ are embedding models, trained simultaneously with the RNN. $\mathbf{W}$, $\mathbf{U}$ are weight matrices and $\mathbf{v}$ is a weight vector.

3. RESULTS

The sparsemax transformation is evaluated using the model defined in (11) in terms of both classification performance and computational performance. It is also used in attention modelling as described in (12). For comparison the results using softmax are also reported.

3.1. Label estimation

Five well-known benchmark datasets are used, as shown in Table 1. The digit dataset MNIST and flower dataset Iris are multi-class classification, while the Scene, Emotions and CAL500 datasets are multi-label classification.[1]

	#Features	#Labels	Train size	Test size
MNIST	784	10	60000	10000
Iris	4	3	135	15
Scene	294	6	1211	1196
Emotions	72	6	391	202
CAL500	68	174	451	51

Table 1: Summary for the five benchmark datasets used.

For comparison of performance the Jensen-Shannon divergence between the predicted distribution and target distribution is reported.

$$\mathbf{JS}(\mathbf{q}, \mathbf{p}) := \frac{1}{2}\mathbf{KL}\left(\mathbf{q}\middle|\middle|\frac{\mathbf{q}+\mathbf{p}}{2}\right) + \frac{1}{2}\mathbf{KL}\left(\mathbf{p}\middle|\middle|\frac{\mathbf{q}+\mathbf{p}}{2}\right)$$

Were $\mathbf{p}, \mathbf{q}$ are the predicted and target distributions, respectively and $\mathbf{KL}$ is the Kullbach-Leibler distance. For both the classifiers an L2 regularizer is used. The optimized regularization parameters can be seen in Table 2. The hyperparameter was tuned using a stratified 5-split cross-validation for the MNIST and Iris datasets and non-stratified for the rest. The resulting experiments are shown in Figure 1. The models were optimized using the Adam algorithm with default parameters, $\lambda = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$.

	Softmax		Sparsemax	
	λ	**JS**	λ	**JS**
MNIST	10^{-2}	0.10 ± 0.002	10^{-2}	0.10 ± 0.003
Iris	10^{-8}	0.13 ± 0.010	10^{-8}	0.09 ± 0.022
Scene	10^{-8}	0.19 ± 0.021	10^{-3}	0.19 ± 0.020
Emotions	10^{-8}	0.28 ± 0.020	10^{-1}	0.28 ± 0.014
CAL500	10^{-8}	0.35 ± 0.003	10^{0}	0.35 ± 0.003

Table 2: Regularization values and the corresponding JS divergence for both classifiers. 5-fold cross validation was used on the training data. The 95% confidence interval is shown.

[1]In multi-class classification the classes are mutually exclusive. In the multi-label case several labels can be associated with one observation.

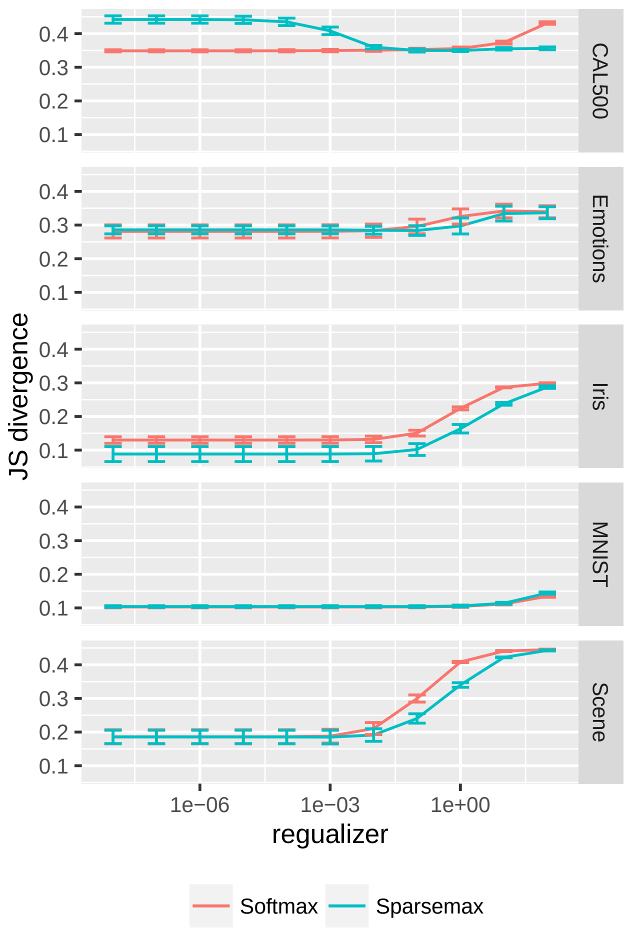

Fig. 1: JS divergence using a range of regularizers on all five datasets for the two regressors. The 95% confidence interval is shown.

The JS divergence on the test set for the five datasets and the error rate for the multi-class datasets is shown in Table 3. The softmax and sparsemax classifiers are very much alike in terms of performance. The Iris dataset shows an improved performance, which is also apparent in the regularization optimization (Table 1) where a statistical significant difference exists for all parameter choices.

	Softmax		Sparsemax	
	JS	error rate	**JS**	error rate
MNIST	0.098	14.5%	0.100	14.5%
Iris	0.138	20.0%	0.104	13.3%
Scene	0.186	–	0.181	–
Emotions	0.272	–	0.270	–
CAL500	0.355	–	0.357	–

Table 3: JS divergence for the five benchmark datasets and the sparsemax classifier as well as the softmax classifier.

3.2. Computational performance

Four implementations were benchmarked using the datasets from Table 1. The implementations can be found on GitHub[2].

- Softmax: A TensorFlow implementation of Softmax Regression.
- Numpy: A pure Numpy implementation of Sparsemax Regression.
- TF Numpy: A TensorFlow implementation where the custom ops associated with Sparsemax have been implemented using only Numpy.
- TF CPU: A TensorFlow implementation where the custom ops associated with Sparsemax have been implemented in C++ for the CPU.
- TF GPU: A TensorFlow implementation where the custom ops associated with Sparsemax have been implemented in C++ and CUDA for the GPU.

In Table 4.1 the computational performance is shown and compared with the native TensorFlow Softmax implementation. Each implementation has been run for 100 epochs. The GPU is an Nvidia® Tesla K40c and the CPU is an Intel® Xeon® CPU E5-2630 v2 (12 cores).

For the large dataset, MNIST, the GPU implementation significantly outperforms all other sparsemax implementations as expected. The Numpy implementation runs significantly slower than the three TensorFlow implementations, also as expected. Compared to softmax, the GPU version of TensorFlow is significantly slower, but not by much.

For the Iris dataset, which is considerably smaller in size, the Numpy and TensorFlow CPU version outperforms both the GPU and Tensorflow Numpy version.

3.3. Encoder-decoder with a sparse attention mechanism

The encoder-decoder model defined in (12) has been tested on a synthetic translation dataset. The input sequence is a sequence of numbers spelled by regular characters e.g. "three seven four". The goal is then to translate this sequence to actual numbers e.g. "374#". Where # is the end of sequence tag.

The weights in the model have all been initialized using a truncated random normal with a standard deviation of 0.1, except for bias terms which have been initialized using a constant initializer of 0.

The model was optimized using the ADAM optimizer with a learning rate of 0.005, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 1\text{e}{-8}$, batch sizes of 100 observations and for a 1000 epochs.

[2]https://github.com/AndreasMadsen/course-02456-sparsemax

	Softmax	Numpy	TF Numpy	TF CPU	TF GPU
MNIST	3.30 ± 0.048	48.29 ± 0.027	10.24 ± 0.110	11.70 ± 0.096	3.73 ± 0.026
Iris	0.18 ± 0.007	0.02 ± 0.002	0.31 ± 0.008	0.07 ± 0.013	0.18 ± 0.007
Scene	0.23 ± 0.005	0.29 ± 0.019	0.64 ± 0.008	0.62 ± 0.020	0.22 ± 0.007
Emotions	0.18 ± 0.006	0.04 ± 0.002	0.37 ± 0.005	0.18 ± 0.007	0.19 ± 0.005
CAL500	0.34 ± 0.006	0.52 ± 0.029	1.63 ± 0.020	1.17 ± 0.020	0.42 ± 0.011

Table 4: Time in seconds with associated confidence intervals.

The results of applying sparse attention compared to soft attention, can be seen in the attention plots Figure 3 and Figure 2. The validation accuracy is stated in the figure caption.

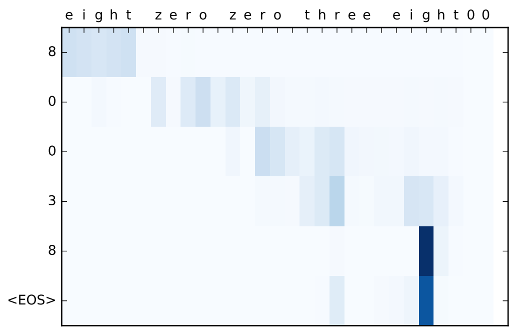

Fig. 2: Softmax attention for an Encoder-Decoder RNN. Validation accuracy after 100k training examples: $\approx 75\%$.

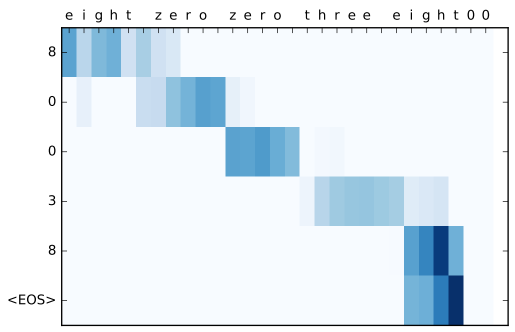

Fig. 3: Sparsemax attention for an Encoder-Decoder RNN. Validation accuracy after 100k training examples: $\approx 98\%$.

4. DISCUSSION

Our TensorFlow implementation of sparsemax has shown that it is a suitable alternative for the popular softmax transformation. As the experiments indicate, it is on par with the softmax transformation in terms of computing performance and predictive ability. It has shown to create sparse attention that improve the performance of the text to digits task. As such, the sparsemax transformation has shown to be interesting in terms of deep learning and attention modelling.

An interesting result is that the multinormal regression on Iris dataset performed much better with sparsemax than with softmax. A likely explanation for this, is that sparsemax creates a slight bias for the probability being either 0 or 1. Because the Iris has very few observations such a bias would help with prediction. This is similar to using a prior distribution in Bayesian statistics. This intuition could perhaps be verified by varying the dataset size for a larger or synthetic dataset.

The attention model turned out to produce much better results than what we could have hoped for, which is very encouraging. A possible issue that, in theory, could have arose, is if the attention vector becomes $\{0, \cdots, 0, 1, 0, \cdots 0\}$. In that case the gradient will also be zero, and it will unfeasible to optimize the model. But as shown in Figure 3 this was not an issue. Note that this will never be a problem for the multinomial regression as the loss function is convex, only for $\text{sparsemax}(\mathbf{z}) = \mathbf{q}$ will the gradient be zero.

In the attention experiment we also tried using sparsemax in the output layer, this resulted in very similar good results. However it had the added effect of being more numerically stable than softmax, because the loss function doesn't use $\log(\cdot)$. Numerical stability is also an advantage of sparsemax that one should consider.

4.1. Implementation

Implementing sparsemax was no trivial task, especially compared to softmax. The implementation here uses an algorithm similar to that covered in Helgason et al. [3] as this is a fairly simple algorithm and it is well suited for parallel implementation on the GPU.

The main challenge when implementing sparsemax on the GPU is the sorting algorithm. This is because TensorFlow and CuDNN does not provide a sorting implementation, it has thus been necessary to define our own custom sorting function in TensorFlow. Parallel sorting algorithms on the GPU are very difficult to implement, thus it's difficult to get a high-performing implementation. Our implementation uses an odd-even sorting algorithm, which has $\mathcal{O}(K)$ complexity. A faster sorting algorithm like bitonic sort which has $\mathcal{O}(\log(K))$ complexity could be considered, however these algorithms are much more difficult to implement. We do not believe that the choice of sorting algorithm has a big performance impact in our case, as the experiments have fairly few labels, and dimensionality in the attention case. When there are few labels the logits fit within the L1 cache, thus the odd-even sort doesn't have to interact with the global GPU memory.

For the CPU case the situation is much different because sequential solution algorithms can easily be used. These have shown to be much more efficient than the naive sorting approach [4]. An obvious choice is thus to use two completely different algorithms for the CPU and GPU implementations. This is not something we have done here, but in future work we would strongly recommend this.

The computational performance from Table , showed as expected that our sparsemax GPU implementation underperforms (statistically significantly) the softmax implementation, but not notably. While the softmax is much simpler than sparsemax, it does involve a lot of exponential calculations, which are slow to compute on a GPU. Sparsemax only uses addition and multiplication. Note that these observations only hold for the MNIST dataset, but given the small size of the other datasets, that is likely also the only relevant dataset to look at.

For the smaller dataset the TensorFlow CPU implementation is the fastest. This is likely because the TensorFlow Numpy version has a communication overhead, between the Tensorflow C++ backend and the Python frontend. Similarly the GPU version has an overhead from transferring data and starting kernels. In particular the latter would be an issue for a small dataset like Iris.

5. CONCLUSION

We have shown how to successfully implement sparsemax in TensorFlow for both the CPU and GPU using C++ and CUDA. More specifically, we have shown that the sparsemax and sparsemax loss can be made suitable for parallel GPU implementation, by applying only minor changes to the algorithms and equations. Our results shows that the sparsemax implementation is slightly but statistically significantly slower than the softmax implementation. But this is expected as it hasn't been optimized as much as the TensorFlow native softmax implementation. We have high hopes for the the computational performance properties of sparsemax because it doesn't use any complex function, such as $\exp(\cdot)$. On the other hand it depends on parallel sorting algorithm, which is difficult for GPUs.

Using this sparsemax implementation, we have applied sparsemax to both multi-class and multi-label classification, as well as to the attention mechanism in a encoder-decoder model. Our results show that for simple classification using sparsemax performs similarly to using softmax. Only for one dataset, Iris, did we get improved results by using sparsemax. We suspect this is because Iris is a very small dataset, thus sparsemax acts as a bias towards 0 and 1, which improves the performance. This would be similar to using a prior in Bayesian statistics.

In the encoder-decoder model results showed a huge performance gain by using sparsemax. For a synthetic seq2seq problem the validation accuracy went from 75% to 98% by using sparsemax. The attention plots also show very accurate and sparse attention weights, when using sparsemax. One initial fear was that if $\mathrm{sparsemax}(\cdot)$ produced an "indicator" vector for the attention weights, then the gradient would become zero and the model would be untrainable. However this did not turn out to be a problem in practice.

Due to the success of our results, we have later published our sparsemax implementation to the TensorFlow project [2] as a Pull Request on GitHub[3].

6. ACKNOWLEDGEMENT

Thanks to Alexander Rosenberg Johansen who supervised us. He originally proposed this project and provided code for the attention model. Thanks to the authors of the "From Softmax to Sparsemax" [1] paper, André F. T. Martins and Ramón Fernandez Astudillo. Thanks to the TensorFlow project [2]. Finally, thanks to DTU's High-Performance-Computing cluster team, who provided hardware, in particular GPUs, such that we could run the experiments.

7. REFERENCES

[1] A. F. T. Martins and R. F. Astudillo, "From softmax to sparsemax: A sparse model of attention and multi-label classification", *CoRR*, vol. abs/1602.02068, 2016. [Online]. Available: `http://arxiv.org/abs/1602.02068`.

[2] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *TensorFlow: Large-scale machine learning on heterogeneous*

[3]https://github.com/tensorflow/tensorflow/pull/6387

systems, Software available from tensorflow.org, 2015. [Online]. Available: `http://tensorflow.org/`.

[3] R. Helgason, J. Kennington, and H. Lall, "A polynomially bounded algorithm for a singly constrained quadratic program", *Mathematical Programming*, vol. 18, no. 1, pp. 338–343, 1980, ISSN: 1436-4646. DOI: `10.1007/BF01588328`. [Online]. Available: `http://dx.doi.org/10.1007/BF01588328`.

[4] M. Liu and Y.-J. Liu, "Fast algorithm for singly linearly constrained quadratic programs with box-like constraints", *Computational Optimization and Applications*, pp. 1–18, 2016, ISSN: 1573-2894. DOI: `10.1007/s10589-016-9863-8`. [Online]. Available: `http://dx.doi.org/10.1007/s10589-016-9863-8`.

[5] P. BRUCKER, "An o(n) algorithm for quadratic knapsack-problems", eng, *Operations Research Letters*, vol. 3, no. 3, pp. 163–166, 1984, ISSN: 18727468, 01676377. DOI: `10.1016/0167-6377(84)90010-5`.

[6] K. C. Kiwiel, "Breakpoint searching algorithms for the continuous quadratic knapsack problem", eng, *Mathematical Programming*, vol. 112, no. 2, pp. 473–491, 2008, ISSN: 14364646, 00255610. DOI: `10.1007/s10107-006-0050-z`.

[7] K. C. Kiwiel, "Variable fixing algorithms for the continuous quadratic knapsack problem", eng, *Journal of Optimization Theory and Applications*, vol. 136, no. 3, pp. 445–458, 2008, ISSN: 15732878, 00223239. DOI: `10.1007/s10957-007-9317-7`.

[8] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate", *CoRR*, vol. abs/1409.0473, 2014. [Online]. Available: `http://arxiv.org/abs/1409.0473`.