Faster Gated Recurrent Units via Conditional Computation

Andrew S. Davis
University of Tennessee, Knoxville
adavis72@tennessee.edu

Itamar Arel
University of Tennessee, Knoxville
itamar@utk.edu

***Abstract*—In this work, we apply the idea of conditional computation to the gated recurrent unit (GRU), a type of recurrent activation function. With slight modifications to the GRU, the number of floating point operations required to calculate the feed-forward pass through the network may be significantly reduced. This allows for more rapid computation, enabling a trade-off between model accuracy and model speed. Such a trade-off may be useful in a scenario where real-time performance is required, allowing for powerful recurrent models to be deployed on compute-limited devices.**

I. Introduction

Conditional computation, originally proposed in [1], is the notion of skipping the computation of some nodes given the values of other nodes in the network. Some training regularization or sparsification techniques such as [2] and [3] tend to skip the required computation in a random or completely fixed manner. [1], on the other hand, proposes to "drop them [the calculation of hidden units] in a learned and optimized way."

The motivation for conditional computation is that of substantially increasing model capacity (i.e., the amount of information a neural network can store) while reducing the growth of computation required to scale to larger models. While other approaches of conditional computation have had applications to feed-forward and convolutional models [4] [5] [6], conditional computation in recurrent models has yet to be explored.

II. Gated Recurrent Unit

The GRU [7] is a recently proposed recurrent unit similar to the LSTM [8], but it contains some simplifications to the recurrent structure of the unit. While the LSTM contains three gates (input gate, forget gate, and output gate), the GRU has two – a forget gate r_t, and an output gate z_t that is used to update the next hidden state h_t as a convex combination of the previous state h_{t-1} and a proposal state $\hat{h}_t$:

$$z_t = \sigma\left(W_z x_t + U_z h_{t-1} + b_z\right) \quad (1)$$

$$r_t = \sigma\left(W_r x_t + U_r h_{t-1} + b_r\right) \quad (2)$$

$$\hat{h}_t = f\left(W_h x_t + U_h\left(r_t \cdot h_{t-1}\right) + b_h\right) \quad (3)$$

$$h_t = \left(1 - z_t\right) \cdot h_{t-1} + z_t \cdot \hat{h}_t \quad (4)$$

where $\sigma(\cdot)$ is some function with a range of $(0, 1)$, $f(\cdot)$ is any elementwise nonlinearity, and $W_{(\cdot)}$, $U_{(\cdot)}$, and $b_{(\cdot)}$ are trainable parameters. The states are vectors $z_t, r_t, \hat{h}_t, h_t \in R^h$, the input vector $x_t \in R^d$, non-recurrent weights $W_{(\cdot)} \in R^{h \times d}$, and recurrent weights $U_{(\cdot)} \in R^{h \times h}$. A common selection for the gating function $\sigma(\cdot)$ is the logistic sigmoid function, and a common selection for the hidden state proposal $f(\cdot)$ is the hyperbolic tangent.

III. Accelerating the Gated Recurrent Unit

Existing models of conditional computation rely on predicting sparsity in the activations to determine which portions of the neural network must be calculated. Sparsity in feed forward networks is typically encouraged by adding an ℓ_1 or a Kullback-Leibler penalty to the activations, which can encourage the rate of sparsity necessary for conditional computation, or can be encouraged much more mildly by dropout regularization. However, recent literature [9] shows that dropout, and thus sparsity, must be carefully applied to recurrent units such as LSTMs. If dropout is not carefully applied, then the model suffers due to the corruption of information flow over many timesteps.

Instead of relying on the sparsity of the hidden activations to reduce computational burden, the model can be modified in order to rely on the sparsity of the gating activation z_t. In Eq. 1, when z_t approaches zero, the influence of the proposal state $\hat{h}_t$ diminishes, and the previous state h_{t-1} is passed forward to the next timestep. If z_t is exactly zero, then no computation is required for an element i of h_t^i, as the value of h_{t-1}^i can simply be copied forward.

In its current configuration, calculating z_t accounts for approximately $^1/_3$ of the floating point operations. If z_t had all zero values, the best case reduction in floating point operations would be only $^1/_3$, allowing for a 3x speed increase. In order to obtain greater acceleration, it is necessary to reduce the number of floating point operations required to compute z_t. Here, we introduce two methods, both of which can be seen as low-rank constraints on W_z and U_z, which reduce the computational requirements of matrix-vector or matrix-matrix operations.

In addition to imposing a low-rank constraint on W_z and U_z, a change to Eq. 3 must be made. Because the objective is to bypass the calculation of individual entries h_t^i of h_t, all elements associated with the computation h_t^i must be able to be bypassed as well. In the computation of the next state proposal $\hat{h}_t$, r_t gates h_{t-1} prior to the linear transformation

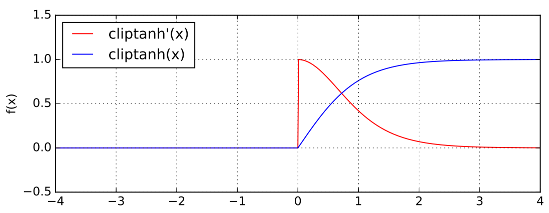

Fig. 1. An illustration of the clipped tanh function and its derivative.

through U_h. Therefore, all elements of r_t must be computed in order to compute $\hat{h}_t$. To decrease the number of floating point operations further, a simple modification of Eq. 3 moves the forget gate r_t to the outside of the linear transformation:

$$\hat{h}_t = f\left(W_h x_t + r_t \cdot U_h h_{t-1} + b_h\right) \tag{5}$$

In this sense, the forget gate now gates the linear transformation $U_h h_{t-1}$ of the previous state h_{t-1} rather than the previous state itself. With these modifications, the potential efficiency gains become primarily a function of the sparsity of the output gating z_t.

IV. Constraining the Sparsity of z_t

Because the potential computational benefits are now mostly dependent on z_t, it is important to introduce a mechanism to control the sparsity of z_t. First, we propose an activation function in the range of $[0,1]$ that produces values that reach zero, instead of merely approaching it in a limit. To this end, we propose replacing the activation function of z_t from a sigmoidal activation to a clipped hyperbolic tangent (illustrated in Fig. 1):

$$f(x) = \begin{cases} \tanh(x) & x > 0 \\ 0 & \textit{otherwise} \end{cases} \tag{6}$$

In the positive range where $x > 0$, the hyperbolic tangent has similar properties to the sigmoidal activation, in that it gradually saturates to a value of 1. On the negative end, however, it behaves like a rectifier, blocking any preactivation with a negative value from propagating forward (or backward, during backpropagation). This property allows for the simple induction of sparsity in z_t.

In order to control the level of sparsity in z_t, we propose a modification of batch normalization [10]:

$$BN(z_t, s) = \frac{z_t - \mu}{\sigma^2} - s \tag{7}$$

where μ and σ^2 correspond to the mean and variance mini-batch statistics. Instead of allowing for trainable vectors β and γ in the affine transformation as in standard batch normalization, we have a scalar hyperparameter s that allows for more direct control of the sparsity of z_t.

V. Block-Sparse Gating versus Unstructured Gating

In the implementation of conditional computation for the purposes of training models that will be faster at test time, the structure of the sparsity is a significant consideration. To this end, there are two types of sparsity that may be employed - that of unstructured sparsity and block sparsity. In the unstructured setting, there are no constraints imposed on the sparsity pattern of the z_t gating. In the block sparse setting, however, the sparsity pattern is constrained in the sense that contiguous sets of activations are active or inactive with respect to each other. Figure 2 illustrates the difference between the two types of sparsity. Given the difference between the nature of the computations, an unstructured gating is well suited to processing one sample at a time (e.g., a mobile phone processing a single voice stream), and block-sparse gating is well suited to processing several examples at a time (e.g., a server batch processing several examples at once.) Because BLAS libraries are very well optimized, we choose to implement the block-sparse and the unstructured gating with BLAS primitives.

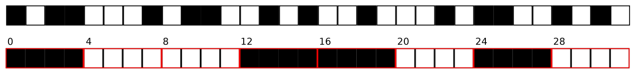

Fig. 2. An illustration contrasting unstructured sparsity (above) with block-sparsity (below). In this case, the block-sparse representation is constrained to be sparse in contiguous chunks of length 4, and the sparsity pattern must align with the red outlines.

A. Unstructured Gating

The unstructured gating is formulated with a targeted use case of single example processing, that is, instead of sending several examples in a minibatch through the GRU, only one sample is sent. Potential use cases involve real-time applications where there is only a single example that can be processed. In such cases, there is a significantly lower degree of parallelism that a CPU or GPU can exploit, so obtaining speed benefits simply by skipping the dot products between the input vector and the weight vectors corresponding to sparsified activations is relatively straightforward, and can be accomplished simply by copying the non-sparse weight vectors to some temporary storage, calling GEMV from the BLAS library, and writing the result back to the appropriate output elements.

If the parameterization of z_t is left as-is, the greatest speed increase we could obtain is around 3x, as the calculation of z_t is roughly one-third of the operations in the GRU state update equation. If z_t outputs a zero vector, then the other two-thirds of the required computations may be skipped, resulting in the 3x speed increase. In order to raise this upper bound, we reparameterize the z_t update with a bottleneck layer. In this case, we will project x_t and h_{t-1} to a lower-dimensional space g, apply a nonlinearity such as ReLU, and then expand this representation to the space corresponding to the hidden state dimensionality h:

$$z_t^{lr} = f\left(W_z^{lr} x_t + U_z^{lr} h_{t-1} + b_z^{lr}\right) \tag{8}$$

$$z_t = \sigma\left(BN\left(W_z z_t^{lr} + b_z\right)\right) \tag{9}$$

where $BN(\cdot)$ may be either of the two batch normalization approaches introduced above. In reparameterizing z_t in such

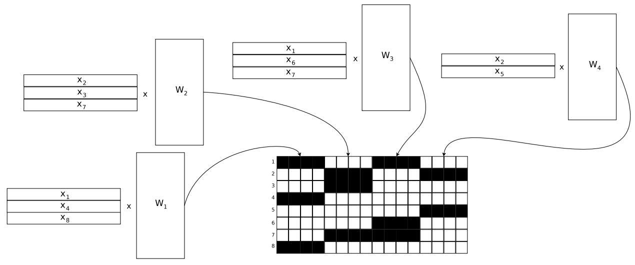

Fig. 3. An illustration of the block-sparse multiplication recast as several matrix-matrix multiplications, given a sparsity mask.

Architecture	1024-1024	1024-1024	1024-1024
Block Size	16	32	64
Gating Dim	64	32	16
Learning Rate	0.001		
Optimizer	ADAM		
z Nonlinearity	CLIPTANH $(\cdot)$		
r Nonlinearity	$\sigma(\cdot)$		
State Nonlinearity	$\tanh(\cdot)$		
$W_{(\cdot)}$ *Init*	Glorot		
$U_{(\cdot)}$ *Init*	Orthogonal		

TABLE I
HYPERPARAMETERS FOR THE BLOCK-SPARSE GATED LANGUAGE MODELS.

Block Size / Gating Dim.	16/64	32/32	64/16
$s = 0.00$	0.73x	1.06x	1.20x
$s = -0.25$	1.88x	1.72x	1.68x
$s = -0.50$	1.72x	1.62x	1.68x
$s = -0.75$	1.57x	1.62x	1.95x

TABLE II
BLOCK-SPARSE MODEL ACCELERATION FACTOR OVER A FULLY-DENSE MODEL. ALL ENTRIES IN THE TABLE ARE AVERAGED OVER 10 TRIALS OF 1000 FEED-FORWARDS.

a way, we are increasing the upper bound of the speedup this approach can yield at the expense of the capacity or expressiveness of the gating function.

B. Block-Sparse Gating

The block-sparse gating is formulated with a targeted use case of batch processing. Unlike the unstructured gating, the block-sparse gating has the potential to be implemented with multiple matrix-matrix multiplications, as illustrated in Figure 3. The block sparse approach is also used in [4] [5] in order to exploit sparsity for speedup in feed forward networks. In such a formulation, the parameterization of z_t can be expressed exactly as in Equation 8 and Equation 9, except W_z and b_z are fixed to non-trainable values:

$$W_z = \begin{bmatrix} \mathbf{1}^T & \mathbf{0}^T & \cdots & \mathbf{0}^T \\ \mathbf{0}^T & \mathbf{1}^T & \cdots & \mathbf{0}^T \\ \vdots & \vdots & & \vdots \\ \mathbf{0}^T & \mathbf{0}^T & \cdots & \mathbf{1}^T \end{bmatrix}, b_z = 0 \qquad (10)$$

where $\mathbf{0}^T, \mathbf{1}^T \in$

$$z_t = \left[\left(\mathbf{z_t^{lr}}\right)_\mathbf{1} \quad \left(\mathbf{z_t^{lr}}\right)_\mathbf{2} \quad \cdots \quad \left(\mathbf{z_t^{lr}}\right)_\mathbf{g} \right] \qquad (11)$$

where $\left(\mathbf{z_t^{lr}}\right)_\mathbf{i}$ is defined as a ${}^{h}/_{g}$-dimensional vector with all entries equal to $\left(z_t^{lr}\right)_i$. Such a transformation can be implemented efficiently with a copy rather than a matrix multiplication, resulting in an operation that requires significantly fewer FLOPs than the fully trainable low-rank bottleneck approach.

VI. EXPERIMENTS

A. Character-Level Language Modeling and TEXT8

In order to study the effects and speed benefits of the alternative GRU parameterizations, we evaluate the models as applied to language modeling on the TEXT8 dataset. In language modeling, the goal is to train a probabilistic model $p\left(x_t \mid \theta, x_{t-1}, \ldots, x_{t-n}\right)$ that estimates the probability of a token x_t occuring given some history of tokens $x_{t-1}, \ldots, x_{t-n}$. Tokens may be specific words in the case of word-level language modeling, or they may be individual characters in the case of character-level language modeling. There are benefits and drawbacks to either approach: handling massive vocabularies and out-of-vocabulary tokens in word-level modeling can pose challenges, but the extra parameters required to memorize particular words and the extra modeling effort required to handle sequences spanning longer-term dependencies make character level modeling less desirable in some applications.

The TEXT8 dataset consists of 10^8 bytes from an abbreviated and cleaned English Wikipedia corpus. The dataset is stripped of all non-alphabetical characters such as XML markup, punctuation, and so forth. In the character-level modeling task, there are 27 tokens - lower-case a-z, as well as a 'space' token to provide separation between individual words. Given the size of the corpus and the diversity of the content, TEXT8 is a common dataset to evaluate language modeling techniques. The first 95% of the dataset is used as training data, and the remaining 5% is used as validation data. The bits-per-character metric (BPC) is used to evaluate the models.

Both the block sparse and the unstructured models are trained with truncated BPTT, backpropagating 50 timesteps per update while retaining the hidden state between sequences. The hidden state is reset every 1000 updates. Both models use a minibatch size of 64. The tokens are represented as one-hot vectors, making the input and output dimensionalities a size of 27. Both models have a softmax output and are trained with categorical cross entropy.

B. Conditional Models - Block Sparse

In order to evaluate the block-sparse approach and to understand how varying the blocks sizes and batch normalization biases impact the overall speed and accuracy of the models, we train twelve networks: the product of choices between the block size $bs = [16, 32, 64]$ and the biases $s = [0.00, -0.25, -0.50, -0.75]$. Further hyperparameters are given in Table I. The training curves of the block-sparse models are given in Fig. 4. The acceleration factors over a densely calculated baseline are given in Table II.

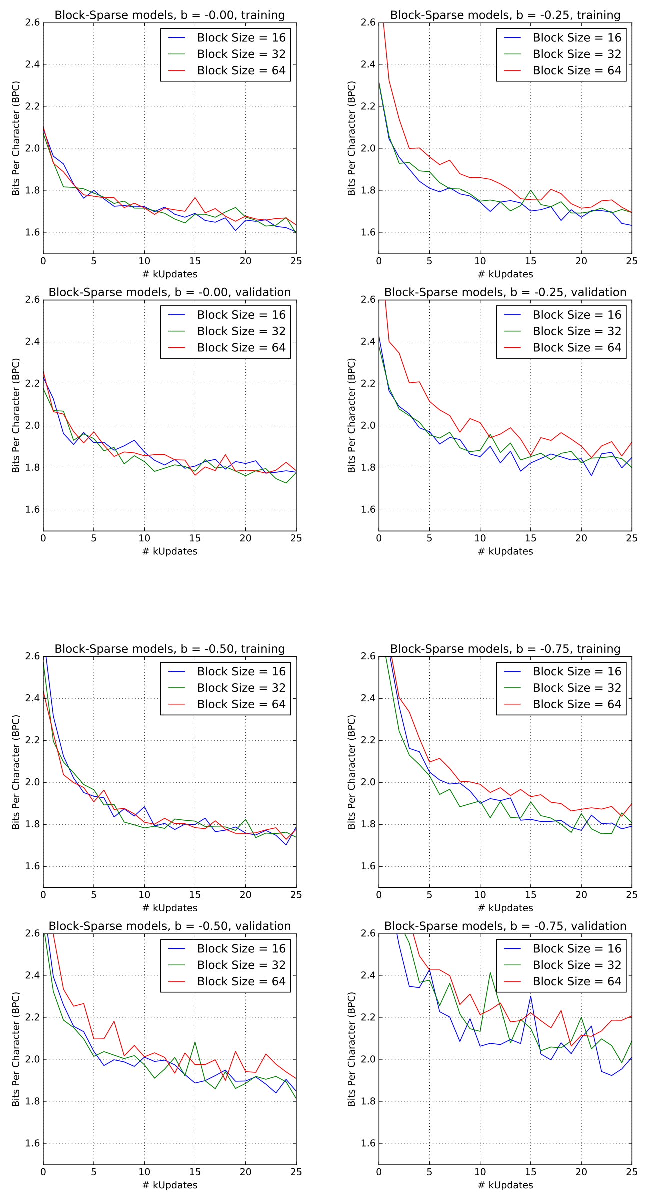

Fig. 4. A plot of the block-sparse model training and validation performances as measured in BPC as the models train.

Architecture	1024-1024	1024-1024	1024-1024
z Gating Rank	16	32	64
z Biases	$[0, -0.25, -0.50, -0.75]$		
Learning Rate	0.001		
Optimizer	ADAM		
z Nonlinearity	CLIPTANH $(\cdot)$		
r Nonlinearity	$\sigma(\cdot)$		
State Nonlinearity	$\tanh(\cdot)$		
$W_{(\cdot)}$ Initialization	Glorot		
$U_{(\cdot)}$ Initialization	Orthogonal		

TABLE III
HYPERPARAMETERS FOR THE BLOCK-SPARSE GATED LANGUAGE MODELS.

Rank Dimensionality	16	32	64
$s = 0.00$	1.55x	1.38x	1.43x
$s = -0.25$	1.87x	1.67x	1.63x
$s = -0.50$	1.86x	1.66x	1.99x
$s = -0.75$	2.55x	2.18x	2.29x

TABLE IV
UNSTRUCTURED SPARSITY MODEL ACCELERATION FACTOR OVER A FULLY-DENSE MODEL. ALL ENTRIES IN THE TABLE ARE AVERAGED OVER 10 TRIALS OF 1000 FEED-FORWARDS.

In general, as the s term becomes more negative, greater speedups are achievable. However, this comes at the cost of less accuracy in the models – as the s term is lowered from 0.00 to -0.25, the validation BPC raises from around 1.80 to around 1.90 for the 64/16 model. Lowering s to -0.5 reduces the BPC for the 16/64 and 32/32 models only slightly, but increases the BPC for the 64/16 model to approximately 1.95. Lowering s to -0.75 introduces instabilities into the validation accuracy and degrades the accuracy of the models significantly.

While the greatest achieved speedup is with the 64/16 model with $s = -0.75$, the most practical model is the 16/64 model with $s = -0.25$. This model realizes a good tradeoff between speed and accuracy, only marginally increasing the validation BPC while resulting in a model that runs around 1.88x faster than the baseline.

C. *Conditional Models - Unstructured*

Similar to the block sparse experiments, we train twelve networks on the product of choices between rank sizes $r = [16, 32, 64]$ and biases $s = [0.00, -0.25, -0.50, -0.75]$. Further hyperparameters are given in Table III. The training curves of the unstructured models are given in Fig. 5. The acceleration factors over a densely calculated baseline are given in Table IV.

In general, the unstructured models fit the data better, likely due to the less significant limitations placed on the gating units. In the unstructured model, all entries of z_t are free to change independently, whereas in the block-sparse model, all entries in z_t of a particular block are constrained to have the same value. As with the block-sparse results, we observe that as s decreases, the potential speedups increase. However, the unstructured parameterization appears to result in models that significantly and unstably overfit when $s \geq -0.50$.

VII. Conclusion

With the block-sparse as well as the unstructured parameterizations, speedups of around 1.8x are possible, but require trading off accuracy compared to slower models. As z_t becomes more sparse as s becomes more negative, the hidden states are forced to pass through their previous activations instead of being allowed to produce new ones. This results in a model that can not react to rapid changes as well as a model with less sparse z_t gatings. Therefore, care must be taken when setting the s hyperparameter, as it is likely highly dependent on the target dataset. This problem is especially pronounced with the block-sparse model, where the z_t gatings are required to take on the same value for each particular block and cannot

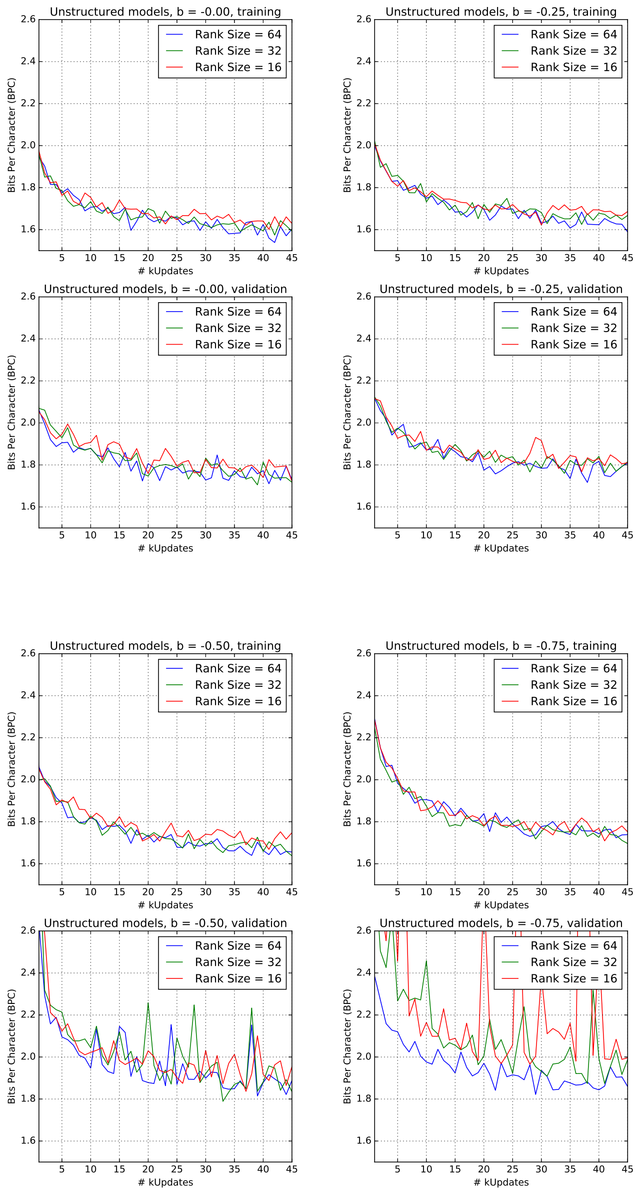

Fig. 5. A plot of the unstructured sparse model training and validation performances as measured in BPC as the models train.

gate individual h_{t-1} activations. In order to be more reduce the strains of this limitation, the block-sparse model could be modified to allow for individual gatings for the elements of z_t that are non-zero. Such a solution would allow for the block-sparsity that enables batch-wise conditional computation to be accelerated, while adding the individual gating behavior that gives unstructured sparsity an edge in BPC performance.

References

[1] Y. Bengio, "Deep learning of representations: Looking forward," in *International Conference on Statistical Language and Speech Processing*. Springer, 2013, pp. 1–37.

[2] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv preprint arXiv:1207.0580*, 2012.

[3] A. Makhzani and B. Frey, "k-sparse autoencoders," *International Conference on Learning Representations*, 2014.

[4] N. Léonard, "Distributed conditional computation," 2015.

[5] E. Bengio, P.-L. Bacon, J. Pineau, and D. Precup, "Conditional computation in neural networks for faster models," *arXiv preprint arXiv:1511.06297*, 2015.

[6] Y. Ioannou, D. Robertson, D. Zikic, P. Kontschieder, J. Shotton, M. Brown, and A. Criminisi, "Decision forests, convolutional networks and the models in-between," *arXiv preprint arXiv:1603.01250*, 2016.

[7] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[8] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[9] W. Zaremba, I. Sutskever, and O. Vinyals, "Recurrent neural network regularization," *arXiv preprint arXiv:1409.2329*, 2014.

[10] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of The 32nd International Conference on Machine Learning*, 2015, pp. 448–456.