Deriving Neural Architectures from Sequence and Graph Kernels

Tao Lei* [1] **Wengong Jin*** [1] **Regina Barzilay** [1] **Tommi Jaakkola** [1]

Abstract

The design of neural architectures for structured objects is typically guided by experimental insights rather than a formal process. In this work, we appeal to kernels over combinatorial structures, such as sequences and graphs, to derive appropriate neural operations. We introduce a class of deep recurrent neural operations and formally characterize their associated kernel spaces. Our recurrent modules compare the input to virtual reference objects (cf. filters in CNN) via the kernels. Similar to traditional neural operations, these reference objects are parameterized and directly optimized in end-to-end training. We empirically evaluate the proposed class of neural architectures on standard applications such as language modeling and molecular graph regression, achieving state-of-the-art results across these applications.

1. Introduction

Many recent studies focus on designing novel neural architectures for structured data such as sequences or annotated graphs. For instance, LSTM (Hochreiter & Schmidhuber, 1997), GRU (Chung et al., 2014) and other complex recurrent units (Zoph & Le, 2016) can be easily adapted to embed structured objects such as sentences (Tai et al., 2015) or molecules (Li et al., 2015; Dai et al., 2016) into vector spaces suitable for later processing by standard predictive methods. The embedding algorithms are typically integrated into an end-to-end trainable architecture so as to tailor the learnable embeddings directly to the task at hand.

The embedding process itself is characterized by a sequence operations summarized in a structure known as the computational graph. Each node in the computational graph identifies the unit/mapping applied while the arcs specify the relative arrangement/order of operations. The process of designing such computational graphs or associated operations for classes of objects is often guided by insights and expertise rather than a formal process.

Recent work has substantially narrowed the gap between desirable computational operations associated with objects and how their representations are acquired. For example, value iteration calculations can be folded into convolutional architectures so as to optimize the representations to facilitate planning (Tamar et al., 2016). Similarly, inference calculations in graphical models about latent states of variables such as atom characteristics can be directly associated with embedding operations (Dai et al., 2016).

We appeal to kernels over combinatorial structures to define the appropriate computational operations. Kernels give rise to well-defined function spaces and possess rules of composition that guide how they can be built from simpler ones. The comparison of objects inherent in kernels is often broken down to elementary relations such as counting of common sub-structures as in

$$\mathcal{K}(\chi, \chi') = \sum_{s \in \mathcal{S}} \mathbf{1}[s \in \chi]\mathbf{1}[s \in \chi'] \quad (1)$$

where $\mathcal{S}$ is the set of possible substructures. For example, in a string kernel (Lodhi et al., 2002), $\mathcal{S}$ may refer to all possible subsequences while a graph kernel (Vishwanathan et al., 2010) would deal with possible paths in the graph. Several studies have highlighted the relation between feed-forward neural architectures and kernels (Hazan & Jaakkola, 2015; Zhang et al., 2016) but we are unaware of any prior work pertaining to kernels associated with neural architectures for structured objects.

In this paper, we introduce a class of deep recurrent neural embedding operations and formally characterize their associated kernel spaces. The resulting kernels are parameterized in the sense that the neural operations relate objects of interest to virtual reference objects through kernels. These reference objects are parameterized and readily optimized for end-to-end performance.

To summarize, the proposed neural architectures, or *Kernel Neural Networks* [1] , enjoy the following advantages:

*Equal contribution [1]MIT Computer Science & Artificial Intelligence Laboratory. Correspondence to: Tao Lei <taolei@csail.mit.edu>, Wengong Jin <wengong@csail.mit.edu>.

Proceedings of the 34^{th} International Conference on Machine Learning, Sydney, Australia, PMLR 70, 2017.

[1]Code available at https://github.com/taolei87/icml17_knn

- The architecture design is grounded in kernel computations.
- Our neural models remain end-to-end trainable to the task at hand.
- Resulting architectures demonstrate state-of-the-art performance against strong baselines.

In the following sections, we will introduce these neural components derived from string and graph kernels, as well as their deep versions. Due to space limitations, we defer proofs to supplementary material.[2]

2. From String Kernels to Sequence NNs

Notations We define a sequence (or a string) of tokens (e.g. a sentence) as $\mathbf{x}_{1:L} = \{\mathbf{x}_i\}_{i=1}^{L}$ where $\mathbf{x}_i \in \mathbb{R}^d$ represents its i^{th} element and $|\mathbf{x}| = L$ denotes the length. Whenever it is clear from the context, we will omit the subscript and directly use $\mathbf{x}$ (and $\mathbf{y}$) to denote a sequence. For a pair of vectors (or matrices) $\mathbf{u}, \mathbf{v}$, we denote $\langle \mathbf{u}, \mathbf{v} \rangle = \sum_k u_k v_k$ as their inner product. For a kernel function $\mathcal{K}_i(\cdot, \cdot)$ with subscript i, we use $\phi_i(\cdot)$ to denote its underlying mapping, i.e. $\mathcal{K}_i(\mathbf{x}, \mathbf{y}) = \langle \phi_i(\mathbf{x}), \phi_i(\mathbf{y}) \rangle = \phi_i(\mathbf{x})^\top \phi_i(\mathbf{y})$.

String Kernel String kernel measures the similarity between two sequences by counting shared subsequences (see Lodhi et al. (2002)). For example, let $\mathbf{x}$ and $\mathbf{y}$ be two strings, a bi-gram string kernel $\mathcal{K}_2(\mathbf{x}, \mathbf{y})$ counts the number of bi-grams $(\mathbf{x}_i, \mathbf{x}_j)$ and $(\mathbf{y}_k, \mathbf{y}_l)$ such that $(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{y}_k, \mathbf{y}_l)$[3],

$$\mathcal{K}_2(\mathbf{x}, \mathbf{y}) = \sum_{\substack{1 \le i < j \le |\mathbf{x}| \\ 1 \le k < l \le |\mathbf{y}|}} \lambda_{\mathbf{x},i,j}\, \lambda_{\mathbf{y},k,l}\, \delta(\mathbf{x}_i, \mathbf{y}_k) \cdot \delta(\mathbf{x}_j, \mathbf{y}_l) \tag{2}$$

where $\lambda_{\mathbf{x},i,j}, \lambda_{\mathbf{y},k,l} \in [0, 1)$ are context-dependent weights and $\delta(x, y)$ is an indicator that returns 1 only when $x = y$. The weight factors can be realized in various ways. For instance, in temporal predictions such as language modeling, substrings (i.e. patterns) which appear later may have higher impact for prediction. Thus a realization $\lambda_{\mathbf{x},i,j} = \lambda^{|\mathbf{x}|-i-1}$ and $\lambda_{\mathbf{y},k,l} = \lambda^{|\mathbf{y}|-k-1}$ (penalizing substrings far from the end) can be used to determine weights given a constant decay factor $\lambda \in (0, 1)$.

In our case, each token in the sequence is a vector (such as one-hot encoding of a word or a feature vector). We shall replace the exact match $\delta(\mathbf{u}, \mathbf{v})$ by the inner product $\langle \mathbf{u}, \mathbf{v} \rangle$. To this end, the kernel function (2) can be rewritten as,

$$\sum_{1 \le i < j \le |\mathbf{x}|} \sum_{1 \le k < l \le |\mathbf{y}|} \lambda_{\mathbf{x},i,j}\, \lambda_{\mathbf{y},k,l}\, \langle \mathbf{x}_i, \mathbf{y}_k \rangle \cdot \langle \mathbf{x}_j, \mathbf{y}_l \rangle$$

[2] An extended version with supplementary material is available at https://arxiv.org/abs/1705.09037

[3] We define n-gram as a **subsequence** of original string (not necessarily consecutive).

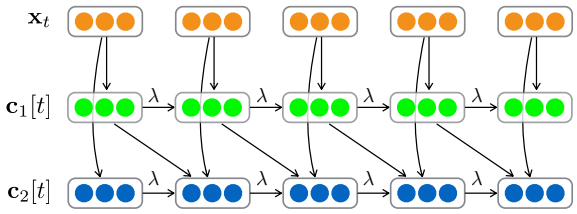

Figure 1. An unrolled view of the derived recurrent module for $\mathcal{K}_2()$. Horizontal lines denote decayed propagation from $\mathbf{c}[t-1]$ to $\mathbf{c}[t]$, while vertical lines represent a linear mapping $\mathbf{W}\mathbf{x}_t$ that is propagated to the internal states $\mathbf{c}[t]$.

$$= \sum_{1 \le i < j \le |\mathbf{x}|} \sum_{1 \le k < l \le |\mathbf{y}|} \lambda_{\mathbf{x},i,j}\, \lambda_{\mathbf{y},k,l}\, \langle \mathbf{x}_i \otimes \mathbf{x}_j, \mathbf{y}_k \otimes \mathbf{y}_l \rangle$$
$$= \left\langle \sum_{i<j} \lambda^{|\mathbf{x}|-i-1}\, \mathbf{x}_i \otimes \mathbf{x}_j,\ \sum_{k<l} \lambda^{|\mathbf{y}|-k-1}\, \mathbf{y}_k \otimes \mathbf{y}_l \right\rangle \tag{3}$$

where $\mathbf{x}_i \otimes \mathbf{x}_j \in \mathbb{R}^{d \times d}$ (and similarly $\mathbf{y}_k \otimes \mathbf{y}_l$) is the outer-product. In other words, the underlying mapping of kernel $\mathcal{K}_2()$ defined above is $\phi_2(\mathbf{x}) = \sum_{1 \le i < j \le |\mathbf{x}|} \lambda^{|\mathbf{x}|-i-1} \mathbf{x}_i \otimes \mathbf{x}_j$. Note we could alternatively use a partial additive scoring $\langle \mathbf{x}_i, \mathbf{y}_k \rangle + \langle \mathbf{x}_j, \mathbf{y}_l \rangle$, and the kernel function can be generalized to n-grams when $n \neq 2$. Again, we commit to one realization in this section.

String Kernel NNs We introduce a class of recurrent modules whose internal feature states embed the computation of string kernels. The modules *project* kernel mapping $\phi(\mathbf{x})$ into multi-dimensional vector space (i.e. internal states of recurrent nets). Owing to the combinatorial structure of $\phi(\mathbf{x})$, such projection can be realized and factorized via efficient computation. For the example kernel discussed above, the corresponding neural component is realized as,

$$\begin{aligned} \mathbf{c}_1[t] &= \lambda \cdot \mathbf{c}_1[t-1] + \left(\mathbf{W}^{(1)} \mathbf{x}_t\right) \\ \mathbf{c}_j[t] &= \lambda \cdot \mathbf{c}_j[t-1] + \left(\mathbf{c}_{j-1}[t-1] \odot \mathbf{W}^{(j)} \mathbf{x}_t\right) \\ \mathbf{h}[t] &= \sigma(\mathbf{c}_n[t]), \qquad\qquad 1 < j \le n \end{aligned} \tag{4}$$

where $\mathbf{c}_j[t]$ are the pre-activation cell states at word $\mathbf{x}_t$, and $\mathbf{h}[t]$ is the (post-activation) hidden vector. $\mathbf{c}_j[0]$ is initialized with a zero vector. $\mathbf{W}^{(1)}, .., \mathbf{W}^{(n)}$ are weight matrices to be learned from training examples.

The network operates like other RNNs by processing each input token and updating the internal states. The element-wise multiplication $\odot$ can be replaced by addition $+$ (corresponding to the partial additive scoring above). As a special case, the additive variant becomes a word-level convolutional neural net (Kim, 2014) when $\lambda = 0$.[4]

[4] $\mathbf{h}[t] = \sigma(\mathbf{W}^{(1)} \mathbf{x}_{t-n+1} + \cdots + \mathbf{W}^{(n)} \mathbf{x}_t)$ when $\lambda = 0$.

2.1. Single Layer as Kernel Computation

Now we state how the proposed class embeds string kernel computation. For $j \in \{1, .., n\}$, let $c_j[t][i]$ be the i-th entry of state vector $\mathbf{c}_j[t]$, $\mathbf{w}_i^{(j)}$ represents the i-th row of matrix $\mathbf{W}^{(j)}$. Define $\mathbf{w}_{i,j} = \{\mathbf{w}_i^{(1)}, \mathbf{w}_i^{(2)}, ..., \mathbf{w}_i^{(j)}\}$ as a "reference sequence" constructed by taking the i-th row from each matrix $\mathbf{W}^{(1)}, .., \mathbf{W}^{(j)}$.

Theorem 1. *Let $\mathbf{x}_{1:t}$ be the prefix of $\mathbf{x}$ consisting of first t tokens, and $\mathcal{K}_j$ be the string kernel of j-gram shown in Eq.(3). Then $\mathbf{c}_j[t][i]$ evaluates kernel function,*

$$\mathbf{c}_j[t][i] \;=\; \mathcal{K}_j\left(\mathbf{x}_{1:t}, \mathbf{w}_{i,j}\right) = \langle \phi_j(\mathbf{x}_{1:t}), \phi_j(\mathbf{w}_{i,j}) \rangle$$

for any $j \in \{1, .., n\}$, $t \in \{1, .., |\mathbf{x}|\}$.

In other words, the network *embeds sequence similarity computation* by assessing the similarity between the input sequence $\mathbf{x}_{1:t}$ and the reference sequence $\mathbf{w}_{i,j}$. This interpretation is similar to that of CNNs, where each filter is a "reference pattern" to search in the input. String kernel NN further takes non-consecutive n-gram patterns into consideration (seen from the summation over all n-grams in Eq.(3)).

Applying Non-linear Activation In practice, a non-linear activation function such as polynomial or sigmoid-like activation is added to the internal states to produce the final output state $\mathbf{h}[t]$. It turns out that many activations are also functions in the reproducing kernel Hilbert space (RKHS) of certain kernel functions (see Shalev-Shwartz et al. (2011); Zhang et al. (2016)). When this is true, the underlying kernel of $\mathbf{h}[t]$ is the composition of string kernel and the kernel containing the activation. We give the formal statements below.

Lemma 1. *Let $\mathbf{x}$ and $\mathbf{w}$ be multi-dimensional vectors with finite norm. Consider the function $f(\mathbf{x}) := \sigma(\mathbf{w}^\top \mathbf{x})$ with non-linear activation $\sigma(\cdot)$. For functions such as polynomials and sigmoid function, there exists kernel functions $\mathcal{K}_\sigma(\cdot, \cdot)$ and the underlying mapping $\phi_\sigma(\cdot)$ such that $f(x)$ is in the reproducing kernel Hilbert space of $\mathcal{K}_\sigma(\cdot, \cdot)$, i.e.,*

$$f(\mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x}) = \langle \phi_\sigma(\mathbf{x}), \psi(\mathbf{w}) \rangle$$

for some mapping $\psi(\mathbf{w})$ constructed from $\mathbf{w}$. In particular, $\mathcal{K}_\sigma(\mathbf{x}, \mathbf{y})$ can be the inverse-polynomial kernel $\frac{1}{2-\langle \mathbf{x}, \mathbf{y} \rangle}$ for the above activations.

Proposition 1. *For one layer string kernel NN with non-linear activation $\sigma(\cdot)$ discussed in Lemma 1, $\mathbf{h}[t][i]$ as a function of input $\mathbf{x}$ belongs to the RKHS introduced by the composition of $\mathcal{K}_\sigma(\cdot, \cdot)$ and string kernel $\mathcal{K}_n(\cdot, \cdot)$. Here a kernel composition $\mathcal{K}_{\sigma,n}(\mathbf{x}, \mathbf{y})$ is defined with the underlying mapping $\mathbf{x} \mapsto \phi_\sigma(\phi_n(\mathbf{x}))$, and hence $\mathcal{K}_{\sigma,n}(\mathbf{x}, \mathbf{y}) = \phi_\sigma(\phi_n(\mathbf{x})) \top \phi_\sigma(\phi_n(\mathbf{y}))$.*

Proposition 1 is the corollary of Lemma 1 and Theorem 1, since $\mathbf{h}[t][i] = \sigma(\mathbf{c}_n[t][i]) = \sigma(\mathcal{K}_n(\mathbf{x}_{1:t}, \mathbf{w}_{i,j})) = \langle \phi_\sigma(\phi_n(\mathbf{x}_{1:t})), \tilde{\mathbf{w}}_{i,j} \rangle$ and $\phi_\sigma(\phi_n(\cdot))$ is the mapping for the composed kernel. The same proof applies when $\mathbf{h}[t]$ is a linear combination of all $\mathbf{c}_i[t]$ since kernel functions are closed under addition.

2.2. Deep Networks as Deep Kernel Construction

We now address the case when multiple layers of the same module are stacked to construct deeper networks. That is, the output states $\mathbf{h}^{(l)}[t]$ of the l-th layer are fed to the $(l+1)$-th layer as the input sequence. We show that layer stacking corresponds to recursive kernel construction (i.e. $(l+1)$-th kernel is defined on top of l-th kernel), which has been proven for feed-forward networks (Zhang et al., 2016).

We first generalize the sequence kernel definition to enable recursive construction. Notice that the definition in Eq.(3) uses the linear kernel (inner product) $\langle \mathbf{x}_i, \mathbf{y}_k \rangle$ as a "subroutine" to measure the similarity between substructures (e.g. tokens) within the sequences. We can therefore replace it with other similarity measures introduced by other "base kernels". In particular, let $\mathcal{K}^{(1)}(\cdot, \cdot)$ be the string kernel (associated with a single layer). The generalized sequence kernel $\mathcal{K}^{(l+1)}(\mathbf{x}, \mathbf{y})$ can be recursively defined as,

$$\sum_{\substack{i<j \\ k<l}} \lambda_{\mathbf{x},i,j}\; \lambda_{\mathbf{y},k,l}\; \mathcal{K}_\sigma^{(l)}(\mathbf{x}_{1:i}, \mathbf{y}_{1:k})\; \mathcal{K}_\sigma^{(l)}(\mathbf{x}_{1:j}, \mathbf{y}_{1:l}) \;\; =$$

$$\left\langle \sum_{i<j} \phi_\sigma^{(l)}(\mathbf{x}_{1:i}) \otimes \phi_\sigma^{(l)}(\mathbf{x}_{1:j}), \sum_{k<l} \phi_\sigma^{(l)}(\mathbf{y}_{1:k}) \otimes \phi_\sigma^{(l)}(\mathbf{y}_{1:l}) \right\rangle$$

where $\phi^{(l)}(\cdot)$ denotes the pre-activation mapping of the l-th kernel, $\phi_\sigma^{(l)}(\cdot) = \phi_\sigma(\phi^{(l)}(\cdot))$ denotes the underlying (post-activation) mapping for non-linear activation $\sigma(\cdot)$, and $\mathcal{K}_\sigma^{(l)}(\cdot, \cdot)$ is the l-th post-activation kernel. Based on this definition, a deeper model can also be interpreted as a kernel computation.

Theorem 2. *Consider a deep string kernel NN with L layers and activation function $\sigma(\cdot)$. Let the final output state $\mathbf{h}^{(L)}[t] = \sigma(\mathbf{c}_n^{(L)}[t])$ (or any linear combination of $\{\mathbf{c}_i^{(l)}[t]\}, i = 1, .., n$). For $l = 1, \cdots, L$,*

(i) $\mathbf{c}_n^{(l)}[t][i]$ as a function of input $\mathbf{x}$ belongs to the RKHS of kernel $\mathcal{K}^{(l)}(\cdot, \cdot)$;

(ii) $\mathbf{h}^{(l)}[t][i]$ belongs to the RKHS of kernel $\mathcal{K}_\sigma^{(l)}(\cdot, \cdot)$.

3. From Graph Kernels to Graph NNs

In the previous section, we encode sequence kernel computation into neural modules and demonstrate possible extensions using different base kernels. The same ideas apply

to other types of kernels and data. Specifically, we derive neural components for graphs in this section.

Notations A graph is defined as $G = (V, E)$, with each vertex $v \in V$ associated with feature vector $\mathbf{f}_v$. The neighbor of node v is denoted as $N(v)$. Following previous notations, for any kernel function $\mathcal{K}_*(\cdot,\cdot)$ with underlying mapping $\phi_*(\cdot)$, we use $\mathcal{K}_{*,\sigma}(\cdot,\cdot)$ to denote the post-activation kernel induced from the composed underlying mapping $\phi_{*,\sigma} = \phi_\sigma(\phi_*(\cdot))$.

3.1. Random Walk Kernel NNs

We start from random walk graph kernels (Gärtner et al., 2003), which count common walks in two graphs. Formally, let $P_n(G)$ be the set of walks $\mathbf{x} = x_1 \cdots x_n$, where $\forall i : (x_i, x_{i+1}) \in E$.[5] Given two graphs G and G', an n-th order random walk graph kernel is defined as:

$$\mathcal{K}^n_W(G, G') = \lambda^{n-1} \sum_{\mathbf{x} \in P_n(G)} \sum_{\mathbf{y} \in P_n(G')} \prod_{i=1}^{n} \langle \mathbf{f}_{\mathbf{x}_i}, \mathbf{f}_{\mathbf{y}_i} \rangle \quad (5)$$

where $\mathbf{f}_{x_i} \in \mathbb{R}^d$ is the feature vector of node $\mathbf{x}_i$ in the walk.

Now we show how to realize the above graph kernel with a neural module. Given a graph G, the proposed neural module is:

$$\begin{aligned} \mathbf{c}_1[v] &= \mathbf{W}^{(1)} \mathbf{f}_v \\ \mathbf{c}_j[v] &= \lambda \sum_{u \in N(v)} c_{j-1}[u] \odot \mathbf{W}^{(j)} \mathbf{f}_v \qquad (6) \\ \mathbf{h}_G &= \sigma(\sum_v \mathbf{c}_n[v]) \qquad 1 < j \le n \end{aligned}$$

where again $\mathbf{c}_*[v]$ is the cell state vector of node v, and $\mathbf{h}_G$ is the representation of graph G aggregated from node vectors. $\mathbf{h}_G$ could then be used for classification or regression.

Now we show the proposed model embeds the random walk kernel. To show this, construct $L_{n,k}$ as a "reference walk" consisting of the row vectors $\{\mathbf{w}^{(1)}_k, \cdots, \mathbf{w}^{(n)}_k\}$ from the parameter matrices. Here $L_{n,k} = (L_V, L_E)$, where $L_V = \{v_0, v_1, \cdots, v_n\}$, $L_E = \{(v_i, v_{i+1})\}$ and v_i's feature vector is $\mathbf{w}^{(i)}_k$. We have the following theorem:

Theorem 3. *For any $n \ge 1$, the state value $c_n[v][k]$ (the k-th coordinate of $c_n[v]$) satisfies:*

$$\sum_v \mathbf{c}_n[v][k] = \mathcal{K}^n_W(G, L_{n,k})$$

thus $\sum_v \mathbf{c}_n[v]$ lies in the RKHS of kernel $\mathcal{K}^n_W$. As a corollary, $\mathbf{h}_G$ lies in the RKHS of kernel $K^n_{W,\sigma}()$.

[5] A single node could appear multiple times in a walk.

3.2. Unified View of Graph Kernels

The derivation of the above neural module could be extended to other classes of graph kernels, such as subtree kernels (cf. (Ramon & Gärtner, 2003; Vishwanathan et al., 2010)). Generally speaking, most of these kernel functions factorize graphs into local sub-structures, i.e.

$$\mathcal{K}(G, G') = \sum_v \sum_{v'} \mathcal{K}_{loc}(v, v') \quad (7)$$

where $\mathcal{K}_{loc}(v, v')$ measures the similarity between local sub-structures centered at node v and v'.

For example, the random walk kernel K^n_W can be equivalently defined with $\mathcal{K}^n_{loc}(v, v') =$

$$\begin{cases} \langle \mathbf{f}_v, \mathbf{f}_{v'} \rangle & \text{if } n = 1 \\ \langle \mathbf{f}_v, \mathbf{f}_{v'} \rangle \cdot \lambda \sum_{u \in N(v)} \sum_{u' \in N(v')} \mathcal{K}^{n-1}_{loc}(u, u') & \text{if } n > 1 \end{cases}$$

Other kernels like subtree kernels could be recursively defined similarly. Therefore, we adopt this unified view of graph kernels for the rest of this paper.

In addition, this definition of random walk kernel could be further generalized and enhanced by aggregating neighbor features non-linearly:

$$\mathcal{K}^n_{loc}(v, v') = \langle \mathbf{f}_v, \mathbf{f}_{v'} \rangle \circ \lambda \sum_{u \in N(v)} \sum_{u' \in N(v')} \mathcal{K}^{n-1}_{loc,\sigma}(u, u')$$

where $\circ$ could be either multiplication or addition. $\sigma(\cdot)$ denotes a non-linear activation and $\mathcal{K}^{n-1}_{loc,\sigma}(\cdot,\cdot)$ denotes the post-activation kernel when $\sigma(\cdot)$ is involved. The generalized kernel could be realized by modifying Eq.(6) into:

$$\mathbf{c}_j[v] = \mathbf{W}^{(j)} \mathbf{f}_v \circ \lambda \sum_{u \in N(v)} \sigma(c_{j-1}[u]) \quad (8)$$

where $\circ$ could be either $+$ or $\odot$ operation.

3.3. Deep Graph Kernels and NNs

Following Section 2, we could stack multiple graph kernel NNs to form a deep network. That is:

$$\begin{aligned} \mathbf{c}^{(l)}_1[v] &= \mathbf{W}^{(l,1)} \mathbf{h}^{(l-1)}[v] \\ \mathbf{c}^{(l)}_j[v] &= \mathbf{W}^{(l,j)} \mathbf{h}^{(l-1)}[v] \circ \lambda \sum_{u \in N(v)} \sigma\left(\mathbf{c}^{(l)}_{j-1}[u]\right) \\ \mathbf{h}^{(l)}[v] &= \sigma(\mathbf{U}^{(l)} \mathbf{c}^{(l)}_n[v]) \qquad 1 \le l \le L, 1 < j \le n \end{aligned}$$

The local kernel function is recursively defined in two dimensions: depth (term $\mathbf{h}^{(l)}$) and width (term $\mathbf{c}_j$). Let the pre-activation kernel in the l-th layer be $\mathcal{K}^{(l)}_{loc}(v, v') = \mathcal{K}^{(l,n)}_{loc}(v, v')$, and the post-activation kernel be $\mathcal{K}^{(l)}_{loc,\sigma}(v, v') = \mathcal{K}^{(l,n)}_{loc,\sigma}(v, v')$. We recursively define

$$\mathcal{K}_{loc}^{(l,j)}(v,v') = \begin{cases} \mathcal{K}_{loc,\sigma}^{(l-1)}(v,v') & \text{if } j=1 \\ \mathcal{K}_{loc,\sigma}^{(l-1)}(v,v') \circ \lambda \sum_{u\in N(v)} \sum_{u'\in N(v')} \mathcal{K}_{loc,\sigma}^{(l,j-1)}(u,u') & \text{if } j>1 \end{cases}$$

for $j = 1, \cdots, n$. Finally, the graph kernel is $\mathcal{K}^{(L,n)}(G,G') = \sum_{v,v'} \mathcal{K}_{loc}^{(L,n)}(v,v')$. Similar to Theorem 2, we have

Theorem 4. *Consider a deep graph kernel NN with L layers and activation function $\sigma(\cdot)$. Let the final output state $\mathbf{h}_G = \sum_v \mathbf{h}^{(L)}[v]$. For $l = 1, \cdots, L; j = 1, \cdots, n$:*

(i) $\mathbf{c}_j^{(l)}[v][i]$ *as a function of input v and graph G belongs to the RKHS of kernel $\mathcal{K}_{loc}^{(l,j)}(\cdot,\cdot)$;*

(ii) $\mathbf{h}^{(l)}[v][i]$ *belongs to the RKHS of kernel $\mathcal{K}_{loc,\sigma}^{(l,n)}(\cdot,\cdot)$.*

(iii) $\mathbf{h}_G[i]$ *belongs to the RKHS of kernel $\mathcal{K}^{(L,n)}(\cdot,\cdot)$.*

3.4. Connection to Weisfeiler-Lehman Kernel

We derived the above deep kernel NN for the purpose of generality. This model could be simplified by setting $n = 2$, without losing representational power (as non-linearity is already involved in depth dimension). In this case, we rewrite the network by reparametrization:

$$\mathbf{h}_v^{(l)} = \sigma\left(\mathbf{U}_1^{(l)}\mathbf{h}_v^{(l-1)} \circ \mathbf{U}_2^{(l)} \sum_{u\in N(v)} \sigma\left(\mathbf{V}^{(l)}\mathbf{h}_u^{(l-1)}\right)\right) \tag{9}$$

In this section, we further show that this model could be enhanced by sharing weight matrices $\mathbf{U}$ and $\mathbf{V}$ across layers. This parameter tying mechanism allows our model to embed Weisfeiler-Lehman kernel (Shervashidze et al., 2011). For clarity, we briefly review basic concepts of Weisfeiler-Lehman kernel below.

Weisfeiler-Lehman Graph Relabeling Weisfeiler-Lehman kernel borrows concepts from the Weisfeiler-Lehman isomorphism test for labeled graphs. The key idea of the algorithm is to augment the node labels by the sorted set of node labels of neighbor nodes, and compress these augmented labels into new, short labels (Figure 2). Such relabeling process is repeated T times. In the i-th iteration, it generates a new labeling $l_i(v)$ for all nodes v in graph G, with initial labeling l_0.

Generalized Graph Relabeling The key observation here is that graph relabeling operation could be viewed as neighbor feature aggregation. As a result, the relabeling process naturally generalizes to the case where nodes are associated with continuous feature vectors. In particular, let r be the relabeling function. For a node $v \in G$:

$$r(v) = \sigma(\mathbf{U}_1\mathbf{f}_v + \mathbf{U}_2 \sum_{u\in N(v)} \sigma(\mathbf{V}\mathbf{f}_u)) \tag{10}$$

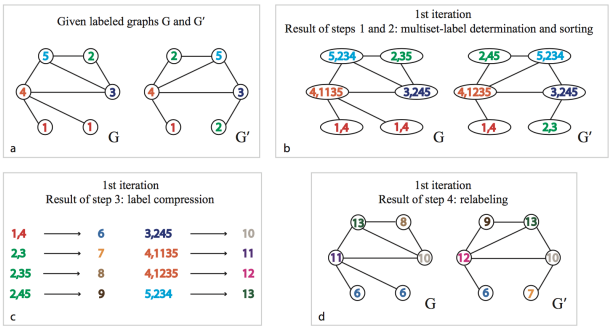

Figure 2. Node relabeling in Weisfeiler-Lehman isomorphism test. Figure taken from Shervashidze et al. (2011)

Note that our definition of $r(v)$ is exactly the same as $\mathbf{h}_v$ in Equation 9, with $\circ$ being additive composition.

Weisfeiler-Lehman Kernel Let $\mathcal{K}$ be any graph kernel (called base kernel). Given a relabeling function r, Weisfeiler-Lehman kernel with base kernel $\mathcal{K}$ and depth L is defined as

$$\mathcal{K}_{WL}^{(L)}(G,G') = \sum_{i=0}^{L} \mathcal{K}(r^i(G), r^i(G')) \tag{11}$$

where $r^0(G) = G$ and $r^i(G), r^i(G')$ are the i-th relabeled graph of G and G' respectively.

Weisfeiler-Lehman Kernel NN Now with the above kernel definition, and random walk kernel as the base kernel, we propose the following recurrent module:

$$\begin{aligned} \mathbf{c}_0^{(l)}[v] &= \mathbf{W}^{(l,0)}\mathbf{h}_v^{(l-1)} \\ \mathbf{c}_j^{(l)}[v] &= \lambda \sum_{u\in N(v)} c_{j-1}^{(l)}[u] \odot \mathbf{W}^{(l,j)}\mathbf{h}_v^{(l-1)} \\ \mathbf{h}_v^{(l)} &= \sigma\left(\mathbf{U}_1\mathbf{h}_v^{(l-1)} + \mathbf{U}_2 \sum_{u\in N(v)} \sigma(\mathbf{V}\mathbf{h}_u^{(l-1)})\right) \\ \mathbf{h}_G^{(l)} &= \sum_v \mathbf{c}_n^{(l)}[v] \qquad 1 \le l \le L, 1 < j \le n \end{aligned}$$

where $\mathbf{h}_v^{(0)} = \mathbf{f}_v$ and $\mathbf{U}_1, \mathbf{U}_2, \mathbf{V}$ are shared across layers. The final output of this network is $\mathbf{h}_G = \sum_{l=1}^{L} \mathbf{h}_G^{(l)}$.

The above recurrent module is still an instance of deep kernel, even though some parameters are shared. A minor difference here is that there is an additional random walk kernel NN that connects i-th layer and the output layer. But this is just a linear combination of L deep random walk kernels (of different depth). Therefore, as an corollary of Theorem 4, we have:

Proposition 2. *For a Weisfeiler-Lehman Kernel NN with L iterations and random walk kernel $\mathcal{K}_W^n$ as base kernel, the final output state $\mathbf{h}_G = \sum_l \mathbf{h}_G^{(l)}$ belongs to the RKHS of kernel $\mathcal{K}_{WL}^{(L)}(\cdot,\cdot)$.*

4. Adaptive Decay with Neural Gates

The sequence and graph kernel (and their neural components) discussed so far use a constant decay value λ regardless of the current input. However, this is often not the case since the importance of the input can vary across the context or the applications. One extension is to make use of neural gates that adaptively control the decay factor. Here we give two illustrative examples:

Gated String Kernel NN By replacing constant decay λ with a sigmoid gate, we modify our single-layer sequence module as:

$$\begin{aligned}
\lambda_t &= \sigma(\mathbf{U}[\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}) \\
\mathbf{c}_1[t] &= \lambda_t \odot \mathbf{c}_1[t-1] + \left(\mathbf{W}^{(1)}\mathbf{x}_t\right) \\
\mathbf{c}_j[t] &= \lambda_t \odot \mathbf{c}_j[t-1] + \left(\mathbf{c}_{j-1}[t-1] \odot \mathbf{W}^{(j)}\mathbf{x}_t\right) \\
\mathbf{h}[t] &= \sigma(\mathbf{c}_n[t]) \qquad\qquad 1 < j \leq n
\end{aligned}$$

As compared with the original string kernel, now the decay factor $\lambda_{\mathbf{x},i,j}$ is no longer $\lambda^{|\mathbf{x}|-i-1}$, but rather an adaptive value based on current context.

Gated Random Walk Kernel NN Similarly, we could introduce gates so that different walks have different weights:

$$\begin{aligned}
\lambda_{u,v} &= \sigma(\mathbf{U}[\mathbf{f}_u, \mathbf{f}_v] + \mathbf{b}) \\
\mathbf{c}_0[v] &= \mathbf{W}^{(0)}\mathbf{f}_v \\
\mathbf{c}_j[v] &= \sum_{u \in N(v)} \lambda_{u,v} \odot c_{j-1}[u] \odot \mathbf{W}^{(j)}\mathbf{f}_v \\
\mathbf{h}_G &= \sigma(\sum_v \mathbf{c}_n[v]) \qquad 1 < j \leq n
\end{aligned}$$

The underlying kernel of the above gated network becomes

$$\mathcal{K}^n_W(G, G') = \sum_{\mathbf{x} \in P_n(G)} \sum_{\mathbf{y} \in P_n(G')} \prod_{i=1}^{n} \lambda_{\mathbf{x}_i, \mathbf{y}_i} \langle \mathbf{f}_{\mathbf{x}_i}, \mathbf{f}_{\mathbf{y}_i} \rangle$$

where each path is weighted by different decay weights, determined by network itself.

5. Related Work

Sequence Networks Considerable effort has gone into designing effective networks for sequence processing. This includes recurrent modules with the ability to carry persistent memories such as LSTM (Hochreiter & Schmidhuber, 1997) and GRU (Chung et al., 2014), as well as non-consecutive convolutional modules (RCNNs, Lei et al. (2015)), and others. More recently, Zoph & Le (2016) exemplified a reinforcement learning-based search algorithm to further optimize the design of such recurrent architectures. Our proposed neural networks offer similar state evolution and feature aggregation functionalities but derive the motivation for the operations involved from well-established kernel computations over sequences.

Recursive neural networks are alternative architectures to model hierarchical structures such as syntax trees and logic forms. For instance, Socher et al. (2013) employs recursive networks for sentence classification, where each node in the dependency tree of the sentence is transformed into a vector representation. Tai et al. (2015) further proposed tree-LSTM, which incorporates LSTM-style architectures as the transformation unit. Dyer et al. (2015; 2016) recently introduced a recursive neural model for transition-based language modeling and parsing. While not specifically discussed in the paper, our ideas do extend to similar neural components for hierarchical objects (e.g. trees).

Graph Networks Most of the current graph neural architectures perform either convolutional or recurrent operations on graphs. Duvenaud et al. (2015) developed Neural Fingerprint for chemical compounds, where each convolution operation is a sum of neighbor node features, followed by a linear transformation. Our model differs from theirs in that our generalized kernels and networks can aggregate neighboring features in a non-linear way. Other approaches, e.g., Bruna et al. (2013) and Henaff et al. (2015), rely on graph Laplacian or Fourier transform.

For recurrent architectures, Li et al. (2015) proposed gated graph neural networks, where neighbor features are aggregated by GRU function. Dai et al. (2016) considers a different architecture where a graph is viewed as a latent variable graphical model. Their recurrent model is derived from Belief Propagation-like algorithms. Our approach is most closely related to Dai et al. (2016), in terms of neighbor feature aggregation and resulting recurrent architecture. Nonetheless, the focus of this paper is on providing a framework for how such recurrent networks could be derived from deep graph kernels.

Kernels and Neural Nets Our work follows recent work demonstrating the connection between neural networks and kernels (Cho & Saul, 2009; Hazan & Jaakkola, 2015). For example, Zhang et al. (2016) showed that standard feedforward neural nets belong to a larger space of recursively constructed kernels (given certain activation functions). Similar results have been made for convolutional neural nets (Anselmi et al., 2015), and general computational graphs (Daniely et al., 2016). We extend prior work to kernels and neural architectures over structured inputs, in particular, sequences and graphs. Another difference is how we train the model. While some prior work appeals to convex optimization through improper learning (Zhang et al., 2016; Heinemann et al., 2016) (since kernel space is larger), we use the proposed networks as building blocks in typical non-convex but flexible neural network training.

6. Experiments

The left-over question is whether the proposed class of operations, despite its formal characteristics, leads to more effective architecture exploration and hence improved performance. In this section, we apply the proposed sequence and graph modules to various tasks and empirically evaluate their performance against other neural network models. These tasks include language modeling, sentiment classification and molecule regression.

6.1. Language Modeling on PTB

Dataset and Setup We use the Penn Tree Bank (PTB) corpus as the benchmark. The dataset contains about 1 million tokens in total. We use the standard train/development/test split of this dataset with vocabulary of size 10,000.

Model Configuration Following standard practice, we use SGD with an initial learning rate of 1.0 and decrease the learning rate by a constant factor after a certain epoch. We back-propagate the gradient with an unroll size of 35 and use dropout (Hinton et al., 2012) as the regularization. Unless otherwise specified, we train 3-layer networks with $n = 1$ and normalized adaptive decay.[6] Following (Zilly et al., 2016), we add highway connections (Srivastava et al., 2015) within each layer:

$$\mathbf{c}^{(l)}[t] = \lambda_t \odot \mathbf{c}^{(l)}[t-1] + (1-\lambda_t) \odot (\mathbf{W}^{(l)}\mathbf{h}^{(l-1)}[t])$$
$$\mathbf{h}^{(l)}[t] = \mathbf{f}_t \odot \mathbf{c}^{(l)}[t] + (1-\mathbf{f}_t) \odot \mathbf{h}^{(l-1)}[t]$$

where $\mathbf{h}^{(0)}[t] = \mathbf{x}_t$, λ_t is the gated decay factor and $\mathbf{f}_t$ is the transformation gate of highway connections.

Results Table 1 compares our model with various state-of-the-art models. Our small model with 5 million parameters achieves a test perplexity of 73.6, already outperforming many results achieved using much larger network. By increasing the network size to 20 million, we obtain a test perplexity of 69.2, with standard dropout. Adding variational dropout (Gal & Ghahramani, 2016) within the recurrent cells further improves the perplexity to 65.5. Finally, the model achieves 63.8 perplexity when the recurrence depth is increased to 4, being state-of-the-art and on par with the results reported in (Zilly et al., 2016; Zoph & Le, 2016). Note that Zilly et al. (2016) uses 10 neural layers and Zoph & Le (2016) adopts a complex recurrent cell found by reinforcement learning based search. Our network is architecturally much simpler.

Figure 3 analyzes several variants of our model. Word-level CNNs are degraded cases ($\lambda = 0$) that ignore non-contiguous n-gram patterns. Clearly, this variant performs worse compared to other recurrent variants with $\lambda > 0$.

[6] See the supplementary sections for a discussion of network variants: https://arxiv.org/abs/1705.09037

Table 1: Comparison with state-of-the-art results on PTB. $|\theta|$ denotes the number of parameters. Following recent work (Press & Wolf, 2016), we share the input and output word embedding matrix. We report the test perplexity (PPL) of each model. Lower number is better.

Model	$\|\theta\|$	**PPL**
LSTM (large) (Zaremba et al., 2014)	66m	78.4
Character CNN (Kim et al., 2015)	19m	78.9
Variational LSTM (Gal & Ghahramani)	20m	78.6
Variational LSTM (Gal & Ghahramani)	66m	73.4
Pointer Sentinel-LSTM (Merity et al.)	21m	70.9
Variational RHN (Zilly et al., 2016)	23m	65.4
Neural Net Search (Zoph & Le, 2016)	25m	64.0
Kernel NN ($\lambda = 0.8$)	5m	84.3
Kernel NN (λ learned as parameter)	5m	76.8
Kernel NN (gated λ)	5m	73.6
Kernel NN (gated λ)	20m	69.2
+ variational dropout	20m	65.5
+ variational dropout, 4 RNN layers	20m	**63.8**

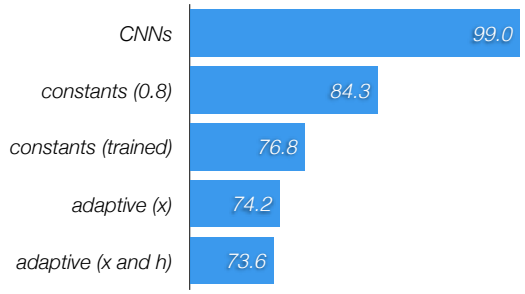

Figure 3. Comparison between kernel NN variants on PTB. $|\theta| =$ 5m for all models. Hyper-parameter search is performed for each variant.

Moreover, the test perplexity improves from 84.3 to 76.8 when we train the constant decay vector as part of the model parameters. Finally, the last two variants utilize neural gates (depending on input $\mathbf{x}_t$ only or on both input $\mathbf{x}_t$ and previous state $\mathbf{h}[t-1]$), improving the performance 2-3 points.

6.2. Sentiment Classification

Dataset and Setup We evaluate our model on the sentence classification task. We use the Stanford Sentiment Treebank benchmark (Socher et al., 2013). The dataset consists of 11855 parsed English sentences annotated at both the root (i.e. sentence) level and the phrase level using 5-class fine-grained labels. We use the standard split for training, development and testing. Following previous work, we also evaluate our model on the binary classification variant of this benchmark, ignoring all neutral sentences.

Table 2: Classification accuracy on Stanford Sentiment Treebank. Block I: recursive networks; Block II: convolutional or recurrent networks; Block III: other baseline methods. Higher number is better.

Model	Fine	Binary
RNN (Socher et al. (2011))	43.2	82.4
RNTN (Socher et al. (2013))	45.7	85.4
DRNN (Irsoy & Cardie (2014))	49.8	86.8
RLSTM (Tai et al. (2015))	51.0	88.0
DCNN (Kalchbrenner et al. (2014))	48.5	86.9
CNN-MC (Kim (2014))	47.4	88.1
Bi-LSTM (Tai et al. (2015))	49.1	87.5
LSTMN (Cheng et al. (2016))	47.9	87.0
PVEC (Le & Mikolov (2014))	48.7	87.8
DAN (Iyyer et al. (2014))	48.2	86.8
DMN (Kumar et al. (2016))	52.1	88.6
Kernel NN, $\lambda = 0.5$	51.2	88.6
Kernel NN, gated λ	**53.2**	**89.9**

Following the recent work of DAN (Iyyer et al., 2015) and RLSTM (Tai et al., 2015), we use the publicly available 300-dimensional GloVe word vectors (Pennington et al., 2014). Unlike prior work which fine tunes the word vectors, we normalize the vectors (i.e. $\|w\|_2^2 = 1$) and fixed them for simplicity.

Model Configuration Our best model is a 3-layer network with $n = 2$ and hidden dimension $d = 200$. We average the hidden states $h[t]$ across t, and concatenate the averaged vectors from the 3 layers as the input of the final softmax layer. The model is optimized with Adam (Kingma & Ba, 2015), and dropout probability of 0.35.

Results Table 2 presents the performance of our model and other networks. We report the best results achieved across 5 independent runs. Our best model obtains 53.2% and 89.9% test accuracies on fine-grained and binary tasks respectively. Our model with only a constant decay factor also obtains quite high accuracy, outperforming other baseline methods shown in the table.

6.3. Molecular Graph Regression

Dataset and Setup We further evaluate our graph NN models on the Harvard Clean Energy Project benchmark, which has been used in Dai et al. (2016); Duvenaud et al. (2015) as their evaluation dataset. This dataset contains 2.3 million candidate molecules, with each molecule labeled with its power conversion efficiency (PCE) value.

We follow exactly the same train-test split as Dai et al. (2016), and the same re-sampling procedure on the training data (but not the test data) to make the algorithm put more emphasis on molecules with higher PCE values, since the data is distributed unevenly.

Table 3: Experiments on Harvard Clean Energy Project. We report Root Mean Square Error(RMSE) on test set. The first block lists the results reported in Dai et al. (2016) for reference. For fair comparison, we reimplemented their best model so that all models are trained under the same setup. Results under our setup is reported in second block.

Model (Dai et al., 2016)	$\|\theta\|$	**RMSE**
Mean Predicator	1	2.4062
Weisfeiler-lehman Kernel, degree=3	1.6m	0.2040
Weisfeiler-lehman Kernel, degree=6	1378m	0.1367
Embedded Mean Field	0.1m	0.1250
Embedded Loopy BP	0.1m	0.1174
Under Our Setup		
Neural Fingerprint	0.26m	0.1409
Embedded Loopy BP	0.26m	0.1065
Weisfeiler Kernel NN	0.26m	0.1058
Weisfeiler Kernel NN, gated λ	0.26m	**0.1043**

We use the same feature set as in Duvenaud et al. (2015) for atoms and bonds. Initial atom features include the atoms element, its degree, the number of attached hydrogens, its implicit valence, and an aromaticity indicator. The bond feature is a concatenation of bond type indicator, whether the bond is conjugated, and whether the bond is in a ring.

Model Configuration Our model is a Weisfeiler-Lehman NN, with 4 recurrent iterations and $n = 2$. All models (including baseline) are optimized with Adam (Kingma & Ba, 2015), with learning rate decay factor 0.9.

Results In Table 3, we report the performance of our model against other baseline methods. Neural Fingerprint (Duvenaud et al., 2015) is a 4-layer convolutional neural network. Convolution is applied to each atom, which sums over its neighbors' hidden state, followed by a linear transformation and non-linear activation. Embedded Loopy BP (Dai et al., 2016) is a recurrent architecture, with 4 recurrent iterations. It maintains message vectors for each atom and bond, and propagates those vectors in a message passing fashion. Table 3 shows our model achieves state-of-the-art against various baselines.

7. Conclusion

We proposed a class of deep recurrent neural architectures and formally characterized its underlying computation using kernels. By linking kernel and neural operations, we have a "template" for deriving new families of neural architectures for sequences and graphs. We hope the theoretical view of kernel neural networks can be helpful for future model exploration.

Acknowledgement

We thank Prof. Le Song for sharing Harvard Clean Energy Project dataset. We also thank Yu Zhang, Vikas Garg, David Alvarez, Tianxiao Shen, Karthik Narasimhan and the reviewers for their helpful comments. This work was supported by the DARPA Make-It program under contract ARO W911NF-16-2-0023.

References

Anselmi, Fabio, Rosasco, Lorenzo, Tan, Cheston, and Poggio, Tomaso. Deep convolutional networks are hierarchical kernel machines. *preprint arXiv:1508.01084*, 2015.

Bruna, Joan, Zaremba, Wojciech, Szlam, Arthur, and LeCun, Yann. Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*, 2013.

Cheng, Jianpeng, Dong, Li, and Lapata, Mirella. Long short-term memory networks for machine reading. *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pp. 551–562, 2016.

Cho, Youngmin and Saul, Lawrence K. Kernel methods for deep learning. In Bengio, Y., Schuurmans, D., Lafferty, J. D., Williams, C. K. I., and Culotta, A. (eds.), *Advances in Neural Information Processing Systems 22*, pp. 342–350. 2009.

Chung, Junyoung, Gulcehre, Caglar, Cho, KyungHyun, and Bengio, Yoshua. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

Dai, Hanjun, Dai, Bo, and Song, Le. Discriminative embeddings of latent variable models for structured data. *arXiv preprint arXiv:1603.05629*, 2016.

Daniely, Amit, Frostig, Roy, and Singer, Yoram. Toward deeper understanding of neural networks: The power of initialization and a dual view on expressivity. *CoRR*, abs/1602.05897, 2016.

Duvenaud, David K, Maclaurin, Dougal, Iparraguirre, Jorge, Bombarell, Rafael, Hirzel, Timothy, Aspuru-Guzik, Alán, and Adams, Ryan P. Convolutional networks on graphs for learning molecular fingerprints. In *Advances in neural information processing systems*, pp. 2224–2232, 2015.

Dyer, Chris, Ballesteros, Miguel, Ling, Wang, Matthews, Austin, and Smith, Noah A. Transition-based dependency parsing with stack long short-term memory. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Beijing, China, July 2015.

Dyer, Chris, Kuncoro, Adhiguna, Ballesteros, Miguel, and Smith, Noah A. Recurrent neural network grammars. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics*, San Diego, California, June 2016.

Gal, Yarin and Ghahramani, Zoubin. A theoretically grounded application of dropout in recurrent neural networks. In *Advances in Neural Information Processing Systems 29 (NIPS)*, 2016.

Gärtner, Thomas, Flach, Peter, and Wrobel, Stefan. On graph kernels: Hardness results and efficient alternatives. In *Learning Theory and Kernel Machines*, pp. 129–143. Springer, 2003.

Hazan, Tamir and Jaakkola, Tommi. Steps toward deep kernel methods from infinite neural networks. *arXiv preprint arXiv:1508.05133*, 2015.

Heinemann, Uri, Livni, Roi, Eban, Elad, Elidan, Gal, and Globerson, Amir. Improper deep kernels. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, pp. 1159–1167, 2016.

Henaff, Mikael, Bruna, Joan, and LeCun, Yann. Deep convolutional networks on graph-structured data. *arXiv preprint arXiv:1506.05163*, 2015.

Hinton, Geoffrey E, Srivastava, Nitish, Krizhevsky, Alex, Sutskever, Ilya, and Salakhutdinov, Ruslan R. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

Hochreiter, Sepp and Schmidhuber, Jürgen. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Irsoy, Ozan and Cardie, Claire. Deep recursive neural networks for compositionality in language. In *Advances in Neural Information Processing Systems*, 2014.

Iyyer, Mohit, Boyd-Graber, Jordan, Claudino, Leonardo, Socher, Richard, and Daumé III, Hal. A neural network for factoid question answering over paragraphs. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 633–644, Doha, Qatar, October 2014.

Iyyer, Mohit, Manjunatha, Varun, Boyd-Graber, Jordan, and Daumé III, Hal. Deep unordered composition rivals syntactic methods for text classification. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2015.

Kalchbrenner, Nal, Grefenstette, Edward, and Blunsom, Phil. A convolutional neural network for modelling sentences. In *Proceedings of the 52th Annual Meeting of the Association for Computational Linguistics*, 2014.

Kim, Yoon. Convolutional neural networks for sentence classification. In *Proceedings of the Empiricial Methods in Natural Language Processing (EMNLP 2014)*, 2014.

Kim, Yoon, Jernite, Yacine, Sontag, David, and Rush, Alexander M. Character-aware neural language models. *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

Kingma, Diederik P and Ba, Jimmy Lei. Adam: A method for stochastic optimization. In *International Conference on Learning Representation*, 2015.

Kumar, Ankit, Irsoy, Ozan, Ondruska, Peter, Iyyer, Mohit, James Bradbury, Ishaan Gulrajani, Zhong, Victor, Paulus, Romain, and Socher, Richard. Ask me anything: Dynamic memory networks for natural language processing. 2016.

Le, Quoc and Mikolov, Tomas. Distributed representations of sentences and documents. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pp. 1188–1196, 2014.

Lei, Tao, Joshi, Hrishikesh, Barzilay, Regina, Jaakkola, Tommi, Tymoshenko, Katerina, Moschitti, Alessandro, and Marquez, Lluis. Semi-supervised question retrieval with gated convolutions. *arXiv preprint arXiv:1512.05726*, 2015.

Li, Yujia, Tarlow, Daniel, Brockschmidt, Marc, and Zemel, Richard. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

Lodhi, Huma, Saunders, Craig, Shawe-Taylor, John, Cristianini, Nello, and Watkins, Chris. Text classification using string kernels. *Journal of Machine Learning Research*, 2(Feb):419–444, 2002.

Merity, Stephen, Xiong, Caiming, Bradbury, James, and Socher, Richard. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

Pennington, Jeffrey, Socher, Richard, and Manning, Christopher D. Glove: Global vectors for word representation. volume 12, 2014.

Press, Ofir and Wolf, Lior. Using the output embedding to improve language models. *arXiv preprint arXiv:1608.05859*, 2016.

Ramon, Jan and Gärtner, Thomas. Expressivity versus efficiency of graph kernels. In *First international workshop on mining graphs, trees and sequences*, pp. 65–74. Citeseer, 2003.

Shalev-Shwartz, Shai, Shamir, Ohad, and Sridharan, Karthik. Learning kernel-based halfspaces with the 0-1 loss. *SIAM Journal on Computing*, 40(6):1623–1646, 2011.

Shervashidze, Nino, Schweitzer, Pascal, Leeuwen, Erik Jan van, Mehlhorn, Kurt, and Borgwardt, Karsten M. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(Sep):2539–2561, 2011.

Socher, Richard, Pennington, Jeffrey, Huang, Eric H, Ng, Andrew Y, and Manning, Christopher D. Semi-supervised recursive autoencoders for predicting sentiment distributions. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pp. 151–161, 2011.

Socher, Richard, Perelygin, Alex, Wu, Jean, Chuang, Jason, Manning, Christopher D., Ng, Andrew Y., and Potts, Christopher. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pp. 1631–1642, October 2013.

Srivastava, Rupesh K, Greff, Klaus, and Schmidhuber, Jürgen. Training very deep networks. In *Advances in neural information processing systems*, pp. 2377–2385, 2015.

Tai, Kai Sheng, Socher, Richard, and Manning, Christopher D. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53th Annual Meeting of the Association for Computational Linguistics*, 2015.

Tamar, Aviv, Levine, Sergey, Abbeel, Pieter, Wu, Yi, and Thomas, Garrett. Value iteration networks. In *Advances in Neural Information Processing Systems*, pp. 2146–2154, 2016.

Vishwanathan, S Vichy N, Schraudolph, Nicol N, Kondor, Risi, and Borgwardt, Karsten M. Graph kernels. *Journal of Machine Learning Research*, 11(Apr):1201–1242, 2010.

Zaremba, Wojciech, Sutskever, Ilya, and Vinyals, Oriol. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

Zhang, Yuchen, Lee, Jason D., and Jordan, Michael I. ℓ_1-regularized neural networks are improperly learnable in polynomial time. In *Proceedings of the 33nd International Conference on Machine Learning*, 2016.

Zilly, Julian Georg, Srivastava, Rupesh Kumar, Koutník, Jan, and Schmidhuber, Jürgen. Recurrent Highway Networks. *arXiv preprint arXiv:1607.03474*, 2016.

Zoph, Barret and Le, Quoc V. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.