Comparing Convolution Kernels and Recursive Neural Networks for Learning Preferences on Structured Data

Sauro Menchetti[1] Fabrizio Costa[1] Paolo Frasconi[1] Massimiliano Pontil[2]

[1]Department of Systems and Computer Science,
Università di Firenze, Italy
{menchett, costa, paolo}@dsi.unifi.it
http://www.dsi.unifi.it/neural/

[2]Department of Computer Science,
University College London, UK
m.pontil@cs.ucl.ac.uk

Abstract

Convolution kernels and recursive neural networks (RNN) are both suitable approaches for supervised learning when the input portion of an instance is a discrete structure like a tree or a graph. We report about an empirical comparison between the two architectures in a large scale preference learning problem related to natural language processing, where instances are candidate incremental parse trees. We found that kernels never outperform RNNs, even when a limited number of examples is employed for learning. We argue that convolution kernels may lead to feature space representations that are too sparse and too general because not focused on the specific learning task. The adaptive encoding mechanism in RNNs in this case allows us to obtain better prediction accuracy at smaller computational cost.

Keywords: *Convolution Kernels, Recursive Neural Networks, Data Structures, Preference Models, Machine Learning.*

1. Introduction

Supervised learning algorithms on discrete structures such as strings, trees, and graphs are very often derived from vector based methods, using a function composition approach. In facts, if we exclude symbolic learning algorithms such as inductive logic programming, any method that learns classification, regression, or preference functions from variable-size discrete structures must first convert structures into a vector space and subsequently apply more "traditional" learning tools to the resulting vectors.

The mapping from discrete structures to vector spaces can be realized in alternative ways. One possible approach is to employ a kernel function, as suggested for example by Haussler [8] or by Jaakkola and Haussler [9]. Similarly, recursive neural networks (RNN) [5] can solve the supervised learning problem when the input portion of the data is a labeled directed acyclic graph (DAG). The two methods have different potential advantages and disadvantages.

The use of kernels allows us to apply large margin classification or regression methods, such as support vector machines (SVM) [14] or the voting perceptron (VP) algorithm [6]. These methods have a solid theoretical foundation and may attain good generalization even with relatively small data sets by searching the solution to the learning problem in an appropriately small hypothesis space. When using kernels, the feature space representation of the input data structure is accessed implicitly and the associated feature space may have very high dimensionality, perhaps even infinite. Separating data in very high dimensional spaces does not necessarily lead to overfitting, since large margin methods such as SVM scale up with the ratio between the sphere containing the data points and the distance between the separating hyperplane and the closest training examples. Another advantage of many kernel-based algorithms is that, unlike neural networks, they typically minimize a convex functional, thus avoiding the intractable problem of dealing with local minima. However, this problem is only partially avoided. In fact, the kernel function usually needs to be tuned/adapted to the problem at hand, a problem which cannot in general be cast as a non-convex optimization problem. The typical example is tuning the variance of a gaussian kernel. Learning the kernel function is still an open problem, particularly in the case of discrete structures. When using convolution kernels, for example, the mapping from discrete structures to the feature space is fixed before learning by the choice of the kernel function. A non-optimally chosen kernel may lead to very sparse representations and outweigh the benefits of the 'subsequent' large margin methods.

On the other hand, a RNN operates by composing two adaptive functions. First, a discrete structure is *recursively* mapped into a low-dimension real vector by a function f. Second, the output is computed by a function g (also implemented by a feedforward neural network) that takes as argument the vector representation computed in

the first step. Thus, the role played by f in RNNs is very similar to the role played by the kernel function in the case of large margin classifiers, but it is an adaptive function, leading to 'feature space' representations that are focused on the particular learning task.

In this paper, we compare the two above approaches in a large scale learning problem that occurs in computational linguistics. The task consists of predicting incremental fragments of parse trees extracted from a treebank [4]. The learning task is a preference problem that consists of selecting the best alternative tree in a forest of competitors. We show how to perform preference learning in this highly structured domain using both learning methods (RNN and VP with convolution kernels) and we enlighten some interesting connections between the two approaches. Finally, we report several experimental comparisons, showing that in spite of their theoretical appeal, convolution kernels never achieve significantly better generalization compared to RNNs, even when smaller data sets are employed for training.

2. Convolution Kernels

An effective method to convert structures into a vector representation, in order to apply traditional learning tools, is to define problem specific kernel functions. Collins and Duffy [2] proposes a convolutional kernel for parse trees generated by a natural language parser. Let $\phi : \mathcal{X} \mapsto \mathcal{H}$ a mapping from discrete structured tree space $\mathcal{X}$ to an high dimensional Hilbert space $\mathcal{H}$ that represents the feature space. Let $\mathcal{H} = \{f_n\}_{n=1}^{N}$ be the set of all the tree fragments [1], where f_n is the n-th tree fragment with the only constraint that a production rule can not be divided into further subparts. Then each parse tree $\boldsymbol{t}$ is represented by an N-dimensional vector $\phi(\boldsymbol{t}) = \{\phi_n(\boldsymbol{t})\}_{n=1}^{N}$ of tree fragments: the n-th feature value $\phi_n(\boldsymbol{t})$ counts the occurrence number of the n-th tree fragment f_n in $\boldsymbol{t}$. This representation can be seen as a *bag of subtrees* representation: the object is mapped into a feature vector of which each component counts the occurrence number of any structure. If we define the inner product between two trees $\boldsymbol{t}_1, \boldsymbol{t}_2 \in \mathcal{X}$ by a kernel $K(\boldsymbol{t}_1, \boldsymbol{t}_2) = \phi(\boldsymbol{t}_1) \cdot \phi(\boldsymbol{t}_2)$ and if we find a solution to compute it efficiently [2], there is no need to compute explicitly the feature vectors $\phi(\boldsymbol{t}_1)$ and $\phi(\boldsymbol{t}_2)$. The main idea is to compute the kernel on a tree as a function of the kernel evaluated on subtrees of the original tree using a dynamic programming algorithm. The computation of the kernel is recursive: the kernel is defined for fundamental structures and more complex objects are split into simple subparts.

A problem of this kernel is its dependency on the number of tree nodes: the large is the number of nodes, the bigger the kernel value will be. It can be overcome by normalizing the kernel value. An another drawback is that the kernel value distribution is very peacked. Since the number of larger fragments is greater than the number of smaller fragments, we can limit the height of the subtrees we use to compute the kernel. Otherwise we can weight all the fragments by a coefficient which decays exponentially with their size, so larger fragments have a smaller coefficient than the smaller ones:

$$K_\lambda(\boldsymbol{t}_1, \boldsymbol{t}_2) = \sum_{n=1}^{N} \lambda^{\text{size}_n} \phi_n(\boldsymbol{t}_1)\phi_n(\boldsymbol{t}_2) \qquad (1)$$

where size_n is the number of productions in the n-th fragment and $0 < \lambda \leq 1$.

Voted Perceptron

Kernels can be used in conjunction with several learning algorithms. SVM is employed when we are interested to find a maximal-margin solution with a good generalization performance, since it is theoretically well-founded and there are guarantees on its convergence to a unique global optimum. Unfortunately SVM is computationally very intensive and, when the number of training examples is very large, it becomes not applicable.

Voted Perceptron (VP) [6] is an online algorithm for binary classification of n-dimensional instances based on Rosenblatt's perceptron algorithm. It is much simpler to implement and more efficient in terms of computational time in respect to SVM classifiers, though there are no convergence guarantees. On the other hand, as experimentally shown by [6], VP and maximal-margin classifiers performance results are very similar. The training procedure uses a labeled training set $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{m}$, where $\boldsymbol{x}_i \in \mathcal{X}$ and $y_i \in \{+1, -1\}$. At each step k, VP classifies a training example $\boldsymbol{x}_i$ using the current vector of weights $\boldsymbol{w}_k$. If the predicted label $\hat{y}_i$ is different from the true label y_i, then the vector $\boldsymbol{w}_k$ is updated and $\boldsymbol{x}_i$ added to a list $\mathcal{L}$ of incorrectly classified examples, else the number c_k of training examples correctly classified by $\boldsymbol{w}_k$ is increased. Let $\phi : \mathcal{X} \mapsto \mathcal{H}$ be a function that maps training examples into high dimensional Hilbert space and let $K : \mathcal{X} \times \mathcal{X} \mapsto I\!R$ be a Mercer kernel, $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \phi(\boldsymbol{x}_i) \cdot \phi(\boldsymbol{x}_j)$. VP outputs a set of E weighted perceptrons $\mathcal{W} = \{(\boldsymbol{w}_k, c_k)\}_{k=1}^{E}$ or, equivalently, a list $\mathcal{L} = \{(\boldsymbol{x}_{q_k}, c_k)\}_{k=1}^{E}$ of incorrectly classified examples. The class of a new instance is predicted as $\hat{y} = \text{sign}[f(\boldsymbol{x})]$ where

$$\begin{aligned} f(\boldsymbol{x}) &= \sum_{k=1}^{E} c_k \Gamma[\boldsymbol{w}_k \cdot \phi(\boldsymbol{x})] \\ &= \sum_{k=1}^{E} c_k \Gamma[\sum_{j=1}^{k} y_{q_j} K(\boldsymbol{x}_{q_j}, \boldsymbol{x})] \end{aligned} \qquad (2)$$

and $\Gamma : I\!R \mapsto I\!R$ is the identity function or the sign function. The worst case complexity of the VP training algorithm is $\mathcal{O}(Tm^2u)$, where T is the epoch number, m is the training set element number and u is the cost to compute the kernel [6].

3. Recursive Neural Networks

Recursive neural networks (RNNs) are a generalization of neural networks (NNs) capable of processing structured data such as directed ordered acyclic graphs (DOAG) [5] where a discrete or real label is associated with each vertex. Recently RNNs have been used to process linguistic data such as syntactic trees in order to model psycholinguistic preferences in the incremental parsing process [4, 13]. The key idea is to replicate a NN for each node of the tree and consider as input to the network both the atomic information represented by the label and the structured information derived by the output of all the networks instantiated for each child node. The process of replicating a NN is called *network unfolding* and as a result of this procedure we obtain a large network having shared weights and whose topology matches that of the input graph. At each node the NN outputs a vector encoding of the whole subtree dominated by that node. More formally, on a labeled ordered m-ary tree with nodes v we denote with $\mathrm{ch}[v]$ the ordered m-tuple of vertices whose elements are v's children and with $I(v)$ the discrete label attached to vertex v. Data processing takes place in recursive fashion, traversing the tree in post-order, using a transition function f such that $\boldsymbol{x}(v) = f(\boldsymbol{x}(\mathrm{ch}[v]), I(v))$ where $\boldsymbol{x}(v) \in I\!R^n$ denotes the state vector associated with node v and $\boldsymbol{x}(\mathrm{ch}[v]) \in I\!R^{m \cdot n}$ is a vector obtained by concatenating the components of the state vectors contained in v's children. The transition function $f : \mathcal{I} \times I\!R^{m \cdot n} \mapsto I\!R^n$ maps states at v's children and the label at v into the state vector at v. A *frontier* state $\overline{\boldsymbol{x}} = 0$ is used as the base step of recursion.

We use a feed-forward neural network to model the transition function f according to the scheme:

$$\begin{aligned} a_j(v) &= \omega_{j0} + \sum_{h=1}^{N} \omega_{jh} z_h(I(v)) + \\ &+ \sum_{k=1}^{m} \sum_{\ell=1}^{n} w_{jk\ell} x_\ell(\mathrm{ch}_k[v]) \qquad (3) \\ x_j(v) &= \tanh(a_j(v)) \quad h = 1, \ldots, n \qquad (4) \end{aligned}$$

where $x_j(v)$ denotes the j-th component of the state vector at vertex v, $z_h(I(v))$ is a one-hot encoding of label symbols with N size of the label set, $\mathrm{ch}_k[v]$ denotes the k-th child of v, and $\omega_{jh}, w_{jk\ell}$ are adjustable weights.

Proceeding in this fashion, the root state computed by the network encodes the whole data structure and can be used for subsequent processing.

4. Ordinal Regression and Preference Model

Work on learning theory has mostly concentrated on classification and regression. However, there are many applications in which it is desirable to order rather than to classify instances: these problems arise frequently in social sciences, in information retrieval, in econometric models and in classical statistics where human preferences play a major role.

In a general learning task, we have an instance space $\mathcal{X}$ with associated a set $\mathcal{Y}$. The goal of learning theory is to compute a function $f : \mathcal{X} \mapsto \mathcal{Y}$ which best models the probabilistic relation between these two spaces. The properties of the set $\mathcal{Y}$ define two different learning problems: (a) if $\mathcal{Y}$ is a finite unordered set, we have a classification problem; (b) if $\mathcal{Y}$ is a metric space, e.g., the set of real numbers, we have a regression problem. Ordinal regression, partial ranking and preference model tasks don't fit in any two previous classes but share properties of both classification and regression problems: (a) $\mathcal{Y}$ is a finite set; (b) the elements of $\mathcal{Y}$ are ordered but $\mathcal{Y}$ is a non-metric space, i.e. the distance $(y_1 - y_2)$ of two elements is not defined. So, as in classification problems, we have to assign a possible label to a new instance, but similar to regression problems, the label set admits a total order relation.

The Utility Function Approach

Let $\mathcal{D} = \{(\boldsymbol{x}_{i1}, y_{i1}), \ldots, (\boldsymbol{x}_{ik_i}, y_{ik_i})\}_{i=1}^{m}$ be a training set, where $(\boldsymbol{x}_{i1}, \ldots, \boldsymbol{x}_{ik_i})$ is the i-th set of competing examples, $\boldsymbol{x}_{ij} \in \mathcal{X} = I\!R^n$, $y_{ij} \in \mathcal{Y} = \{1, \ldots, k_i\}$ is the rank of $\boldsymbol{x}_{ij}$. In this setting, $\boldsymbol{x}_{ij}$ is better than $\boldsymbol{x}_{ik}$ (written $\boldsymbol{x}_{ij} \succ \boldsymbol{x}_{ik}$) if $y_{ij} < y_{ik}$. We can model the importance of an instance by introducing an utility function $U : \mathcal{X} = I\!R^n \mapsto I\!R$ so that given $\boldsymbol{x}, \boldsymbol{z} \in \mathcal{X}$, then $\boldsymbol{x} \succ \boldsymbol{z} \Leftrightarrow U(\boldsymbol{x}) > U(\boldsymbol{z})$. Since U can be used to rank all the instances of a set $X = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_r\}$, we have converted the ordinal regression and preference model problems to the problem of approximating the utility function U with a function $\hat{U}$.

Since $\boldsymbol{x}_i \succ \boldsymbol{x}_j \Leftrightarrow U(\boldsymbol{x}_i) > U(\boldsymbol{x}_j)$, we can compute the rank $y_i \in \{1, \ldots, r\}$ of each instance $\boldsymbol{x}_i$ as

$$y_i = \sum_{j=1}^{r} \theta(\hat{U}(\boldsymbol{x}_j) - \hat{U}(\boldsymbol{x}_i)) \qquad (5)$$

where $\theta : I\!R \mapsto \{1, 0\}$ is the Heavyside function defined as $\theta(x) = 1$ if $x \geq 0$, 0 otherwise. In the case of a preference model, we just want to select the best instance $\boldsymbol{x}^*$ and we look at minimum among y_i computed in (5) or, equivalently, $\boldsymbol{x}^* = \arg\max_{\boldsymbol{x}_i} \hat{U}(\boldsymbol{x}_i)$.

The overall ranking error on a dataset of m sets can be defined in terms of the utility function U:

$$E = \sum_{k=1}^{m} L \left[\sum_{i,j: y_{ki} < y_{kj}} G\left(U(\boldsymbol{x}_{ki}) - U(\boldsymbol{x}_{kj})\right) \right] \qquad (6)$$

where is G is a non-decreasing function that weights pairwise differences between predicted utilities and L a non-decreasing function of the cumulative error. For example, if L is the identity function and $G(a) = \theta(a)$, equation (6) counts the number of misranked instances. In case of

a preference model, assuming that $\boldsymbol{x}_{k1}$ is the best alternative, we have:

$$E = \sum_{k=1}^{m} L\left[\sum_{j=2}^{r_k} G\left(U(\boldsymbol{x}_{kj}) - U(\boldsymbol{x}_{k1})\right)\right] \quad . \tag{7}$$

Kernel Preference Model

We consider a simple model and we assume that the utility function U is linear and parametrized by an n-dimensional vector $\boldsymbol{w} = (w_1, \ldots, w_n)$:

$$U(\boldsymbol{x}) = \sum_{i=1}^{n} w_i x_i \quad .$$

If $\boldsymbol{x} \succ \boldsymbol{z}$, then

$$\begin{aligned} U(\boldsymbol{x}) > U(\boldsymbol{z}) \quad &\Leftrightarrow \quad \boldsymbol{w} \cdot \boldsymbol{x} > \boldsymbol{w} \cdot \boldsymbol{z} \\ &\Leftrightarrow \quad \boldsymbol{w} \cdot (\boldsymbol{x} - \boldsymbol{z}) > 0 \quad . \end{aligned}$$

Then in a complete rank problem, the solution $\boldsymbol{w}$ satisfies the following constraints:

$$\begin{aligned} &\boldsymbol{w} \cdot (\boldsymbol{x}_{ki} - \boldsymbol{x}_{kj}) > 0 \\ &k = 1, \ldots, m \text{ and } i, j = 1, \ldots, r_k : y_{ki} < y_{kj} \end{aligned} \tag{8}$$

This is equivalent to binary classification of pairwise differences between instances. In a preference model, instead, assuming that $\boldsymbol{x}_{k1}$ is the best alternative, the constraints that must be satisfied by $\boldsymbol{w}$ are:

$$\begin{aligned} &\boldsymbol{w} \cdot (\boldsymbol{x}_{k1} - \boldsymbol{x}_{kj}) > 0 \\ &k = 1, \ldots, m \text{ and } j = 2, \ldots, r_k \quad . \end{aligned} \tag{9}$$

We see that the number of constraint in a complete rank grows quadratically in the size of the alternatives, while it is linear in a preference model.

Nonlinear utility function can be easily realized by introducing a kernel function and mapping instances into the associated feature space. In this case:

$$\begin{aligned} &\boldsymbol{w}_t \cdot [\phi(\boldsymbol{x}_{k1}) - \phi(\boldsymbol{x}_{kj})] = \\ &= \sum_{h=1}^{t} y_{q_h p_h} [K(\boldsymbol{x}_{q_h 1}, \boldsymbol{x}_{k1}) - K(\boldsymbol{x}_{q_h 1}, \boldsymbol{x}_{kj}) + \\ &- K(\boldsymbol{x}_{q_h p_h}, \boldsymbol{x}_{k1}) + K(\boldsymbol{x}_{q_h p_h}, \boldsymbol{x}_{kj})] \end{aligned}$$

for appropriate indexes q_h and p_h. In a preference model or in an ordinal regression task, the Γ function is set to the identity function to avoid ties between competing instances generated by the sign function.

In the end learning, the classification task yields to minimize an error function of the kind:

$$E = \sum_{k=1}^{m} \sum_{j=2}^{r_k} \theta(\boldsymbol{w} \cdot [\phi(\boldsymbol{x}_{kj}) - \phi(\boldsymbol{x}_{k1})]) \tag{10}$$

which is an instance of (7) where L is the identity function, G is the Heavyside function and $U(\boldsymbol{x}) = \boldsymbol{w} \cdot \phi(\boldsymbol{x})$. Since the preference learning task is reduced to a standard classification problem on pairs of examples, high classification accuracy ensures a small classification error (10).

Recursive Neural Networks Preference Model

For the connectionist approach, we will consider only the preference task. We implement U with a neural network architecture formed by an encoder that maps a tree into a real vector representation (the state vector computed by the recursive network on the root node of the tree itself) and a feed-forward output network that performs the final mapping into a real number [4].

We now model the probability for $\boldsymbol{x}_1 \in X$, where $X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_r\}$, to be selected as the preferred element by our model with parameters ω, in the following way:

$$P(\boldsymbol{x}_1 | X, \omega) = \frac{e^{U(\boldsymbol{x}_1)}}{\sum_{j=1}^{r} e^{U(\boldsymbol{x}_j)}} \quad . \tag{11}$$

Using this probability estimate, we can write the likelihood of the training data as $\prod_{k=1}^{m} P(\boldsymbol{x}_{k1} | X^k, \omega)$. Learning proceeds maximizing the log-likelihood of the training data (i.e. minimizing the negative log-likelihood[1]) with respect to the model parameters, which can easily be shown to be equivalent to optimize ω in order to minimizing the ranking error:

$$\begin{aligned} &\sum_{k=1}^{m} -\log \frac{e^{U(\boldsymbol{x}_{k1})}}{\sum_{j=1}^{r_k} e^{U(\boldsymbol{x}_{kj})}} = \\ &= \sum_{k=1}^{m} \log \sum_{j=1}^{r_k} e^{U(\boldsymbol{x}_{kj}) - U(\boldsymbol{x}_{k1})} \quad . \end{aligned} \tag{12}$$

Minimization is achieved by a variant of the gradient descend backpropagation algorithm [7]. Note that (12) is an instance of (7) where $L(x) = \log(x)$, $G(x) = e^x$ and U is the combination of the recursive and the output neural network.

5. Experimental Results

In order to compare the kernel and the connectionist methods on a preference learning task, we have chosen the linguistically relevant problem of first pass attachment prediction. We performed two experiments: the first using a large dataset to compare the two models performance and the second on several smaller dataset splits to assess the different generalization properties in a regime of data scarcity for training.

5.1. The Learning Problem: First Pass Attachment

An interesting problem in psycholinguistic is to determine structural preferences exhibited while interpreting a sentence in a sequential fashion. In [10] a dynamic grammar is developed to formalize this task, which is called *first pass attachment disambiguation* task in the

[1] For a complete derivation see [4].

psycholinguistic literature. Under this framework, the problem becomes that to process a sentence one word at a time, from left to right, in an incremental way. At any stage of the elaboration, the interpretation (i.e the syntactic parse tree) is a fully connected structure, as opposed to traditional parsing procedures that determine smaller sub-structures to be joined at later stages. The key idea of the dynamic grammar is to associate to each lexical item the part of syntactic tree needed to connect that item to the structure built up to that point: these parts are called *connection paths* and can be automatically extracted from a corpus of parsed sentences [11]. Note that the lexical items in the remainder of this paper are represented by their grammatical category (like verb, noun, etc.), called *part of speech* (POS) rather than by the actual word. Given a syntactic interpretation for a fragment of a sentence, called *left context*, and a new item that we want to incorporate in the interpretation (i.e. to *attach* to the left context), we have to decide: (a) at which point of the left context to attach the connection path (the *anchor*[2] point of the attachment); and (b) which connection path to choose among several possibilities. The dynamic grammar introduces therefore some ambiguities that in the end generate several licensed parse trees, though only one of these analyses will be the correct one. To have an idea of the possible ambiguities, consider Fig. 1.

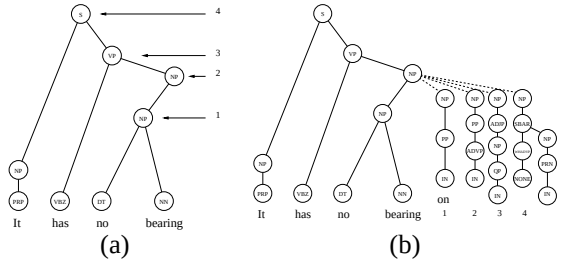

Figure 1. Ambiguities: (a) possible anchor points, (b) possible connection paths

Once the connection path has been attached to a left context, we obtain an *incremental tree*. The first pass attachment disambiguation task consists in choosing at each stage the correct incremental tree and it can be naturally casted into a preference or ranking task.

5.2. Heuristics for Prediction Enhancement

As shown in the analysis carried out in [4], the incremental disambiguation task can be significantly enhanced adopting two heuristics. The first heuristic is called *tree reduction* and consists in removing nodes from the syntactic parse that are considered irrelevant for discriminating between alternative incremental trees. Intuitively these nodes are deep nodes, where the depth is measured in respect of the part of the left context where the attachment process takes place[3]. This reduction of the complexity of the trees has been proved beneficial in increasing the prediction accuracy.

The second heuristic consists in *specializing* the first pass attachment prediction in respect to the class of the item being attached. The idea is to train and employ specialized predictors for each different word classes (nouns, verbs, etc.). The heuristic is applicable since the different classes naturally partition the dataset into non-overlapping sets (for example, what is learned on attaching nouns is not relevant for attachment decisions on verbs or punctuation).

5.3. Experimental Setup and Evaluation

The dataset used in our comparison experiments is the Wall Street Journal (WSJ) section of Penn TreeBank [12], chosen because of its large size (about 40,000 parsed sentences, for a total of 1 million words) and because of its status as a recognized benchmark in computational linguistics.

In the first experiment, we used the literature-standard split into train (sections 2–21), validation (section 24) and test set (section 23). All parse trees have been preprocessed with the reduction heuristic (see Sect. 5.2) and divided in accordance to the POS tag of the word being attached. The WSJ treebank uses 45 different POS tags that we have grouped into 10 classes [4] (one for all tags referring to nouns, one for verbs, etc.). Note that the input data consists of a much larger number of trees since each parse tree is split into incremental trees (one for each word of the sentence) and each incremental tree in turn is just part of a set where a single correct instance has to be chosen among several candidate alternatives generated by the dynamic grammar. The real dataset size can be estimated considering that the average sentence length is 24 words and that for each word we have on average of 120 alternative incremental trees (ranging from a minimum of 2 to a maximum of 2000 alternatives), obtaining a total of 100 million trees.

The VP and the RNN prediction accuracy in the preference task have been compared varying the training set size. In this way, we have estimated the learning curve of the two approaches together with the prediction performance. We expected the VP to show better generalization properties and to outperform the RNN when trained on small datasets due to the regularization guarantees of the kernel method. We created incremental datasets with 100, 500, 2,000, 10,000 and 40,000 sentences, randomly extracting them from the training set. Each of these incremental datasets has then been partitioned into the 10 classes and a different VP and RNN has been trained for

[2]More precisely the anchor is the node in common between the connection path and the left context once the connection path has been joined.

[3]More precisely the discarded nodes are selected on the basis of linguistically motivated characteristics, such as the notion of c-command, though the result is the same.

each class of each dataset. The ranking error is computed as the normalized number of correct elements ranked in the first position in respect of the total number of sets.

Due to the high computational cost of the validation procedure, the working and models parameters have been optimized on a fixed validation set of 1,000 sentences. These parameters have then been used for all the models in the presented experimentation. Specifically, for the kernel VP the regularization parameter λ is set to 0.5; for the RNN the adequate state vector size has been fixed to 25 units (i.e. the minimum size to ensure enough expressive power to learn the training set with high accuracy), a learning rate η ranging from 10^{-2} to 10^{-4} and a momentum $m = 0.1$. Other parameters for the RNN are: weight initialization with random weights in $[-0.01, +0.01]$, hyperbolic tangent as the non-linear squashing function, maximum node outdegree set to 15. Fixing the maximum outdegree is an architectural constraint which, in our implementation, has as a consequence the pruning of those syntactic trees with very long production rules. Note that since each child position is associated with its own set of weights, pruning long productions avoids poor estimates of the weights associated with very infrequent rules. Allowing rules containing no more than 15 symbols, only 0.3% of the possible productions are pruned.

For the VP, training is carried out presenting all the input elements in the training sets only once (one epoch). For the RNN, training is continued until the error reaches a minimum threshold. A 1,000 sentences validation set is then used to determine the optimal epoch and select the best weight configuration for generalization (early stopping criterion). The number of iterations needed to reach an error minimum varies with the size of the incremental training sets from 5,000 to 5.

The results of the comparison are shown in Table 1. The RNN exhibit 1% better prediction accuracy in respect to VP and no evidence is found for the superiority kernel VP when trained on a small dataset. In addition kernel VP method has the drawback of high computational costs: training over 5 splits of 100 sentences takes about a week on a 2 GHz CPU and moreover VP does not scale linearly with the number of examples as the RNN does. For small datasets, the CPU time of kernel VP is about the CPU time of RNN, while for larger datasets the elaboration times are much higher: in the first experiments, VP took over 2 months to complete an epoch but RNN learns in 1–2 epoch (about 3 days with a 2 GHz CPU). This high computational cost has forced us to train the VP for only one epoch in the full experiment with the 40,000 sentences. In order to verify the performance on small datasets, we have carried out a more robust experiment. To avoid differences due to random properties of the chosen datasets, we train the two models on 5 splits of 100 sentences randomly chosen from WSJ sections 2–21. For this experiment no class partition has been performed, though sentences have been preprocessed with the reduction heuristic. Model parameters have been kept identical

VP after 1 Epoch						
POS	Size	100	500	2000	10000	40000
Noun	33.0	87.6	91.6	92.9	94.5	95.6
Verb	13.4	87.4	91.1	93.2	94.9	96.1
Prep	12.6	52.4	57.7	61.9	65.5	68.4
Art	12.5	73.5	82.6	85.8	88.9	90.4
Punc	11.7	49.1	61.0	66.7	74.3	77.5
Adj	7.5	75.8	81.8	85.3	87.9	89.0
Adv	4.3	34.2	41.2	44.3	49.0	53.6
Conj	2.3	53.7	61.3	68.1	71.9	75.0
Pos	2.0	92.5	93.5	94.6	95.6	97.1
Othe	0.7	35.1	45.3	65.4	74.2	78.2
Total	100	72.7	78.7	81.7	84.7	86.6
RNN						
POS	Size	100	500	2000	10000	40000
Noun	33.0	88.6	91.8	91.5	94.1	95.7
Verb	13.4	82.1	90.9	94.0	94.9	96.8
Prep	12.6	56.4	61.8	62.3	64.2	67.5
Art	12.5	69.6	85.2	87.7	89.4	91.0
Punc	11.7	63.4	68.2	74.3	79.0	80.8
Adj	7.5	77.0	83.6	83.2	87.0	89.5
Adv	4.3	34.8	45.1	51.9	56.1	59.4
Conj	2.3	60.8	59.6	68.5	76.8	78.7
Pos	2.0	60.0	89.8	94.1	97.5	97.1
Othe	0.7	11.0	51.7	65.5	77.7	68.6
Total	100	73.4	80.6	82.7	85.5	87.4

Table 1. VP and RNN in the first-pass attachment prediction task. Modularization in 10 POS tag categories and results for various training set sizes

to the previous experimentation. Results are reported in Table 2, where column "Relative Error Reduction %" of RNN respect to VP is computed as $(y-x)/(100-x)*100$, where y is the accuracy of RNN and x is the accuracy of VP. The results confirm the hypothesis on the significant superiority of RNN against the kernel VP even in regime of scarce data available for training.

An advantage of the kernel VP is its smoothness in respect to training iterations, i.e. validating the performance on a working set yields a smooth, single-maximum function. In contrast the RNN is much more sensitive, making it hard to decide for a good generalization point.

6. Conclusion

The experimental results do not indicate any better performance of the kernel VP in respect to the RNN; on the contrary, the connectionist approach outperforms the other method, even when trained on a small training dataset. It appears that RNNs are to be preferred, unless good knowledge is available for the design of the right kernel. In other terms, it is better to rely on a method

Split	VP	RNN	Rel. Error Reduction %
1	76.0	76.3	1.3
2	75.7	78.6	11.9
3	75.4	77.2	7.3
4	75.3	75.9	2.4
5	74.5	77.1	10.2
Mean	75.4	77.0	6.5

Table 2. VP and RNN in the first-pass attachment prediction task: 5 splits of 100 sentences

capable of building an adaptive representation of data, rather than using a kernel method when it is not clear how good is the kernel for the problem. A solution would be to learn the kernel function, but learning the kernel is still an open problem in the case of discrete structures.

Future work includes Collins reranking task [3], where the problem is to rank parse trees generated and scored with a parser. The task can be casted into a preference problem over complete parse trees, and so it is similar to the one studied in this paper. From an initial experimentation on small datasets, it seems that also in this task RNNs are superior to kernel VP.

Acknowledgments

Thanks to Alessio Ceroni, Alessandro Vullo, Andrea Passerini and Giovanni Soda for their fruitful comments.

References

[1] R. Bod. What is the Minimal Set of Fragments that Achieves Maximal Parse Accuracy? In *Proceedings of ACL*, 2001.

[2] M. Collins and N. Duffy. Convolution Kernels for Natural Language. In *Proc. Neural Information Processing Systems*. NIPS 14, 2001.

[3] M. Collins and N. Duffy. New Ranking Algorithms for Parsing and Tagging: Kernels over Discrete Structures, and the Voted Perceptron. In *Proc. of ACL*, 2002.

[4] F. Costa, P. Frasconi, V.Lombardo, and G. Soda. Towards incremental parsing of natural language using recursive neural networks. *Applied Intelligence*, 19(1/2):9–25, 2003.

[5] P. Frasconi, M. Gori, and A. Sperduti. A General Framework for Adaptative Processing of Data Structure. *IEEE Transaction on Neural Networks*, 9(5):768–786, 1998.

[6] Y. Freund and R. E. Schapire. Large Margin Classification using the Perceptron Algorithm. *Machine Learning*, 37(3):277–296, 1999.

[7] C. Goller and A. Kuechler. Learning Task-dependent Distributed Structure-representations by Back-propagation through Structure. In *IEEE International Conference on Neural networks*, pages 347–352, 1996.

[8] D. Haussler. Convolution Kernels on Discrete Structures. Technical Report UCSC-CLR-99-10, University of California at Santa Cruz, 1999.

[9] T. Jaakkola and D. Haussler. Exploiting Generative Models in Discriminative Classifiers. In *Proc. Neural Information Processing Systems*. NIPS 10, 1998.

[10] V. Lombardo, L. Lesmo, L. Ferraris, and C. Seidenari. Incremental Processing and Lexicalized Grammars. In *Proceedings of the XXI Annual Meeting of the Cognitive Science Society*, 1998.

[11] V. Lombardo and P. Sturt. Incrementality and Lexicalism: a Treebank Study. In S. Stevenson and P. Merlo, editors, *Lexical Representations in Sentence Processing*, Computational Psycholingusitics Series. John Benjamins, in press.

[12] M. Marcus, B. Santorini, and M. Marcinkiewicz. Building a Large Annotated Corpus of English: the Penn Treebank. *Computational Linguistics*, 19:313–330, 1993.

[13] P. Sturt, F. Costa, V. Lombardo, and P. Frasconi. Learning First-Pass Structural Attachment Preferences with Dynamic Grammars and Recursive Neural Networks. *Cognition*, 2003. In press.

[14] V. Vapnik. *Statistical Learning Theory*. John Wiley and Sons, New York, 1998.