Transfer Learning High-Order Word Representations

Konstantinos Kogkalidis

konstantinos@riseup.net

1 Preface

This document is meant to act as a brief technical overview of my work on transfer-learning high-order tensorial representations for transitive verbs. As the project itself is still work in progress, the document will also be evolving over time, tracking issues and solutions as they appear.

1.1 Changelog

Date	Changes
25/06/2018	First Draft

2 Introduction

Distributional Compositional Semantics are an active area of computational linguistics research, concerned with the modeling of word interactions to form larger linguistic units. Inspired by formal semantics and distributional models of meaning, it proposes a functional merging of the two. The core idea is that, given a grammar formalism $\mathcal{G}$, such as pregroup grammars or combinatory categorial grammars, we may move to the category of finite-dimensional vector spaces, $\mathbf{FdVect}$ via a structure-preserving map (homomorphism) $\mathcal{F} : \mathcal{G} \rightarrow \mathbf{FdVect}$.

The grammar is responsible for deriving proofs for well-formed linguistic units. In such a grammar, words are assigned types descriptive of their syntactic role. These types may either be atomic (e.g. S for sentences or NP for noun-phrases) or complex, defined as the structured compositions of atomic ones (e.g. NP/NP for adjectives, "consuming" a noun-phrase to their right to produce a new, larger noun-phrase).

Using $\mathcal{F}$, we may assign vector spaces to simple types of $\mathcal{G}$. Complex types can then be translated to high-order tensors (or equivallently multi-linear maps), which operate on their arguments (as dictated by $\mathcal{G}$), to produce syntactically-informed vectorial constructions for larger phrases. Crucially, this allows us to benefit from both the applied neuro-distributional approach and the more theoretically grounded aspects of computational linguistics in meaningfully constructing phrase or sentence numerical representations.

A key problem in the field is the lack of an existing, experimentally verified, framework towards obtaining these high-order tensors for complex word types. This work proposes an alternative to thus far explored solutions, namely transforming the problem setting to that of supervised learning.

Word Type	$\mathcal{G}$ Type	$\mathcal{F}$ Translation
Noun	NP	$\lceil \text{NP} \rceil$
Adjective	NP / NP	$\lceil \text{NP} \rceil \rightarrow \lceil \text{NP} \rceil$

Table 1: *Example translations from grammatic types to $\mathcal{G}$ types to vector spaces. If noun-phrases occupy an arbitrary vector space, $\lceil \text{NP} \rceil$, then adjectives are an endomorphism of that space.*

3 Towards a Supervised Setting

3.1 Overview

Supervised Learning can be abstractly summarized as the search process over a set of continuously differentiable functions F_p with predetermined signature $A \to B$, parameterized over a set of trainable weights p

$$F_p = \{f_p : A \to B\}$$

with the goal being finding the one function $\hat{f}_p$ that best approximates some ideal (but unknown) function $g : A \to B$.

3.2 Dataset

To begin a supervised learning endeavour, we need to first find samples from the aforementioned ideal function g, organized in a tuple $(X, g(X))$. Finding such samples and evaluating their potential to drive learning is by itself a difficult task. For this work, we will be working with the Paraphrase Database (PPDB) phrasal corpus. This contains a large amount of phrases organized in paraphrastic pairs, i.e. phrases that are semantically close to each other.

Before actually utilizing the dataset, a lot of preprocessing was necessary for a variety of reasons summarized below.

Parsing Working with any potential syntactic type would require a dynamically adaptive graph, a functionality that is only being provided by a small number of relevant libraries on an experimental level. Additionally, it would incur heavy computational costs (as explained in Section 5) and would make debugging and evaluating performance hard. For these reasons, it was deemed a minor compromise to constrain the experimentation domain on just transitive verbs and their objects, and the question we set out to answer is whether we can find a way to learn meaningful verb matrices (i.e. order-2 tensors). We use a (very) simplified grammar consisting of two simple types:

1. **Noun Phrases** NP
2. **Actions** A

Where the type A encompasses actions potentially undertaken by some subject agent. Then a monotransitive verb, which is the focal point of this work, can be seen as a the complex type $\lceil a/np \rceil$ and its $\mathcal{F}$ translation as a morphism from the space of noun phrases to that of actions. Under this light, we define and use the below type translations:

1. **Objects** $\lceil np \rceil = \mathbb{R}^{300}$
2. **Actions** $\lceil a \rceil = \mathbb{R}^{100}$
3. **Monotransitive Verbs** $\lceil a/o \rceil = \mathbb{R}^{60 \times 300}$

To enable experimentation, all of our samples need to follow this single derivation. This is accomplished by filtering the entirety of the PPDB corpus by spaCy's syntactic parser. The choice of the concrete vector spaces used is somewhat arbitrary but justifiable. Objects are chosen to live in $\mathbb{R}^{300}$ since that is the standard word vector dimensionality used by popular pretrained word embedding systems such as word2vec, GloVe and fastText, thus giving us the ability to inherit these vectors without further training. The action space is set to $\mathbb{R}^{100}$; fewer dimensions would be overly compressive, reducing the capacity for disambiguation and prediction, whereas higher dimensions would make the network more prone to overfitting and harder to train (more on Section 5).

Backtranslation The syntactic filtering vastly reduces the number of available samples. A large number of different verbs, but also different verb-object combinations, is necessary for training to succeed. Consequently, we need a way to recover from this sample loss. A good way around the problem is to use backtranslation, a popular technique for generating synthetic new paraphrastic phrases out of existing ones. The procedure involves using a pretrained machine translation black box which allows translating an arbitrary phrase to and from a set of languages. For the purposes of this work, google cloud services' translation was used. Given such a system, we can initiate a (potentially randomized) set of candidate languages and draw a randomized transition graph (potentially including loops) passing through those. The stochastic and slightly inaccurate nature of neural translation and the expressive ambiguity between different languages are guaranteed to introduce a minor semantic shift with each translation. After each phrase is chain-translated up to the graph's end-point, we can reverse translate back to the original language in a single step. This usually results in a phrase that is paraphrastic to our starting one, yet not identical to it. This process can be repeated until our dataset size is sufficient for training.

Statistical Filtering PPDB is itself automatically annotated, meaning that it contains a significant amount of error. Syntactic parsing and backtranslation are also not 100% accurate, therefore introducing additional errors with each step of the process. This error is hard to control; manually checking the entirety of the samples would be unfeasibly demanding. A cost-efficient way of reducing the error to acceptable rates is to apply statistical filtering on our samples. Since we are seeking to learn verb representations, it would make sense to only keep verbs that appear adequately often in our samples, yet are not extremely frequent (thereby carrying more noise than information). The distribution of unique verbs in our dataset follows a very sharp, uneven distribution; the head and tail of these are truncated to even out verb frequency. Additionally, verbs co-existing with a very small number of objects carry little information and are prone to overfitting against their objects and are also cut out.

Machine Readability The last preprocessing step is to make the dataset machine-readable; we initiate two injective dictionaries, mapping verbs and objects to and from unique naturals:

$$\mathcal{V} : \{v_1 : 1, v_2 : 2, ..., v_V : V\}$$
$$\mathcal{O} : \{o_1 : 1, o_2 : 2, ..., o_O : O\}$$

where V, O the total number of unique verbs and objects, respectively.

Via this encoding, we can now formulate our ideal function as a relation $\mathcal{P}$; it is a function that accepts two tuples of two naturals each (one tuple uniquely determines a verb-object pair), and returns either 1 if the two encoded phrases are paraphrastic or 0 otherwise:

$$\mathcal{P} : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \{0, 1\}$$
$$\mathcal{P}(i, j, k, l) = \mathcal{P}(k, l, i, j) = \begin{cases} 1 & v_i o_j \sim v_k o_l \\ 0 & v_i o_j \nsim v_k o_l \end{cases}$$

The verb(/object) identifiers can be one-hot encoded before being fed as an input to any embedding layer. One-hot encoding translates a natural n from a strictly ordered set of size N to a N-dimensional vector, whose only non-zero element is its n-th one (being equal to 1). This allows each verb(/object) to be assigned a tensorial representation independent of how the rest of the words are represented. The output is already in an ideal form for a classification task.

4 Student Network

Given the ideal function $\mathcal{P}$, we may begin assembling a network. The goal we set was to train some verb embedding function, which takes a verb's identifier i and returns its correspondig matrix $\mathbf{V}_i$:

$$\varepsilon_{verb} : \mathbb{N} \to \mathbb{R}^{100 \times 300}$$

Leaving the one-hot encoding ($\mathbb{N} \rightarrow [0,1]^V$) implicit, the function is simply a trainable linear map implemented by a feedforward, linearly activated layer with no bias and weights:

$$\mathbf{E}_{verb} \in \mathbb{R}^{V \times 30{,}000{,}000}$$

followed by a reshape. For reasons of brevity and readability, the one-hot encoding and reshaping processes will be implied from now on.

The function we wish to train, ε_{verb}, seems to have little in common with our ideal function's $\mathcal{P}$ signature. This poses no significant issue, as long as we find a functional decomposition of our approximation model F_p, such that it includes ε_{verb}:

$$F_p = f_1 \circ f_2 \circ \cdots \circ \varepsilon_{verb} \circ \dots$$

To simplify the problem, we can assume that the object embeddings can be acquired "for free" via a pretrained embedding layer, which similarly accepts an identifier i and returns an object vector $\mathbf{o}_i$:

$$\varepsilon_{object} : \mathbb{N} \rightarrow \mathbb{R}^{300}$$

Given such a function, we know that an "action" embedding may be acquired by simply applying an object vector to a verb matrix. Two such embeddings can then be compared via any measure of vector distance to produce a scalar. The most common solution, and the one employed in this particular instantiation, is cosine similarity:

$$cos(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{||\mathbf{x}|| \cdot ||\mathbf{y}||}$$

The above sequence of transformations result in a network signature that is fully matching our samples. It can be visually inspected in Fig.1.

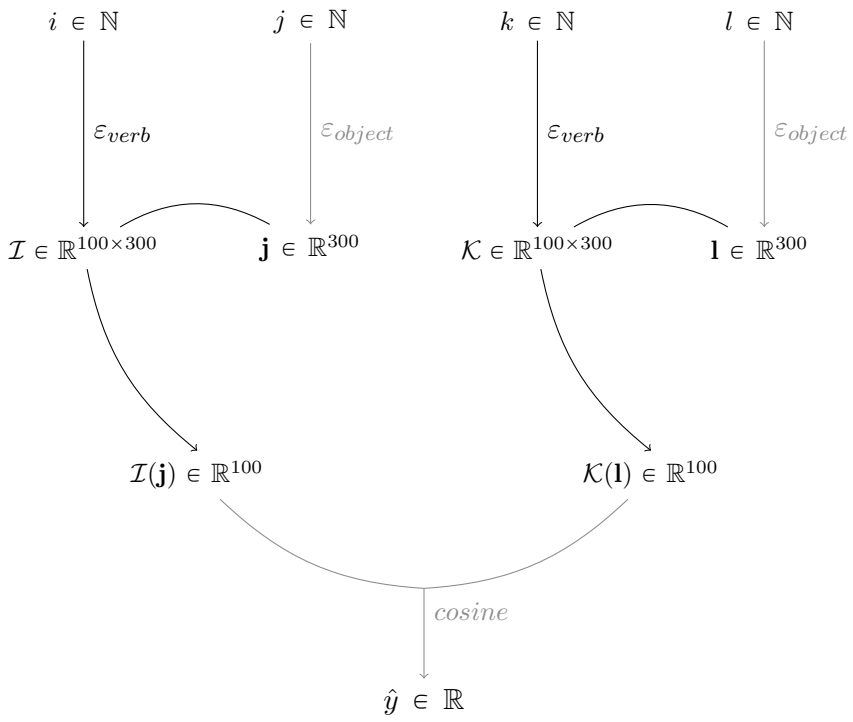

Figure 1: *Abstract view of the student network's data flow and its transformations. Grayed out arrows represent non-trainable functions.*

5 Objective Function

The network, as depicted above, type checks and is fully differentiable. Given some binary loss operator $\mathcal{L}[.,.]$, the objective function can be formulated as:

$$\min_{\forall i,j,k,l} \mathcal{L}\{[cos(\mathbf{V}_i(\mathbf{o}_j), cos(\mathbf{V}_k, \mathbf{o}_l)],\ \mathbf{P}(i,j,k,l)\}$$

Potential candidates for $\mathcal{L}$ might be mean square error, binary crossentropy, categorial hinge or any linear combination thereof. A short discussion on how each particular choice affects the training process can be found in Section 8.

Disregarding the particular choice of $\mathcal{L}$, let us translate the objective function into natural language. We want to minimize some measure of classification penalty between the paraphrase prediction of two phrases and their original label, as given by our training set. More concretely, we want to obtain a phrase embedding from each phrase, constructed by applying an object vector to a verb matrix, such that when we compare two phrase vectors against one another via cosine similarity the result is close to 1 if these two phrases are labeled as paraphrastic or 0 otherwise.

5.1 Intractability

The complexity of the raw optimization problem should be apparent by its formulation alone. The objective is in fact intractable, i.e. practically impossible to optimize over. To better understand why, we need to address a number of key points.

Number of parameters Our verb embedding function is a fully-connected feedforward layer. Fully connected layers have a trainable multiplicative weight associating each input neuron with each output neuron. In our case, the input is the one-hot encoding of each verb (i.e. a $||V||$-dimensional vector space) and the output space is that of linear functions $\mathbb{R}^{NP} \rightarrow \mathbb{R}^{A}$ (i.e. $(NP \times A)$-dimensional representations). Given the fact that the dataset contains $\sim 1,000$ unique verbs, and our choice of spaces for the objects and actions, the resulting layer enumerates $30,000,000$ trainable parameters.

Quantifying Over Multiple Spaces The objective function quantifies over two directly interacting spaces; the space of verbs and the space of actions. This is problematic in many ways; essentially, each space will be sensitive to perturbations of the other (at best), or their joint optimization might be infeasible due to one's minimum being at odds to the other's (at worst).

Non-Convexity More importantly, however, both of the two spaces that we are optimizing over are non-convex. This means that the each of them may have a non-unique global minimum and possibly a large number of local minima. Intuitively, the paraphrase space requires that the vectors of two non-paraphrastic phrases are perpendicular to one another. In a two dimensional space, a simple rotation operation applied on all the vectors would still yield a new solution from an existing one. This is even further attenuated in a higher-dimensional space, where a vector's perpendiculars may occupy a huge hypersurface. At the same time, the verb space requires that an object vector $\mathbf{o}$ applied on a verb matrix $\mathbf{V}$ would result in some predefined phrase vector $\mathbf{p}$. Analytically solving $\mathbf{Vo} = \mathbf{p}$ w.r.t. to $\mathbf{V}$ is impossible, as there exist infinite possible linear combinations of $\mathbf{o}$ that result in $\mathbf{p}$. More realistically, we have a system of linear equations $\mathbf{Vo}_1 = \mathbf{p}_1, \ldots \mathbf{Vo}_k = \mathbf{p}_k$ for all k unique objects vectors applied on the same verb $\mathbf{V}$ yielding different phrase vectors, thus constraining our solution space; however, there is still limited quarantee of the solutions being unique.

6 Teacher Network

The above are all clear indicators of the impracticality of attempting to directly train the network. Even though our operations are end-to-end differentiable, theoretically allowing gradient-based methods to

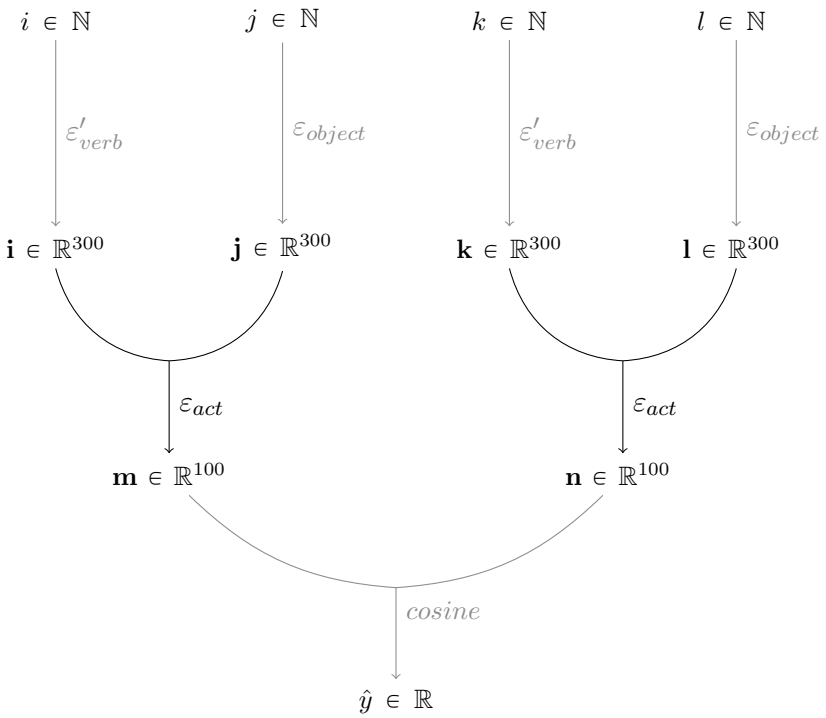

Figure 2: *Abstract view of the teacher network's data flow and its transformations. Grayed out arrows represent non-trainable functions.*

work, the solution space is so vast, interdependent and noisy that we have no quarantee of convergence.

This problem can be bypassed by breaking the optimization process down into two smaller, individually easier to solve, components; one relating to finding the action space vectors, and a second one for transfering this knowledge to the verb space matrices. To do so, we can treat the original network as a *student network*, supervised by an oracle, or *teacher network*, in a process called transfer learning.

To begin with, we can simplify things by assigning each verb a pretrained vector, in a manner similar to our object embedding function. We therefore can assume a new pretreained verb embedding function, that accepts a verb's identifier i and yields a vector rather $\mathbf{v}_i$:

$$\varepsilon'_{verb} : \mathbb{N} \to \mathbb{R}^{300}$$

A verb vector and an object vector can be vertically stacked to form an order-2 vector sequence $[\mathbf{v}, \mathbf{o}] \in \mathbb{R}^{2\times300}$. Such sequences are ideal inputs for recurrent neural networks, and we can construct one such network to process the sequence. The exact implementation is that of a long short-term memory autoencoder, which is standard practice in language related tasks. From a high level perspective, the autoencoder can be seen as an action embedding function:

$$\varepsilon_{act} : \mathbb{R}^{300} \times \mathbb{R}^{300} \to \mathbb{R}^{100}$$

The action embeddings of two phrases, as given by the autoencoder, can be contrasted with one another, again using cosine similarity, yielding a scalar prediction of paraphrasing. Visually, the oracle network can be viewed in Fig.2

Given free access over high-quality pretrained vectors for the verb embeddings significantly accelarates training. Compared to our original implementation, this network has far less trainable parameters and is thus much easier to train.

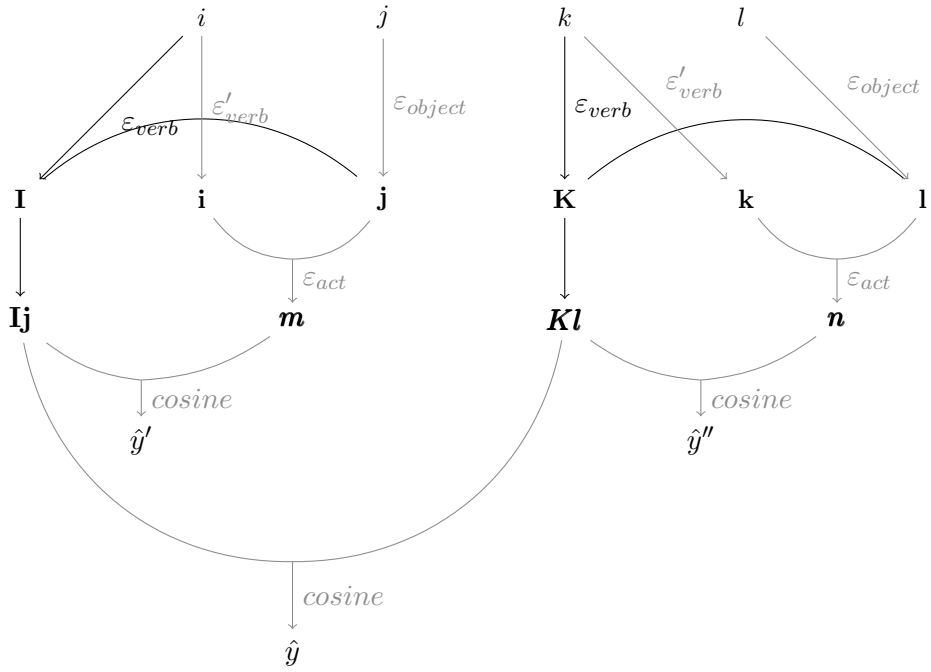

Figure 3: *Abstract view of the composition of the two networks. The cosine distance between the teacher's and the student's action embeddings yield two new predictions, $\hat{y}'$ and $\hat{y}''$, which should always be close to* 1*. Note that ε_{act} is grayed out, as its weights are now frozen.*

7 Composing Networks

The question now turns to utilizing the trained teacher's intermediate representations. Since it has managed to adequately solve the paraphrase detection problem, we can assume that its action embeddings are of high-quality and therefore could act as a good guide for the student network to follow.

First, the two networks are internally connected with one another, and the teacher network's weights are frozen; from now on the teacher can be treated as a fixed function. Then the student network's action embeddings are compared with the teacher's, giving us access to one new additive loss that needs to be minimized (or two in real-time, as the network is simultaneously operating and optimizing on two phrases):

$$\min_{\forall i,j} \mathcal{L}\{[cos(\mathbf{V}_i(\mathbf{o}_j), \varepsilon_{act}(i,j)], 1\}$$

Intuitively, we let the student train on its paraphrase detection objective, while also forcing it to construct action vectors that are as parallel to the teacher's as possible. Even though it may not always be able to do so (remember that we are trying to approximate a non-linear function with a linear one), the extra error signal pushes it away from local minima in the global error landscape. Since the oracle embeddings are already capable of predicting paraphrasing successfully, and the student is trained to mimic the oracle, it follows that it should also learn to predict paraphrasing. To negate the overabundance of positive labels induced by these extra error signals and force variation in the learned representations, we may progressively increase the ratio of negative over positive samples provided during the training process.

8 Technical Details

Recurrent Autoencoder Our ε_{act} is a neural function implemented by a long short-term memory unit followed by a fully connected layer. The LSTM accepts a vector sequence $[\mathbf{v}, \mathbf{o}]$ as its input and produces an output vector $\mathbf{h}_2$. Its operations are described by the following dynamic state equations:

$\mathbf{i}_1 = \sigma(\mathbf{W}_i[\mathbf{v};\mathbf{0}] + \boldsymbol{\beta}_i)$	(Input Gate : t=1)	$\mathbf{i}_2 = \sigma(\mathbf{W}_i[\mathbf{o};\mathbf{h}] + \boldsymbol{\beta}_i)$	(Input Gate : t=2)
$\mathbf{f}_1 = \sigma(W_f[\mathbf{v};\mathbf{0}] + \boldsymbol{\beta}_f)$	(Forget Gate : t=1)	$\mathbf{f}_2 = \sigma(\mathbf{W}_f[\mathbf{o};\mathbf{h}] + \boldsymbol{\beta}_f)$	(Forget Gate : t=2)
$\mathbf{c}_1 = \mathbf{i}_1 tanh(\mathbf{W}_c[\mathbf{v};\mathbf{0}] + \boldsymbol{\beta}_c)$	(Memory Cell: t=1)	$\mathbf{c}_2 = \mathbf{i}_2 tanh(\mathbf{W}_c[\mathbf{o};\boldsymbol{h}] + \boldsymbol{\beta}_c)$	(Memory Cell: t=2)
$\mathbf{g}_1 = \sigma(\mathbf{W}_o[\mathbf{v};\mathbf{0}] + \boldsymbol{\beta}_o)$	(Output Gate: t=1)	$\mathbf{g}_2 = \sigma(\mathbf{W}_o[\mathbf{o};\mathbf{h}] + \boldsymbol{\beta}_o)$	(Output Gate: t=2)
$\mathbf{h} = \mathbf{g}_1 tanh(\mathbf{c}_1)$	(Hidden Vector)	$\mathbf{h}_2 = \mathbf{g}_2 tanh(\boldsymbol{c}_2)$	(Output Vector)

Where $\mathbf{W}$, $\boldsymbol{\beta}$ denote trainable weight matrices and vectors, respectively, σ is the sigmoid function and $tanh$ the hyperbolic tangent. The output vector is set to be 300-dimensional so as to retain the original input vector size. It is then compressed via a fully-connected layer to our phrase embedding $\mathbf{p}$, whose dimensionaly we specify as 100:

$$\mathbf{p} = tanh(\mathbf{W}_p\mathbf{h}_2 + \boldsymbol{\beta}_p)$$

The choice of $tanh$ as the activation function here is important; the original GloVe vectors reside in $\mathbb{R}^{300}$, i.e. they may contain negative values. That means that when we obtain a phrase embedding through matrix-vector multiplication, it will also be in $\mathbb{R}^{300}$, therefore the autoencoder's output must match that space to remain compatible with the student network. Since $tanh$ has range $[-1, 1]$, it makes for a suitable choice.

Paraphrasing Prediction Two options arise when trying to predict paraphrasing from two action embeddings (regardless of their source). The first and simplest one would to compute their cosine distance and directly use the result as the probability of paraphrasing, to be optimized. This may seem appealing but it entails two problems. The cosine similarity of two real valued vectors has range $[-1, 1]$, whereas our labels take binary values $\{0, 1\}$. This means that:

1. Its continuous nature contradicts the discrete one of our labels, in turn necessitating its use as a probability measure.
2. Its negative range disallows it from directly fitting a probability distribution, as doing so would leave half its representational capacity unused.

An easy solution would be to simply normalize it to $[0, 1]$. However, it is more efficient to instead allow a more general, trainable transformation. A shallow feedforward network is used, implementing:

$$\hat{y} = \sigma(\mathbf{W}_y cos(\mathbf{p}_1, \mathbf{p}_2) + \beta_y)$$

Where $\mathbf{p}_1, \mathbf{p}_2$ our compared vectors. The impact of choice between normalization or arbitrary transformation seems negligible but is hard to concretely measure. It seems that the transformation allows for higher peak performance scores and yields more visually convincing end results.

Loss Function Another crucial choice is that of a loss function $\mathcal{L}$. The available options are binary crossentropy, mean square error and categorial hinge; each of which affects the optimization process in different ways. The most immediate choice would be binary crossentropy, since our training task is binary classification. However, it could be argued that the training task is just a means towards the construction of informative verb matrices, therefore we have limited interest in the classification performance itself. It would then make sense to use mean square error between the cosine similarity (normalized or otherwise) and the prediction label, as the cosine similarity space should itself be uniform continuous. A third alternative would be to use categorial hinge, which is essentially fitting the equivalent of a SVM kernel on top of our network, in attempt to maximize the separability of positive and negative paraphrastic predictions. Categorial hinge is unique in its ability to assign less importance to the exact value of a prediction, as long as it remains above(/below) the correct classification threshold; it has also been successfully used before in paraphrasing prediction tasks.

Quantifying the impact of the loss function is very hard, given its interactions with the rest of the design decisions and the stochastic nature of the process. From an experimental point of view, binary

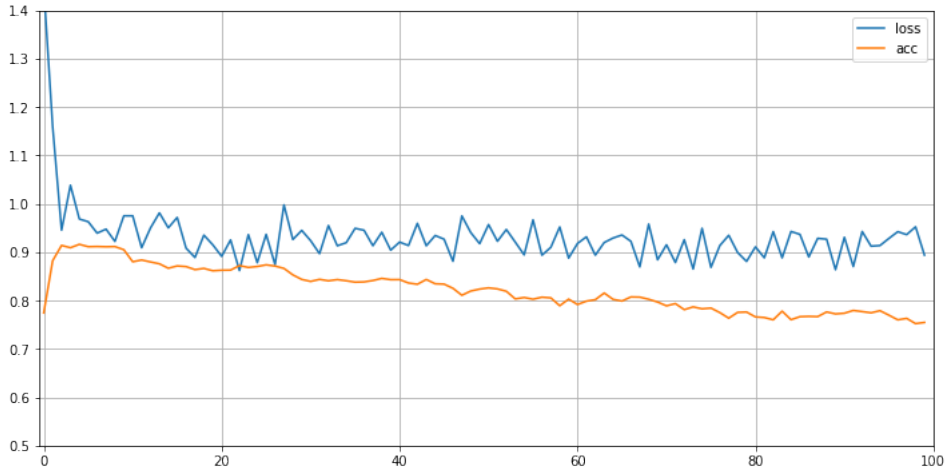

Figure 4: *Independent training of the student network with no additional supervision. Even though it appears to be learning fast during the first few epochs, its accuracy (orange line) quickly starts decreasing as it shown more samples. Notice the perturbations in the loss function (blue line), indicators of the noisy nature of the optimization landscape.*

crossentropy seems to yield higher performance scores when utilizing a regression layer on top of the cosine computation, whereas mean square error improves training speed on a simple normalized cosine. A weighted sum of the two apparently combines the best of both worlds and outperforms each of them individualy, as well as categorial hinge.

Optimization Algorithm The optimization is performed using Adam with a learning rate of 10^{-5}.

Negative Sampling As PPDB contains only paraphrastic pairs (i.e. positive examples), non-paraphrastic pairs (i.e. negative examples) have to be generated by random sampling. Two randomly selected phrases are assumed to be non-paraphrastic by default, unless they co-occur in a paraphrase pair or the both of them are mutually paraphrastic with respect to a third, shared phrase. Obviously this can potentially result to inaccuracies; for instance, two phrases may be semantically close but not be annotated as such, or their paraphrases may in turn be paraphrastic with one another (i.e. transitivity of the paraphrase relation). Measures may be taken towards alleviating the last issue, but in general the quality criterion for negative samples is not of extreme importance.

Curriculum Learning The ratio of negative samples against positive samples plays an important role in the end results. Especially prominent when using mean square error as the loss function, the network tends to aggressively fit any over-represented class in the training set. This can be negated by dynamically adapting the ratio throughout training. We may start with a class-balanced dataset, which allows the network to learn the majority of the positive examples. Then upon convergence, the percentage of negative examples can be increased by a small amount. Iteratively repeating the above steps significantly increases peak performance metrics.

9 Training & Evaluation

Figures 5 through 7 show the training curves of the networks. It is obvious that the teacher network is much easier to train than the student network, which in turn is positively affected by the additional supervision. The performance metrics of the teacher network and the (supervised) student can be seen in Table 2.

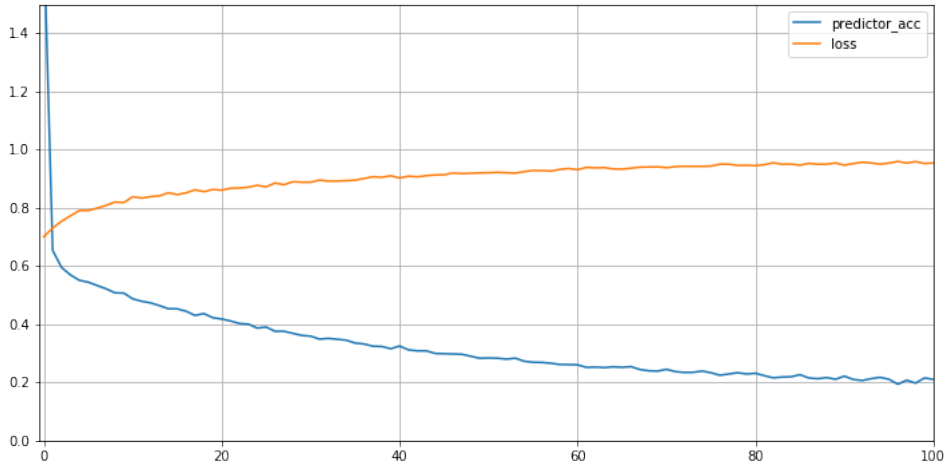

Figure 5: *Training of the teacher network. Both loss and accuracy improve smoothly over time until convergence.*

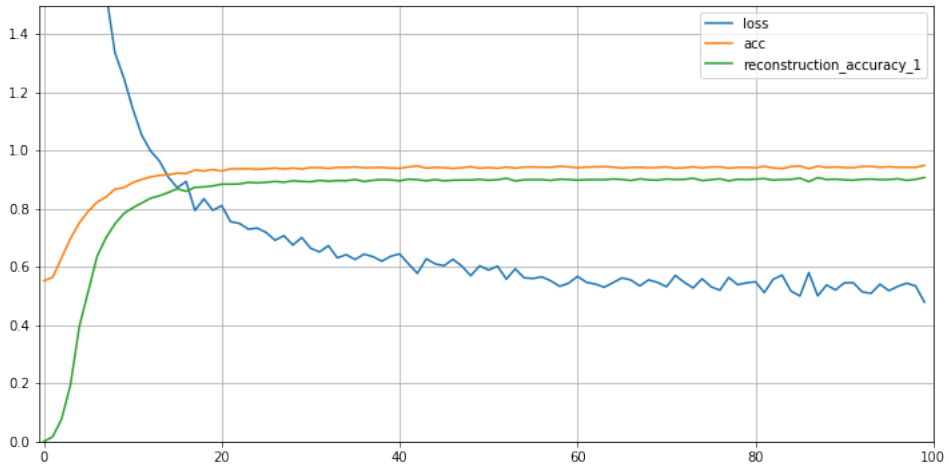

Figure 6: *Training of the student network under the teacher's supervision. Perturbations are still present, but are far less severe. Accuracy no longer diverges, and actually surpasses the percentage of agreement between teacher and student (green line).*

Metric \ Network	Teacher	Student
ACC	0.98	0.94
TPR	0.99	0.89
TNR	0.98	1

Table 2: *Performance metrics of the teacher and supervised student networks. Accuracy is measured at a class-balanced setting. TPR and TNR refer to the percentage of positive and negative samples correctly identified, respectively.*

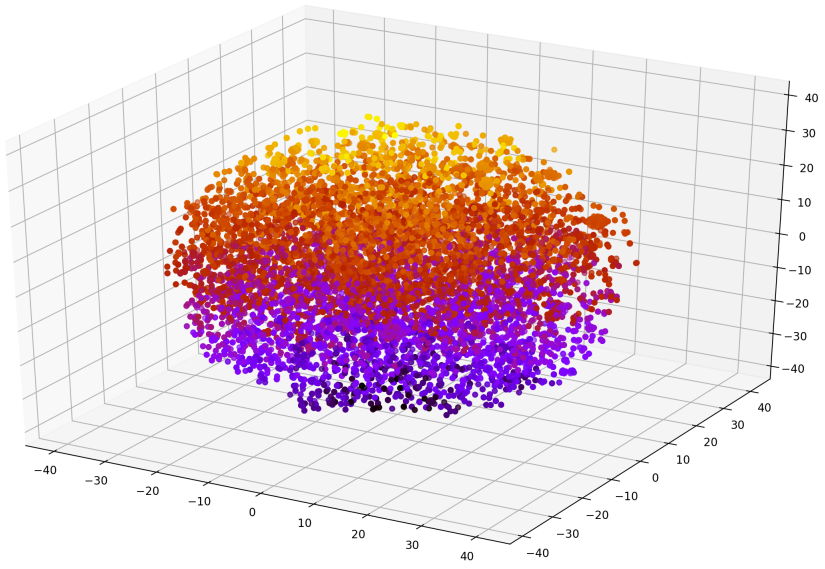

Figure 7: *Phrasal embeddings as produced by the supervised student, visualized with 3-dimensional tSNE. The embeddings uniformly occupy a spherical volume, which is the most volume-efficient 3d shape.*

Task-specific metrics give us an overview of the small-scale structure of the learned space, i.e. the pair-wise distances between vectors inhabiting this space.