A Primer on Neural Network Models for Natural Language Processing

Yoav Goldberg
Draft as of October 5, 2015.

The most up-to-date version of this manuscript is available at `http://www.cs.biu.ac.il/~yogo/nnlp.pdf`. Major updates will be published on arxiv periodically.
I welcome any comments you may have regarding the content and presentation. If you spot a missing reference or have relevant work you'd like to see mentioned, do let me know.
`first.last@gmail`

Abstract

Over the past few years, neural networks have re-emerged as powerful machine-learning models, yielding state-of-the-art results in fields such as image recognition and speech processing. More recently, neural network models started to be applied also to textual natural language signals, again with very promising results. This tutorial surveys neural network models from the perspective of natural language processing research, in an attempt to bring natural-language researchers up to speed with the neural techniques. The tutorial covers input encoding for natural language tasks, feed-forward networks, convolutional networks, recurrent networks and recursive networks, as well as the computation graph abstraction for automatic gradient computation.

1. Introduction

For a long time, core NLP techniques were dominated by machine-learning approaches that used linear models such as support vector machines or logistic regression, trained over very high dimensional yet very sparse feature vectors.

Recently, the field has seen some success in switching from such linear models over sparse inputs to non-linear neural-network models over dense inputs. While most of the neural network techniques are easy to apply, sometimes as almost drop-in replacements of the old linear classifiers, there is in many cases a strong barrier of entry. In this tutorial I attempt to provide NLP practitioners (as well as newcomers) with the basic background, jargon, tools and methodology that will allow them to understand the principles behind the neural network models and apply them to their own work. This tutorial is expected to be self-contained, while presenting the different approaches under a unified notation and framework. It repeats a lot of material which is available elsewhere. It also points to external sources for more advanced topics when appropriate.

This primer is not intended as a comprehensive resource for those that will go on and develop the next advances in neural-network machinery (though it may serve as a good entry point). Rather, it is aimed at those readers who are interested in taking the existing, useful technology and applying it in useful and creative ways to their favourite NLP problems. For more in-depth, general discussion of neural networks, the theory behind them, advanced

optimization methods and other advanced topics, the reader is referred to other existing resources. In particular, the book by Bengio et al (2015) is highly recommended.

Scope The focus is on applications of neural networks to language processing tasks. However, some subareas of language processing with neural networks were decidedly left out of scope of this tutorial. These include the vast literature of language modeling and acoustic modeling, the use of neural networks for machine translation, and multi-modal applications combining language and other signals such as images and videos (e.g. caption generation). Caching methods for efficient runtime performance, methods for efficient training with large output vocabularies and attention models are also not discussed. Word embeddings are discussed only to the extent that is needed to understand in order to use them as inputs for other models. Other unsupervised approaches, including autoencoders and recursive autoencoders, also fall out of scope. While some applications of neural networks for language modeling and machine translation are mentioned in the text, their treatment is by no means comprehensive.

A Note on Terminology The word "feature" is used to refer to a concrete, linguistic input such as a word, a suffix, or a part-of-speech tag. For example, in a first-order part-of-speech tagger, the features might be "current word, previous word, next word, previous part of speech". The term "input vector" is used to refer to the actual input that is fed to the neural-network classifier. Similarly, "input vector entry" refers to a specific value of the input. This is in contrast to a lot of the neural networks literature in which the word "feature" is overloaded between the two uses, and is used primarily to refer to an input-vector entry.

Mathematical Notation I use bold upper case letters to represent matrices ($\mathbf{X}$, $\mathbf{Y}$, $\mathbf{Z}$), and bold lower-case letters to represent vectors ($\mathbf{b}$). When there are series of related matrices and vectors (for example, where each matrix corresponds to a different layer in the network), superscript indices are used ($\mathbf{W^1}$, $\mathbf{W^2}$). For the rare cases in which we want indicate the power of a matrix or a vector, a pair of brackets is added around the item to be exponentiated: $(\mathbf{W})^2, (\mathbf{W^3})^2$. Unless otherwise stated, vectors are assumed to be row vectors. We use $[\mathbf{v_1}; \mathbf{v_2}]$ to denote vector concatenation.

2. Neural Network Architectures

Neural networks are powerful learning models. We will discuss two kinds of neural network architectures, that can be mixed and matched – feed-forward networks and Recurrent / Recursive networks. Feed-forward networks include networks with fully connected layers, such as the multi-layer perceptron, as well as networks with convolutional and pooling layers. All of the networks act as classifiers, but each with different strengths.

Fully connected feed-forward neural networks (Section 4) are non-linear learners that can, for the most part, be used as a drop-in replacement wherever a linear learner is used. This includes binary and multiclass classification problems, as well as more complex structured prediction problems (Section 8). The non-linearity of the network, as well as the ability to easily integrate pre-trained word embeddings, often lead to superior classification accuracy. A series of works (Chen & Manning, 2014; Weiss, Alberti, Collins, & Petrov, 2015; Pei, Ge, & Chang, 2015; Durrett & Klein, 2015) managed to obtain improved syntactic parsing results by simply replacing the linear model of a parser with a fully connected feed-forward network. Straight-forward applications of a feed-forward network as a classifier replacement (usually coupled with the use of pre-trained word vectors) provide benefits also for CCG supertagging (Lewis & Steedman, 2014), dialog state tracking (Henderson, Thomson, & Young, 2013), pre-ordering for statistical machine translation (de Gispert, Iglesias, & Byrne, 2015) and language modeling (Bengio, Ducharme, Vincent, & Janvin, 2003; Vaswani, Zhao, Fossum, & Chiang, 2013). Iyyer et al (2015) demonstrate that multi-layer feed-forward networks can provide competitive results on sentiment classification and factoid question answering.

Networks with convolutional and pooling layers (Section 9) are useful for classification tasks in which we expect to find strong local clues regarding class membership, but these clues can appear in different places in the input. For example, in a document classification task, a single key phrase (or an ngram) can help in determining the topic of the document (Johnson & Zhang, 2015). We would like to learn that certain sequences of words are good indicators of the topic, and do not necessarily care where they appear in the document. Convolutional and pooling layers allow the model to learn to find such local indicators, regardless of their position. Convolutional and pooling architecture show promising results on many tasks, including document classification (Johnson & Zhang, 2015), short-text categorization (Wang, Xu, Xu, Liu, Zhang, Wang, & Hao, 2015a), sentiment classification (Kalchbrenner, Grefenstette, & Blunsom, 2014; Kim, 2014), relation type classification between entities (Zeng, Liu, Lai, Zhou, & Zhao, 2014; dos Santos, Xiang, & Zhou, 2015), event detection (Chen, Xu, Liu, Zeng, & Zhao, 2015; Nguyen & Grishman, 2015), paraphrase identification (Yin & Schütze, 2015) semantic role labeling (Collobert, Weston, Bottou, Karlen, Kavukcuoglu, & Kuksa, 2011), question answering (Dong, Wei, Zhou, & Xu, 2015), predicting box-office revenues of movies based on critic reviews (Bitvai & Cohn, 2015) modeling text interestingness (Gao, Pantel, Gamon, He, & Deng, 2014), and modeling the relation between character-sequences and part-of-speech tags (Santos & Zadrozny, 2014).

In natural language we often work with structured data of arbitrary sizes, such as sequences and trees. We would like to be able to capture regularities in such structures, or to model similarities between such structures. In many cases, this means encoding the structure as a fixed width vector, which we can then pass on to another statistical

learner for further processing. While convolutional and pooling architectures allow us to encode arbitrary large items as fixed size vectors capturing their most salient features, they do so by sacrificing most of the structural information. Recurrent (Section 10) and recursive (Section 12) architectures, on the other hand, allow us to work with sequences and trees while preserving a lot of the structural information. Recurrent networks (Elman, 1990) are designed to model sequences, while recursive networks (Goller & Küchler, 1996) are generalizations of recurrent networks that can handle trees. We will also discuss an extension of recurrent networks that allow them to model stacks (Dyer, Ballesteros, Ling, Matthews, & Smith, 2015; Watanabe & Sumita, 2015).

Recurrent models have been shown to produce very strong results for language modeling, including (Mikolov, Karafiát, Burget, Cernocky, & Khudanpur, 2010; Mikolov, Kombrink, Lukáš Burget, Černocky, & Khudanpur, 2011; Mikolov, 2012; Duh, Neubig, Sudoh, & Tsukada, 2013; Adel, Vu, & Schultz, 2013; Auli, Galley, Quirk, & Zweig, 2013; Auli & Gao, 2014); as well as for sequence tagging (Irsoy & Cardie, 2014; Xu, Auli, & Clark, 2015; Ling, Dyer, Black, Trancoso, Fermandez, Amir, Marujo, & Luis, 2015b), machine translation (Sundermeyer, Alkhouli, Wuebker, & Ney, 2014; Tamura, Watanabe, & Sumita, 2014; Sutskever, Vinyals, & Le, 2014; Cho, van Merrienboer, Gulcehre, Bahdanau, Bougares, Schwenk, & Bengio, 2014b), dependency parsing (Dyer et al., 2015; Watanabe & Sumita, 2015), sentiment analysis (Wang, Liu, SUN, Wang, & Wang, 2015b), noisy text normalization (Chrupala, 2014), dialog state tracking (Mrkšić, Ó Séaghdha, Thomson, Gasic, Su, Vandyke, Wen, & Young, 2015), response generation (Sordoni, Galley, Auli, Brockett, Ji, Mitchell, Nie, Gao, & Dolan, 2015), and modeling the relation between character sequences and part-of-speech tags (Ling et al., 2015b).

Recursive models were shown to produce state-of-the-art or near state-of-the-art results for constituency (Socher, Bauer, Manning, & Andrew Y., 2013) and dependency (Le & Zuidema, 2014; Zhu, Qiu, Chen, & Huang, 2015a) parse re-ranking, discourse parsing (Li, Li, & Hovy, 2014), semantic relation classification (Hashimoto, Miwa, Tsuruoka, & Chikayama, 2013; Liu, Wei, Li, Ji, Zhou, & WANG, 2015), political ideology detection based on parse trees (Iyyer, Enns, Boyd-Graber, & Resnik, 2014b), sentiment classification (Socher, Perelygin, Wu, Chuang, Manning, Ng, & Potts, 2013; Hermann & Blunsom, 2013), target-dependent sentiment classification (Dong, Wei, Tan, Tang, Zhou, & Xu, 2014) and question answering (Iyyer, Boyd-Graber, Claudino, Socher, & Daumé III, 2014a).

3. Feature Representation

Before discussing the network structure in more depth, it is important to pay attention to how features are represented. For now, we can think of a feed-forward neural network as a function $NN(\mathbf{x})$ that takes as input a d_{in} dimensional vector $\mathbf{x}$ and produces a d_{out} dimensional output vector. The function is often used as a *classifier*, assigning the input $\mathbf{x}$ a degree of membership in one or more of d_{out} classes. The function can be complex, and is almost always non-linear. Common structures of this function will be discussed in Section 4. Here, we focus on the input, $\mathbf{x}$. When dealing with natural language, the input $\mathbf{x}$ encodes features such as words, part-of-speech tags or other linguistic information. Perhaps the biggest jump when moving from sparse-input linear models to neural-network based models is to stop representing each feature as a unique dimension (the so called one-hot representation) and representing them instead as dense vectors. That is, each core feature is *embedded* into a d dimensional space, and represented as a vector in that space.[1] The embeddings (the vector representation of each core feature) can then be trained like the other parameter of the function NN. Figure 1 shows the two approaches to feature representation.

The feature embeddings (the values of the vector entries for each feature) are treated as *model parameters* that need to be trained together with the other components of the network. Methods of training (or obtaining) the feature embeddings will be discussed later. For now, consider the feature embeddings as given.

The general structure for an NLP classification system based on a feed-forward neural network is thus:

1. Extract a set of core linguistic features $f_1, \ldots, f_k$ that are relevant for predicting the output class.
2. For each feature f_i of interest, retrieve the corresponding vector $v(f_i)$.
3. Combine the vectors (either by concatenation, summation or a combination of both) into an input vector $\mathbf{x}$.
4. Feed $\mathbf{x}$ into a non-linear classifier (feed-forward neural network).

The biggest change in the input, then, is the move from sparse representations in which each feature is its own dimension, to a dense representation in which each feature is mapped to a vector. Another difference is that we extract only *core features* and not feature combinations. We will elaborate on both these changes briefly.

Dense Vectors vs. One-hot Representations What are the benefits of representing our features as vectors instead of as unique IDs? Should we always represent features as dense vectors? Let's consider the two kinds of representations:

One Hot Each feature is its own dimension.

- Dimensionality of one-hot vector is same as number of distinct features.

1. Different feature types may be embedded into different spaces. For example, one may represent word features using 100 dimensions, and part-of-speech features using 20 dimensions.

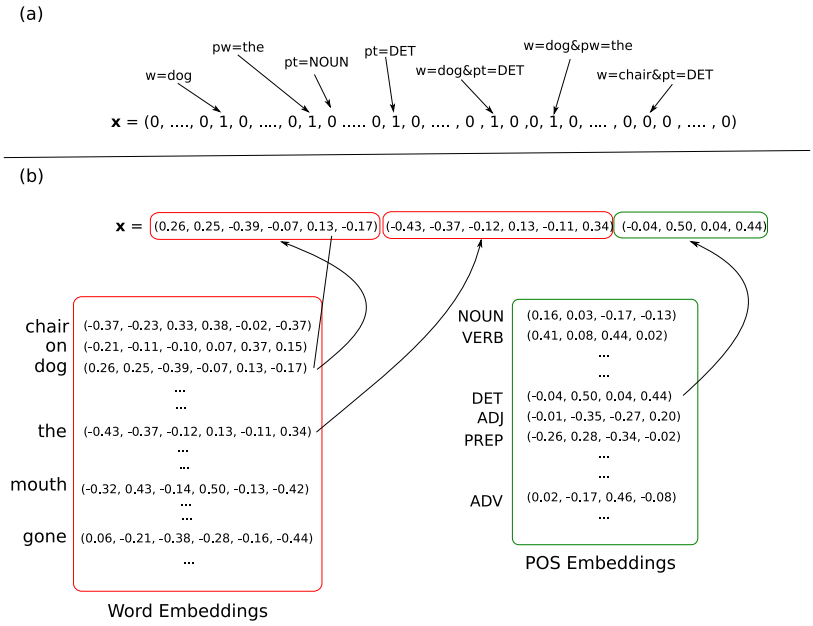


Figure 1: **Sparse vs. dense feature representations**. Two encodings of the information: *current word is "dog"; previous word is "the"; previous pos-tag is "DET"*. (a) Sparse feature vector. Each dimension represents a feature. Feature combinations receive their own dimensions. Feature values are binary. Dimensionality is very high. (b) Dense, embeddings-based feature vector. Each core feature is represented as a vector. Each feature corresponds to several input vector entries. No explicit encoding of feature combinations. Dimensionality is low. The feature-to-vector mappings come from an embedding table.

- Features are completely independent from one another. The feature "word is 'dog' " is as dis-similar to "word is 'thinking' " than it is to "word is 'cat' ".

Dense Each feature is a d-dimensional vector.

- Dimensionality of vector is d.
- Similar features will have similar vectors – information is shared between similar features.

One benefit of using dense and low-dimensional vectors is computational: the majority of neural network toolkits do not play well with very high-dimensional, sparse vectors. However, this is just a technical obstacle, which can be resolved with some engineering effort.

The main benefit of the dense representations is in generalization power: if we believe some features may provide similar clues, it is worthwhile to provide a representation that is able to capture these similarities. For example, assume we have observed the word 'dog' many times during training, but only observed the word 'cat' a handful of times, or not at

all. If each of the words is associated with its own dimension, occurrences of 'dog' will not tell us anything about the occurrences of 'cat'. However, in the dense vectors representation the learned vector for 'dog' may be similar to the learned vector from 'cat', allowing the model to share statistical strength between the two events. This argument assumes that "good" vectors are somehow given to us. Section 5 describes ways of obtaining such vector representations.

In cases where we have relatively few distinct features in the category, and we believe there are no correlations between the different features, we may use the one-hot representation. However, if we believe there are going to be correlations between the different features in the group (for example, for part-of-speech tags, we may believe that the different verb inflections `VB` and `VBZ` may behave similarly as far as our task is concerned) it may be worthwhile to let the network figure out the correlations and gain some statistical strength by sharing the parameters. It may be the case that under some circumstances, when the feature space is relatively small and the training data is plentiful, or when we do not wish to share statistical information between distinct words, there are gains to be made from using the one-hot representations. However, this is still an open research question, and there are no strong evidence to either side. The majority of work (pioneered by (Collobert & Weston, 2008; Collobert et al., 2011; Chen & Manning, 2014)) advocate the use of dense, trainable embedding vectors for all features. For work using neural network architecture with sparse vector encodings see (Johnson & Zhang, 2015).

Finally, it is important to note that representing features as dense vectors is an integral part of the neural network framework, and that consequentially the differences between using sparse and dense feature representations are subtler than they may appear at first. In fact, using sparse, one-hot vectors as input when training a neural network amounts to dedicating the first layer of the network to learning a dense embedding vector for each feature based on the training data. We touch on this in Section 4.4.

Variable Number of Features: Continuous Bag of Words Feed-forward networks assume a fixed dimensional input. This can easily accommodate the case of a feature-extraction function that extracts a fixed number of features: each feature is represented as a vector, and the vectors are concatenated. This way, each region of the resulting input vector corresponds to a different feature. However, in some cases the number of features is not known in advance (for example, in document classification it is common that each word in the sentence is a feature). We thus need to represent an unbounded number of features using a fixed size vector. One way of achieving this is through a so-called *continuous bag of words* (CBOW) representation (Mikolov, Chen, Corrado, & Dean, 2013). The CBOW is very similar to the traditional bag-of-words representation in which we discard order information, and works by either summing or averaging the embedding vectors of the corresponding features:[2]

2. Note that if the $v(f_i)$s were one-hot vectors rather than dense feature representations, the $CBOW$ and $WCBOW$ equations above would reduce to the traditional (weighted) bag-of-words representations, which is in turn equivalent to a sparse feature-vector representation in which each binary indicator feature corresponds to a unique "word".

$$CBOW(f_1, ..., f_k) = \frac{1}{k} \sum_{i=1}^{k} v(f_i)$$

A simple variation on the CBOW representation is weighted CBOW, in which different vectors receive different weights:

$$WCBOW(f_1, ..., f_k) = \frac{1}{\sum_{i=1}^{k} a_i} \sum_{i=1}^{k} a_i v(f_i)$$

Here, each feature f_i has an associated weight a_i, indicating the relative importance of the feature. For example, in a document classification task, a feature f_i may correspond to a word in the document, and the associated weight a_i could be the word's TF-IDF score.

Distance and Position Features The linear distance in between two words in a sentence may serve as an informative feature. For example, in an event extraction task[3] we may be given a trigger word and a candidate argument word, and asked to predict if the argument word is indeed an argument of the trigger. The distance (or relative position) between the trigger and the argument is a strong signal for this prediction task. In the "traditional" NLP setup, distances are usually encoded by binning the distances into several groups (i.e. 1, 2, 3, 4, 5–10, 10+) and associating each bin with a one-hot vector. In a neural architecture, where the input vector is not composed of binary indicator features, it may seem natural to allocate a single input vector entry to the distance feature, where the numeric value of that entry is the distance. However, this approach is not taken in practice. Instead, distance features are encoded similarly to the other feature types: each bin is associated with a d-dimensional vector, and these distance-embedding vectors are then trained as regular parameters in the network (Zeng et al., 2014; dos Santos et al., 2015; Zhu et al., 2015a; Nguyen & Grishman, 2015).

Feature Combinations Note that the feature extraction stage in the neural-network settings deals only with extraction of *core* features. This is in contrast to the traditional linear-model-based NLP systems in which the feature designer had to manually specify not only the core features of interests but also interactions between them (e.g., introducing not only a feature stating "word is X" and a feature stating "tag is Y" but also combined feature stating "word is X and tag is Y" or sometimes even "word is X, tag is Y and previous word is Z"). The combination features are crucial in linear models because they introduce more dimensions to the input, transforming it into a space where the data-points are closer to being linearly separable. On the other hand, the space of possible combinations is very large, and the feature designer has to spend a lot of time coming up with an effective set of feature combinations. One of the promises of the non-linear neural network models is that one needs to define only the core features. The non-linearity of the classifier, as defined by the network structure, is expected to take care of finding the indicative feature combinations, alleviating the need for feature combination engineering.

3. The event extraction task involves identification of events from a predefined set of event types. For example identification of "purchase" events or "terror-attack" events. Each event type can be triggered by various triggering words (commonly verbs), and has several slots (arguments) that needs to be filled (i.e. who purchased? what was purchased? at what amount?).

Kernel methods (Shawe-Taylor & Cristianini, 2004), and in particular polynomial kernels (Kudo & Matsumoto, 2003), also allow the feature designer to specify only core features, leaving the feature combination aspect to the learning algorithm. In contrast to neural-network models, kernels methods are convex, admitting exact solutions to the optimization problem. However, the classification efficiency in kernel methods scales linearly with the size of the training data, making them too slow for most practical purposes, and not suitable for training with large datasets. On the other hand, neural network classification efficiency scales linearly with the size of the network, regardless of the training data size.

Dimensionality How many dimensions should we allocate for each feature? Unfortunately, there are no theoretical bounds or even established best-practices in this space. Clearly, the dimensionality should grow with the number of the members in the class (you probably want to assign more dimensions to word embeddings than to part-of-speech embeddings) but how much is enough? In current research, the dimensionality of word-embedding vectors range between about 50 to a few hundreds, and, in some extreme cases, thousands. Since the dimensionality of the vectors has a direct effect on memory requirements and processing time, a good rule of thumb would be to experiment with a few different sizes, and choose a good trade-off between speed and task accuracy.

Vector Sharing Consider a case where you have a few features that share the same vocabulary. For example, when assigning a part-of-speech to a given word, we may have a set of features considering the previous word, and a set of features considering the next word. When building the input to the classifier, we will concatenate the vector representation of the previous word to the vector representation of the next word. The classifier will then be able to distinguish the two different indicators, and treat them differently. But should the two features share the same vectors? Should the vector for "dog:previous-word" be the same as the vector of "dog:next-word"? Or should we assign them two distinct vectors? This, again, is mostly an empirical question. If you believe words behave differently when they appear in different positions (e.g., word X behaves like word Y when in the previous position, but X behaves like Z when in the next position) then it may be a good idea to use two different vocabularies and assign a different set of vectors for each feature type. However, if you believe the words behave similarly in both locations, then something may be gained by using a shared vocabulary for both feature types.

Network's Output For multi-class classification problems with k classes, the network's output is a k-dimensional vector in which every dimension represents the strength of a particular output class. That is, the output remains as in the traditional linear models – scalar scores to items in a discrete set. However, as we will see in Section 4, there is a $d \times k$ matrix associated with the output layer. The columns of this matrix can be thought of as d dimensional embeddings of the output classes. The vector similarities between the vector representations of the k classes indicate the model's learned similarities between the output classes.

Historical Note Representing words as dense vectors for input to a neural network was introduced by Bengio et al (Bengio et al., 2003) in the context of neural language modeling. It was introduced to NLP tasks in the pioneering work of Collobert, Weston and colleagues

(2008, 2011). Using embeddings for representing not only words but arbitrary features was popularized following Chen and Manning (2014).

4. Feed-forward Neural Networks

A Brain-inspired metaphor As the name suggest, neural-networks are inspired by the brain's computation mechanism, which consists of computation units called neurons. In the metaphor, a neuron is a computational unit that has scalar inputs and outputs. Each input has an associated weight. The neuron multiplies each input by its weight, and then sums[4] them, applies a non-linear function to the result, and passes it to its output. The neurons are connected to each other, forming a network: the output of a neuron may feed into the inputs of one or more neurons. Such networks were shown to be very capable computational devices. If the weights are set correctly, a neural network with enough neurons and a non-linear activation function can approximate a very wide range of mathematical functions (we will be more precise about this later).

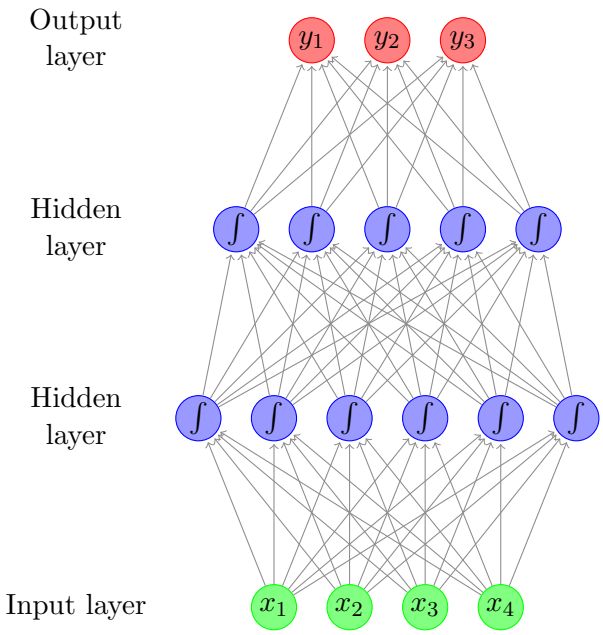


Figure 2: Feed-forward neural network with two hidden layers.

A typical feed-forward neural network may be drawn as in Figure 2. Each circle is a neuron, with incoming arrows being the neuron's inputs and outgoing arrows being the neuron's outputs. Each arrow carries a weight, reflecting its importance (not shown). Neurons are arranged in layers, reflecting the flow of information. The bottom layer has no incoming arrows, and is the input to the network. The top-most layer has no outgoing arrows, and is the output of the network. The other layers are considered "hidden". The sigmoid shape inside the neurons in the middle layers represent a non-linear function (typically a $1/(1+e^{-x})$) that is applied to the neuron's value before passing it to the output. In the figure, each neuron is connected to all of the neurons in the next layer – this is called a *fully-connected layer* or an *affine layer*.

4. While summing is the most common operation, other functions, such as a max, are also possible

While the brain metaphor is sexy and intriguing, it is also distracting and cumbersome to manipulate mathematically. We therefore switch to using more concise mathematic notation. The values of each row of neurons in the network can be thought of as a vector. In Figure 2 the input layer is a 4 dimensional vector ($\mathbf{x}$), and the layer above it is a 6 dimensional vector ($\mathbf{h^1}$). The fully connected layer can be thought of as a linear transformation from 4 dimensions to 6 dimensions. A fully-connected layer implements a vector-matrix multiplication, $\mathbf{h} = \mathbf{xW}$ where the weight of the connection from the ith neuron in the input row to the jth neuron in the output row is W_{ij}.[5] The values of $\mathbf{h}$ are then transformed by a non-linear function g that is applied to each value before being passed on to the next input. The whole computation from input to output can be written as: $(g(\mathbf{xW^1}))\mathbf{W^2}$ where $\mathbf{W^1}$ are the weights of the first layer and $\mathbf{W^2}$ are the weights of the second one.

In Mathematical Notation From this point on, we will abandon the brain metaphor and describe networks exclusively in terms of vector-matrix operations.
The simplest neural network is the perceptron, which is a linear function of its inputs:

$$NN_{Perceptron}(\mathbf{x}) = \mathbf{xW} + \mathbf{b}$$
$$\mathbf{x} \in \mathbb{R}^{d_{in}},\ \ \mathbf{W} \in \mathbb{R}^{d_{in} \times d_{out}},\ \ \mathbf{b} \in \mathbb{R}^{d_{out}}$$

$\mathbf{W}$ is the weight matrix, and $\mathbf{b}$ is a bias term.[6] In order to go beyond linear functions, we introduce a non-linear hidden layer (the network in Figure 2 has two such layers), resulting in the 1-layer Multi Layer Perceptron (MLP1). A one-layer feed-forward neural network has the form:

$$NN_{MLP1}(\mathbf{x}) = g(\mathbf{xW^1} + \mathbf{b^1})\mathbf{W^2} + \mathbf{b^2}$$
$$\mathbf{x} \in \mathbb{R}^{d_{in}},\ \ \mathbf{W^1} \in \mathbb{R}^{d_{in} \times d_1},\ \ \mathbf{b^1} \in \mathbb{R}^{d_1},\ \ \mathbf{W^2} \in \mathbb{R}^{d_1 \times d_2},\ \ \mathbf{b^2} \in \mathbb{R}^{d_2}$$

Here $\mathbf{W^1}$ and $\mathbf{b^1}$ are a matrix and a bias term for the first linear transformation of the input, g is a non-linear function that is applied element-wise (also called a *non-linearity* or an *activation function*), and $\mathbf{W^2}$ and $\mathbf{b^2}$ are the matrix and bias term for a second linear transform.

Breaking it down, $\mathbf{xW^1}+\mathbf{b^1}$ is a linear transformation of the input $\mathbf{x}$ from d_{in} dimensions to d_1 dimensions. g is then applied to each of the d_1 dimensions, and the matrix $\mathbf{W^2}$ together with bias vector $\mathbf{b}^2$ are then used to transform the result into the d_2 dimensional output vector. The non-linear activation function g has a crucial role in the network's ability to represent complex functions. Without the non-linearity in g, the neural network can only represent linear transformations of the input.[7]

We can add additional linear-transformations and non-linearities, resulting in a 2-layer MLP (the network in Figure 2 is of this form):

$$NN_{MLP2}(\mathbf{x}) = (g^2(g^1(\mathbf{xW^1} + \mathbf{b^1})\mathbf{W^2} + \mathbf{b^2}))\mathbf{W^3}$$

5. To see why this is the case, denote the weight of the ith input of the jth neuron in $\mathbf{h}$ as w_{ij}. The value of h_j is then $h_j = \sum_{i=1}^{4} x_i \cdot w_{ij}$.
6. The network in figure 2 does not include bias terms. A bias term can be added to a layer by adding to it an additional neuron that does not have any incoming connections, whose value is always 1.
7. To see why, consider that a sequence of linear transformations is still a linear transformation.

It is perhaps clearer to write deeper networks like this using intermediary variables:

$$\begin{aligned} NN_{MLP2}(\mathbf{x}) =& \mathbf{y} \\ \mathbf{h^1} =& g^1(\mathbf{x}\mathbf{W^1} + \mathbf{b^1}) \\ \mathbf{h^2} =& g^2(\mathbf{h^1}\mathbf{W^2} + \mathbf{b^2}) \\ \mathbf{y} =& \mathbf{h^2}\mathbf{W^3} \end{aligned}$$

The vector resulting from each linear transform is referred to as a *layer*. The outer-most linear transform results in the *output layer* and the other linear transforms result in *hidden layers*. Each hidden layer is followed by a non-linear activation. In some cases, such as in the last layer of our example, the bias vectors are forced to 0 ("dropped").

Layers resulting from linear transformations are often referred to as *fully connected*, or *affine*. Other types of architectures exist. In particular, image recognition problems benefit from *convolutional* and *pooling* layers. Such layers have uses also in language processing, and will be discussed in Section 9. Networks with more than one hidden layer are said to be *deep* networks, hence the name *deep learning*.

When describing a neural network, one should specify the *dimensions* of the layers and the input. A layer will expect a d_{in} dimensional vector as its input, and transform it into a d_{out} dimensional vector. The dimensionality of the layer is taken to be the dimensionality of its output. For a fully connected layer $l(\mathbf{x}) = \mathbf{x}\mathbf{W} + \mathbf{b}$ with input dimensionality d_{in} and output dimensionality d_{out}, the dimensions of $\mathbf{x}$ is $1 \times d_{in}$, of $\mathbf{W}$ is $d_{in} \times d_{out}$ and of $\mathbf{b}$ is $1 \times d_{out}$.

The output of the network is a d_{out} dimensional vector. In case $d_{out} = 1$, the network's output is a scalar. Such networks can be used for regression (or scoring) by considering the value of the output, or for binary classification by consulting the sign of the output. Networks with $d_{out} = k > 1$ can be used for k-class classification, by associating each dimension with a class, and looking for the dimension with maximal value. Similarly, if the output vector entries are positive and sum to one, the output can be interpreted as a distribution over class assignments (such output normalization is typically achieved by applying a softmax transformation on the output layer, see Section 4.3).

The matrices and the bias terms that define the linear transformations are the *parameters* of the network. It is common to refer to the collection of all parameters as θ. Together with the input, the parameters determine the network's output. The training algorithm is responsible for setting their values such that the network's predictions are correct. Training is discussed in Section 6.

4.1 Representation Power

In terms of representation power, it was shown by (Hornik, Stinchcombe, & White, 1989; Cybenko, 1989) that MLP1 is a universal approximator – it can approximate with any desired non-zero amount of error a family of functions[8] that include all continuous functions

8. Specifically, a feed-forward network with linear output layer and at least one hidden layer with a "squashing" activation function can approximate any Borel measurable function from one finite dimensional space to another.

on a closed and bounded subset of $\mathbb{R}^n$, and any function mapping from any finite dimensional discrete space to another. This may suggest there is no reason to go beyond MLP1 to more complex architectures. However, the theoretical result does not state how large the hidden layer should be, nor does it say anything about the learnability of the neural network (it states that a representation exists, but does not say how easy or hard it is to set the parameters based on training data and a specific learning algorithm). It also does not guarantee that a training algorithm will find the *correct* function generating our training data. Since in practice we train neural networks on relatively small amounts of data, using a combination of the backpropagation algorithm and variants of stochastic gradient descent, and use hidden layers of relatively modest sizes (up to several thousands), there is benefit to be had in trying out more complex architectures than MLP1. In many cases, however, MLP1 does indeed provide very strong results. For further discussion on the representation power of feed-forward neural networks, see (Bengio et al., 2015, Section 6.5).

4.2 Common Non-linearities

The non-linearity g can take many forms. There is currently no good theory as to which non-linearity to apply in which conditions, and choosing the correct non-linearity for a given task is for the most part an empirical question. I will now go over the common non-linearities from the literature: the sigmoid, tanh, hard tanh and the rectified linear unit (ReLU). Some NLP researchers also experimented with other forms of non-linearities such as cube and tanh-cube.

Sigmoid The sigmoid activation function $\sigma(x) = 1/(1 + e^{-x})$ is an S-shaped function, transforming each value x into the range $[0, 1]$.

Hyperbolic tangent (tanh) The hyperbolic tangent $tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$ activation function is an S-shaped function, transforming the values x into the range $[-1, 1]$.

Hard tanh The hard-tanh activation function is an approximation of the *tanh* function which is faster to compute and take derivatives of:

$$hardtanh(x) = \begin{cases} -1 & x < -1 \\ 1 & x > 1 \\ x & \text{otherwise} \end{cases}$$

Rectifier (ReLU) The Rectifier activation function (Glorot, Bordes, & Bengio, 2011), also known as the rectified linear unit is a very simple activation function that is easy to work with and was shown many times to produce excellent results.[9] The ReLU unit clips each value $x < 0$ at 0. Despite its simplicity, it performs well for many tasks, especially when combined with the dropout regularization technique (see Section 6.4).

9. The technical advantages of the ReLU over the sigmoid and tanh activation functions is that it does not involve expensive-to-compute functions, and more importantly that it does not saturate. The sigmoid and tanh activation are capped at 1, and the gradients at this region of the functions are near zero, driving the entire gradient near zero. The ReLU activation does not have this problem, making it especially suitable for networks with multiple layers, which are susceptible to the vanishing gradients problem when trained with the saturating units.

$$ReLU(x) = \max(0, x) = \begin{cases} 0 & x < 0 \\ x & \text{otherwise} \end{cases}$$

As a rule of thumb, ReLU units work better than tanh, and tanh works better than sigmoid.[10]

4.3 Output Transformations

In many cases, the output layer vector is also transformed. A common transformation is the *softmax*:

$$\begin{aligned} \mathbf{x} =& x_1, \ldots, x_k \\ softmax(x_i) =& \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}} \end{aligned}$$

The result is a vector of non-negative real numbers that sum to one, making it a discrete probability distribution over k possible outcomes.

The $softmax$ output transformation is used when we are interested in modeling a probability distribution over the possible output classes. To be effective, it should be used in conjunction with a probabilistic training objective such as cross-entropy (see Section 4.5 below).

When the softmax transformation is applied to the output of a network without a hidden layer, the result is the well known multinomial logistic regression model, also known as a maximum-entropy classifier.

4.4 Embedding Layers

Up until now, the discussion ignored the source of $\mathbf{x}$, treating it as an arbitrary vector. In an NLP application, $\mathbf{x}$ is usually composed of various embeddings vectors. We can be explicit about the source of $\mathbf{x}$, and include it in the network's definition. We introduce $c(\cdot)$, a function from core features to an input vector.

It is common for c to extract the embedding vector associated with each feature, and concatenate them:

10. In addition to these activation functions, recent works from the NLP community experiment with and reported success with other forms of non-linearities. The **Cube** activation function, $g(x) = (x)^3$, was suggested by (Chen & Manning, 2014), who found it to be more effective than other non-linearities in a feed-forward network that was used to predict the actions in a greedy transition-based dependency parser. The **tanh cube** activation function $g(x) = \tanh((x)^3 + x)$ was proposed by (Pei et al., 2015), who found it to be more effective than other non-linearities in a feed-forward network that was used as a component in a structured-prediction graph-based dependency parser.

    The cube and tanh-cube activation functions are motivated by the desire to better capture interactions between different features. While these activation functions are reported to improve performance in certain situations, their general applicability is still to be determined.

$$\begin{aligned}\mathbf{x} = c(f_1, f_2, f_3) =& [v(f_1); v(f_2); v(f_3)] \\ NN_{MLP1}(\mathbf{x}) =& NN_{MLP1}(c(f_1, f_2, f_3)) \\ =& NN_{MLP1}([v(f_1); v(f_2); v(f_3)]) \\ =& (g([v(f_1); v(f_2); v(f_3)]\mathbf{W^1} + \mathbf{b^1}))\mathbf{W^2} + \mathbf{b^2}\end{aligned}$$

Another common choice is for c to sum the embedding vectors (this assumes the embedding vectors all share the same dimensionality):

$$\begin{aligned}\mathbf{x} = c(f_1, f_2, f_3) =& v(f_1) + v(f_2) + v(f_3) \\ NN_{MLP1}(\mathbf{x}) =& NN_{MLP1}(c(f_1, f_2, f_3)) \\ =& NN_{MLP1}(v(f_1) + v(f_2) + v(f_3)) \\ =& (g((v(f_1) + v(f_2) + v(f_3))\mathbf{W^1} + \mathbf{b^1}))\mathbf{W^2} + \mathbf{b^2}\end{aligned}$$

The form of c is an essential part of the network's design. In many papers, it is common to refer to c as part of the network, and likewise treat the word embeddings $v(f_i)$ as resulting from an "embedding layer" or "lookup layer". Consider a vocabulary of $|V|$ words, each embedded as a d dimensional vector. The collection of vectors can then be thought of as a $|V| \times d$ embedding matrix $\mathbf{E}$ in which each row corresponds to an embedded feature. Let $\mathbf{f_i}$ be a $|V|$-dimensional vector, which is all zeros except from one index, corresponding to the value of the ith feature, in which the value is 1 (this is called a one-hot vector). The multiplication $\mathbf{f_i E}$ will then "select" the corresponding row of $\mathbf{E}$. Thus, $v(f_i)$ can be defined in terms of $\mathbf{E}$ and $\mathbf{f_i}$:

$$v(f_i) = \mathbf{f_i E}$$

And similarly:

$$CBOW(f_1, ..., f_k) = \sum_{i=1}^{k} (\mathbf{f_i E}) = (\sum_{i=1}^{k} \mathbf{f_i})\mathbf{E}$$

The input to the network is then considered to be a collection of one-hot vectors. While this is elegant and well defined mathematically, an efficient implementation typically involves a hash-based data structure mapping features to their corresponding embedding vectors, without going through the one-hot representation.

In this tutorial, we take c to be separate from the network architecture: the network's inputs are always dense real-valued input vectors, and c is applied before the input is passed the network, similar to a "feature function" in the familiar linear-models terminology. However, when training a network, the input vector $\mathbf{x}$ does remember how it was constructed, and can propagate error gradients back to its component embedding vectors, as appropriate.

A note on notation When describing network layers that get concatenated vectors $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ as input, some authors use explicit concatenation ($[\mathbf{x}; \mathbf{y}; \mathbf{z}]\mathbf{W} + \mathbf{b}$) while others use an affine transformation ($\mathbf{xU} + \mathbf{yV} + \mathbf{zW} + \mathbf{b}$). If the weight matrices $\mathbf{U}$, $\mathbf{V}$, $\mathbf{W}$ in the affine transformation are different than one another, the two notations are equivalent.

A note on sparse vs. dense features Consider a network which uses a "traditional" sparse representation for its input vectors, and no embedding layer. Assuming the set of all available features is V and we have k "on" features $f_1, \ldots, f_k$, $f_i \in V$, the network's input is:

$$\mathbf{x} = \sum_{i=1}^{k} \mathbf{f_i} \qquad \mathbf{x} \in \mathbb{N}_+^{|V|}$$

and so the first layer (ignoring the non-linear activation) is:

$$\mathbf{x}\mathbf{W} + \mathbf{b} = (\sum_{i=1}^{k} \mathbf{f_i})\mathbf{W}$$

$$\mathbf{W} \in \mathbb{R}^{|V| \times d}, \quad \mathbf{b} \in \mathbb{R}^d$$

This layer selects rows of $\mathbf{W}$ corresponding to the input features in $\mathbf{x}$ and sums them, then adding a bias term. This is very similar to an embedding layer that produces a CBOW representation over the features, where the matrix $\mathbf{W}$ acts as the embedding matrix. The main difference is the introduction of the bias vector $\mathbf{b}$, and the fact that the embedding layer typically does not undergo a non-linear activation but rather passed on directly to the first layer. Another difference is that this scenario forces each feature to receive a separate vector (row in $\mathbf{W}$) while the embedding layer provides more flexibility, allowing for example for the features "next word is dog" and "previous word is dog" to share the same vector. However, these differences are small and subtle. When it comes to multi-layer feed-forward networks, the difference between dense and sparse inputs is smaller than it may seem at first sight.

4.5 Loss Functions

When training a neural network (more on training in Section 6 below), much like when training a linear classifier, one defines a loss function $L(\hat{\mathbf{y}}, \mathbf{y})$, stating the loss of predicting $\hat{\mathbf{y}}$ when the true output is $\mathbf{y}$. The training objective is then to minimize the loss across the different training examples. The loss $L(\hat{\mathbf{y}}, \mathbf{y})$ assigns a numerical score (a scalar) for the network's output $\hat{\mathbf{y}}$ given the true expected output $\mathbf{y}$.[11] The loss is always positive, and should be zero only for cases where the network's output is correct.

The parameters of the network (the matrices $\mathbf{W^i}$, the biases $\mathbf{b^i}$ and commonly the embeddings $\mathbf{E}$) are then set in order to minimize the loss L over the training examples (usually, it is the sum of the losses over the different training examples that is being minimized).

The loss can be an arbitrary function mapping two vectors to a scalar. For practical purposes of optimization, we restrict ourselves to functions for which we can easily compute gradients (or sub-gradients). In most cases, it is sufficient and advisable to rely on a common loss function rather than defining your own. For a detailed discussion on loss functions for neural networks see (LeCun, Chopra, Hadsell, Ranzato, & Huang, 2006; LeCun & Huang, 2005; Bengio et al., 2015). We now discuss some loss functions that are commonly used in neural networks for NLP.

11. In our notation, both the model's output and the expected output are vectors, while in many cases it is more natural to think of the expected output as a scalar (class assignment). In such cases, $\mathbf{y}$ is simply the corresponding one-hot vector.

Hinge (binary) For binary classification problems, the network's output is a single scalar $\hat{y}$ and the intended output y is in $\{+1, -1\}$. The classification rule is $sign(\hat{y})$, and a classification is considered correct if $y \cdot \hat{y} > 0$, meaning that y and $\hat{y}$ share the same sign. The hinge loss, also known as margin loss or SVM loss, is defined as:

$$L_{hinge(binary)}(\hat{y}, y) = \max(0, 1 - y \cdot \hat{y})$$

The loss is 0 when y and $\hat{y}$ share the same sign and $|\hat{y}| \geq 1$. Otherwise, the loss is linear. In other words, the binary hinge loss attempts to achieve a correct classification, with a margin of at least 1.

Hinge (multiclass) The hinge loss was extended to the multiclass setting by Crammer and Singer (2002). Let $\hat{\mathbf{y}} = \hat{y}_1, \ldots, \hat{y}_n$ be the network's output vector, and $\mathbf{y}$ be the one-hot vector for the correct output class.

The classification rule is defined as selecting the class with the highest score:

$$\text{prediction} = \arg\max_i \hat{y}_i$$

,

Denote by $t = \arg\max_i y_i$ the correct class, and by $k = \arg\max_{i \neq t} \hat{y}_i$ the highest scoring class such that $k \neq t$. The multiclass hinge loss is defined as:

$$L_{hinge(multiclass)}(\hat{\mathbf{y}}, \mathbf{y}) = \max(0, 1 - (\hat{y}_t - \hat{y}_k))$$

The multiclass hinge loss attempts to score the correct class above all other classes with a margin of at least 1.

Both the binary and multiclass hinge losses are intended to be used with a linear output layer. The hinge losses are useful whenever we require a hard decision rule, and do not attempt to model class membership probability.

Log loss The log loss is a common variation of the hinge loss, which can be seen as a "soft" version of the hinge loss with an infinite margin (LeCun et al., 2006).

$$L_{log}(\hat{\mathbf{y}}, \mathbf{y}) = \log(1 + exp(-(\hat{y}_t - \hat{y}_k))$$

Categorical cross-entropy loss The categorical cross-entropy loss (also referred to as *negative log likelihood*) is used when a probabilistic interpretation of the scores is desired.

Let $\mathbf{y} = y_1, \ldots, y_n$ be a vector representing the true multinomial distribution over the labels $1, \ldots, n$, and let $\hat{\mathbf{y}} = \hat{y_1}, \ldots, \hat{y_n}$ be the network's output, which was transformed by the $softmax$ activation function, and represent the class membership conditional distribution $\hat{y}_i = P(y = i|\mathbf{x})$. The categorical cross entropy loss measures the dissimilarity between the true label distribution $\mathbf{y}$ and the predicted label distribution $\hat{\mathbf{y}}$, and is defined as cross entropy:

$$L_{cross-entropy}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_i y_i \log(\hat{y}_i)$$

For hard classification problems in which each training example has a single correct class assignment, $\mathbf{y}$ is a one-hot vector representing the true class. In such cases, the cross entropy can be simplified to:

$$L_{cross-entropy(\text{hard classification})}(\hat{\mathbf{y}}, \mathbf{y}) = -\log(\hat{y}_t)$$

where t is the correct class assignment. This attempts to set the probability mass assigned to the correct class t to 1. Because the scores $\hat{\mathbf{y}}$ have been transformed using the $softmax$ function and represent a conditional distribution, increasing the mass assigned to the correct class means decreasing the mass assigned to all the other classes.

The cross-entropy loss is very common in the neural networks literature, and produces a multi-class classifier which does not only predict the one-best class label but also predicts a distribution over the possible labels. When using the cross-entropy loss, it is assumed that the network's output is transformed using the $softmax$ transformation.

Ranking losses In some settings, we are not given supervision in term of labels, but rather as pairs of correct and incorrect items $\mathbf{x}$ and $\mathbf{x}'$, and our goal is to score correct items above incorrect ones. Such training situations arise when we have only positive examples, and generate negative examples by corrupting a positive example. A useful loss in such scenarios is the margin-based ranking loss, defined for a pair of correct and incorrect examples:

$$L_{ranking(margin)}(\mathbf{x}, \mathbf{x}') = \max(0, 1 - (NN(\mathbf{x}) - NN(\mathbf{x}')))$$

where $NN(\mathbf{x})$ is the score assigned by the network for input vector $\mathbf{x}$. The objective is to score (rank) correct inputs over incorrect ones with a margin of at least 1.

A common variation is to use the log version of the ranking loss:

$$L_{ranking(log)}(\mathbf{x}, \mathbf{x}') = \log(1 + exp(-(NN(\mathbf{x}) - NN(\mathbf{x}'))))$$

Examples using the ranking hinge loss in language tasks include training with the auxiliary tasks used for deriving pre-trained word embeddings (see section 5), in which we are given a correct word sequence and a corrupted word sequence, and our goal is to score the correct sequence above the corrupt one (Collobert & Weston, 2008). Similarly, Van de Cruys (2014) used the ranking loss in a selectional-preferences task, in which the network was trained to rank correct verb-object pairs above incorrect, automatically derived ones, and (Weston, Bordes, Yakhnenko, & Usunier, 2013) trained a model to score correct (head,relation,trail) triplets above corrupted ones in an information-extraction setting. An example of using the ranking log loss can be found in (Gao et al., 2014). A variation of the ranking log loss allowing for a different margin for the negative and positive class is given in (dos Santos et al., 2015).

5. Word Embeddings

A main component of the neural-network approach is the use of embeddings – representing each feature as a vector in a low dimensional space. But where do the vectors come from? This section will survey the common approaches.

5.1 Random Initialization

When enough supervised training data is available, one can just treat the feature embeddings the same as the other model parameters: initialize the embedding vectors to random values, and let the network-training procedure tune them into "good" vectors.

Some care has to be taken in the way the random initialization is performed. The method used by the effective word2vec implementation (Mikolov et al., 2013; Mikolov, Sutskever, Chen, Corrado, & Dean, 2013) is to initialize the word vectors to uniformly sampled random numbers in the range $[-\frac{1}{2d}, \frac{1}{2d}]$ where d is the number of dimensions. Another option is to use *xavier initialization* (see Section 6.3) and initialize with uniformly sampled values from $\left[-\frac{\sqrt{6}}{\sqrt{d}}, \frac{\sqrt{6}}{\sqrt{d}}\right]$.

In practice, one will often use the random initialization approach to initialize the embedding vectors of commonly occurring features, such as part-of-speech tags or individual letters, while using some form of supervised or unsupervised pre-training to initialize the potentially rare features, such as features for individual words. The pre-trained vectors can then either be treated as fixed during the network training process, or, more commonly, treated like the randomly-initialized vectors and further tuned to the task at hand.

5.2 Supervised Task-specific Pre-training

If we are interested in task A, for which we only have a limited amount of labeled data (for example, syntactic parsing), but there is an auxiliary task B (say, part-of-speech tagging) for which we have much more labeled data, we may want to pre-train our word vectors so that they perform well as predictors for task B, and then use the trained vectors for training task A. In this way, we can utilize the larger amounts of labeled data we have for task B. When training task A we can either treat the pre-trained vectors as fixed, or tune them further for task A. Another option is to train jointly for both objectives, see Section 7 for more details.

5.3 Unsupervised Pre-training

The common case is that we do not have an auxiliary task with large enough amounts of annotated data (or maybe we want to help bootstrap the auxiliary task training with better vectors). In such cases, we resort to "unsupervised" methods, which can be trained on huge amounts of unannotated text.

The techniques for training the word vectors are essentially those of supervised learning, but instead of supervision for the task that we care about, we instead create practically

unlimited number of supervised training instances from raw text, hoping that the tasks that we created will match (or be close enough to) the final task we care about.[12]

The key idea behind the unsupervised approaches is that one would like the embedding vectors of "similar" words to have similar vectors. While word similarity is hard to define and is usually very task-dependent, the current approaches derive from the distributional hypothesis (Harris, 1954), stating that *words are similar if they appear in similar contexts*. The different methods all create supervised training instances in which the goal is to either predict the word from its context, or predict the context from the word.

An important benefit of training word embeddings on large amounts of unannotated data is that it provides vector representations for words that do not appear in the supervised training set. Ideally, the representations for these words will be similar to those of related words that do appear in the training set, allowing the model to generalize better on unseen events. It is thus desired that the similarity between word vectors learned by the unsupervised algorithm captures the same aspects of similarity that are useful for performing the intended task of the network.

Common unsupervised word-embedding algorithms include `word2vec` [13] (Mikolov et al., 2013, 2013), `GloVe` (Pennington, Socher, & Manning, 2014) and the Collobert and Weston (2008, 2011) embeddings algorithm. These models are inspired by neural networks and are based on stochastic gradient training. However, they are deeply connected to another family of algorithms which evolved in the NLP and IR communities, and that are based on matrix factorization (see (Levy & Goldberg, 2014b; Levy et al., 2015) for a discussion).

Arguably, the choice of auxiliary problem (what is being predicted, based on what kind of context) affects the resulting vectors much more than the learning method that is being used to train them. We thus focus on the different choices of auxiliary problems that are available, and only skim over the details of the training methods. Several software packages for deriving word vectors are available, including word2vec[14] and Gensim[15] implementing the word2vec models with word-windows based contexts, word2vecf[16] which is a modified version of word2vec allowing the use of arbitrary contexts, and GloVe[17] implementing the GloVe model. Many pre-trained word vectors are also available for download on the web.

While beyond the scope of this tutorial, it is worth noting that the word embeddings derived by unsupervised training algorithms have a wide range of applications in NLP beyond using them for initializing the word-embeddings layer of a neural-network model.

5.4 Training Objectives

Given a word w and its context c, different algorithms formulate different auxiliary tasks. In all cases, each word is represented as a d-dimensional vector which is initialized to a random value. Training the model to perform the auxiliary tasks well will result in good

12. The interpretation of creating auxiliary problems from raw text is inspired by Ando and Zhang (Ando & Zhang, 2005a, 2005b).
13. While often treated as a single algorithm, `word2vec` is actually a software package including various training objectives, optimization methods and other hyperparameters. See (Rong, 2014; Levy, Goldberg, & Dagan, 2015) for a discussion.
14. `https://code.google.com/p/word2vec/`
15. `https://radimrehurek.com/gensim/`
16. `https://bitbucket.org/yoavgo/word2vecf`
17. `http://nlp.stanford.edu/projects/glove/`

word embeddings for relating the words to the contexts, which in turn will result in the embedding vectors for similar words to be similar to each other.

Language-modeling inspired approaches such as those taken by (Mikolov et al., 2013; Mnih & Kavukcuoglu, 2013) as well as GloVe (Pennington et al., 2014) use auxiliary tasks in which the goal is to predict the word given its context. This is posed in a probabilistic setup, trying to model the conditional probability $P(w|c)$.

Other approaches reduce the problem to that of binary classification. In addition to the set D of observed word-context pairs, a set $\bar{D}$ is created from random words and context pairings. The binary classification problem is then: does the given (w, c) pair come from D or not? The approaches differ in how the set $\bar{D}$ is constructed, what is the structure of the classifier, and what is the objective being optimized. Collobert and Weston (2008, 2011) take a margin-based binary ranking approach, training a feed-forward neural network to score correct (w, c) pairs over incorrect ones. Mikolov et al (2013, 2014) take instead a probabilistic version, training a log-bilinear model to predict the probability $P((w, c) \in D|w, c)$ that the pair come from the corpus rather than the random sample.

5.5 The Choice of Contexts

In most cases, the contexts of a word are taken to be other words that appear in its surrounding, either in a short window around it, or within the same sentence, paragraph or document. In some cases the text is automatically parsed by a syntactic parser, and the contexts are derived from the syntactic neighbourhood induced by the automatic parse trees. Sometimes, the definitions of words and context change to include also parts of words, such as prefixes or suffixes.

Neural word embeddings originated from the world of language modeling, in which a network is trained to predict the next word based on a sequence of preceding words (Bengio et al., 2003). There, the text is used to create auxiliary tasks in which the aim is to predict a word based on a context the k previous words. While training for the language modeling auxiliary prediction problems indeed produce useful embeddings, this approach is needlessly restricted by the constraints of the language modeling task, in which one is allowed to look only at the previous words. If we do not care about language modeling but only about the resulting embeddings, we may do better by ignoring this constraint and taking the context to be a symmetric window around the focus word.

5.5.1 Window Approach

The most common approach is a sliding window approach, in which auxiliary tasks are created by looking at a sequence of $2k + 1$ words. The middle word is callled the *focus word* and the k words to each side are the *contexts*. Then, either a single task is created in which the goal is to predict the focus word based on all of the context words (represented either using CBOW (Mikolov et al., 2013) or vector concatenation (Collobert & Weston, 2008)), or $2k$ distinct tasks are created, each pairing the focus word with a different context word. The $2k$ tasks approach, popularized by (Mikolov et al., 2013) is referred to as a *skip-gram* model. Skip-gram based approaches are shown to be robust and efficient to train (Mikolov et al., 2013; Pennington et al., 2014), and often produce state of the art results.

Effect of Window Size The size of the sliding window has a strong effect on the resulting vector similarities. Larger windows tend to produce more topical similarities (i.e. "dog", "bark" and "leash" will be grouped together, as well as "walked", "run" and "walking"), while smaller windows tend to produce more functional and syntactic similarities (i.e. "Poodle", "Pitbull", "Rottweiler", or "walking","running","approaching").

Positional Windows When using the CBOW or skip-gram context representations, all the different context words within the window are treated equally. There is no distinction between context words that are close to the focus words and those that are farther from it, and likewise there is no distinction between context words that appear before the focus words to context words that appear after it. Such information can easily be factored in by using *positional contexts*: indicating for each context word also its relative position to the focus words (i.e. instead of the context word being "the" it becomes "the:+2", indicating the word appears two positions to the right of the focus word). The use of positional context together with smaller windows tend to produce similarities that are more syntactic, with a strong tendency of grouping together words that share a part of speech, as well as being functionally similar in terms of their semantics. Positional vectors were shown by (Ling, Dyer, Black, & Trancoso, 2015a) to be more effective than window-based vectors when used to initialize networks for part-of-speech tagging and syntactic dependency parsing.

Variants Many variants on the window approach are possible. One may lemmatize words before learning, apply text normalization, filter too short or too long sentences, or remove capitalization (see, e.g., the pre-processing steps described in (dos Santos & Gatti, 2014). One may sub-sample part of the corpus, skipping with some probability the creation of tasks from windows that have too common or too rare focus words. The window size may be dynamic, using a different window size at each turn. One may weigh the different positions in the window differently, focusing more on trying to predict correctly close word-context pairs than further away ones. Each of these choices will effect the resulting vectors. Some of these hyperparameters (and others) are discussed in (Levy et al., 2015).

5.5.2 Sentences, Paragraphs or Documents

Using a skip-grams (or CBOW) approach, one can consider the contexts of a word to be all the other words that appear with it in the same sentence, paragraph or document. This is equivalent to using very large window sizes, and is expected to result in word vectors that capture topical similarity (words from the same topic, i.e. words that one would expect to appear in the same document, are likely to receive similar vectors).

5.5.3 Syntactic Window

Some work replace the linear context within a sentence with a syntactic one (Levy & Goldberg, 2014a; Bansal, Gimpel, & Livescu, 2014). The text is automatically parsed using a dependency parser, and the context of a word is taken to be the words that are in its proximity in the parse tree, together with the syntactic relation by which they are connected. Such approaches produce highly *functional* similarities, grouping together words than can fill the same role in a sentence (e.g. colors, names of schools, verbs of movement).

The grouping is also syntactic, grouping together words that share an inflection (Levy & Goldberg, 2014a).

5.5.4 Multilingual

Another option is using multilingual, translation based contexts (Hermann & Blunsom, 2014; Faruqui & Dyer, 2014). For example, given a large amount of sentence-aligned parallel text, one can run a bilingual alignment model such as the IBM model 1 or model 2 (i.e. using the GIZA++ software), and then use the produced alignments to derive word contexts. Here, the context of a word instance are the foreign language words that are aligned to it. Such alignments tend to result in synonym words receiving similar vectors. Some authors work instead on the sentence alignment level, without relying on word alignments. An appealing method is to mix a monolingual window-based approach with a multilingual approach, creating both kinds of auxiliary tasks. This is likely to produce vectors that are similar to the window-based approach, but reducing the somewhat undesired effect of the window-based approach in which antonyms (e.g. hot and cold, high and low) tend to receive similar vectors (Faruqui & Dyer, 2014).

5.5.5 Character-based and Sub-word Representations

An interesting line of work attempts to derive the vector representation of a word from the characters that compose it. Such approaches are likely to be particularly useful for tasks which are syntactic in nature, as the character patterns within words are strongly related to their syntactic function. These approaches also have the benefit of producing very small model sizes (only one vector for each character in the alphabet together with a handful of small matrices needs to be stored), and being able to provide an embedding vector for every word that may be encountered. dos Santos and Gatti (2014) and dos Santos and Zadrozny (2014) model the embedding of a word using a convolutional network (see Section 9) over the characters. Ling et al (2015b) model the embedding of a word using the concatenation of the final states of two RNN (LSTM) encoders (Section 10), one reading the characters from left to right, and the other from right to left. Both produce very strong results for part-of-speech tagging. The work of Ballesteros et al (2015) show that the two-LSTMs encoding of (Ling et al., 2015b) is beneficial also for representing words in dependency parsing of morphologically rich languages.

Deriving representations of words from the representations of their characters is motivated by the *unknown words problem* – what do you do when you encounter a word for which you do not have an embedding vector? Working on the level of characters alleviates this problem to a large extent, as the vocabulary of possible characters is much smaller than the vocabulary of possible words. However, working on the character level is very challenging, as the relationship between form (characters) and function (syntax, semantics) in language is quite loose. Restricting oneself to stay on the character level may be an unnecessarily hard constraint. Some researchers propose a middle-ground, in which a word is represented as a combination of a vector for the word itself with vectors of sub-word units that comprise it. The sub-word embeddings then help in sharing information between different words with similar forms, as well as allowing back-off to the subword level when the word is not observed. At the same time, the models are not forced to rely solely on

form when enough observations of the word are available. Botha and Blunsom (2014) suggest to model the embedding vector of a word as a sum of the word-specific vector if such vector is available, with vectors for the different morphological components that comprise it (the components are derived using Morfessor (Creutz & Lagus, 2007), an unsupervised morphological segmentation method). Gao et al (Gao et al., 2014) suggest using as core features not only the word form itself but also a unique feature (hence a unique embedding vector) for each of the letter-trigrams in the word.

6. Neural Network Training

Neural network training is done by trying to minimize a loss function over a training set, using a gradient-based method. Roughly speaking, all training methods work by repeatedly computing an estimate of the error over the dataset, computing the gradient with respect to the error, and then moving the parameters in the direction of the gradient. Models differ in how the error estimate is computed, and how "moving in the direction of the gradient" is defined. We describe the basic algorithm, *stochastic gradient descent* (SGD), and then briefly mention the other approaches with pointers for further reading. Gradient calculation is central to the approach. Gradients can be efficiently and automatically computed using reverse mode differentiation on a computation graph – a general algorithmic framework for automatically computing the gradient of any network and loss function.

6.1 Stochastic Gradient Training

The common approach for training neural networks is using the stochastic gradient descent (SGD) algorithm (Bottou, 2012; LeCun, Bottou, Orr, & Muller, 1998a) or a variant of it. SGD is a general optimization algorithm. It receives a function f parameterized by θ, a loss function, and desired input and output pairs. It then attempts to set the parameters θ such that the loss of f with respect to the training examples is small. The algorithm works as follows:

Algorithm 1 Online Stochastic Gradient Descent Training

1: **Input:** Function $f(\mathbf{x};\theta)$ parameterized with parameters θ.
2: **Input:** Training set of inputs $\mathbf{x_1},\ldots,\mathbf{x_n}$ and outputs $\mathbf{y_1},\ldots,\mathbf{y_n}$.
3: **Input:** Loss function L.
4: **while** stopping criteria not met **do**
5: Sample a training example $\mathbf{x_i},\mathbf{y_i}$
6: Compute the loss $L(f(\mathbf{x_i};\theta),\mathbf{y_i})$
7: $\hat{\mathbf{g}} \leftarrow$ gradients of $L(f(\mathbf{x_i};\theta),\mathbf{y_i})$ w.r.t θ
8: $\theta \leftarrow \theta + \eta_k \hat{\mathbf{g}}$
9: **return** θ

The goal of the algorithm is to set the parameters θ so as to minimize the total loss $\sum_{i=1}^{n} L(f(\mathbf{x_i};\theta),\mathbf{y_i})$ over the training set. It works by repeatedly sampling a training example and computing the gradient of the error on the example with respect to the parameters θ (line 7) – the input and expected output are assumed to be fixed, and the loss is treated as a function of the parameters θ. The parameters θ are then updated in the direction of the gradient, scaled by a learning rate η_k (line 8). For further discussion on setting the learning rate, see Section 6.3.

Note that the error calculated in line 6 is based on a single training example, and is thus just a rough estimate of the corpus-wide loss that we are aiming to minimize. The noise in the loss computation may result in inaccurate gradients. A common way of reducing this noise is to estimate the error and the gradients based on a sample of m examples. This gives rise to the *minibatch SGD* algorithm:

Algorithm 2 Minibatch Stochastic Gradient Descent Training

1: **Input:** Function $f(\mathbf{x};\theta)$ parameterized with parameters θ.
2: **Input:** Training set of inputs $\mathbf{x_1},\ldots,\mathbf{x_n}$ and outputs $\mathbf{y_1},\ldots,\mathbf{y_n}$.
3: **Input:** Loss function L.
4: **while** stopping criteria not met **do**
5: Sample a minibatch of m examples $\{(\mathbf{x_1},\mathbf{y_1}),\ldots,(\mathbf{x_m},\mathbf{y_m})\}$
6: $\hat{\mathbf{g}} \leftarrow 0$
7: **for** $i = 1$ to m **do**
8: Compute the loss $L(f(\mathbf{x_i};\theta),\mathbf{y_i})$
9: $\hat{\mathbf{g}} \leftarrow \hat{\mathbf{g}} +$ gradients of $\frac{1}{m}L(f(\mathbf{x_i};\theta),\mathbf{y_i})$ w.r.t θ
10: $\theta \leftarrow \theta + \eta_k\hat{\mathbf{g}}$
11: **return** θ

In lines 6 – 9 the algorithm estimates the gradient of the corpus loss based on the minibatch. After the loop, $\hat{\mathbf{g}}$ contains the gradient estimate, and the parameters θ are updated toward $\hat{\mathbf{g}}$. The minibatch size can vary in size from $m = 1$ to $m = n$. Higher values provide better estimates of the corpus-wide gradients, while smaller values allow more updates and in turn faster convergence. Besides the improved accuracy of the gradients estimation, the minibatch algorithm provides opportunities for improved training efficiency. For modest sizes of m, some computing architectures (i.e. GPUs) allow an efficient parallel implementation of the computation in lines 6–9. With a small enough learning rate, SGD is guaranteed to converge to a global optimum if the function is convex. However, it can also be used to optimize non-convex functions such as neural-network. While there are no longer guarantees of finding a global optimum, the algorithm proved to be robust and performs well in practice.

When training a neural network, the parameterized function f is the neural network, and the parameters θ are the layer-transfer matrices, bias terms, embedding matrices and so on. The gradient computation is a key step in the SGD algorithm, as well as in all other neural network training algorithms. The question is, then, how to compute the gradients of the network's error with respect to the parameters. Fortunately, there is an easy solution in the form of the *backpropagation algorithm* (Rumelhart, Hinton, & Williams, 1986; Lecun, Bottou, Bengio, & Haffner, 1998b). The backpropagation algorithm is a fancy name for methodologically computing the derivatives of a complex expression using the chain-rule, while caching intermediary results. More generally, the backpropagation algorithm is a special case of the reverse-mode automatic differentiation algorithm (Neidinger, 2010, Section 7), (Baydin, Pearlmutter, Radul, & Siskind, 2015; Bengio, 2012).The following section describes reverse mode automatic differentiation in the context of the *computation graph* abstraction.

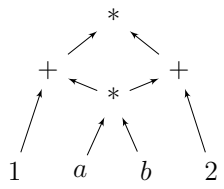
Beyond SGD While the SGD algorithm can and often does produce good results, more advanced algorithms are also available. The *SGD+Momentum* (Polyak, 1964) and *Nesterov Momentum* (Sutskever, Martens, Dahl, & Hinton, 2013) algorithms are variants of SGD in which previous gradients are accumulated and affect the current update. Adaptive learning rate algorithms including AdaGrad (Duchi, Hazan, & Singer, 2011), AdaDelta (Zeiler, 2012),

RMSProp (Tieleman & Hinton, 2012) and Adam (Kingma & Ba, 2014) are designed to select the learning rate for each minibatch, sometimes on a per-coordinate basis, potentially alleviating the need of fiddling with learning rate scheduling. For details of these algorithms, see the original papers or (Bengio et al., 2015, Sections 8.3, 8.4). As many neural-network software frameworks provide implementations of these algorithms, it is easy and sometimes worthwhile to try out different variants.

6.2 The Computation Graph Abstraction

While one can compute the gradients of the various parameters of a network by hand and implement them in code, this procedure is cumbersome and error prone. For most purposes, it is preferable to use automatic tools for gradient computation (Bengio, 2012). The computation-graph abstraction allows us to easily construct arbitrary networks, evaluate their predictions for given inputs (forward pass), and compute gradients for their parameters with respect to arbitrary scalar losses (backward pass).

A computation graph is a representation of an arbitrary mathematical computation as a graph. It is a directed acyclic graph (DAG) in which nodes correspond to mathematical operations or (bound) variables and edges correspond to the flow of intermediary values between the nodes. The graph structure defines the order of the computation in terms of the dependencies between the different components. The graph is a DAG and not a tree, as the result of one operation can be the input of several continuations. Consider for example a graph for the computation of $(a * b + 1) * (a * b + 2)$:



The computation of $a * b$ is shared. We restrict ourselves to the case where the computation graph is connected.

Since a neural network is essentially a mathematical expression, it can be represented as a computation graph.

For example, Figure 3a presents the computation graph for a 1-layer MLP with a softmax output transformation. In our notation, oval nodes represent mathematical operations or functions, and shaded rectangle nodes represent parameters (bound variables). Network inputs are treated as constants, and drawn without a surrounding node. Input and parameter nodes have no incoming arcs, and output nodes have no outgoing arcs. The output of each node is a matrix, the dimensionality of which is indicated above the node.

This graph is incomplete: without specifying the inputs, we cannot compute an output. Figure 3b shows a complete graph for an MLP that takes three words as inputs, and predicts the distribution over part-of-speech tags for the third word. This graph can be used for prediction, but not for training, as the output is a vector (not a scalar) and the graph does not take into account the correct answer or the loss term. Finally, the graph in 3c shows the computation graph for a specific training example, in which the inputs are the (embeddings

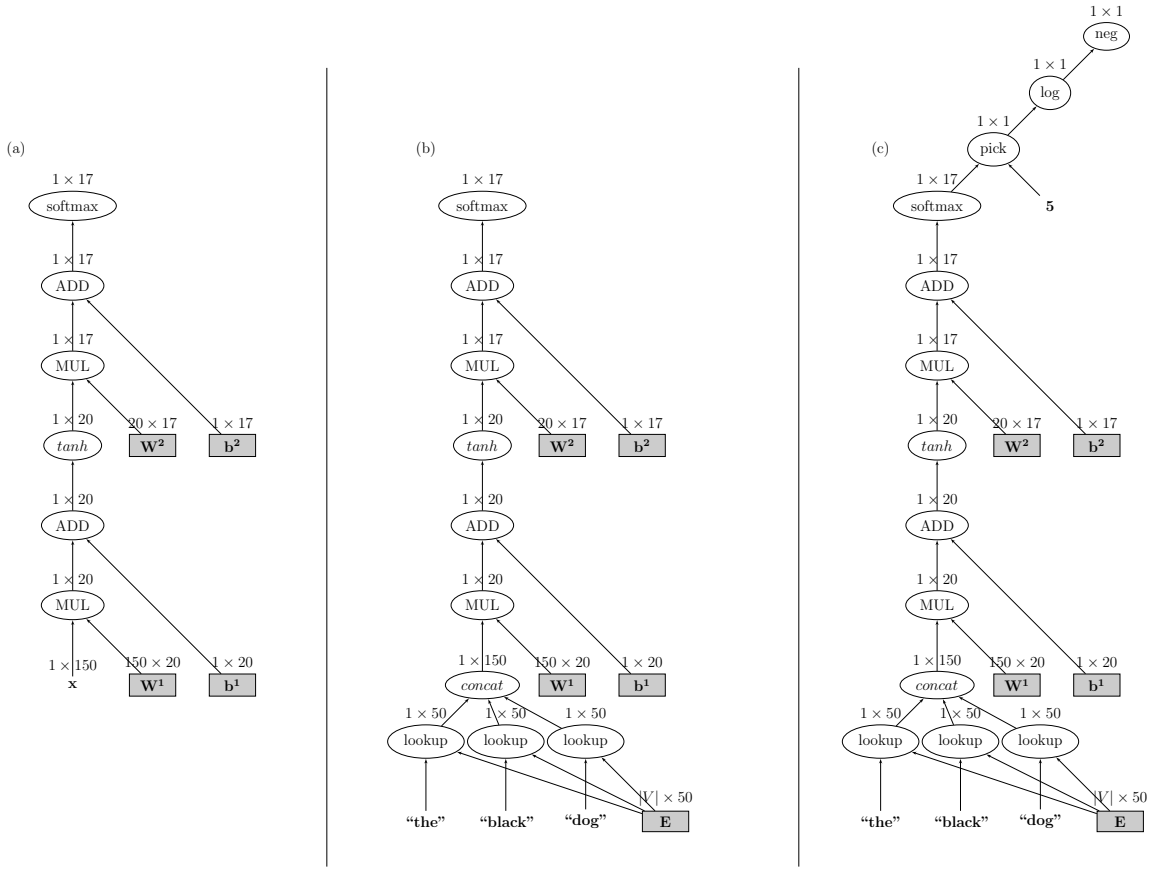


Figure 3: **Computation Graph for MLP1.** (a) Graph with unbound input. (b) Graph with concrete input. (c) Graph with concrete input, expected output, and loss node.

of) the words "the", "black", "dog", and the expected output is "NOUN" (whose index is 5).

Once the graph is built, it is straightforward to run either a forward computation (compute the result of the computation) or a backward computation (computing the gradients), as we show below. Constructing the graphs may look daunting, but is actually very easy using dedicated software libraries and APIs.

Forward Computation The forward pass computes the outputs of the nodes in the graph. Since each node's output depends only on itself and on its incoming edges, it is trivial to compute the outputs of all nodes by traversing the nodes in a topological order and computing the output of each node given the already computed outputs of its predecessors.

More formally, in a graph of N nodes, we associate each node with an index i according to their topological ordering. Let f_i be the function computed by node i (e.g. *multiplication*. *addition*, ...). Let $\pi(i)$ be the parent nodes of node i, and $\pi^{-1}(i) = \{j \mid i \in \pi(j)\}$ the children nodes of node i (these are the arguments of f_i). Denote by $v(i)$ the output of node

i, that is, the application of f_i to the output values of its arguments $\pi^{-1}(i)$. For variable and input nodes, f_i is a constant function and $\pi^{-1}(i)$ is empty. The Forward algorithm computes the values $v(i)$ for all $i \in [1, N]$.

Algorithm 3 Computation Graph Forward Pass

1: **for** i = 1 to N **do**
2: Let $a_1, \ldots, a_m = \pi^{-1}(i)$
3: $v(i) \leftarrow f_i(v(a_1), \ldots, v(a_m))$

Backward Computation (Derivatives, Backprop) The backward pass begins by designating a node N with scalar (1×1) output as a loss-node, and running forward computation up to that node. The backward computation will computes the gradients with respect to that node's value. Denote by $d(i)$ the quantity $\frac{\partial N}{\partial i}$. The backpropagation algorithm is used to compute the values $d(i)$ for all nodes i.
The backward pass fills a table $d(i)$ as follows:

Algorithm 4 Computation Graph Backward Pass (Backpropagation)

1: $d(N) \leftarrow 1$
2: **for** i = N-1 to 1 **do**
3: $d(i) \leftarrow \sum_{j \in \pi(i)} d(j) \cdot \frac{\partial f_j}{\partial i}$

The quantity $\frac{\partial f_j}{\partial i}$ is the partial derivative of $f_j(\pi^{-1}(j))$ w.r.t the argument $i \in \pi^{-1}(j)$. This value depends on the function f_j and the values $v(a_1), \ldots, v(a_m)$ (where $a_1, \ldots, a_m = \pi^{-1}(j)$) of its arguments, which were computed in the forward pass.

Thus, in order to define a new kind of node, one need to define two methods: one for calculating the forward value $v(i)$ based on the nodes inputs, and the another for calculating $\frac{\partial f_i}{\partial x}$ for each $x \in \pi^{-1}(i)$.

For further information on automatic differentiation see (Neidinger, 2010, Section 7), (Baydin et al., 2015). For more in depth discussion of the backpropagation algorithm and computation graphs (also called flow graphs) see (Bengio et al., 2015, Section 6.4), (Lecun et al., 1998b; Bengio, 2012). For a popular yet technical presentation, see Chris Olah's description at `http://colah.github.io/posts/2015-08-Backprop/`.

Software Several software packages implement the computation-graph model, including Theano[18], Chainer[19], penne[20] and CNN/pyCNN[21]. All these packages support all the essential components (node types) for defining a wide range of neural network architectures, covering the structures described in this tutorial and more. Graph creation is made almost transparent by use of operator overloading. The framework defines a type for representing graph nodes (commonly called *expressions*), methods for constructing nodes for inputs and

18. `http://deeplearning.net/software/theano/`
19. `http://chainer.org`
20. `https://bitbucket.org/ndnlp/penne`
21. `https://github.com/clab/cnn`

parameters, and a set of functions and mathematical operations that take expressions as input and result in more complex expressions. For example, the python code for creating the computation graph from Figure (3c) using the pyCNN framework is:

```
from pycnn import *
# model initialization.
model = Model()
model.add_parameters("W1", (20,150))
model.add_parameters("b1", 20)
model.add_parameters("W2", (17,20))
model.add_parameters("b2", 17)
model.add_lookup_parameters("words", (100, 50))

# Building the computation graph:
renew_cg() # create a new graph.
# Wrap the model parameters as graph-nodes.
W1 = parameter(model["W1"])
b1 = parameter(model["b1"])
W2 = parameter(model["W2"])
b2 = parameter(model["b2"])
def get_index(x): return 1
# Generate the embeddings layer.
vthe   = lookup(model["words"], get_index("the"))
vblack = lookup(model["words"], get_index("black"))
vdog   = lookup(model["words"], get_index("dog"))

# Connect the leaf nodes into a complete graph.
x = concatenate([vthe, vblack, vdog])
output = softmax(W2*(tanh(W1*x)+b1)+b2)
loss = -log(pick(output, 5))

loss_value = loss.forward()
loss.backward() # the gradient is computed
                # and stored in the corresponding
                # parameters.
```

Most of the code involves various initializations: the first block defines model parameters that are be shared between different computation graphs (recall that each graph corresponds to a specific training example). The second block turns the model parameters into the graph-node (Expression) types. The third block retrieves the Expressions for the embeddings of the input words. Finally, the fourth block is where the graph is created. Note how transparent the graph creation is – there is an almost a one-to-one correspondence between creating the graph and describing it mathematically. The last block shows a forward and backward pass. The other software frameworks follow similar patterns.

Theano involves an optimizing compiler for computation graphs, which is both a blessing and a curse. On the one hand, once compiled, large graphs can be run efficiently on either the CPU or a GPU, making it ideal for large graphs with a fixed structure, where only the inputs change between instances. However, the compilation step itself can be costly, and it makes the interface a bit cumbersome to work with. In contrast, the other packages focus on building large and dynamic computation graphs and executing them "on the fly" without a compilation step. While the execution speed may suffer with respect to Theano's optimized version, these packages are especially convenient when working with the recurrent and

recursive networks described in Sections 10, 12 as well as in structured prediction settings as described in Section 8.

Implementation Recipe Using the computation graph abstraction, the pseudo-code for a network training algorithm is given in Algorithm 5.

Algorithm 5 Neural Network Training with Computation Graph Abstraction (using mini-batches of size 1)

```
Define network parameters.
for iteration = 1 to N do
    for Training example x_i, y_i in dataset do
        loss_node ← build_computation_graph(x_i, y_i, parameters)
        loss_node.forward()
        gradients ← loss_node().backward()
        parameters ← update_parameters(parameters, gradients)
return parameters.
```

Here, build_computation_graph is a user-defined function that builds the computation graph for the given input, output and network structure, returning a single loss node. update_parameters is an optimizer specific update rule. The recipe specifies that a new graph is created for each training example. This accommodates cases in which the network structure varies between training example, such as recurrent and recursive neural networks, to be discussed in Sections 10 – 12. For networks with fixed structures, such as an MLPs, it may be more efficient to create one base computation graph and vary only the inputs and expected outputs between examples.

Network Composition As long as the network's output is a vector ($1 \times k$ matrix), it is trivial to compose networks by making the output of one network the input of another, creating arbitrary networks. The computation graph abstractions makes this ability explicit: a node in the computation graph can itself be a computation graph with a designated output node. One can then design arbitrarily deep and complex networks, and be able to easily evaluate and train them thanks to automatic forward and gradient computation. This makes it easy to define and train networks for structured outputs and multi-objective training, as we discuss in Section 7, as well as complex recurrent and recursive networks, as discussed in Sections 10–12.

6.3 Optimization Issues

Once the gradient computation is taken care of, the network is trained using SGD or another gradient-based optimization algorithm. The function being optimized is not convex, and for a long time training of neural networks was considered a "black art" which can only be done by selected few. Indeed, many parameters affect the optimization process, and care has to be taken to tune these parameters. While this tutorial is not intended as a comprehensive guide to successfully training neural networks, we do list here a few of the prominent issues. For further discussion on optimization techniques and algorithms for neural networks, refer to (Bengio et al., 2015, Chapter 8). For some theoretical discussion and analysis, refer

to (Glorot & Bengio, 2010). For various practical tips and recommendations, see (LeCun et al., 1998a; Bottou, 2012).

Initialization The non-convexity of the loss function means the optimization procedure may get stuck in a local minimum or a saddle point, and that starting from different initial points (e.g. different random values for the parameters) may result in different results. Thus, it is advised to run several restarts of the training starting at different random initializations, and choosing the best one based on a development set.[22] The amount of variance in the results is different for different network formulations and datasets, and cannot be predicted in advance.

The magnitude of the random values has an important effect on the success of training. An effective scheme due to Glorot and Bengio (2010), called *xavier initialization* after Glorot's first name, suggests initializing a weight matrix $\mathbf{W} \in \mathbb{R}^{d_{in} \times d_{out}}$ as:

$$\mathbf{W} \sim U\left[-\frac{\sqrt{6}}{\sqrt{d_{in}+d_{out}}}, +\frac{\sqrt{6}}{\sqrt{d_{in}+d_{out}}}\right]$$

where $U[a,b]$ is a uniformly sampled random value in the range $[a,b]$. This advice works well on many occasions, and is the preferred default initialization method by many.

Analysis by He et al (2015) suggests that when using ReLU non-linearities, the weights should be initialized by sampling from a zero-mean Gaussian distribution whose standard deviation is $\sqrt{\frac{2}{d_{in}}}$. This initialization was found by He et al to work better than xavier initialization in an image classification task, especially when deep networks were involved.

Vanishing and Exploding Gradients In deep networks, it is common for the error gradients to either vanish (become exceedingly close to 0) or explode (become exceedingly high) as they propagate back through the computation graph. The problem becomes more severe in deeper networks, and especially so in recursive and recurrent networks (Pascanu, Mikolov, & Bengio, 2012). Dealing with the vanishing gradients problem is still an open research question. Solutions include making the networks shallower, step-wise training (first train the first layers based on some auxiliary output signal, then fix them and train the upper layers of the complete network based on the real task signal), or specialized architectures that are designed to assist in gradient flow (e.g., the LSTM and GRU architectures for recurrent networks, discussed in Section 11). Dealing with the exploding gradients has a simple but very effective solution: clipping the gradients if their norm exceeds a given threshold. Let $\hat{\mathbf{g}}$ be the gradients of all parameters in the network, and $\|\hat{\mathbf{g}}\|$ be their L_2 norm. Pascanu et al (2012) suggest to set: $\hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|}\hat{\mathbf{g}}$ if $\|\hat{\mathbf{g}}\| > threshold$.

Saturation and Dead Neurons Layers with *tanh* and *sigmoid* activations can become saturated – resulting in output values for that layer that are all close to one, the upper-limit of the activation function. Saturated neurons have very small gradients, and should be avoided. Layers with the ReLU activation cannot be saturated, but can "die" – most or all values are negative and thus clipped at zero for all inputs, resulting in a gradient of zero for that layer. If your network does not train well, it is advisable to monitor the network for layers with many saturated or dead neurons. Saturated neurons are caused by too large

22. When debugging, and for reproducibility of results, it is advised to used a fixed random seed.

values entering the layer. This may be controlled for by changing the initialization, scaling the range of the input values, or changing the learning rate. Dead neurons are caused by all weights entering the layer being negative (for example this can happen after a large gradient update). Reducing the learning rate will help in this situation. For saturated layers, another option is to normalize the values in the saturated layer after the activation, i.e. instead of $g(\mathbf{h}) = tanh(\mathbf{h})$ using $g(\mathbf{h}) = \frac{tanh(\mathbf{h})}{\|tanh(\mathbf{h})\|}$. Layer normalization is an effective measure for countering saturation, but is also expensive in terms of gradient computation.

Shuffling The order in which the training examples are presented to the network is important. The SGD formulation above specifies selecting a random example in each turn. In practice, most implementations go over the training example in order. It is advised to shuffle the training examples before each pass through the data.

Learning Rate Selection of the learning rate is important. Too large learning rates will prevent the network from converging on an effective solution. Too small learning rates will take very long time to converge. As a rule of thumb, one should experiment with a range of initial learning rates in range $[0, 1]$, e.g. 0.001, 0.01, 0.1, 1. Monitor the network's loss over time, and decrease the learning rate once the network seem to be stuck in a fixed region. *Learning rate scheduling* decrease the rate as a function of the number of observed minibatches. A common schedule is dividing the initial learning rate by the iteration number. Léon Bottou (2012) recommends using a learning rate of the form $\eta_t = \eta_0(1 + \eta_0\lambda t)^{-1}$ where η_0 is the initial learning rate, η_t is the learning rate to use on the tth training example, and λ is an additional hyperparameter. He further recommends determining a good value of η_0 based on a small sample of the data prior to running on the entire dataset.

Minibatches Parameter updates occur either every training example (minibatches of size 1) or every k training examples. Some problems benefit from training with larger minibatch sizes. In terms of the computation graph abstraction, one can create a computation graph for each of the k training examples, and then connecting the k loss nodes under an averaging node, whose output will be the loss of the minibatch. Large minibatched training can also be beneficial in terms of computation efficiency on specialized computing architectures such as GPUs. This is beyond the scope of this tutorial.

6.4 Regularization

Neural network models have many parameters, and overfitting can easily occur. Overfitting can be alleviated to some extent by *regularization*. A common regularization method is L_2 regularization, placing a squared penalty on parameters with large values by adding an additive $\frac{\lambda}{2}\|\theta\|^2$ term to the objective function to be minimized, where θ is the set of model parameters, $\|\cdot\|^2$ is the squared L_2 norm (sum of squares of the values), and λ is a hyperparameter controlling the amount of regularization.

A recently proposed alternative regularization method is *dropout* (Hinton, Srivastava, Krizhevsky, Sutskever, & Salakhutdinov, 2012). The dropout method is designed to prevent the network from learning to rely on specific weights. It works by randomly dropping (setting to 0) half of the neurons in the network (or in a specific layer) in each training example. Work by Wager et al (2013) establishes a strong connection between the dropout method

and L_2 regularization. Gal and Gharamani (2015) show that a multi-layer perceptron with dropout applied at every layer can be interpreted as Bayesian model averaging.

The dropout technique is one of the key factors contributing to very strong results of neural-network methods on image classification tasks (Krizhevsky, Sutskever, & Hinton, 2012), especially when combined with ReLU activation units (Dahl, Sainath, & Hinton, 2013). The dropout technique is effective also in NLP applications of neural networks.

7. Cascading and Multi-task Learning

The combination of online training methods with automatic gradient computations using the computation graph abstraction allows for an easy implementation of model cascading, parameter sharing and multi-task learning.

Model cascading is a powerful technique in which large networks are built by composing them out of smaller component networks. For example, we may have a feed-forward network for predicting the part of speech of a word based on its neighbouring words and/or the characters that compose it. In a pipeline approach, we would use this network for predicting parts of speech, and then feed the predictions as input features to neural network that does syntactic chunking or parsing. Instead, we could think of the hidden layers of this network as an encoding that captures the relevant information for predicting the part of speech. In a cascading approach, we take the hidden layers of this network and connect them (and not the part of speech prediction themselves) as the inputs for the syntactic network. We now have a larger network that takes as input sequences of words and characters, and outputs a syntactic structure. The computation graph abstraction allows us to easily propagate the error gradients from the syntactic task loss all the way back to the characters.

To combat the vanishing gradient problem of deep networks, as well as to make better use of available training material, the individual component network's parameters can be bootstrapped by training them separately on a relevant task, before plugging them in to the larger network for further tuning. For example, the part-of-speech predicting network can be trained to accurately predict parts-of-speech on a relatively large annotated corpus, before plugging its hidden layer into the syntactic parsing network for which less training data is available. In case the training data provide direct supervision for both tasks, we can make use of it during training by creating a network with two outputs, one for each task, computing a separate loss for each output, and then summing the losses into a single node from which we backpropagate the error gradients.

Model cascading is very common when using convolutional, recursive and recurrent neural networks, where, for example, a recurrent network is used to encode a sentence into a fixed sized vector, which is then used as the input of another network. The supervision signal of the recurrent network comes primarily from the upper network that consumes the recurrent network's output as it inputs.

Multi-task learning is used when we have related prediction tasks that do not necessarily feed into one another, but we do believe that information that is useful for one type of prediction can be useful also to some of the other tasks. For example, chunking, named entity recognition (NER) and language modeling are examples of synergistic tasks. Information for predicting chunk boundaries, named-entity boundaries and the next word in the sentence all rely on some shared underlying syntactic-semantic representation. Instead of training a separate network for each task, we can create a single network with several outputs. A common approach is to have a multi-layer feed-forward network, whose final hidden layer (or a concatenation of all hidden layers) is then passed to different output layers. This way, most of the parameters of the network are shared between the different tasks. Useful information learned from one task can then help to disambiguate other tasks. Again, the computation graph abstraction makes it very easy to construct such networks and compute

the gradients for them, by computing a separate loss for each available supervision signal, and then summing the losses into a single loss that is used for computing the gradients. In case we have several corpora, each with different kind of supervision signal (e.g. we have one corpus for NER and another for chunking), the training procedure will shuffle all of the available training example, performing gradient computation and updates with respect to a different loss in every turn. Multi-task learning in the context of language-processing is introduced and discussed in (Collobert et al., 2011).

8. Structured Output Prediction

Many problems in NLP involve structured outputs: cases where the desired output is not a class label or distribution over class labels, but a structured object such as a sequence, a tree or a graph. Canonical examples are sequence tagging (e.g. part-of-speech tagging) sequence segmentation (chunking, NER), and syntactic parsing. In this section, we discuss how feed-forward neural network models can be used for structured tasks. In later sections we discuss specialized neural network models for dealing with sequences (Section 10) and trees (Section 12).

8.1 Greedy Structured Prediction

The greedy approach to structured prediction is to decompose the structure prediction problem into a sequence of local prediction problems and training a classifier to perform each local decision. At test time, the trained classifier is used in a greedy manner. Examples of this approach are left-to-right tagging models (Giménez & Màrquez, 2004) and greedy transition-based parsing (Nivre, 2008). Such approaches are easily adapted to use neural networks by simply replacing the local classifier from a linear classifier such as an SVM or a logistic regression model to a neural network, as demonstrated in (Chen & Manning, 2014; Lewis & Steedman, 2014).

The greedy approaches suffer from error propagation, where mistakes in early decisions carry over and influence later decisions. The overall higher accuracy achievable with non-linear neural network classifiers helps in offsetting this problem to some extent. In addition, training techniques were proposed for mitigating the error propagation problem by either attempting to take easier predictions before harder ones (the easy-first approach (Goldberg & Elhadad, 2010)) or making training conditions more similar to testing conditions by exposing the training procedure to inputs that result from likely mistakes (Hal Daumé III, Langford, & Marcu, 2009; Goldberg & Nivre, 2013). These are effective also for training greedy neural network models, as demonstrated by Ma et al (Ma, Zhang, & Zhu, 2014) (easy-first tagger) and (?) (dynamic oracle training for greedy dependency parsing).

8.2 Search Based Structured Prediction

The common approach to predicting natural language structures is search based. For in-depth discussion of search-based structure prediction in NLP, see the book by Smith (Smith, 2011). The techniques can easily be adapted to use a neural-network. In the neural-networks literature, such models were discussed under the framework of *energy based learning* (LeCun et al., 2006, Section 7). They are presented here using setup and terminology familiar to the NLP community.

Search-based structured prediction is formulated as a search problem over possible structures:

$$predict(x) = \arg\max_{y \in \mathcal{Y}(x)} score(x, y)$$

where x is an input structure, y is an output over x (in a typical example x is a sentence and y is a tag-assignment or a parse-tree over the sentence), $\mathcal{Y}(x)$ is the set of all valid

structures over x, and we are looking for an output y that will maximize the score of the x, y pair.

The scoring function is defined as a linear model:

$$score(x, y) = \Phi(x, y) \cdot \mathbf{w}$$

where Φ is a feature extraction function and $\mathbf{w}$ is a weight vector.

In order to make the search for the optimal y tractable, the structure y is decomposed into parts, and the feature function is defined in terms of the parts, where $\phi(p)$ is a part-local feature extraction function:

$$\Phi(x, y) = \sum_{p \in parts(x,y)} \phi(p)$$

Each part is scored separately, and the structure score is the sum of the component parts scores:

$$score(x, y) = \mathbf{w} \cdot \Phi(x, y) = \mathbf{w} \cdot \sum_{p \in y} \phi(p) = \sum_{p \in y} \mathbf{w} \cdot \phi(p) = \sum_{p \in y} score(p)$$

where $p \in y$ is a shorthand for $p \in parts(x, y)$. The decomposition of y into parts is such that there exists an inference algorithm that allows for efficient search for the best scoring structure given the scores of the individual parts.

One can now trivially replace the linear scoring function over parts with a neural-network:

$$score(x, y) = \sum_{p \in y} score(p) = \sum_{p \in y} NN(c(p))$$

where $c(p)$ maps the part p into a d_{in} dimensional vector.

In case of a one hidden-layer feed-forward network:

$$score(x, y) = \sum_{p \in y} NN_{MLP1}(c(p)) = \sum_{p \in y} (g(c(p)\mathbf{W^1} + \mathbf{b^1}))\mathbf{w}$$

$c(p) \in \mathbb{R}^{d_{in}}$, $\mathbf{W^1} \in \mathbb{R}^{d_{in} \times d_1}$, $\mathbf{b^1} \in \mathbb{R}^{d_1}$, $\mathbf{w} \in \mathbb{R}^{d_1}$. A common objective in structured prediction is making the gold structure y score higher than any other structure y', leading to the following (generalized perceptron) loss:

$$\max_{y'} score(x, y') - score(x, y)$$

In terms of implementation, this means: create a computation graph CG_p for each of the possible parts, and calculate its score. Then, run inference over the scored parts to find the best scoring structure y'. Connect the output nodes of the computation graphs corresponding to parts in the gold (predicted) structure y (y') into a summing node CG_y (CG'_y). Connect CG_y and CG'_y using a "minus" node, CG_l, and compute the gradients.

As argued in (LeCun et al., 2006, Section 5), the generalized perceptron loss may not be a good loss function when training structured prediction neural networks as it does not have a margin, and a margin-based hinge loss is preferred:

$$max(0, m + score(x, y) - \max_{y' \neq y} score(x, y'))$$

It is trivial to modify the implementation above to work with the hinge loss.

Note that in both cases we lose the nice properties of the linear model. In particular, the model is no longer convex. This is to be expected, as even the simplest non-linear neural network is already non-convex. Nonetheless, we could still use standard neural-network optimization techniques to train the structured model.

Training and inference is slower, as we have to evaluate the neural network (and take gradients) $|parts(x, y)|$ times.

Structured prediction is a vast field and is beyond the scope of this tutorial, but loss functions, regularizers and methods described in, e.g., (Smith, 2011), such as cost-augmented decoding, can be easily applied or adapted to the neural-network framework.[23]

Probabilistic objective (CRF) In a probabilistic framework ("CRF"), we treat each of the parts scores as a *clique potential* (see (Smith, 2011)) and define the score of each structure y to be:

$$score_{CRF}(x, y) = P(y|x) = \frac{\sum_{p \in y} e^{score(p)}}{\sum_{y' \in \mathcal{Y}(x)} \sum_{p \in y'} e^{score(p)}} = \frac{\sum_{p \in y} e^{NN(c(p))}}{\sum_{y' \in \mathcal{Y}(x)} \sum_{p \in y'} e^{NN(c(p))}}$$

The scoring function defines a conditional distribution $P(y|x)$, and we wish to set the parameters of the network such that corpus conditional log likelihood $\sum_{(x_i, y_i) \in training} \log P(y_i|x_i)$ is maximized.

The loss for a given training example (x, y) is then: $-\log score_{CRF}(x, y)$. Taking the gradient with respect to the loss is as involved as building the associated computation graph. The tricky part is the denominator (the *partition function*) which requires summing over the potentially exponentially many structures in $\mathcal{Y}$. However, for some problems, a dynamic programming algorithm exists for efficiently solving the summation in polynomial time. When such an algorithm exists, it can be adapted to also create a polynomial-size computation graph.

When an efficient enough algorithm for computing the partition function is not available, approximate methods can be used. For example, one may use beam search for inference, and for the partition function sum over the structures remaining in the beam instead of over the exponentially large $\mathcal{Y}(x)$.

A hinge based approached was used by Pei et al (2015) for arc-factored dependency parsing, and the probabilistic approach by Durrett and Klein (Durrett & Klein, 2015) for a CRF constituency parser. The approximate beam-based partition function was effectively used by Zhou et al (2015) in a transition based parser.

Reranking When searching over all possible structures is intractable, inefficient or hard to integrate into a model, reranking methods are often used. In the reranking framework (Charniak & Johnson, 2005; Collins & Koo, 2005) a base model is used to produce a

23. One should keep in mind that the resulting objectives are no longer convex, and so lack the formal guarantees and bounds associated with convex optimization problems. Similarly, the theory, learning bounds and guarantees associated with the algorithms do not automatically transfer to the neural versions.

list of the k-best scoring structures. A more complex model is then trained to score the candidates in the k-best list such that the best structure with respect to the gold one is scored highest. As the search is now performed over k items rather than over an exponential space, the complex model can condition on (extract features from) arbitrary aspects of the scored structure. Reranking methods are natural candidates for structured prediction using neural-network models, as they allow the modeler to focus on the feature extraction and network structure, while removing the need to integrate the neural network scoring into a decoder. Indeed, reranking methods are often used for experimenting with neural models that are not straightforward to integrate into a decoder, such as convolutional, recurrent and recursive networks, which will be discussed in later sections. Works using the reranking approach include (Socher et al., 2013; Auli et al., 2013; Le & Zuidema, 2014; Zhu et al., 2015a)

MEMM and hybrid approaches Other formulations are, of course, also possible. For example, an MEMM (McCallum, Freitag, & Pereira, 2000) can be trivially adapted to the neural network world by replacing the logistic regression ("Maximum Entropy") component with an MLP.

Hybrid approaches between neural networks and linear models are also explored. In particular, Weiss et al (Weiss et al., 2015) report strong results for transition-based dependency parsing in a two-stage model. In the first stage, a static feed-forward neural network (MLP2) is trained to perform well on each of the individual decisions of the structured problem in isolation. In the second stage, the neural network model is held fixed, and the different layers (output as well as hidden layer vectors) for each input are then concatenated and used as the input features of a linear structured perceptron model (Collins, 2002) that is trained to perform beam-search for the best resulting structure. While it is not clear that such training regime is more effective than training a single structured-prediction neural network, the use of two simpler, isolated models allowed the researchers to perform a much more extensive hyper-parameter search (e.g. tuning layer sizes, activation functions, learning rates and so on) for each model than is feasible with more complicated networks.

9. Convolutional Layers

Sometimes we are interested in making predictions based on ordered sets of items (e.g. the sequence of words in a sentence, the sequence of sentences in a document and so on). Consider for example predicting the sentiment (positive, negative or neutral) of a sentence. Some of the sentence words are very informative of the sentiment, other words are less informative, and to a good approximation, an informative clue is informative regardless of its position in the sentence. We would like to feed all of the sentence words into a learner, and let the training process figure out the important clues. One possible solution is feeding a CBOW representation into a fully connected network such as an MLP. However, a downside of the CBOW approach is that it ignores the ordering information completely, assigning the sentences "it was not good, it was actually quite bad" and "it was not bad, it was actually quite good" the exact same representation. While the global position of the indicators "not good" and "not bad" does not matter for the classification task, the local ordering of the words (that the word "not" appears right before the word "bad") is very important. A naive approach would suggest embedding word-pairs (bi-grams) rather than words, and building a CBOW over the embedded bigrams. While such architecture could be effective, it will result in huge embedding matrices, will not scale for longer n-grams, and will suffer from data sparsity problems as it does not share statistical strength between different n-grams (the embedding of "quite good" and "very good" are completely independent of one another, so if the learner saw only one of them during training, it will not be able to deduce anything about the other based on its component words). The convolution-and-pooling (also called convolutional neural networks, or CNNs) architecture is an elegant and robust solution to this modeling problem. A convolutional neural network is designed to identify indicative local predictors in a large structure, and combine them to produce a fixed size vector representation of the structure, capturing these local aspects that are most informative for the prediction task at hand.

Convolution-and-pooling architectures (LeCun & Bengio, 1995) evolved in the neural networks vision community, where they showed great success as object detectors – recognizing an object from a predefined category ("cat", "bicycles") regardless of its position in the image (Krizhevsky et al., 2012). When applied to images, the architecture is using 2-dimensional (grid) convolutions. When applied to text, NLP we are mainly concerned with 1-d (sequence) convolutions. Convolutional networks were introduced to the NLP community in the pioneering work of Collobert, Weston and Colleagues (2011) who used them for semantic-role labeling, and later by Kalchbrenner et al (2014) and Kim (Kim, 2014) who used them for sentiment and question-type classification.

9.1 Basic Convolution + Pooling

The main idea behind a convolution and pooling architecture for language tasks is to apply a non-linear (learned) function over each instantiation of a k-word sliding window over the sentence. This function (also called "filter") transforms a window of k words into a d dimensional vector that captures important properties of the words in the window (each dimension is sometimes referred to in the literature as a "channel"). Then, a "pooling" operation is used combine the vectors resulting from the different windows into a single d-dimensional vector, by taking the max or the average value observed in each of the d

channels over the different windows. The intention is to focus on the most important "features" in the sentence, regardless of their location. The d-dimensional vector is then fed further into a network that is used for prediction. The gradients that are propagated back from the network's loss during the training process are used to tune the parameters of the filter function to highlight the aspects of the data that are important for the task the network is trained for. Intuitively, when the sliding window is run over a sequence, the filter function learns to identify informative k-grams.

More formally, consider a sequence of words $\mathbf{x} = x_1, \ldots, x_n$, each with their corresponding d_{emb} dimensional word embedding $v(x_i)$. A 1d convolution layer[24] of width k works by moving a sliding window of size k over the sentence, and applying the same "filter" to each window in the sequence $(v(x_i); v(x_{i+1}); \ldots; v(x_{i+k-1}))$. The filter function is usually a linear transformation followed by a non-linear activation function.

Let the concatenated vector of the ith window be $\mathbf{w_i} = v(x_i); v(x_{i+1}); \ldots; v(x_{i+k-1})$, $\mathbf{w_i} \in \mathbb{R}^{kd_{emb}}$. Depending on whether we pad the sentence with $k-1$ words to each side, we may get either $m = n - k + 1$ (*narrow convolution*) or $m = n + k + 1$ windows (*wide convolution*) (Kalchbrenner et al., 2014). The result of the convolution layer is m vectors $\mathbf{p_1}, \ldots, \mathbf{p_m}$, $\mathbf{p_i} \in \mathbb{R}^{d_{conv}}$ where:

$$\mathbf{p_i} = g(\mathbf{w_i}\mathbf{W} + \mathbf{b})$$

.

g is a non-linear activation function that is applied element-wise, $\mathbf{W} \in \mathbb{R}^{k \cdot d_{emb} \times d_{conv}}$ and $\mathbf{b} \in \mathbb{R}^{d_{conv}}$ are parameters of the network. Each $\mathbf{p_i}$ is a d_{conv} dimensional vector, encoding the information in $\mathbf{w_i}$. Ideally, each dimension captures a different kind of indicative information. The m vectors are then combined using a *max pooling layer*, resulting in a single d_{conv} dimensional vector $\mathbf{c}$.

$$c_j = \max_{1<i\leq m} \mathbf{p_i}[j]$$

$\mathbf{p_i}[j]$ denotes the jth component of $\mathbf{p_i}$. The effect of the max-pooling operation is to get the most salient information across window positions. Ideally, each dimension will "specialize" in a particular sort of predictors, and max operation will pick on the most important predictor of each type.

Figure 4 provides an illustration of the process.

The resulting vector $\mathbf{c}$ is a representation of the sentence in which each dimension reflects the most salient information with respect to some prediction task. $\mathbf{c}$ is then fed into a downstream network layers, perhaps in parallel to other vectors, culminating in an output layer which is used for prediction. The training procedure of the network calculates the loss with respect to the prediction task, and the error gradients are propagated all the way back through the pooling and convolution layers, as well as the embedding layers. [25]

24. 1d here refers to a convolution operating over 1-dimensional inputs such as sequences, as opposed to 2d convolutions which are applied to images.

25. Besides being useful for prediction, a by-product of the training procedure is a set of parameters $\mathbf{W}$, $\mathbf{B}$ and embeddings $v()$ that can be used in a convolution and pooling architecture to encode arbitrary length sentences into fixed-size vectors, such that sentences that share the same kind of predictive information will be close to each other.

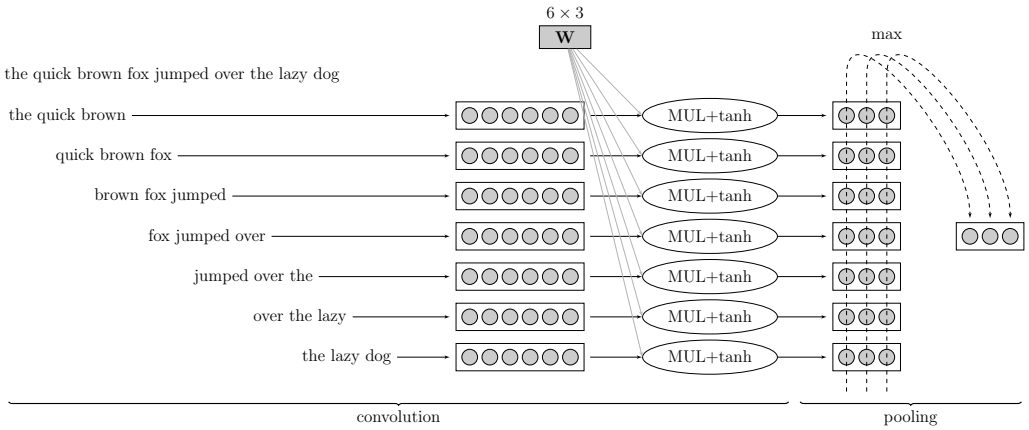


Figure 4: 1d convolution+pooling over the sentence "the quick brown fox jumped over the lazy dog". This is a narrow convolution (no padding is added to the sentence) with a window size of 3. Each word is translated to a 2-dim embedding vector (not shown). The embedding vectors are then concatenated, resulting in 6-dim window representations. Each of the seven windows is transfered through a 6×3 filter (linear transformation followed by element-wise tanh), resulting in seven 3-dimensional filtered representations. Then, a max-pooling operation is applied, taking the max over each dimension, resulting in a final 3-dimensional pooled vector.

While max-pooling is the most common pooling operation in text applications, other pooling operations are also possible, the second most common operation being *average pooling*, taking the average value of each index instead of the max.

9.2 Dynamic, Hierarchical and k-max Pooling

Rather than performing a single pooling operation over the entire sequence, we may want to retain some positional information based on our domain understanding of the prediction problem at hand. To this end, we can split the vectors $\mathbf{p_i}$ into ℓ distinct groups, apply the pooling separately on each group, and then concatenate the ℓ resulting d_{conv} vectors $\mathbf{c_1}, \ldots, \mathbf{c}_\ell$. The division of the $\mathbf{p_i}$s into groups is performed based on domain knowledge. For example, we may conjecture that words appearing early in the sentence are more indicative than words appearing late. We can then split the sequence into ℓ equally sized regions, applying a separate max-pooling to each region. For example, Johnson and Zhang (Johnson & Zhang, 2014) found that when classifying documents into topics, it is useful to have 20 average-pooling regions, clearly separating the initial sentences (where the topic is usually introduced) from later ones, while for a sentiment classification task a single max-pooling operation over the entire sentence was optimal (suggesting that one or two very strong signals are enough to determine the sentiment, regardless of the position in the sentence).

Similarly, in a relation extraction kind of task we may be given two words and asked to determine the relation between them. We could argue that the words before the first word, the words after the second word, and the words between them provide three different kinds of information (Chen et al., 2015). We can thus split the $\mathbf{p_i}$ vectors accordingly, pooling separately the windows resulting from each group.

Another variation is performing *hierarchical pooling*, in which we have a succession of convolution and pooling layers, where each stage applies a convolution to a sequence, pools every k neighboring vectors, performs a convolution on the resulting pooled sequence, applies another convolution and so on. This architecture allows sensitivity to increasingly larger structures.

Finally, (Kalchbrenner et al., 2014) introduced a *k-max pooling* operation, in which the top k values in each dimension are retained instead of only the best one, while preserving the order in which they appeared in the text. For example a, consider the following matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 9 & 6 & 5 \\ 2 & 3 & 1 \\ 7 & 8 & 1 \\ 3 & 4 & 1 \end{bmatrix}$$

A 1-max pooling over the column vectors will result in $\begin{bmatrix} 9 & 8 & 5 \end{bmatrix}$, while a 2-max pooling will result in the following matrix: $\begin{bmatrix} 9 & 6 & 3 \\ 7 & 8 & 5 \end{bmatrix}$ whose rows will then be concatenated to $\begin{bmatrix} 9 & 6 & 3 & 7 & 8 & 5 \end{bmatrix}$

The k-max pooling operation makes it possible to pool the k most active indicators that may be a number of positions apart; it preserves the order of the features, but is insensitive to their specific positions. It can also discern more finely the number of times the feature is highly activated (Kalchbrenner et al., 2014).

9.3 Variations

Rather than a single convolutional layer, several convolutional layers may be applied in parallel. For example, we may have four different convolutional layers, each with a different window size in the range 2–5, capturing n-gram sequences of varying lengths. The result of each convolutional layer will then be pooled, and the resulting vectors concatenated and fed to further processing (Kim, 2014).

The convolutional architecture need not be restricted into the linear ordering of a sentence. For example, Ma et al (2015) generalize the convolution operation to work over syntactic dependency trees. There, each window is around a node in the syntactic tree, and the pooling is performed over the different nodes. Similarly, Liu et al (2015) apply a convolutional architecture on top of dependency paths extracted from dependency trees. Le and Zuidema (2015) propose to perform max pooling over vectors representing the different derivations leading to the same chart item in a chart parser.

10. Recurrent Neural Networks – Modeling Sequences and Stacks

When dealing with language data, it is very common to work with sequences, such as words (sequences of letters), sentences (sequences of words) and documents. We saw how feed-forward networks can accommodate arbitrary feature functions over sequences through the use of vector concatenation and vector addition (CBOW). In particular, the CBOW representations allows to encode arbitrary length sequences as fixed sized vectors. However, the CBOW representation is quite limited, and forces one to disregard the order of features. The convolutional networks also allow encoding a sequence into a fixed size vector. While representations derived from convolutional networks are an improvement above the CBOW representation as they offer some sensitivity to word order, their order sensitivity is restricted to mostly local patterns, and disregards the order of patterns that are far apart in the sequence.

Recurrent neural networks (RNNs) (Elman, 1990) allow representing arbitrarily sized structured inputs in a fixed-size vector, while paying attention to the structured properties of the input.

10.1 The RNN Abstraction

We use $\mathbf{x_{i:j}}$ to denote the sequence of vectors $\mathbf{x_i}, \ldots, \mathbf{x_j}$. The RNN abstraction takes as input an ordered list of input vectors $\mathbf{x_1}, ..., \mathbf{x_n}$ together with an initial *state vector* $\mathbf{s_0}$, and returns an ordered list of state vectors $\mathbf{s_1}, ..., \mathbf{s_n}$, as well as an ordered list of *output vectors* $\mathbf{y_1}, ..., \mathbf{y_n}$. An output vector $\mathbf{y_i}$ is a function of the corresponding state vector $\mathbf{s_i}$. The input vectors $\mathbf{x_i}$ are presented to the RNN in a sequential fashion, and the state vector $\mathbf{s_i}$ and output vector $\mathbf{y_i}$ represent the state of the RNN after observing the inputs $\mathbf{x_{1:i}}$. The output vector $\mathbf{y_i}$ is then used for further prediction. For example, a model for predicting the conditional probability of an event e given the sequence $\mathbf{m_{1:i}}$ can be defined as $p(e = j|\mathbf{x_{1:i}}) = softmax(\mathbf{y_i W} + \mathbf{b})[j]$. The RNN model provides a framework for conditioning on the entire history $\mathbf{x_1}, \ldots, \mathbf{x_i}$ without resorting to the Markov assumption which is traditionally used for modeling sequences. Indeed, RNN-based language models result in very good perplexity scores when compared to n-gram based models.

Mathematically, we have a recursively defined function R that takes as input a state vector $\mathbf{s_i}$ and an input vector $\mathbf{x_{i+1}}$, and results in a new state vector $\mathbf{s_{i+1}}$. An additional function O is used to map a state vector $\mathbf{s_i}$ to an output vector $\mathbf{y_i}$. When constructing an RNN, much like when constructing a feed-forward network, one has to specify the dimension of the inputs $\mathbf{x_i}$ as well as the dimensions of the outputs $\mathbf{y_i}$. The dimensions of the states $\mathbf{s_i}$ are a function of the output dimension.[26]

26. While RNN architectures in which the state dimension is independent of the output dimension are possible, the current popular architectures, including the Simple RNN, the LSTM and the GRU do not follow this flexibility.

$$
\begin{aligned}
RNN(\mathbf{s_0}, \mathbf{x_{1:n}}) = & \mathbf{s_{1:n}},\ \mathbf{y_{1:n}} \\
& \mathbf{s_i} = R(\mathbf{s_{i-1}}, \mathbf{x_i}) \\
& \mathbf{y_i} = O(\mathbf{s_i})
\end{aligned}
$$

$$
\mathbf{x_i} \in \mathbb{R}^{d_{in}},\ \ \mathbf{y_i} \in \mathbb{R}^{d_{out}},\ \ \mathbf{s_i} \in \mathbb{R}^{f(d_{out})}
$$

The functions R and O are the same across the sequence positions, but the RNN keeps track of the states of computation through the state vector that is kept and being passed between invocations of R.

Graphically, the RNN has been traditionally presented as in Figure 5.

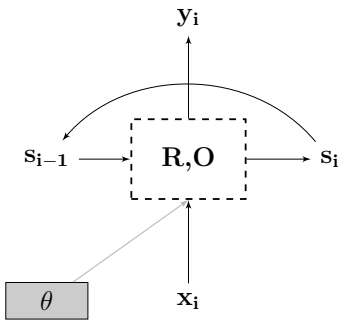


Figure 5: Graphical representation of an RNN (recursive).

This presentation follows the recursive definition, and is correct for arbitrary long sequences. However, for a finite sized input sequence (and all input sequences we deal with are finite) one can *unroll* the recursion, resulting in the structure in Figure 6.

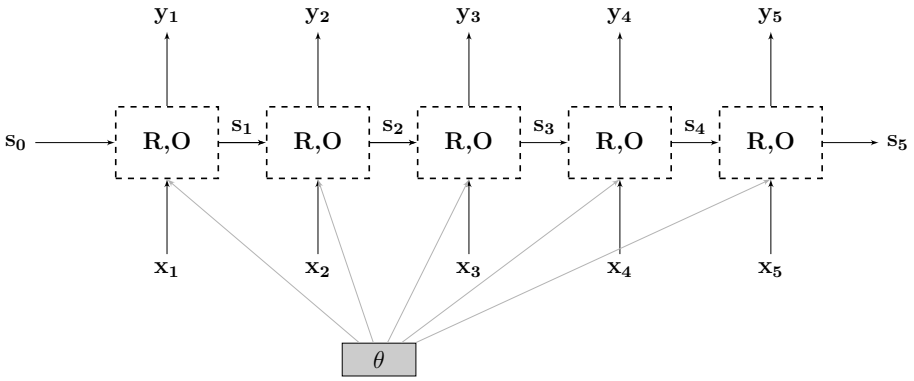


Figure 6: Graphical representation of an RNN (unrolled).

While not usually shown in the visualization, we include here the parameters θ in order to highlight the fact that the same parameters are shared across all time steps. Different

instantiations of R and O will result in different network structures, and will exhibit different properties in terms of their running times and their ability to be trained effectively using gradient-based methods. However, they all adhere to the same abstract interface. We will provide details of concrete instantiations of R and O – the Simple RNN, the LSTM and the GRU – in Section 11. Before that, let's consider modeling with the RNN abstraction.

First, we note that the value of $\mathbf{s_i}$ is based on the entire input $\mathbf{x_1}, ..., \mathbf{x_i}$. For example, by expanding the recursion for $i = 4$ we get:

$$\begin{aligned}
\mathbf{s_4} =& R(\mathbf{s_3}, \mathbf{x_4}) \\
=& R(\overbrace{R(\mathbf{s_2}, \mathbf{x_3})}^{\mathbf{s_3}}, \mathbf{x_4}) \\
=& R(R(\overbrace{R(\mathbf{s_1}, \mathbf{x_2})}^{\mathbf{s_2}}, \mathbf{x_3}), \mathbf{x_4}) \\
=& R(R(R(\overbrace{R(\mathbf{s_0}, \mathbf{x_1})}^{\mathbf{s_1}}, \mathbf{x_2}), \mathbf{x_3}), \mathbf{x_4})
\end{aligned}$$

Thus, $\mathbf{s_n}$ (as well as $\mathbf{y_n}$) could be thought of as *encoding* the entire input sequence.[27] Is the encoding useful? This depends on our definition of usefulness. The job of the network training is to set the parameters of R and O such that the state conveys useful information for the task we are tying to solve.

10.2 RNN Training

Viewed as in Figure 6 it is easy to see that an unrolled RNN is just a very deep neural network (or rather, a very large *computation graph* with somewhat complex nodes), in which the same parameters are shared across many parts of the computation. To train an RNN network, then, all we need to do is to create the unrolled computation graph for a given input sequence, add a loss node to the unrolled graph, and then use the backward (backpropagation) algorithm to compute the gradients with respect to that loss. This procedure is referred to in the RNN literature as *backpropagation through time*, or BPTT (Werbos, 1990).[28] There are various ways in which the supervision signal can be applied.

Acceptor One option is to base the supervision signal only on the final output vector, $\mathbf{y_n}$. Viewed this way, the RNN is an *acceptor*. We observe the final state, and then decide

27. Note that, unless R is specifically designed against this, it is likely that the later elements of the input sequence have stronger effect on $\mathbf{s_n}$ than earlier ones.

28. Variants of the BPTT algorithm include unrolling the RNN only for a fixed number of input symbols at each time: first unroll the RNN for inputs $\mathbf{x_{1:k}}$, resulting in $\mathbf{s_{1:k}}$. Compute a loss, and backpropagate the error through the network (k steps back). Then, unroll the inputs $\mathbf{x_{k+1:2k}}$, this time using $\mathbf{s_k}$ as the initial state, and again backpropagate the error for k steps, and so on. This strategy is based on the observations that for the Simple-RNN variant, the gradients after k steps tend to vanish (for large enough k), and so omitting them is negligible. This procedure allows training of arbitrarily long sequences. For RNN variants such as the LSTM or the GRU that are designed specifically to mitigate the vanishing gradients problem, this fixed size unrolling is less motivated, yet it is still being used, for example when doing language modeling over a book without breaking it into sentences.

on an outcome.[29] For example, consider training an RNN to read the characters of a word one by one and then use the final state to predict the part-of-speech of that word (this is inspired by (Ling et al., 2015b)), an RNN that reads in a sentence and, based on the final state decides if it conveys positive or negative sentiment (this is inspired by (Wang et al., 2015b)) or an RNN that reads in a sequence of words and decides whether it is a valid noun-phrase. The loss in such cases is defined in terms of a function of $\mathbf{y_n} = O(\mathbf{s_n})$, and the error gradients will backpropagate through the rest of the sequence (see Figure 7).[30] The loss can take any familiar form – cross entropy, hinge, margin, etc.

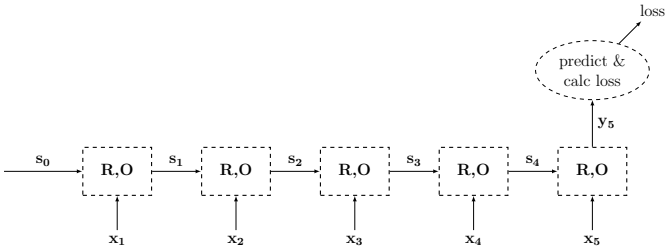


Figure 7: Acceptor RNN Training Graph.

Encoder Similar to the acceptor case, an encoder supervision uses only the final output vector, $\mathbf{y_n}$. However, unlike the acceptor, where a prediction is made solely on the basis of the final vector, here the final vector is treated as an encoding of the information in the sequence, and is used as additional information together with other signals. For example, an extractive document summarization system may first run over the document with an RNN, resulting in a vector $\mathbf{y_n}$ summarizing the entire document. Then, $\mathbf{y_n}$ will be used together with other features in order to select the sentences to be included in the summarization.

Transducer Another option is to treat the RNN as a transducer, producing an output for each input it reads in. Modeled this way, we can compute a local loss signal $L_{local}(\hat{\mathbf{y}}_\mathbf{i}, \mathbf{y_i})$ for each of the outputs $\hat{\mathbf{y}}_\mathbf{i}$ based on a true label $\mathbf{y_i}$. The loss for unrolled sequence will then be: $L(\hat{\mathbf{y}}_{\mathbf{1:n}}, \mathbf{y_{1:n}}) = \sum_{i=1}^{n} L_{local}(\hat{\mathbf{y}}_\mathbf{i}, \mathbf{y_i})$, or using another combination rather than a sum such as an average or a weighted average (see Figure 8). One example for such a transducer is a sequence tagger, in which we take $\mathbf{x_{i:n}}$ to be feature representations for the n words of a sentence, and $\mathbf{y_i}$ as an input for predicting the tag assignment of word i based on words 1:i. A CCG super-tagger based on such an architecture provides state-of-the art CCG super-tagging results (Xu et al., 2015).

A very natural use-case of the transduction setup is for language modeling, in which the sequence of words $\mathbf{x_{1:i}}$ is used to predict a distribution over the $i+1$th word. RNN based

29. The terminology is borrowed from Finite-State Acceptors. However, the RNN has a potentially infinite number of states, making it necessary to rely on a function other than a lookup table for mapping states to decisions.

30. This kind of supervision signal may be hard to train for long sequences, especially so with the Simple-RNN, because of the vanishing gradients problem. It is also a generally hard learning task, as we do not tell the process on which parts of the input to focus.

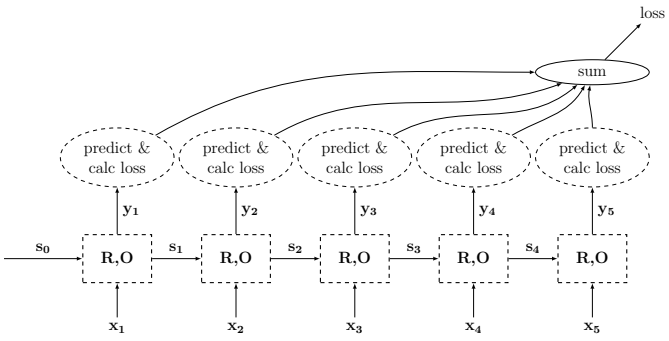


Figure 8: Transducer RNN Training Graph.

language models are shown to provide much better perplexities than traditional language models (Mikolov et al., 2010; Sundermeyer, Schlüter, & Ney, 2012; Mikolov, 2012).

Using RNNs as transducers allows us to relax the Markov assumption that is traditionally taken in language models and HMM taggers, and condition on the entire prediction history. The power of the ability to condition on arbitrarily long histories is demonstrated in generative character-level RNN models, in which a text is generated character by character, each character conditioning on the previous ones (Sutskever, Martens, & Hinton, 2011). The generated texts show sensitivity to properties that are not captured by n-gram language models, including line lengths and nested parenthesis balancing. For a good demonstration and analysis of the properties of RNN-based character level language models, see (Karpathy, Johnson, & Li, 2015).

Encoder - Decoder Finally, an important special case of the encoder scenario is the Encoder-Decoder framework (Cho, van Merrienboer, Bahdanau, & Bengio, 2014a; Sutskever et al., 2014). The RNN is used to encode the sequence into a vector representation $\mathbf{y_n}$, and this vector representation is then used as auxiliary input to another RNN that is used as a decoder. For example, in a machine-translation setup the first RNN encodes the source sentence into a vector representation $\mathbf{y_n}$, and then this state vector is fed into a separate (decoder) RNN that is trained to predict (using a transducer-like language modeling objective) the words of the target language sentence based on the previously predicted words as well as $\mathbf{y_n}$. The supervision happens only for the decoder RNN, but the gradients are propagated all the way back to the encoder RNN (see Figure 9).

Such an approach was shown to be surprisingly effective for Machine Translation (Sutskever et al., 2014) using LSTM RNNs. In order for this technique to work, Sutskever et al found it effective to input the source sentence in reverse, such that $\mathbf{x_n}$ corresponds to the first word of the sentence. In this way, it is easier for the second RNN to establish the relation between the first word of the source sentence to the first word of the target sentence. Another use-case of the encoder-decoder framework is for sequence transduction. Here, in order to generate tags $t_1, \ldots, t_n$, an encoder RNN is first used to encode the sentence $\mathbf{x_{1:n}}$ into fixed sized vector. This vector is then fed as the initial state vector of another (transducer) RNN, which is used together with $\mathbf{x_{1:n}}$ to predict the label t_i at each position i. This approach