Embeddings, neural networks and language models

James A. Duffy

Chapters 6, 7 and 9 of
Jurafsky & Martin, *Speech and Language Processing*

Outline

1. Embeddings: words as euclidean vectors
2. Neural networks
3. Language models and NNs
4. Recursive NNs

- Ultimate objectives?
  - to mathematically encode the relationships between / meaning of words $\Longrightarrow$ embeddings
  - to train a *language model*, a model that is able to predict the next word in a sentence, given the immediately preceding words $\Longrightarrow$ neural language models

Vector semantics and embeddings

- Meaning of / relationship between words has been conceptualised in many ways (see Sec. 6.1)
- Vector semantics identifies the meaning of a word with its *distribution* in language use:
  - essentially, the relative frequency with which it occurs in proximity to other words
  - i.e. its *co-occurrence* with other words
- *Embeddings* represent the distribution of a word in terms of a vector in Euclidean space
  - 'sparse' embeddings (long vectors with many zeros): tf-idf or PPMI
  - 'dense' embeddings (shorter vectors): word2vec
- Representation is exceedingly useful, because it renders the 'meaning' of a word as a mathematical object
- Encodes words in a manner suitable for input into a language model, neural network, etc.

Vectors and documents

- Suppose we have a corpus of documents, and we want to quantify the similarities / differences between them
- Ultimate objective could be document retrieval: you provide the system with a document, and ask it to retrieve similar documents.
- *Term-document matrix* lists the frequencies with with which words appear in each document

	As You Like It	**Twelfth Night**	**Julius Caesar**	**Henry V**
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

Figure 6.2 The term-document matrix for four words in four Shakespeare plays. Each cell contains the number of times the (row) word occurs in the (column) document.

- Here the corpus is four works; the vocabulary V consists of four words ($|V| = 4$)

Vectors and documents

- Column vectors describe ('embed') the documents

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

Figure 6.3 The term-document matrix for four words in four Shakespeare plays. The red boxes show that each document is represented as a column vector of length four.

- How to measure similarity between two documents / vectors, $\boldsymbol{v}$ and $\boldsymbol{w}$?
  - ordinary euclidean distance $\|\boldsymbol{v} - \boldsymbol{w}\|$ inappropriate, because dependent on magnitudes of entries
  - we should first normalise the vectors to have unit length, i.e. $\boldsymbol{v}/\|\boldsymbol{v}\|$, etc., then use euclidean distance, or cosine similarity

$$\cos\theta = \frac{\boldsymbol{v}^{\mathsf{T}}\boldsymbol{w}}{\|\boldsymbol{v}\|\|\boldsymbol{w}\|}$$

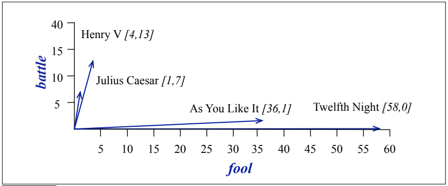

Figure 6.4 A spatial visualization of the document vectors for the four Shakespeare play documents, showing just two of the dimensions, corresponding to the words *battle* and *fool*. The comedies have high values for the *fool* dimension and low values for the *battle* dimension.

Vectors and words

- Row vectors could be used to represent the meaning of words

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

Figure 6.5 The term-document matrix for four words in four Shakespeare plays. The red boxes show that each word is represented as a row vector of length four.

- But gives a very coarse-grained measure of meaning (particularly if each document is large!)

Vectors and words

- Better approach is to construct a *term-term* matrix
  - choose the 'context': a fixed window length, e.g. ± 4 words
  - for each word w in the vocabulary V, record how many times another word $v \in V$ appears within w's context, across a corpus
  - yields a $|V|$-length vector of co-occurrences

	aardvark	...	**computer**	**data**	**result**	**pie**	**sugar**	...
cherry	0	...	2	8	9	442	25	...
strawberry	0	...	0	0	1	60	19	...
digital	0	...	1670	1683	85	5	4	...
information	0	...	3325	3982	378	5	13	...

Figure 6.6 Co-occurrence vectors for four words in the Wikipedia corpus, showing six of the dimensions (hand-picked for pedagogical purposes). The vector for *digital* is outlined in red. Note that a real vector would have vastly more dimensions and thus be much sparser.

- Vectors are *sparse*: if $V = 50,000$, most words will *never* appear in the neighbourhood of most others

Vectors and words

- As with documents, we can use cosine to gauge similarity between words

	pie	data	computer
cherry	442	8	2
digital	5	1683	1670
information	5	3982	3325

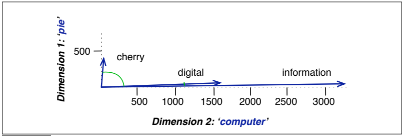

Figure 6.8 A (rough) graphical demonstration of cosine similarity, showing vectors for three words (*cherry*, *digital*, and *information*) in the two dimensional space defined by counts

- Raw frequencies overly skewed by high co-occurrences with words that are uninformative about meaning, e.g. *the*, *it*, *they*, etc.
- Whereas words that occur very infrequently may be *highly* informative about the meaning of neighbouring words
- Weighting schemes (td-idf) or the PPMI algorithm provide an alternative way of producing (sparse) vectors, that are less affected by these problems (Sec. 6.5–6.7)

Dense embeddings

- However, a better approach in practice appears to be to use dense embedding vectors (of length around 300 rather than $30,000$)
- Appear better able e.g. to capture synonymy between words, which is lost by a sparse vector that treats very similar words (e.g. *car* and *automobile*) as entirely separate entries of the vocabulary
- Leading example is the word2vec algorithm, which is based on a classification / prediction problem

Word2vec

- Suppose we take the *context* of a word w to a ± 2 word window, as e.g.

```
... lemon,  a [tablespoon of apricot jam,     a] pinch ...
               c1         c2    w     c3      c4
```

- Given the word w, what is the probability $\mathbb{P}(+ \mid w, c)$ that some other word $c \in V$ appears in w's context?
- Let $\boldsymbol{w}$ and $\boldsymbol{c}$ denote (dense) $\mathbb{R}^d$-valued embeddings for these words. Then

$$\mathbb{P}(+ \mid w, c) = \frac{1}{1 + \exp(-\boldsymbol{c}^\top \boldsymbol{w})} = \sigma(\boldsymbol{c}^\top \boldsymbol{w})$$

  so the probability is highest for words that are 'similar' in the sense that $\boldsymbol{c}^\top \boldsymbol{w}$ is large and positive
- Ultimately, the collection of $\boldsymbol{w}$'s and $\boldsymbol{c}$'s, stacked (columnwise) in the matrices $\boldsymbol{W}$ and $\boldsymbol{C}$, will provide our embeddings for the words in V
- The problem then is to estimate $\boldsymbol{W}$ and $\boldsymbol{C}$, i.e. to 'train the classifier' on a corpus of text

Word2vec

```
... lemon,  a [tablespoon of apricot jam,      a] pinch ...
               c1          c2    w     c3      c4
```

- Want to construct a quasi-likelihood / loss function to estimate the model. What do we learn when observe the above?

1. $c_1, \ldots, c_L \in V$ appear in the context of w; if we assume (heroically!) that context words appear independently of each other

$$\mathbb{P}(+ \mid w, c_1, \ldots, c_L) = \prod_{i=1}^{L} \mathbb{P}(+ \mid w, c_i) = \prod_{i=1}^{L} \sigma(\boldsymbol{c}_i^\top \boldsymbol{w})$$

2. $c \in V \setminus \{c_1, \ldots, c_L\}$ did *not* appear in the context for w; for a *single* word c, this occurs with probability

$$\mathbb{P}(- \mid w, c) = 1 - \sigma(\boldsymbol{c}^\top \boldsymbol{w});$$

- Assuming independence, the log quasi-likelihood of observing w in the context of $(c_1, \ldots, c_L)$ would be

$$\sum_{i=1}^{L} \log \sigma(\boldsymbol{c}_i^\top \boldsymbol{w}) + \sum_{c \in V \setminus \{c_1, \ldots, c_L\}} \log[1 - \sigma(\boldsymbol{c}^\top \boldsymbol{w})]$$

Word2vec

$$\sum_{i=1}^{L} \log \sigma(\boldsymbol{c}_i^\top \boldsymbol{w}) + \sum_{c \in V \setminus \{c_1, \ldots, c_L\}} \log[1 - \sigma(\boldsymbol{c}^\top \boldsymbol{w})]$$

- Problem: objective is overwhelmed by the second term
- So instead, replace the by a random selection of k 'noise' words, chosen in proportion to some weighted frequency measure, e.g.

```
... lemon,  a [tablespoon of apricot jam,      a] pinch ...
               c1          c2    w      c3      c4
```

positive examples +

w	c_{pos}
apricot	tablespoon
apricot	of
apricot	jam
apricot	a

negative examples -

w	c_{neg}	w	c_{neg}
apricot	aardvark	apricot	seven
apricot	my	apricot	forever
apricot	where	apricot	dear
apricot	coaxial	apricot	if

- Leads to, for $c_{*,i}$ drawn randomly from $V \setminus \{c_1, \ldots, c_L\}$:

$$\sum_{i=1}^{L} \log \sigma(\boldsymbol{c}_i^\top \boldsymbol{w}) + \sum_{i=1}^{kL} \log[1 - \sigma(\boldsymbol{c}_{*,i}^\top \boldsymbol{w})]$$

Word2vec

$$\sum_{i=1}^{L} \log \sigma(\boldsymbol{c}_i^\mathsf{T} \boldsymbol{w}) + \sum_{i=1}^{kL} \log[1 - \sigma(\boldsymbol{c}_{*,i}^\mathsf{T} \boldsymbol{w})]$$

- We then sum this over all word w and context $(c_1, \ldots, c_L)$ pairs, and maximise with the aid of stochastic gradient ascent.
- Yields a collection of word $\boldsymbol{w}_i$ and context $\boldsymbol{c}_i$ parameter vectors / embeddings of length d; for each word $w_i \in V$
- We may take $\boldsymbol{w}_i$ or e.g. $\boldsymbol{w}_i + \boldsymbol{c}_i$ to be the word2vec embedding
- [How should we choose d? Cross-validation / information criteria?]
- For some of the 'nice' semantic properties of these embeddings, see Sec. 6.10

Next step: language models

- Now we have a way to (usefully) represent words as vectors
- We can start to mathematically model the dependence between words in sentences
- But this dependence may be very complicated ...

Neural networks: motivation

- The mapping from context to (the distribution of the next) word

$$\underbrace{\text{I have to make sure the cat gets}}_{\text{context}} \begin{cases} \text{fed} \\ \text{spayed} \\ \text{???} \end{cases}$$

  - is potentially highly nonlinear with unknown functional forms
  - we have a potentially enormous large corpus of text from which to estimate it
  - but little a priori theoretical guidance as to what class a 'good' model might come from
- A nonparametric estimation problem?
  - we need a flexible class of models capable of approximating a wide range of functions
  - neural networks provide a nonlinear universal approximator

Computational units

- Simple NNs are composed of (layers of) units of the form

$$a = g(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}) = g\left(\sum_{i=1}^{n} w_i x_i\right)$$

- $\boldsymbol{x} = (x_1, \ldots, x_n)$ is a vector of n inputs (includes a constant input)
- a is the real-valued output
- g is monotone, typically either:
    - sigmoid: $\sigma(z) = 1/(1 + \mathrm{e}^{-z})$, maps to $[0, 1]$
    - $\tanh(z) = (\mathrm{e}^z - \mathrm{e}^{-z})/(\mathrm{e}^z + \mathrm{e}^{-z})$, maps to $[-1, 1]$
    - 'rectified linear': $\mathrm{ReLU}(z) = \max\{z, 0\}$, maps to $[0, \infty)$.

Computational units

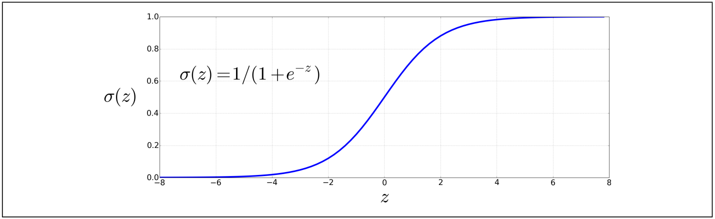

Figure 7.1 The sigmoid function takes a real value and maps it to the range $(0,1)$. It is nearly linear around 0 but outlier values get squashed toward 0 or 1.

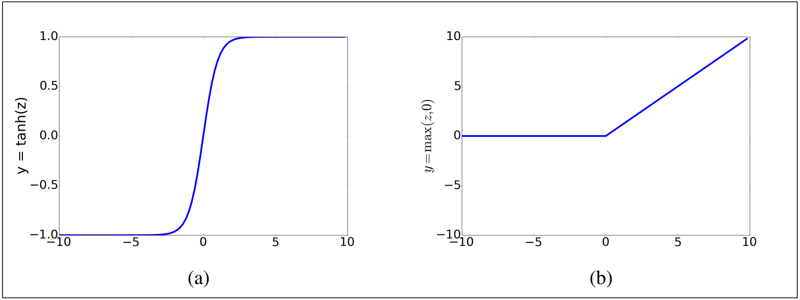

Figure 7.3 The tanh and ReLU activation functions.

Feedforward neural networks

- One unit cannot approximate much on its own: [see their XOR example]
  - it is merely a transformed linear (affine) function
  - the extent of the possible nonlinearity is extremely circumscribed
- We can do much better by ‘nesting’ multiple units within each other
  - hierarchy of units: taking inputs from previous ‘layers’, providing output to subsequent ‘layers’ of units
  - basis for feedforward neural networks (NNs): multiple layers, but no cycles
- Nonlinearity is important: multiple nested layers of *linear* units are equivalent to a single linear function

Feedforward neural networks

- 2-layer example: one 'hidden' and one 'output' layer
- n_0 inputs given by $\boldsymbol{x} = (x_1, \ldots, x_{n_0})$
- n_1 units in the hidden layer, of the form

$$\boldsymbol{h} = \begin{bmatrix} h_1 \\ \vdots \\ h_{n_1} \end{bmatrix} = \begin{bmatrix} g(\boldsymbol{w}_1^\top \boldsymbol{x}) \\ \vdots \\ g(\boldsymbol{w}_{n_1}^\top \boldsymbol{x}) \end{bmatrix} = \boldsymbol{g}(\boldsymbol{W}\boldsymbol{x}), \qquad \boldsymbol{W} = \begin{bmatrix} \boldsymbol{w}_1^\top \\ \vdots \\ \boldsymbol{w}_{n_1}^\top \end{bmatrix}$$

  - produces a *representation* of the input
- output layer:
  - takes linear combinations $\boldsymbol{z} = \boldsymbol{U}\boldsymbol{h}$ of the outputs of the hidden layer
  - produces a $\mathbb{R}^{n_2}$-valued output, where n_2 (and the transformation of $\boldsymbol{z}$ used to get it) depends on the problem
  - e.g. if we want to produce a probability distribution over the next word, use

$$[\text{softmax}(\boldsymbol{z})]_i = \frac{\exp(z_i)}{\sum_{j=1}^{d} \exp(z_j)}$$

  for $\boldsymbol{z} = (z_1, \ldots, z_d) \in \mathbb{R}^d$; the softmax function [the multinomial logit pmf]

Feedforward neural networks

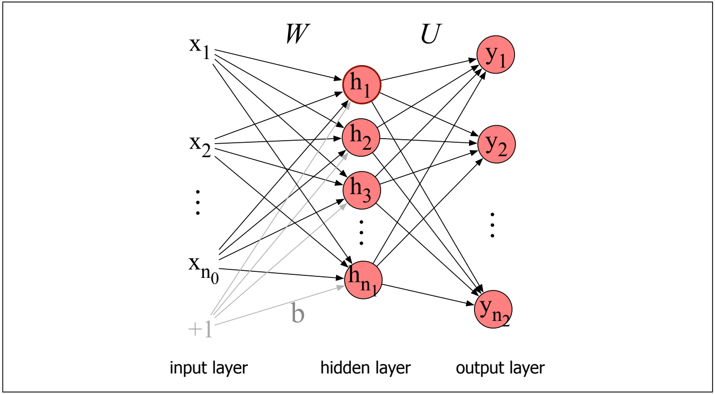

Figure 7.8 A simple 2-layer feedforward network, with one hidden layer, one output layer, and one input layer (the input layer is usually not counted when enumerating layers).

$$\text{hidden:} \quad \boldsymbol{h} = \boldsymbol{g}(\boldsymbol{W}\boldsymbol{x})$$

$$\text{output:} \quad \boldsymbol{y} = \boldsymbol{f}(\boldsymbol{U}\boldsymbol{h})$$

Training [estimation]

- Data: suppose we observe pairs of the form $(\boldsymbol{y}, \boldsymbol{x})$
  - $\boldsymbol{y}$ is a K-vector of all zeros, except for one element equal to unity
  - nonzero element indicates which of the K outcomes actually 'happened'
  - 'self supervised' learning, because the data automatically contains the correct outcomes
- NN yields a ('probabilistic') prediction $\hat{\boldsymbol{y}} = \hat{\boldsymbol{y}}(\boldsymbol{x})$ of $\boldsymbol{y}$
- 'Loss function': *cross entropy loss*; for a single observation

$$
\begin{aligned}
L_{\mathrm{CE}}(\boldsymbol{W}, \boldsymbol{U}; \boldsymbol{y}, \boldsymbol{x}) &= -\sum_{k=1}^{K} y_k \log \hat{y}_k(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{U}) \\
&= -\sum_{k=1}^{K} 1\{y_k = 1\} \log \hat{y}_k(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{U})
\end{aligned}
$$

  - equals conditional probability assigned by the model to the *actual* outcome
  - just the negative of the (quasi-)loglikelihood, for a model in which $\boldsymbol{y}$ has MNL distribution with probabilities given by $\hat{\boldsymbol{y}} = \hat{\boldsymbol{y}}(\boldsymbol{x})$, conditional on $\boldsymbol{x}$

Training [estimation]

$$L_{\text{CE}}(\boldsymbol{W}, \boldsymbol{U}; \boldsymbol{y}, \boldsymbol{x}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{U})$$

- Want to minimise the loss / maximise the quasi-likelihood
- Can we use (stochastic) gradient descent?
  - $\hat{y}_k(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{U})$ is a potentially complicated nonlinear function of $\boldsymbol{W}$ and $\boldsymbol{U}$: multiple units in multiple layers
  - but all the constituent units involve only *smooth* transformations, so derivatives always exist
- Calculation of the gradient:
  - can be broken down into manageable pieces via 'backwards differentiation'
  - basically, compute gradient at each layer, and then combine as per the chain rule
- Non-convex objective, so potentially highly sensitive to starting values

Language models and NNs

- Language models:
  - aim to predict the next next word in a sentence, based on the preceding words, the *context*

    I have to make sure the cat gets ???

  - mathematically, model the conditional distribution of the tth word w_t given $w_{t-1}, w_{t-2}, \ldots$
- N-gram language models: [Ch. 3]
  - suppose this dependence is Markovian, for some window length N

$$p(w_t \mid w_{t-1}, w_{t-2}, \ldots) = p(w_t \mid w_{t-1}, \ldots, w_{t-N+1}) = p(w_t \mid \boldsymbol{c}_{t-1})$$

  - estimate $p(w_t \mid \boldsymbol{c}_{t-1})$ 'nonparametrically' using observed relative frequencies (or modifications thereof)
- Weaknesses of N-gram models:
  - we may see very few occurrences of the exact $(w_t, \boldsymbol{c}_{t-1})$ even in *huge* datasets
  - no way of exploiting similarities between meanings of words to learn about $p(w_t \mid \boldsymbol{c}_{t-1})$ from 'similar' $(w_t', \boldsymbol{c}_{t-1}')$ [e.g. '... dog gets fed' above]

Language models and NNs

- Neural language models:
  - may make the same Markovian assumption as N-gram models
  - but work with *word embeddings*, which encode approximate similarities between word meanings
  - embeddings encoded in a matrix $\boldsymbol{E} \in \mathbb{R}^{d \times |V|}$, where d is the dimension of the embedding, and $|V|$ the length of the vocabulary
- Structure otherwise that of a generic neural network

$$\begin{aligned} \text{input:} \quad & \boldsymbol{e} = [\boldsymbol{E}\boldsymbol{x}_{t-N+1}; \ldots; \boldsymbol{E}\boldsymbol{x}_{t-1}] \\ \text{hidden:} \quad & \boldsymbol{h} = \boldsymbol{g}(\boldsymbol{W}\boldsymbol{e}) \\ \text{output:} \quad & \boldsymbol{y} = \text{softmax}(\boldsymbol{U}\boldsymbol{h}) \end{aligned}$$

where $x_{it} = 1$ if observed word $w_t =$ the ith entry in the vocabulary V, and 0 otherwise

  - output = conditional probability distribution over w_t, given $w_{t-1}, \ldots, w_{t-N+1}$
- Parameters to be estimated: $\boldsymbol{W}$, $\boldsymbol{U}$ (as before), and (possibly) also $\boldsymbol{E}$

Language models and NNs

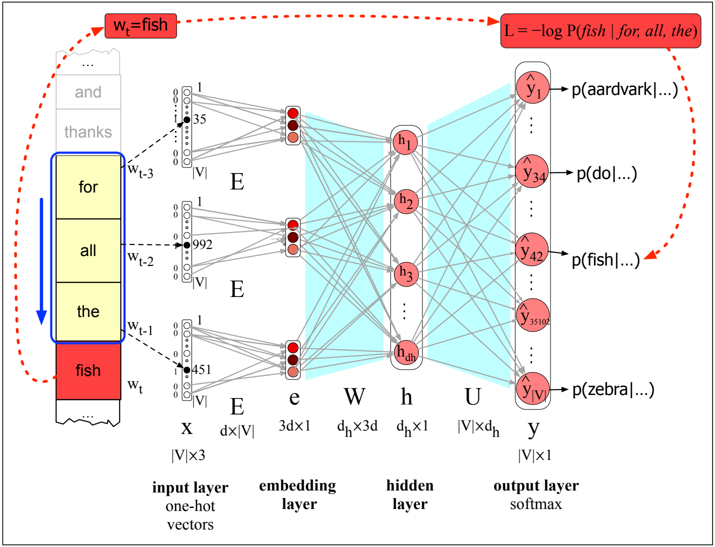

Figure 7.18 Learning all the way back to embeddings. Again, the embedding matrix **E** is shared among the 3 context words.

Recursive NNs: background

- How can we improve on the preceding?
- Feedforward neural language models impose *finite dependence* of w_t on the context $\boldsymbol{c}_{t-1} = (w_{t-1}, \ldots, w_{t-N+1})$
  - only the previous N words matter, e.g. if $N = 3$

    I have to make sure the cat gets ???

    anything prior to 'the' is entirely forgotten
  - a consequence of the Markov assumption, recall

$$p(w_t \mid w_{t-1}, w_{t-2}, \ldots) = p(w_t \mid w_{t-1}, \ldots, w_{t-N+1}) = p(w_t \mid \boldsymbol{c}_{t-1})$$

  - but this isn't how any language works!
- How can we build a model with potentially longer-lived dependence? We need to relax the Markov assumption!

Aside: linear state-space models

- The linear counterpart of a feedforward NN is an AR(N) model

$$
\begin{aligned}
w_t &= \theta_1 w_{t-1} + \theta_2 w_{t-2} + \cdots + \theta_N w_{t-N+1} + u_t \\
&= \theta w_{t-1} + u_t
\end{aligned}
$$

  taking $N = 1$ for simplicity; here u_t is (say) i.i.d.
  - *given* w_{t-1}, the conditional distribution of w_t does not depend on w_{t-s} for any $s \geq 2$
- Compare with the state-space model

$$
\begin{aligned}
w_{t+1} &= \beta h_t + u_{t+1} \\
h_t &= \phi h_{t-1} + w_t
\end{aligned}
$$

  $\{w_t\}$ is the observed process, h_t the unobserved state
  - if $h_0 = 0$, can be 'unrolled' back to period $t = 0$ as

$$
w_{t+1} = \beta h_t + u_{t+1} = \beta \sum_{i=0}^{t-1} \phi^i w_{t-i} + u_{t+1}
$$

  - now the conditional distribution of w_t depends on *every* past w_{t-i} (which decreases if $|\phi| < 1$)
- Recursive NN: extend this idea to neural network models, by introducing (a vector of) latent hidden autoregressive states

Recursive NNs

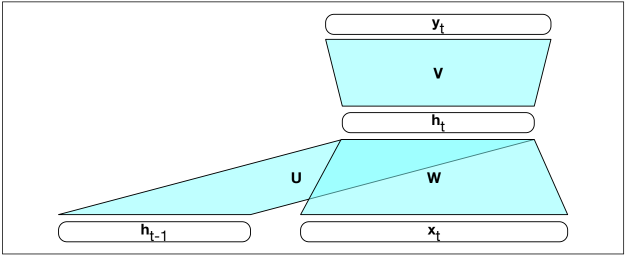

Figure 9.2 Simple recurrent neural network illustrated as a feedforward network.

- Basic structure:

$$\text{hidden:} \quad \boldsymbol{h}_t = \boldsymbol{g}(\boldsymbol{U}\boldsymbol{h}_{t-1} + \boldsymbol{W}\boldsymbol{x}_t)$$

$$\text{output:} \quad \boldsymbol{y}_{t+1} = \boldsymbol{f}(\boldsymbol{V}\boldsymbol{h}_t)$$

  - if $\boldsymbol{U} = 0$, this reduces to a feedforward NN
  - e.g. $\boldsymbol{f} = \text{softmax}$
- May have multiple layers, e.g. we may build ‘stacked RNNs’
- Can be ‘unrolled’ in the same way as a linear state-space model

Recursive NNs: unrolling

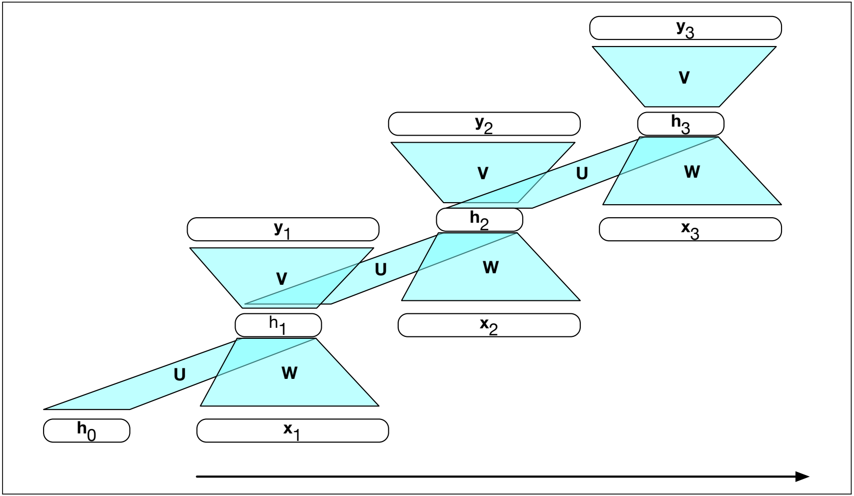

Figure 9.4 A simple recurrent neural network shown unrolled in time. Network layers are recalculated for each time step, while the weights **U**, **V** and **W** are shared across all time steps.

Language models and RNNs

- As with a feedforward neural LM, inputs are vector embeddings of words

$$\begin{aligned} \text{input:} \quad & \boldsymbol{e}_t = \boldsymbol{E}\boldsymbol{x}_t \\ \text{hidden:} \quad & \boldsymbol{h}_t = \boldsymbol{g}(\boldsymbol{U}\boldsymbol{h}_{t-1} + \boldsymbol{W}\boldsymbol{e}_t) \\ \text{output:} \quad & \boldsymbol{y}_{t+1} = \text{softmax}(\boldsymbol{V}\boldsymbol{h}_t) \end{aligned}$$

  - or $\boldsymbol{e} = [\boldsymbol{E}\boldsymbol{x}_{t-N+1}; \ldots; \boldsymbol{E}\boldsymbol{x}_{t-1}]$, etc.; each $\boldsymbol{x}_t$ selects a column from $\boldsymbol{E}$
  - by construction, $\boldsymbol{h}_t$ depends on *all* previous inputs $\boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \ldots$
- 'Forward inference' / prediction is straightforward: requires some initialisation for $\boldsymbol{h}_0$, e.g. $\boldsymbol{h}_0 = 0$
- Training / estimation proceeds as for a feedforward NN, using cross-entropy loss

$$L_{\mathrm{CE}}(\boldsymbol{W}, \boldsymbol{U}, \boldsymbol{V}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{U}, \boldsymbol{V})$$

- 'Weight tying':
  - $\boldsymbol{V}$ is a $|V| \times d_h$ matrix that 'scores' the relative conditional probability of the next word, given its context
  - rows provide embeddings for each word in the vocabulary
  - performs the same role as $\boldsymbol{E}$ (= dim of $\boldsymbol{E}^{\mathsf{T}}$); we may force $\boldsymbol{V} = \boldsymbol{E}^{\mathsf{T}}$ to reduce number of model parameters

Generative AI

- Usage as 'generative AI': use the model to recursively predict, until the end of a sentence is reached

1. Initialise by setting $\boldsymbol{x}_0$ to the symbol $<\mathrm{s}>$ (or some more task-appropriate context) for the beginning of a sentence; $\boldsymbol{e}_0$ the corresponding embedding
2. At the tth step, take $\boldsymbol{x}_{t+1}$ to indicate the element of the vocabulary for which the corresponding element of $\boldsymbol{y}_{t+1} = \mathrm{softmax}(\boldsymbol{V}\boldsymbol{h}_t)$ is highest
3. The next input, $\boldsymbol{e}_{t+1} = \boldsymbol{E}\boldsymbol{x}_{t+1}$ is the embedding corresponding to $\boldsymbol{x}_{t+1}$

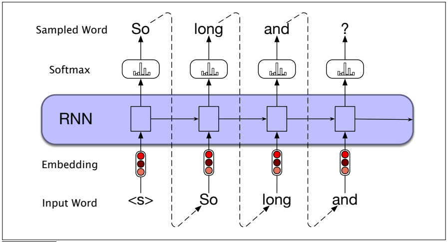

Figure 9.9 Autoregressive generation with an RNN-based neural language model.

- Continue until the end of sentence marker $</\mathrm{s}>$ is output.