Distributed Representation

Language Modeling

Motivation: suppose we want to build a speech recognition system.

We'd like to be able to infer a likely sentence $\mathbf{s}$ given the observed speech signal $\mathbf{a}$. The generative approach is to build two components:

- An observation model, represented as $p(\mathbf{a} \mid \mathbf{s})$, which tells us how likely the sentence $\mathbf{s}$ is to lead to the acoustic signal $\mathbf{a}$.
- A prior, represented as $p(\mathbf{s})$, which tells us how likely a given sentence $\mathbf{s}$ is. E.g., it should know that "recognize speech" is more likely that "wreck a nice beach."

Language Modeling

Motivation: suppose we want to build a speech recognition system.

We'd like to be able to infer a likely sentence $\mathbf{s}$ given the observed speech signal $\mathbf{a}$. The generative approach is to build two components:

- An observation model, represented as $p(\mathbf{a} \mid \mathbf{s})$, which tells us how likely the sentence $\mathbf{s}$ is to lead to the acoustic signal $\mathbf{a}$.
- A prior, represented as $p(\mathbf{s})$, which tells us how likely a given sentence $\mathbf{s}$ is. E.g., it should know that "recognize speech" is more likely that "wreck a nice beach."

Given these components, we can use Bayes' Rule to infer a posterior distribution over sentences given the speech signal:

$$p(\mathbf{s} \mid \mathbf{a}) = \frac{p(\mathbf{s})p(\mathbf{a} \mid \mathbf{s})}{\sum_{\mathbf{s}'} p(\mathbf{s}')p(\mathbf{a} \mid \mathbf{s}')}.$$

Language Modeling

In this lecture, we focus on learning a good distribution $p(\mathbf{s})$ of sentences. This problem is known as language modeling.

Assume we have a corpus of sentences $\mathbf{s}^{(1)}, \ldots, \mathbf{s}^{(N)}$. The maximum likelihood criterion says we want our model to maximize the probability our model assigns to the observed sentences. We assume the sentences are independent, so that their probabilities multiply.

$$\max \prod_{i=1}^{N} p(\mathbf{s}^{(i)}).$$

Language Modeling

In maximum likelihood training, we want to maximize $\prod_{i=1}^{N} p(\mathbf{s}^{(i)})$.

The probability of generating the whole training corpus is vanishingly small — like monkeys typing all of Shakespeare.

- The log probability is something we can work with more easily. It also conveniently decomposes as a sum:

$$\log \prod_{i=1}^{N} p(\mathbf{s}^{(i)}) = \sum_{i=1}^{N} \log p(\mathbf{s}^{(i)}).$$

- Let's use *negative* log probabilities, so that we're working with positive numbers.

Language Modeling

In maximum likelihood training, we want to maximize $\prod_{i=1}^{N} p(\mathbf{s}^{(i)})$.

The probability of generating the whole training corpus is vanishingly small — like monkeys typing all of Shakespeare.

- The log probability is something we can work with more easily. It also conveniently decomposes as a sum:

$$\log \prod_{i=1}^{N} p(\mathbf{s}^{(i)}) = \sum_{i=1}^{N} \log p(\mathbf{s}^{(i)}).$$

- Let's use *negative* log probabilities, so that we're working with positive numbers.

Language Modeling

- Probability of a sentence? What does that even mean?
  - A sentence is a sequence of words $w_1, w_2, \ldots, w_T$. Using the chain rule of conditional probability, we can decompose the probability as

$$p(\mathbf{s}) = p(w_1, \ldots, w_T) = p(w_1)p(w_2 \mid w_1) \cdots p(w_T \mid w_1, \ldots, w_{T-1}).$$

  - Therefore, the language modeling problem is equivalent to being able to predict the next word!
- We typically make a Markov assumption, i.e. that the distribution over the next word only depends on the preceding few words. I.e., if we use a context of length 3,

$$p(w_t \mid w_1, \ldots, w_{t-1}) = p(w_t \mid w_{t-3}, w_{t-2}, w_{t-1}).$$

  - Such a model is called memoryless.
  - Now it's basically a supervised prediction problem. We need to predict the conditional distribution of each word given the previous K.
  - When we decompose it into separate prediction problems this way, it's called an autoregressive model.

N-Gram Language Models

- One sort of Markov model we can learn uses a conditional probability table, i.e.

	cat	and	city	⋯
the fat	0.21	0.003	0.01	
four score	0.0001	0.55	0.0001	⋯
New York	0.002	0.0001	0.48	
⋮		⋮		

- Maybe the simplest way to estimate the probabilities is from the empirical distribution:

$$p(w_3 = \text{cat} \mid w_1 = \text{the}, w_2 = \text{fat}) = \frac{\text{count}(\text{the fat cat})}{\text{count}(\text{the fat})}$$

- This is the maximum likelihood solution; we'll see why later in the course.
- The phrases we're counting are called n-grams (where n is the length), so this is an n-gram language model.
  - Note: the above example is considered a 3-gram model, not a 2-gram

N-Gram Language Models

Shakespeare:

1 gram	–To him swallowed confess hear both. Which. Of save on trail for are ay device and rote life have –Hill he late speaks; or! a more to leg less first you enter
2 gram	–Why dost stand forth thy canopy, forsooth; he is this palpable hit the King Henry. Live king. Follow. –What means, sir. I confess she? then all sorts, he is trim, captain.
3 gram	–Fly, and will rid me these news of price. Therefore the sadness of parting, as they say, 'tis done. –This shall forbid it should be branded, if renown made it empty.
4 gram	–King Henry. What! I will go seek the traitor Gloucester. Exeunt some of the watch. A great banquet serv'd in; –It cannot be but so.

Jurafsky and Martin, *Speech and Language Processing*

N-Gram Language Models

Wall Street Journal:

1 gram	Months the my and issue of year foreign new exchange's september were recession exchange new endorsed a acquire to six executives
2 gram	Last December through the way to preserve the Hudson corporation N. B. E. C. Taylor would seem to complete the major central planners one point five percent of U. S. E. has already old M. X. corporation of living on information such as more frequently fishing to keep her
3 gram	They also point to ninety nine point six billion dollars from two hundred four oh six three percent of the rates of interest stores as Mexico and Brazil on market conditions

Jurafsky and Martin, *Speech and Language Processing*

N-Gram Language Models

- Problems with n-gram language models

N-Gram Language Models

- Problems with n-gram language models
  - The number of entries in the conditional probability table is exponential in the context length.
  - Data sparsity: most n-grams never appear in the corpus, even if they are possible.

N-Gram Language Models

- Problems with n-gram language models
  - The number of entries in the conditional probability table is exponential in the context length.
  - Data sparsity: most n-grams never appear in the corpus, even if they are possible.
- Ways to deal with data sparsity

N-Gram Language Models

- Problems with n-gram language models
  - The number of entries in the conditional probability table is exponential in the context length.
  - Data sparsity: most n-grams never appear in the corpus, even if they are possible.
- Ways to deal with data sparsity
  - Use a short context (but this means the model is less powerful)
  - Smooth the probabilities, e.g. by adding imaginary counts
  - Make predictions using an ensemble of n-gram models with different n

Distributed Representations

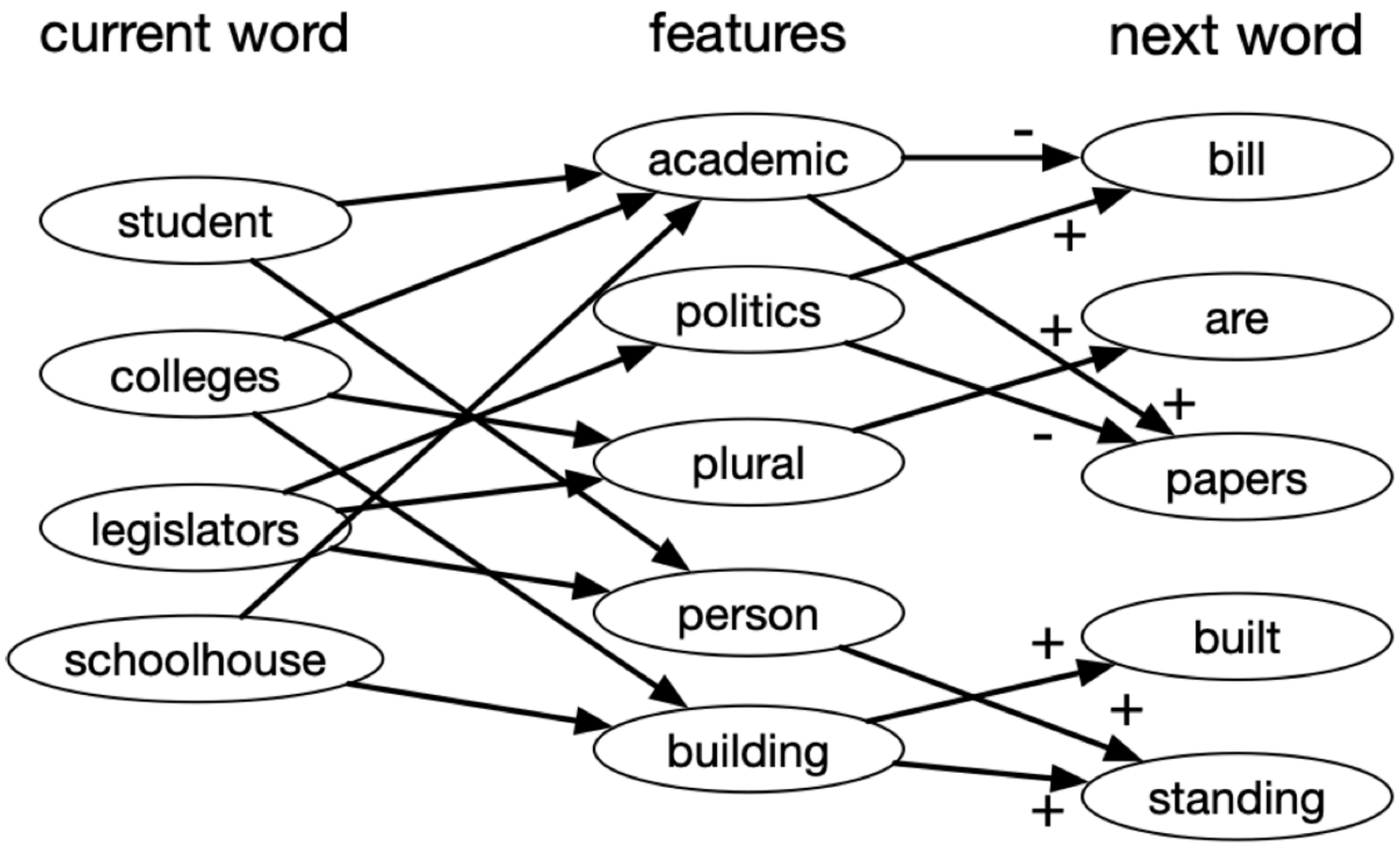
- Conditional probability tables are a kind of localist representation: all the information about a particular word is stored in one place, i.e. a column of the table.
- But different words are related, so we ought to be able to share information between them. For instance, consider this matrix of word attributes:

	academic	**politics**	**plural**	**person**	**building**
students	1	0	1	1	0
colleges	1	0	1	0	1
legislators	0	1	1	1	0
schoolhouse	1	0	0	0	1

- And this matrix of how each attribute influences the next word:

	bill	**is**	**are**	**papers**	**built**	**standing**
academic	−			+		
politics	+			−		
plural		−	+			
person						+
building					+	+

- Imagine these matrices are layers in an MLP. (One-hot representations of words, softmax over next word.)



- Here, the information about a given word is distributed throughout the representation. We call this a distributed representation.
- In general, when we train an MLP with backprop, the hidden units won't have intuitive meanings like in this cartoon. But this is a useful intuition pump for what MLPs can represent.

Distributed Representations

- We would like to be able to share information between related words. E.g., suppose we've seen the sentence

  The cat got squashed in the garden on Friday.

- This should help us predict the words in the sentence

  The dog got flattened in the yard on Monday.

- An n-gram model can't generalize this way, but a distributed representation might let us do so.

Neural Language Model

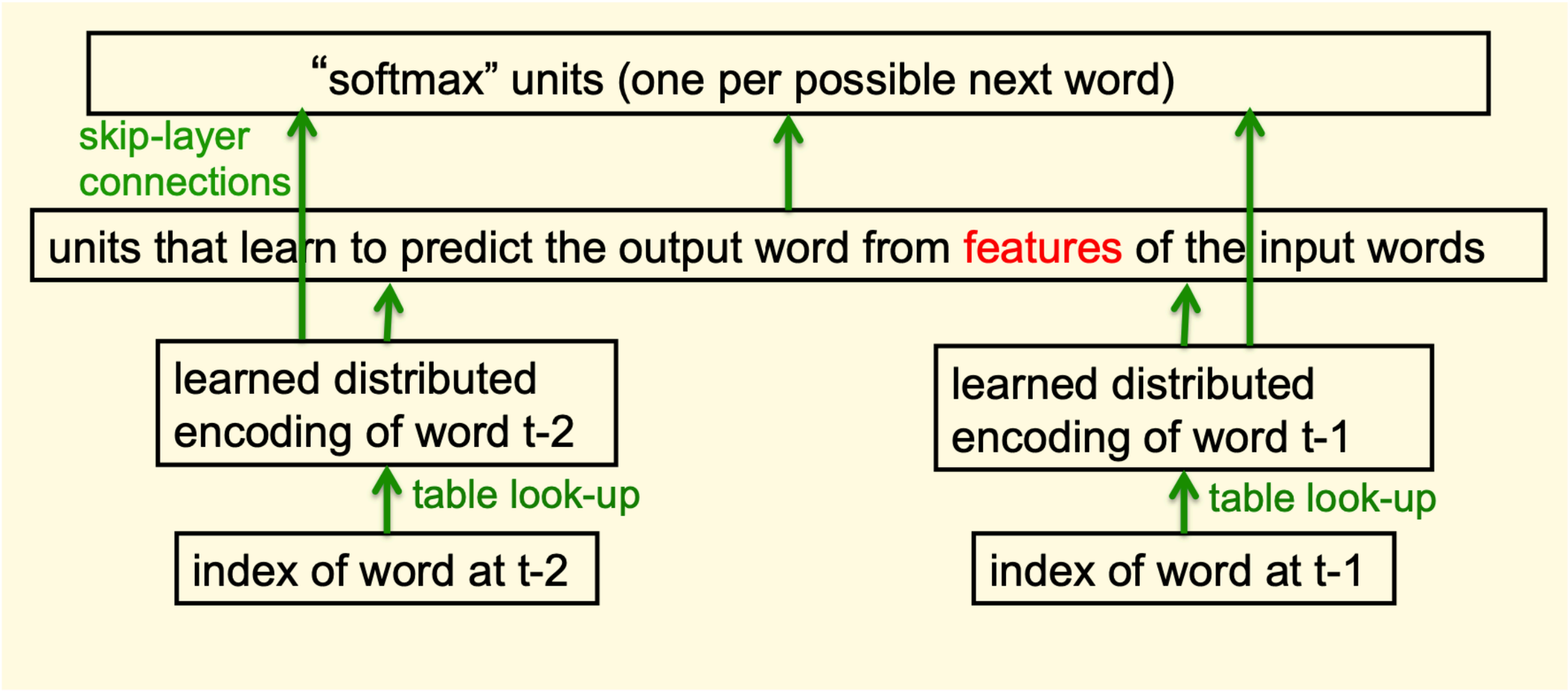
- Predicting the distribution of the next word given the previous K is just a multiway classification problem.
- **Inputs:** previous K words
- **Target:** next word
- **Loss:** cross-entropy. Recall that this is equivalent to maximum likelihood:

$$
\begin{aligned}
-\log p(\mathbf{s}) &= -\log \prod_{t=1}^{T} p(w_t \mid w_1, \ldots, w_{t-1}) \\
&= -\sum_{t=1}^{T} \log p(w_t \mid w_1, \ldots, w_{t-1}) \\
&= -\sum_{t=1}^{T} \sum_{v=1}^{V} t_{tv} \log y_{tv},
\end{aligned}
$$

where t_{iv} is the one-hot encoding for the ith word and y_{iv} is the predicted probability for the ith word being index v.

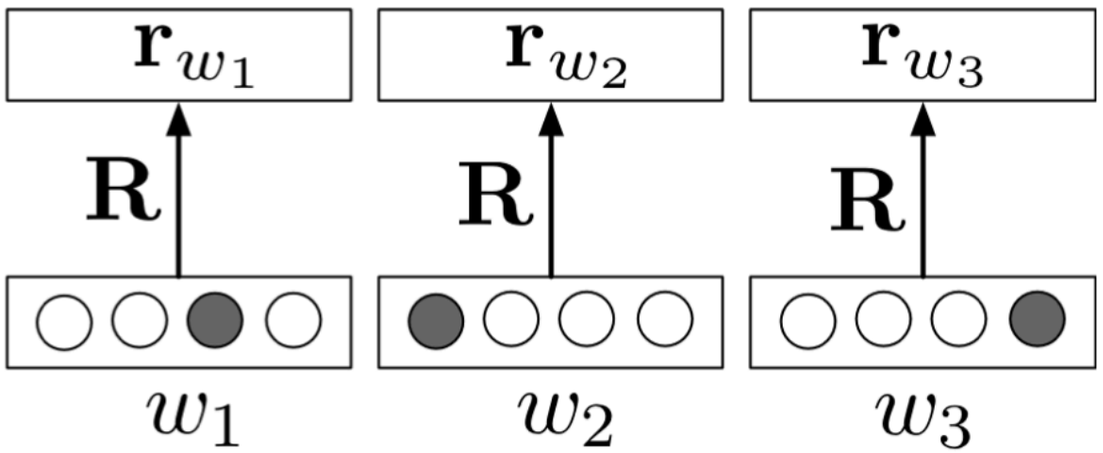
Neural Language Model

- Here is a classic neural probabilistic language model, or just neural language model:



Neural Language Model

- If we use a 1-of-K encoding for the words, the first layer can be thought of as a linear layer with tied weights.



- The weight matrix basically acts like a lookup table. Each column is the representation of a word, also called an embedding, feature vector, or encoding.
  - "Embedding" emphasizes that it's a location in a high-dimensonal space; words that are closer together are more semantically similar
  - "Feature vector" emphasizes that it's a vector that can be used for making predictions, just like other feature mappigns we've looked at (e.g. polynomials)

Neural Language Model

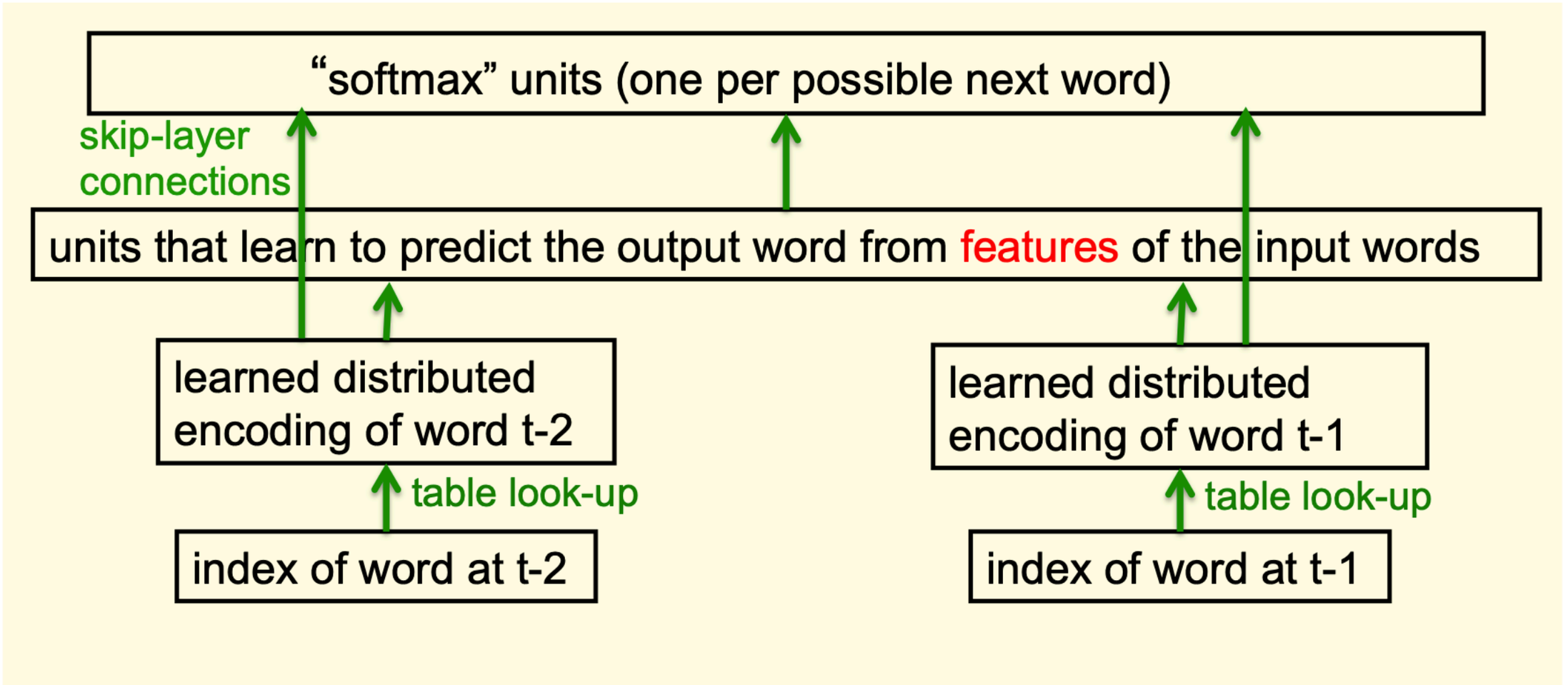
- We can measure the similarity or dissimilarity of two words using
  - the dot product $\mathbf{r}_1^\top \mathbf{r}_2$
  - Euclidean distance $\|\mathbf{r}_1 - \mathbf{r}_2\|$
- If the vectors have unit norm, the two are equivalent:

$$\begin{aligned}\|\mathbf{r}_1 - \mathbf{r}_2\|^2 &= (\mathbf{r}_1 - \mathbf{r}_2)^\top (\mathbf{r}_1 - \mathbf{r}_2) \\ &= \mathbf{r}_1^\top \mathbf{r}_1 - 2\mathbf{r}_1^\top \mathbf{r}_2 + \mathbf{r}_2^\top \mathbf{r}_2 \\ &= 2 - 2\mathbf{r}_1^\top \mathbf{r}_2\end{aligned}$$

- In this case, the dot product is called cosine similarity.

Neural Language Model

- This model is very compact: the number of parameters is *linear* in the context size, compared with exponential for n-gram models.



Neural Language Model

- What do these word embeddings look like?
- It's hard to visualize an n-dimensional space, but there are algorithms for mapping the embeddings to two dimensions.
- The following 2-D embeddings are done with an algorithm called tSNE which tries to make distnaces in the 2-D embedding match the original 30-D distances as closely as possible.
- Note: the visualizations are from a slightly different model.

Neural Language Model

Neural Language Model

Neural Language Model

Neural Language Model

- Thinking about high-dimensional embeddings
  - Most vectors are nearly orthogonal (i.e. dot product is close to 0)
  - Most points are far away from each other
  - "In a 30-dimensional grocery store, anchovies can be next to fish and next to pizza toppings." – Geoff Hinton
- The 2-D embeddings might be fairly misleading, since they can't preserve the distance relationships from a higher-dimensional embedding. (I.e., unrelated words might be close together in 2-D, but far apart in 30-D.)

Neural language model

When we train a neural language model, is that supervised or unsupervised learning? Does it have elements of both?

GloVe

- Fitting language models is really hard:
  - It's really important to make good predictions about relative probabilities of rare words.
  - Computing the predictive distribution requires a large softmax.
- Maybe this is overkill if all you want is word representations.
- Global Vector (GloVe) embeddings are a simpler and faster approach based on a matrix factorization similar to principal component analysis (PCA).
  - First fit the distributed word representations using GloVe, then plug them into a neural net that does some other task (e.g. language modeling, translation).

GloVe

- Distributional hypothesis: words with similar distributions have similar meanings ("judge a word by the company it keeps")
- Consider a co-occurrence matrix $\mathbf{X}$, which counts the number of times two words appear nearby (say, less than 5 positions apart)
- This is a $V \times V$ matrix, where V is the vocabulary size (very large)
- **Intuition pump:** suppose we fit a rank-K approximation $\mathbf{X} \approx \mathbf{R}\tilde{\mathbf{R}}^\top$, where $\mathbf{R}$ and $\tilde{\mathbf{R}}$ are $V \times K$ matrices.
  - Each row $\mathbf{r}_i$ of $\mathbf{R}$ is the K-dimensional representation of a word
  - Each entry is approximated as $x_{ij} \approx \mathbf{r}_i^\top \tilde{\mathbf{r}}_j$
  - Hence, more similar words are more likely to co-occur
  - Minimizing the squared Frobenius norm $\|\mathbf{X} - \mathbf{R}\tilde{\mathbf{R}}^\top\|_F^2 = \sum_{i,j}(x_{ij} - \mathbf{r}_i^\top \tilde{\mathbf{r}}_j)^2$ is basically PCA.

GloVe

- **Problem 1: X** is extremely large, so fitting the above factorization uisng least squares is infeasible.

GloVe

- **Problem 1: X** is extremely large, so fitting the above factorization uisng least squares is infeasible.
    - **Solution:** Reweight the entries so that only nonzero counts matter

GloVe

- **Problem 1: X** is extremely large, so fitting the above factorization uisng least squares is infeasible.
  - **Solution:** Reweight the entries so that only nonzero counts matter
- **Problem 2:** Word counts are a heavy-tailed distribution, so the most common words will dominate the cost function.

GloVe

- **Problem 1: X** is extremely large, so fitting the above factorization uisng least squares is infeasible.
  - **Solution:** Reweight the entries so that only nonzero counts matter
- **Problem 2:** Word counts are a heavy-tailed distribution, so the most common words will dominate the cost function.
  - **Solution:** Approximate $\log x_{ij}$ instead of x_{ij}.

GloVe

- **Problem 1: X** is extremely large, so fitting the above factorization uisng least squares is infeasible.
  - **Solution:** Reweight the entries so that only nonzero counts matter
- **Problem 2:** Word counts are a heavy-tailed distribution, so the most common words will dominate the cost function.
  - **Solution:** Approximate $\log x_{ij}$ instead of x_{ij}.
- Global Vector (GloVe) embedding cost function:

$$\mathcal{J}(\mathbf{R}) = \sum_{i,j} f(x_{ij})(\mathbf{r}_i^\top \tilde{\mathbf{r}}_j + b_i + \tilde{b}_j - \log x_{ij})^2$$

$$f(x_{ij}) = \begin{cases} \left(\frac{x_{ij}}{100}\right)^{3/4} & \text{if } x_{ij} < 100 \\ 1 & \text{if } x_{ij} \geq 100 \end{cases}$$

GloVe

- **Problem 1: X** is extremely large, so fitting the above factorization uisng least squares is infeasible.
  - **Solution:** Reweight the entries so that only nonzero counts matter
- **Problem 2:** Word counts are a heavy-tailed distribution, so the most common words will dominate the cost function.
  - **Solution:** Approximate $\log x_{ij}$ instead of x_{ij}.
- Global Vector (GloVe) embedding cost function:

$$\mathcal{J}(\mathbf{R}) = \sum_{i,j} f(x_{ij})(\mathbf{r}_i^\top \tilde{\mathbf{r}}_j + b_i + \tilde{b}_j - \log x_{ij})^2$$

$$f(x_{ij}) = \begin{cases} \left(\frac{x_{ij}}{100}\right)^{3/4} & \text{if } x_{ij} < 100 \\ 1 & \text{if } x_{ij} \geq 100 \end{cases}$$

- b_i and $\tilde{b}_j$ are bias parameters.
- We can avoid computing $\log 0$ since $f(0) = 0$.
- We only need to consider the nonzero entries of **X**. This gives a big computational savings since **X** is extremely sparse!

GloVe

- **Problem 1:** $\mathbf{X}$ is extremely large, so fitting the above factorization uisng least squares is infeasible.
  - **Solution:** Reweight the entries so that only nonzero counts matter
- **Problem 2:** Word counts are a heavy-tailed distribution, so the most common words will dominate the cost function.
  - **Solution:** Approximate $\log x_{ij}$ instead of x_{ij}.
- Global Vector (GloVe) embedding cost function:

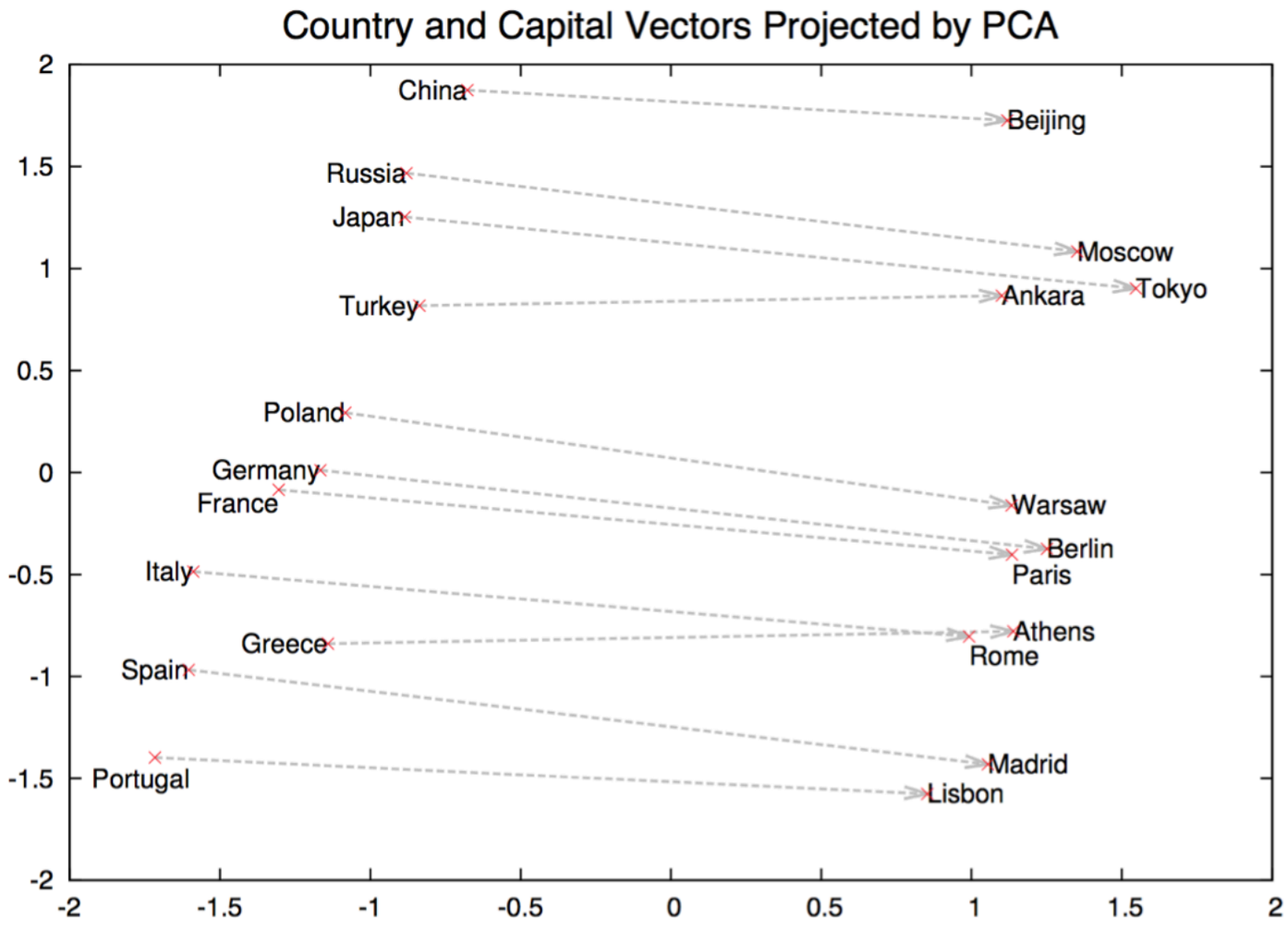
$$\mathcal{J}(\mathbf{R}) = \sum_{i,j} f(x_{ij})(\mathbf{r}_i^\top \tilde{\mathbf{r}}_j + b_i + \tilde{b}_j - \log x_{ij})^2$$

$$f(x_{ij}) = \begin{cases} \left(\frac{x_{ij}}{100}\right)^{3/4} & \text{if } x_{ij} < 100 \\ 1 & \text{if } x_{ij} \geq 100 \end{cases}$$

- b_i and $\tilde{b}_j$ are bias parameters.
- We can avoid computing $\log 0$ since $f(0) = 0$.
- We only need to consider the nonzero entries of $\mathbf{X}$. This gives a big computational savings since $\mathbf{X}$ is extremely sparse!

Word Analogies

- Here's a linear projection of word representations for cities and capitals into 2 dimensions.
- The mapping $\mathrm{city} \rightarrow \mathrm{capital}$ corresponds roughly to a single direction in the vector space:



- Note: this figure actually comes from skip-grams, a predecessor to GloVe.

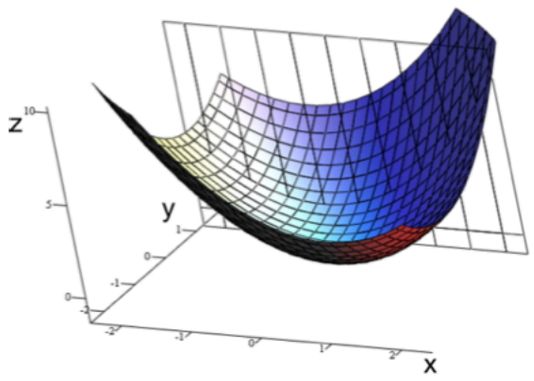
Word Analogies

- In other words,
  $\text{vector}(\text{Paris}) - \text{vector}(\text{France}) \approx \text{vector}(\text{London}) - \text{vector}(\text{England})$
- This means we can analogies by doing arithmetic on word vectors:
  - e.g. "Paris is to France as London is to _____"
  - Find the word whose vector is closest to
    $\text{vector}(\text{France}) - \text{vector}(\text{Paris}) + \text{vector}(\text{London})$
- Example analogies:

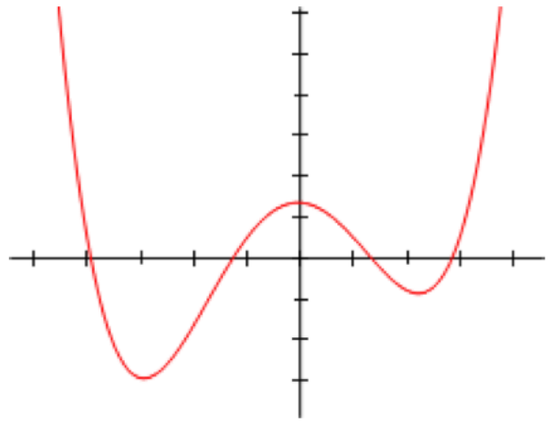
Relationship	Example 1	Example 2	Example 3
France - Paris	Italy: Rome	Japan: Tokyo	Florida: Tallahassee
big - bigger	small: larger	cold: colder	quick: quicker
Miami - Florida	Baltimore: Maryland	Dallas: Texas	Kona: Hawaii
Einstein - scientist	Messi: midfielder	Mozart: violinist	Picasso: painter
Sarkozy - France	Berlusconi: Italy	Merkel: Germany	Koizumi: Japan
copper - Cu	zinc: Zn	gold: Au	uranium: plutonium
Berlusconi - Silvio	Sarkozy: Nicolas	Putin: Medvedev	Obama: Barack
Microsoft - Windows	Google: Android	IBM: Linux	Apple: iPhone
Microsoft - Ballmer	Google: Yahoo	IBM: McNealy	Apple: Jobs
Japan - sushi	Germany: bratwurst	France: tapas	USA: pizza

Optimization

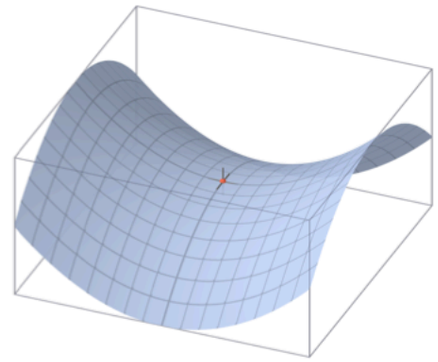
Features of the Optimization Landscape



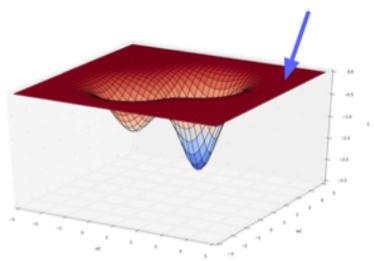
convex functions



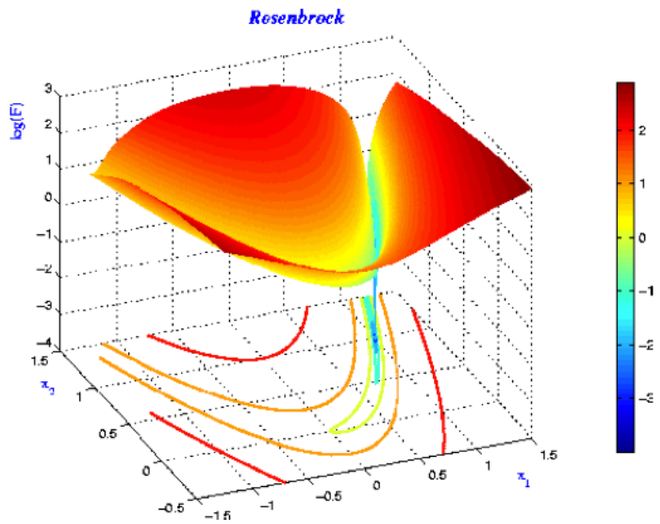
local minima



saddle points



plateaux



narrow ravines

Review: Hessian Matrix

- The Hessian matrix, denoted $\mathbf{H}$, or $\nabla^2 \mathcal{J}$ is the matrix of second derivatives:

$$\mathbf{H} = \nabla^2 \mathcal{J} = \begin{pmatrix} \frac{\partial^2 \mathcal{J}}{\partial \theta_1^2} & \frac{\partial^2 \mathcal{J}}{\partial \theta_1 \partial \theta_2} & \cdots & \frac{\partial^2 \mathcal{J}}{\partial \theta_1 \partial \theta_D} \\ \frac{\partial^2 \mathcal{J}}{\partial \theta_2 \partial \theta_1} & \frac{\partial^2 \mathcal{J}}{\partial \theta_2^2} & \cdots & \frac{\partial^2 \mathcal{J}}{\partial \theta_2 \partial \theta_D} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 \mathcal{J}}{\partial \theta_D \partial \theta_1} & \frac{\partial^2 \mathcal{J}}{\partial \theta_D \partial \theta_2} & \cdots & \frac{\partial^2 \mathcal{J}}{\partial \theta_D^2} \end{pmatrix}$$

- It's a symmetric matrix because $\frac{\partial^2 \mathcal{J}}{\partial \theta_i \partial \theta_j} = \frac{\partial^2 \mathcal{J}}{\partial \theta_j \partial \theta_i}$.

Review: Hessian Matrix

- Locally, a function can be approximated by its second-order Taylor approximation around a point $\boldsymbol{\theta}_0$:

$$\mathcal{J}(\boldsymbol{\theta}) \approx \mathcal{J}(\boldsymbol{\theta}_0) + \nabla \mathcal{J}(\boldsymbol{\theta}_0)^\top (\boldsymbol{\theta} - \boldsymbol{\theta}_0) + \tfrac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \mathbf{H}(\boldsymbol{\theta}_0)(\boldsymbol{\theta} - \boldsymbol{\theta}_0).$$

- A critical point is a point where the gradient is zero. In that case,

$$\mathcal{J}(\boldsymbol{\theta}) \approx \mathcal{J}(\boldsymbol{\theta}_0) + \tfrac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \mathbf{H}(\boldsymbol{\theta}_0)(\boldsymbol{\theta} - \boldsymbol{\theta}_0).$$

Review: Hessian Matrix

- A lot of important features of the optimization landscape can be characterized by the eigenvalues of the Hessian $\mathbf{H}$.
- Recall that a symmetric matrix (such as $\mathbf{H}$) has only real eigenvalues, and there is an orthogonal basis of eigenvectors.
- This can be expressed in terms of the spectral decomposition:

$$\mathbf{H} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top,$$

where $\mathbf{Q}$ is an orthogonal matrix (whose columns are the eigenvectors) and $\mathbf{\Lambda}$ is a diagonal matrix (whose diagonal entries are the eigenvalues).

Review: Hessian Matrix

- We often refer to $\mathbf{H}$ as the curvature of a function.
- Suppose you move along a line defined by $\boldsymbol{\theta} + t\mathbf{v}$ for some vector $\mathbf{v}$.
- Second-order Taylor approximation:

$$\mathcal{J}(\boldsymbol{\theta} + t\mathbf{v}) \approx \mathcal{J}(\boldsymbol{\theta}) + t\nabla\mathcal{J}(\boldsymbol{\theta})^\top\mathbf{v} + \frac{t^2}{2}\mathbf{v}^\top\mathbf{H}(\boldsymbol{\theta})\mathbf{v}$$

- Hence, in a direction where $\mathbf{v}^\top\mathbf{H}\mathbf{v} > 0$, the cost function curves upwards, i.e. has positive curvature. Where $\mathbf{v}^\top\mathbf{H}\mathbf{v} < 0$, it has negative curvature.

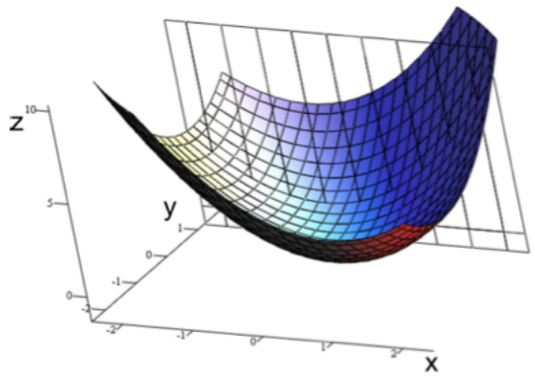
Review: Hessian Matrix

- A matrix $\mathbf{A}$ is positive definite if $\mathbf{v}^\top \mathbf{A}\mathbf{v} > 0$ for all $\mathbf{v} \neq 0$. (I.e., it curves upwards in all directions.)
  - It is positive semidefinite (PSD) if $\mathbf{v}^\top \mathbf{A}\mathbf{v} \geq 0$ for all $\mathbf{v} \neq 0$.
- Equivalently: a matrix is positive definite iff all its eigenvalues are positive. It is PSD iff all its eigenvalues are nonnegative. (Exercise: show this using the Spectral Decomposition.)
- For any critical point $\boldsymbol{\theta}_*$, if $\mathbf{H}(\boldsymbol{\theta}_*)$ exists and is positive definite, then $\boldsymbol{\theta}_*$ is a local minimum (since all directions curve upwards).

Convex Functions

- Recall: a set $\mathcal{S}$ is convex if for any $\mathbf{x}_0, \mathbf{x}_1 \in \mathcal{S}$,

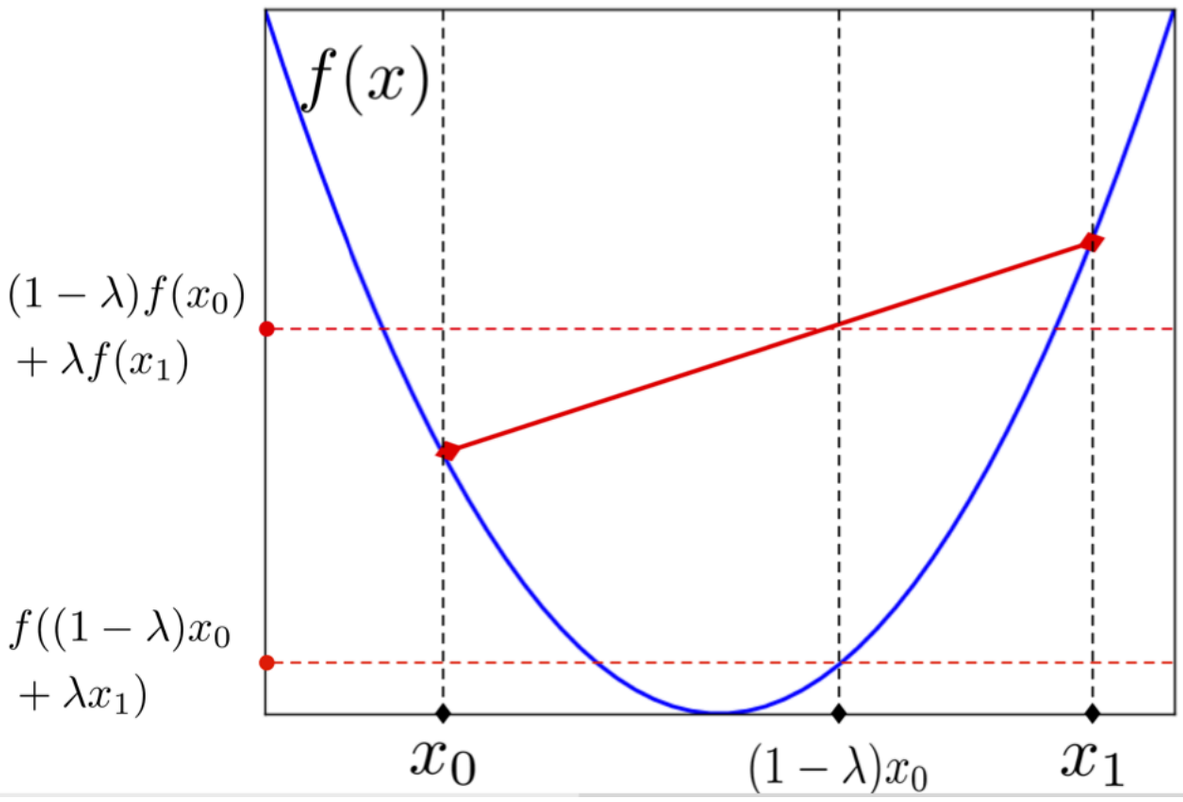
$$(1-\lambda)\mathbf{x}_0 + \lambda\mathbf{x}_1 \in \mathcal{S} \quad \text{for } 0 \leq \lambda \leq 1.$$



- A function f is convex if for any $\mathbf{x}_0, \mathbf{x}_1$,

$$f((1-\lambda)\mathbf{x}_0 + \lambda\mathbf{x}_1) \leq (1-\lambda)f(\mathbf{x}_0) + \lambda f(\mathbf{x}_1)$$

- Equivalently, the set of points lying above the graph of f is convex.
- Intuitively: the function is bowl-shaped.

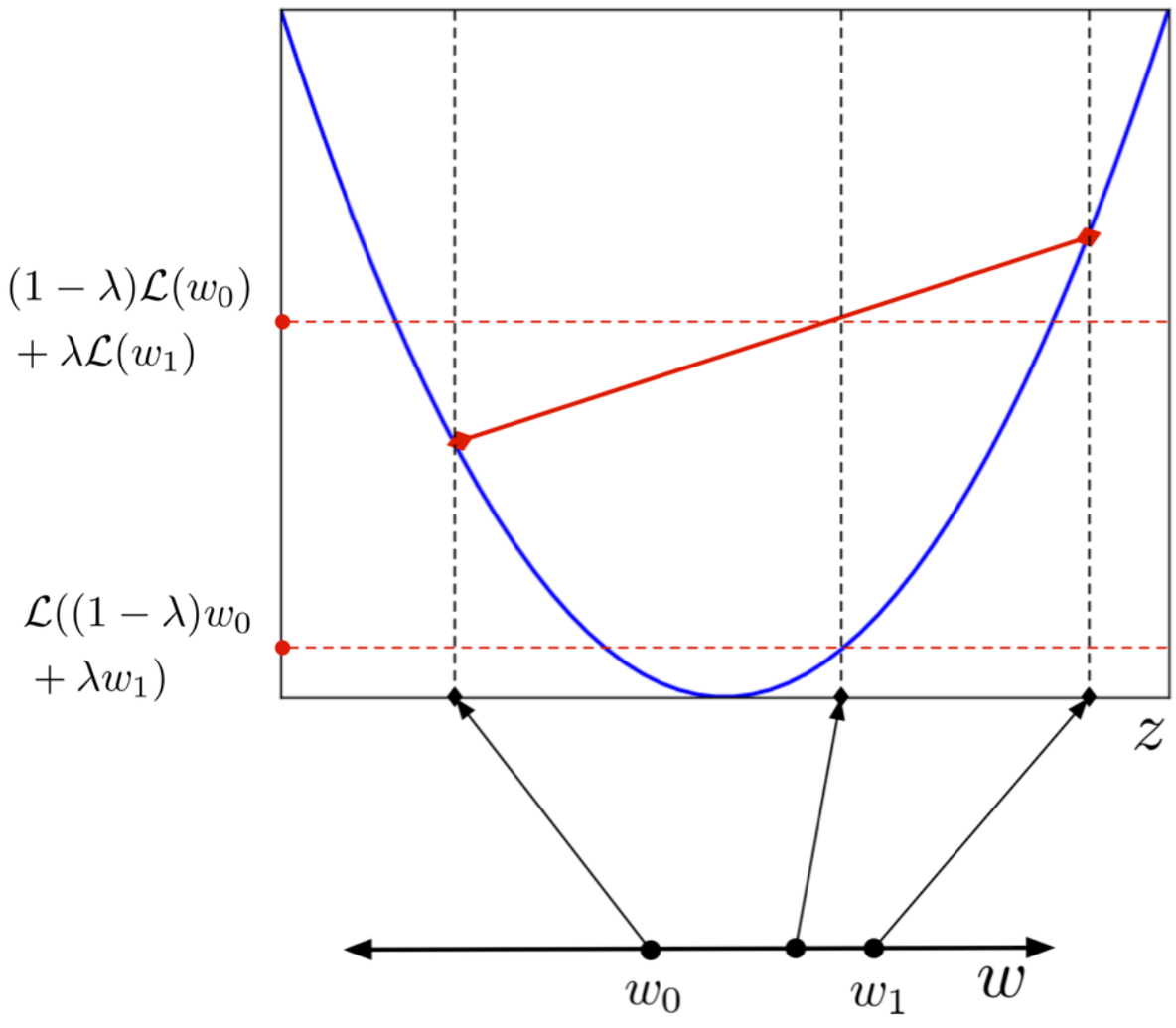


Convex Functions

- If $\mathcal{J}$ is smooth (more precisely, twice differentiable), there's an equivalent characterization in terms of $\mathbf{H}$:
  - A smooth function is convex iff its Hessian is positive semidefinite everywhere.
  - **Special case:** a univariate function is convex iff its second derivative is nonnegative everywhere.
- **Exercise:** show that squared error, logistic-cross-entropy, and softmax-cross-entropy losses are convex (as a function of the network outputs) by taking second derivatives.

Convex Functions

- For a linear model, $z = \mathbf{w}^\top \mathbf{x} + b$ is a linear function of $\mathbf{w}$ and b. If the loss function is convex as a function of z, then it is convex as a function of $\mathbf{w}$ and b.
- Hence, linear regression, logistic regression, and softmax regression are convex.

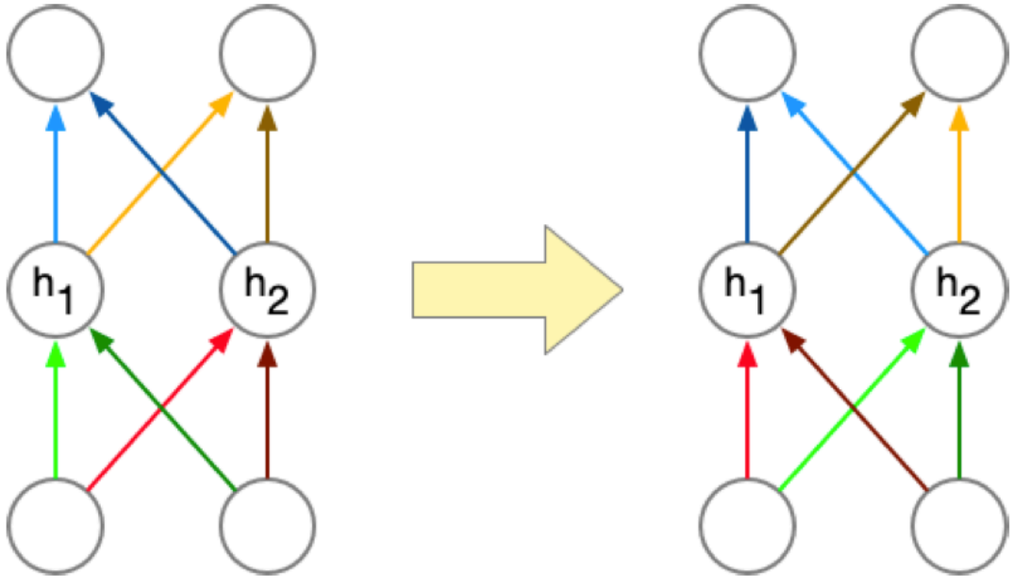


Local Minima

- If a function is convex, it has no spurious local minima, i.e. any local minimum is also a global minimum.
- This is very convenient for optimization since if we keep going downhill, we'll eventually reach a global minimum.

Local Minima

- If a function is convex, it has no spurious local minima, i.e. any local minimum is also a global minimum.
- This is very convenient for optimization since if we keep going downhill, we'll eventually reach a global minimum.
- Unfortunately, training a network with hidden units cannot be convex because of permutation symmetries.
  - I.e., we can re-order the hidden units in a way that preserves the function computed by the network.



Local Minima

- By definition, if a function $\mathcal{J}$ is convex, then for any set of points $\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_N$ in its domain,

$$\mathcal{J}(\lambda_1 \boldsymbol{\theta}_1 + \cdots + \lambda_N \boldsymbol{\theta}_N) \leq \lambda_1 \mathcal{J}(\boldsymbol{\theta}_1) + \cdots + \lambda_N \mathcal{J}(\boldsymbol{\theta}_N) \quad \text{for } \lambda_i \geq 0, \sum_i \lambda_i = 1.$$

- Because of permutation symmetry, there are $K!$ permutations of the hidden units in a given layer which all compute the same function.
- Suppose we average the parameters for all $K!$ permutations. Then we get a degenerate network where all the hidden units are identical.
- If the cost function were convex, this solution would have to be better than the original one, which is ridiculous!
- Hence, training multilayer neural nets is non-convex.

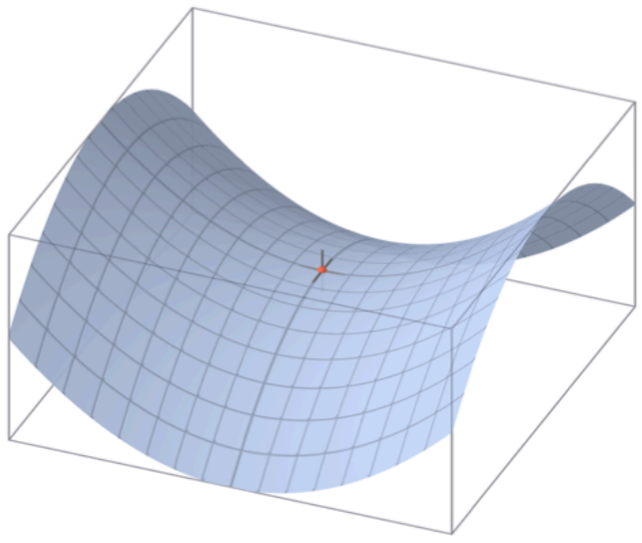
Local Minima (optional, informal)

- Generally, local minima aren't something we worry much about when we train most neural nets.
  - They're normally only a problem if there are local minima "in function space". E.g., CycleGANs (covered later in this course) have a bad local minimum where they learn the wrong color mapping between domains.

Local Minima (optional, informal)

- Generally, local minima aren't something we worry much about when we train most neural nets.
  - They're normally only a problem if there are local minima "in function space". E.g., CycleGANs (covered later in this course) have a bad local minimum where they learn the wrong color mapping between domains.
- It's possible to construct arbitrarily bad local minima even for ordinary classification MLPs. It's poorly understood why these don't arise in practice.

Saddle points

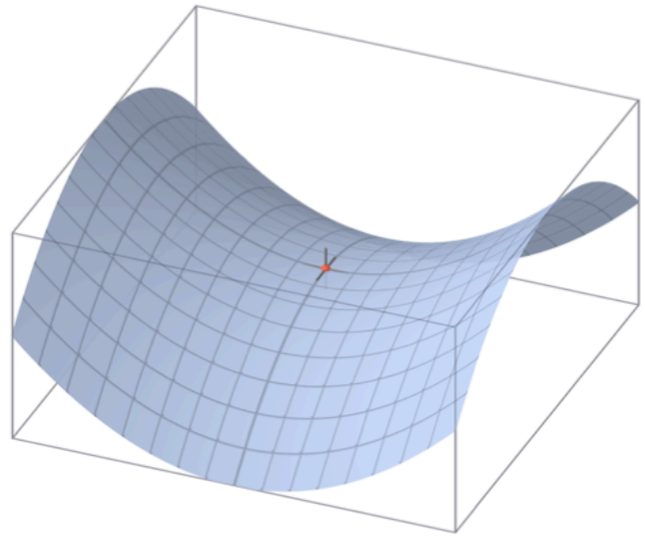


A saddle point is a point where:

- $\nabla \mathcal{J}(\boldsymbol{\theta}) = \mathbf{0}$
- $\mathbf{H}(\boldsymbol{\theta})$ has some positive and some negative eigenvalues, i.e. some directions with positive curvature and some with negative curvature.

When would saddle points be a problem?

Saddle points



A saddle point is a point where:

- $\nabla \mathcal{J}(\boldsymbol{\theta}) = \mathbf{0}$
- $\mathbf{H}(\boldsymbol{\theta})$ has some positive and some negative eigenvalues, i.e. some directions with positive curvature and some with negative curvature.

When would saddle points be a problem?

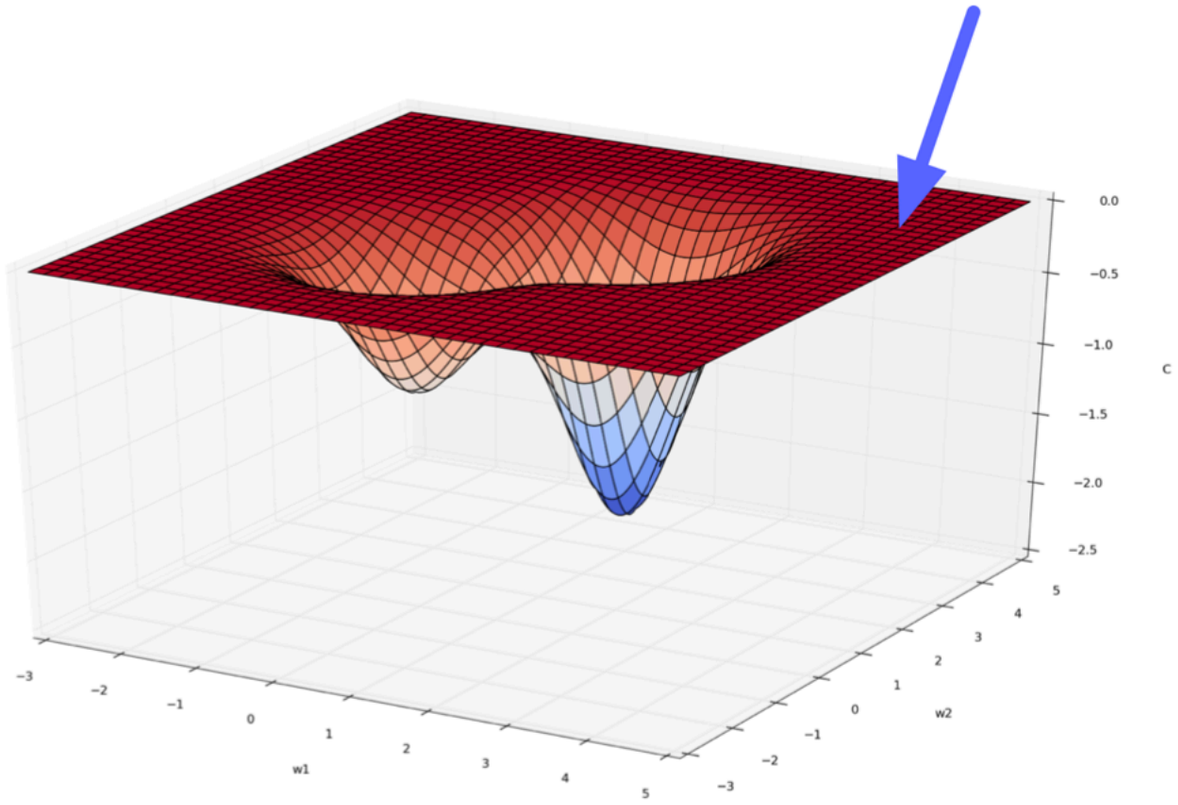
- If we're exactly on the saddle point, then we're stuck.
- If we're slightly to the side, then we can get unstuck.

Saddle points

- Suppose you have two hidden units with identical incoming and outgoing weights.
- After a gradient descent update, they will still have identical weights. By induction, they'll always remain identical.
- But if you perturbed them slightly, they can start to move apart.
- Important special case: don't initialize all your weights to zero!
  - Instead, break the symmetry by using small random values.

Plateaux

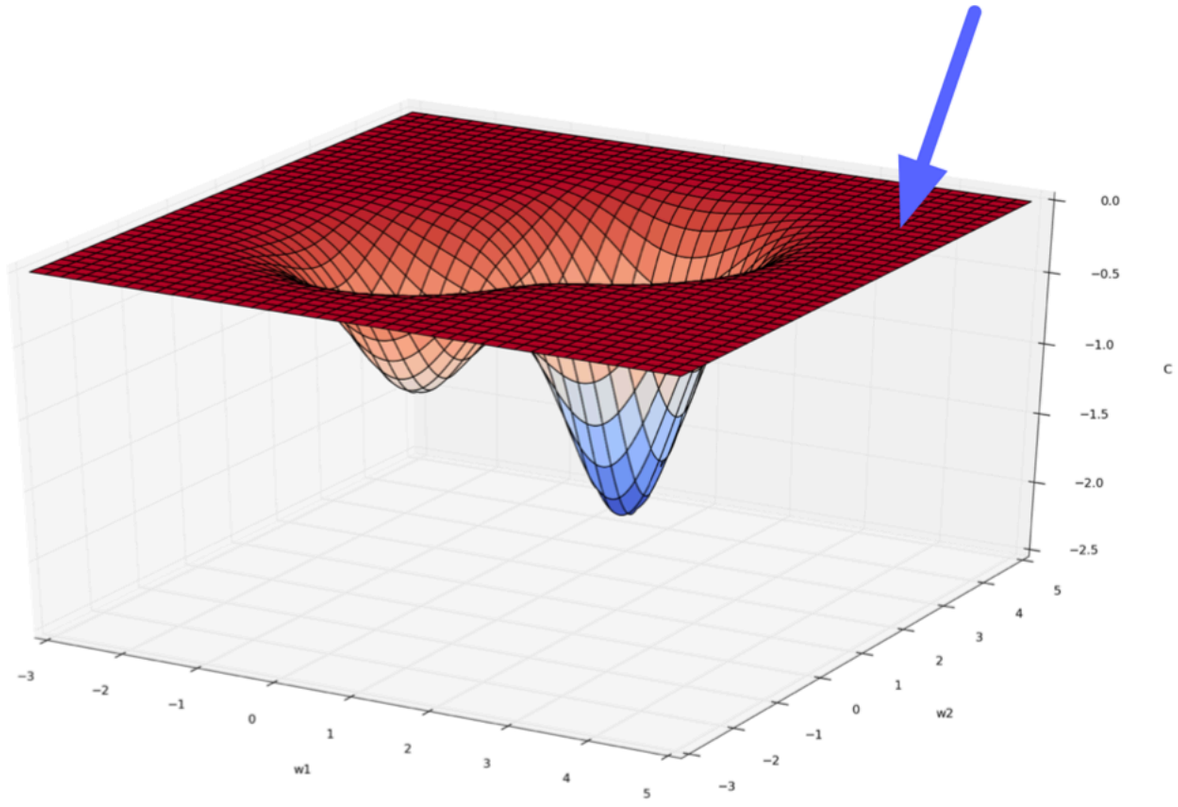
A flat region is called a plateau. (Plural: plateaux)



Can you think of examples?

Plateaux

A flat region is called a plateau. (Plural: plateaux)



Can you think of examples?

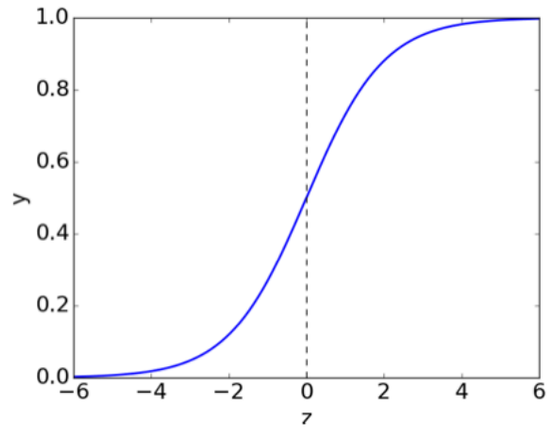
- 0–1 loss
- hard threshold activations
- logistic activations & least squares

Plateaux

- An important example of a plateau is a saturated unit. This is when it is in the flat region of its activation function. Recall the backprop equation for the weight derivative:

$$\overline{z_i} = \overline{h_i}\,\phi'(z)$$

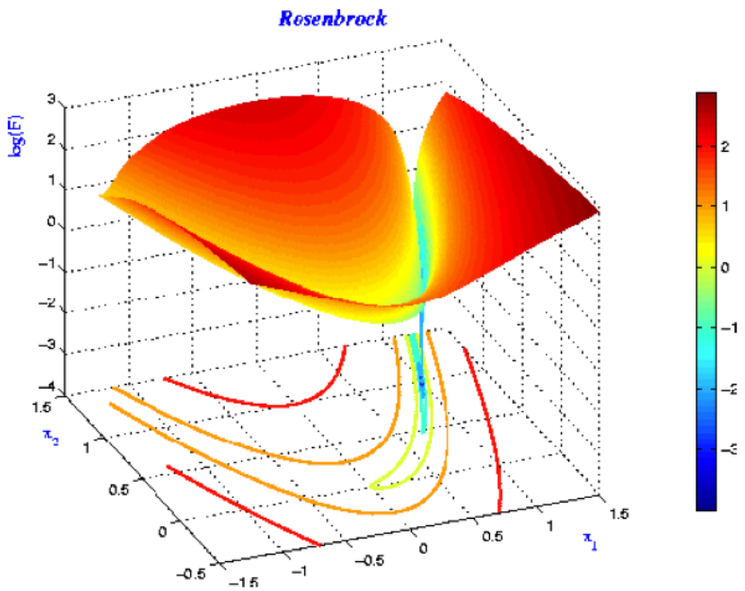
$$\overline{w_{ij}} = \overline{z_i}\,x_j$$

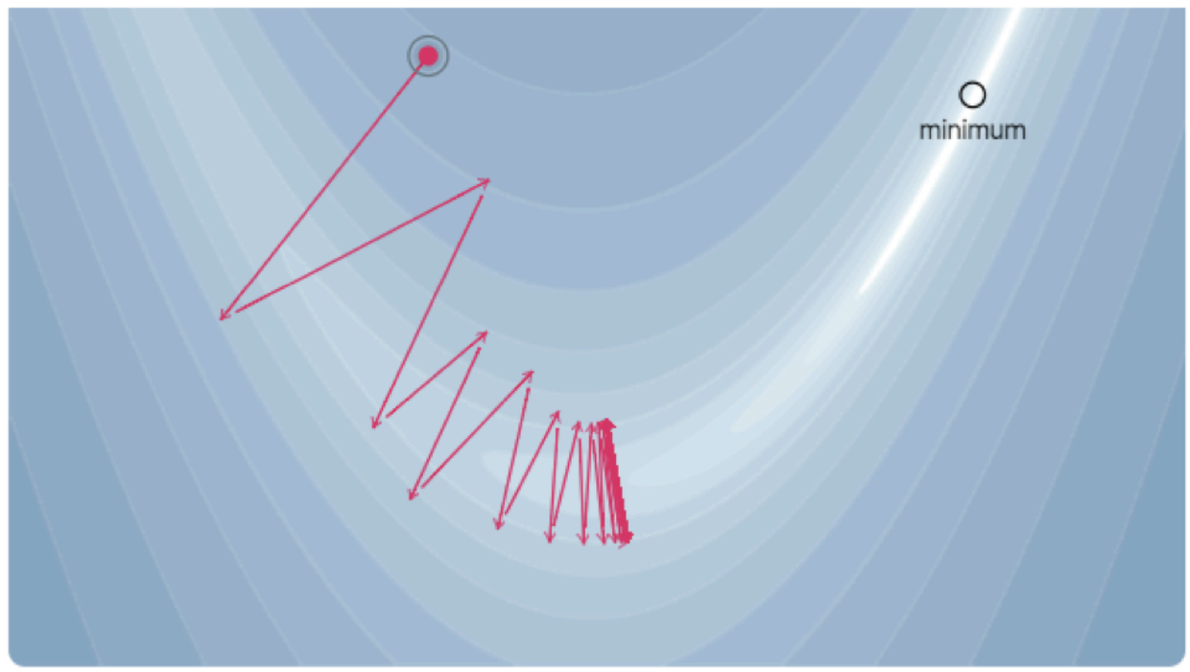


- If $\phi'(z_i)$ is always close to zero, then the weights will get stuck.
- If there is a ReLU unit whose input z_i is always negative, the weight derivatives will be *exactly* 0. We call this a dead unit.

Ill-conditioned curvature

Long, narrow ravines:



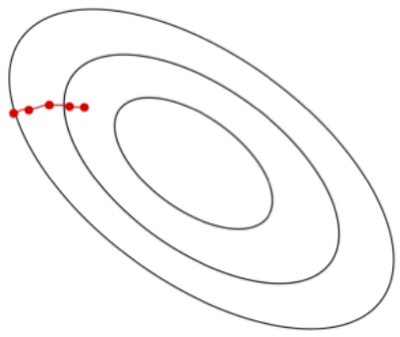


- Suppose **H** has some large positive eigenvalues (i.e. high-curvature directions) and some eigenvalues close to 0 (i.e. low-curvature directions).
- Gradient descent bounces back and forth in high curvature directions and makes slow progress in low curvature directions.
    - To interpret this visually: the gradient is perpendicular to the contours.
- This is known as ill-conditioned curvature. It's very common in neural net training.

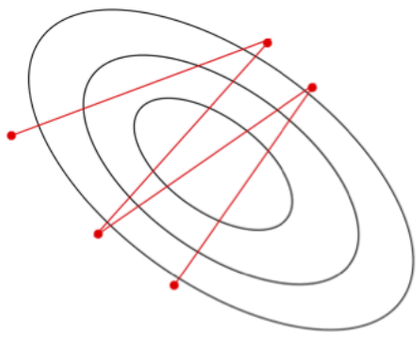
Optimization:
learning steps, mini-batches

Learning Rate

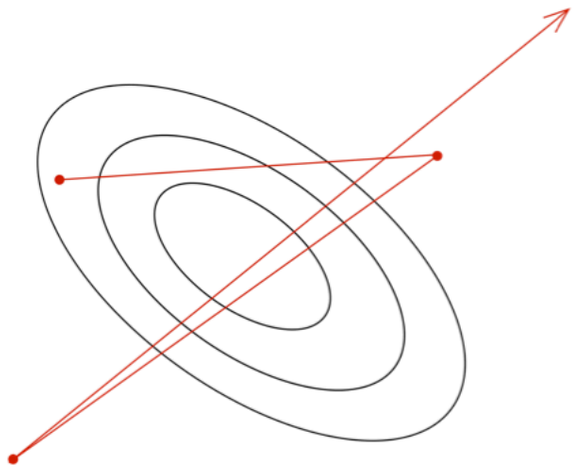
- The learning rate α is a hyperparameter we need to tune. Here are the things that can go wrong in batch mode:



α too small:
slow progress



α too large:
oscillations

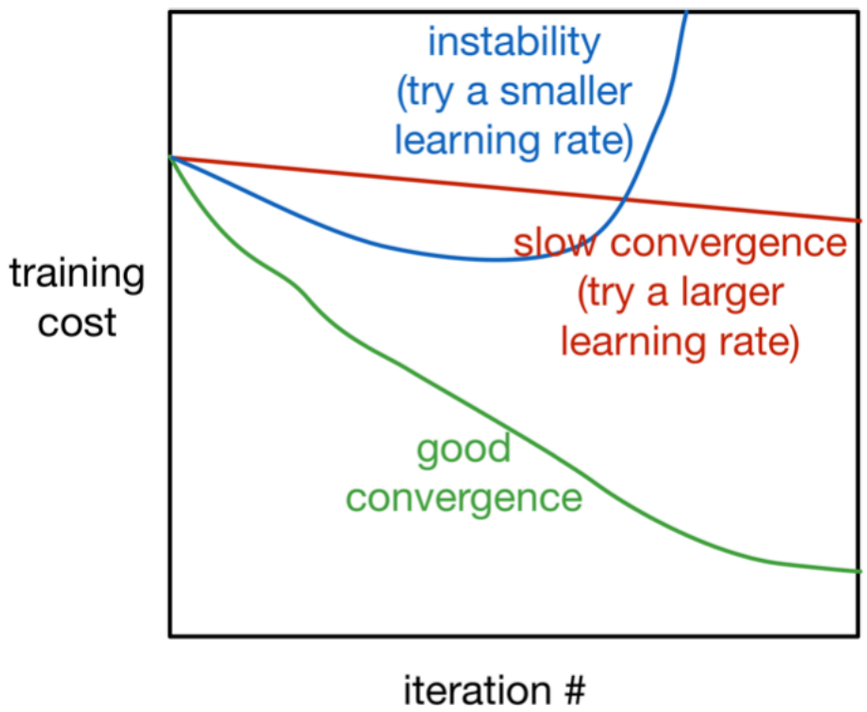


α much too large:
instability

- Good values are typically between 0.001 and 0.1. You should do a grid search if you want good performance (i.e. try $0.1, 0.03, 0.01, \ldots$).

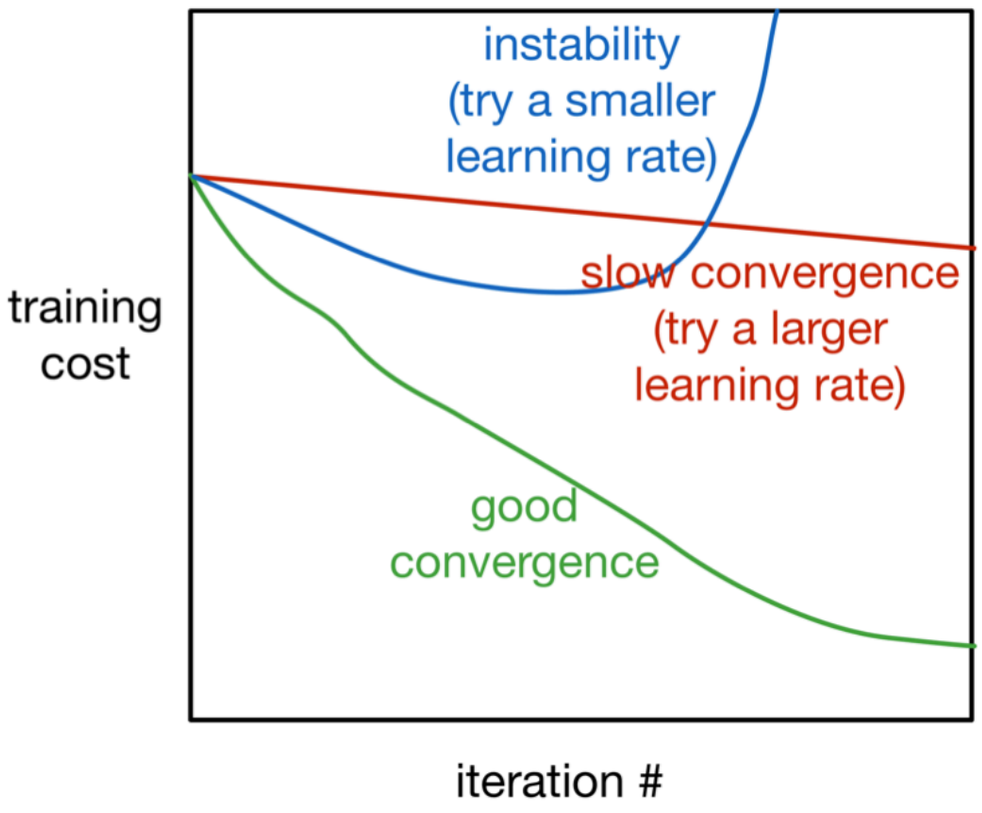
Training Curves

- To diagnose optimization problems, it's useful to look at training curves: plot the training cost as a function of iteration.
- **Gotcha:** use a fixed subset of the training data to monitor the training error. Evaluating on a different batch (e.g. the current one) in each iteration adds a *lot* of noise to the curve!
- **Gotcha:** it's very hard to tell from the training curves whether an optimizer has converged. They can reveal major problems, but they can't guarantee convergence.



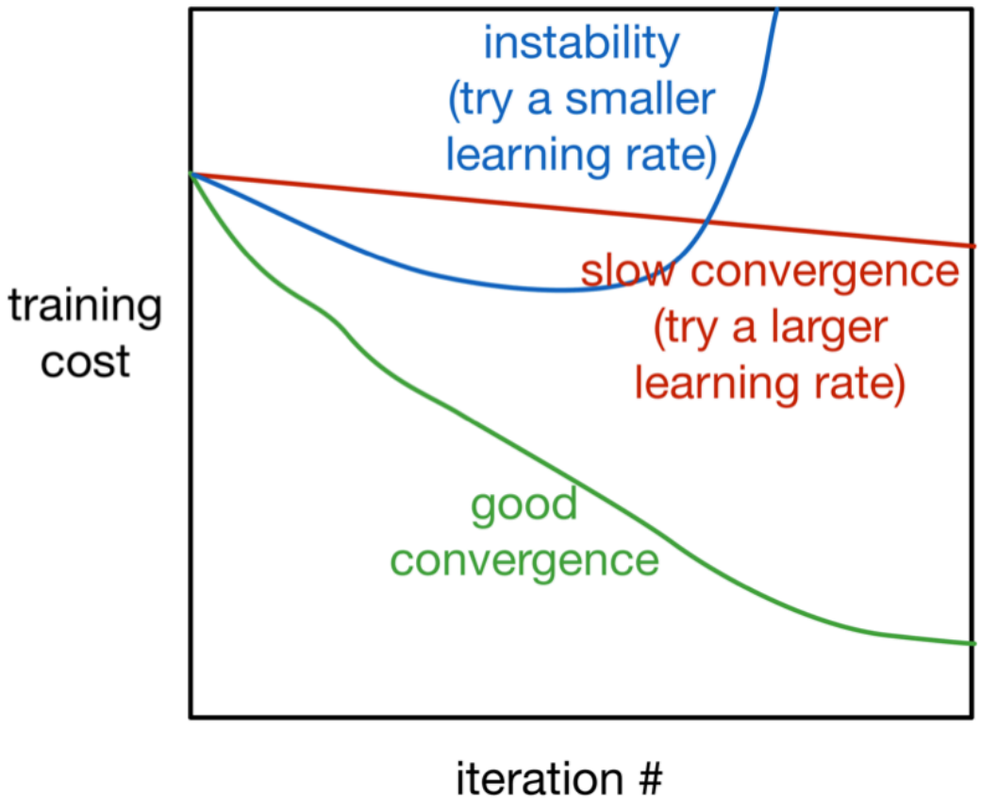
Training Curves

- To diagnose optimization problems, it's useful to look at training curves: plot the training cost as a function of iteration.
- **Gotcha:** use a fixed subset of the training data to monitor the training error. Evaluating on a different batch (e.g. the current one) in each iteration adds a *lot* of noise to the curve!
- **Gotcha:** it's very hard to tell from the training curves whether an optimizer has converged. They can reveal major problems, but they can't guarantee convergence.



Training Curves

- To diagnose optimization problems, it's useful to look at training curves: plot the training cost as a function of iteration.
- **Gotcha:** use a fixed subset of the training data to monitor the training error. Evaluating on a different batch (e.g. the current one) in each iteration adds a *lot* of noise to the curve!
- **Gotcha:** it's very hard to tell from the training curves whether an optimizer has converged. They can reveal major problems, but they can't guarantee convergence.



Stochastic Gradient Descent

- So far, the cost function $\mathcal{J}$ has been the average loss over the training examples:

$$\mathcal{J}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{J}^{(i)}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(y(\mathbf{x}^{(i)}, \boldsymbol{\theta}), t^{(i)}).$$

- By linearity,

$$\nabla\mathcal{J}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N}\nabla\mathcal{J}^{(i)}(\boldsymbol{\theta}).$$

- Computing the gradient requires summing over *all* of the training examples. This is known as batch training.
- Batch training is impractical if you have a large dataset (e.g. millions of training examples)!

Stochastic Gradient Descent

- Stochastic gradient descent (SGD): update the parameters based on the gradient for a single training example:

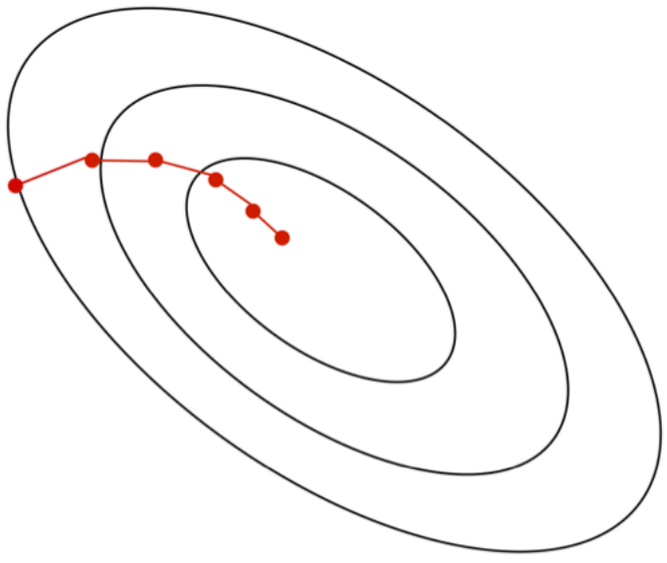
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla \mathcal{J}^{(i)}(\boldsymbol{\theta})$$

- SGD can make significant progress before it has even looked at all the data!
- Mathematical justification: if you sample a training example at random, the stochastic gradient is an unbiased estimate of the batch gradient:

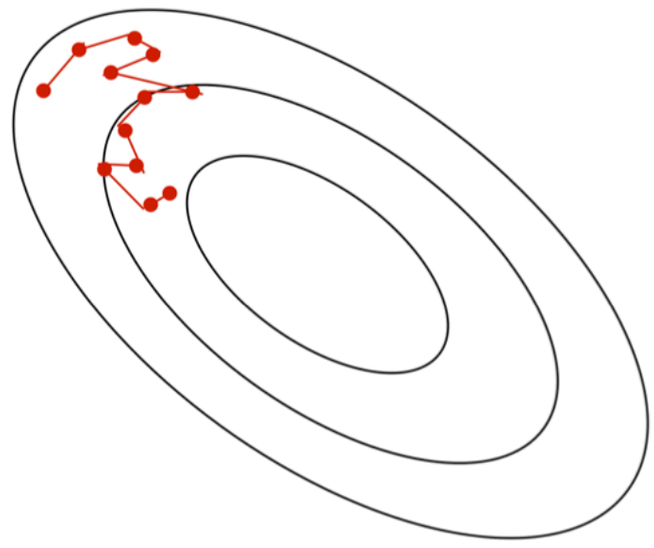
$$\mathbb{E}_i \left[\nabla \mathcal{J}^{(i)}(\boldsymbol{\theta}) \right] = \frac{1}{N} \sum_{i=1}^{N} \nabla \mathcal{J}^{(i)}(\boldsymbol{\theta}) = \nabla \mathcal{J}(\boldsymbol{\theta}).$$

Stochastic Gradient Descent

- Batch gradient descent moves directly downhill. SGD takes steps in a noisy direction, but moves downhill on average.



batch gradient descent



stochastic gradient descent

Stochastic Gradient Descent

- **Problem:** if we only look at one training example at a time, we can't exploit efficient vectorized operations.
- **Compromise approach:** compute the gradients on a medium-sized set of training examples, called a mini-batch.
- Each entire pass over the dataset is called an epoch.
- Stochastic gradients computed on larger mini-batches have smaller variance:

$$\mathrm{Var}\left[\frac{1}{S}\sum_{i=1}^{S}\frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j}\right] = \frac{1}{S^2}\mathrm{Var}\left[\sum_{i=1}^{S}\frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j}\right] = \frac{1}{S}\mathrm{Var}\left[\frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j}\right]$$

- The mini-batch size S is a hyperparameter. Typical values are 10 or 100.

Stochastic Gradient Descent

- **Problem:** if we only look at one training example at a time, we can't exploit efficient vectorized operations.
- **Compromise approach:** compute the gradients on a medium-sized set of training examples, called a mini-batch.
- Each entire pass over the dataset is called an epoch.
- Stochastic gradients computed on larger mini-batches have smaller variance:

$$\mathrm{Var}\left[\frac{1}{S}\sum_{i=1}^{S}\frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j}\right] = \frac{1}{S^2}\mathrm{Var}\left[\sum_{i=1}^{S}\frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j}\right] = \frac{1}{S}\mathrm{Var}\left[\frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j}\right]$$

- The mini-batch size S is a hyperparameter. Typical values are 10 or 100.

Stochastic Gradient Descent: Batch Size

- The mini-batch size S is a hyperparameter that needs to be set.
  - **Large batches:** converge in fewer weight updates because each stochastic gradient is less noisy.
  - **Small batches:** perform more weight updates per second because each one requires less computation.
- **Claim:** If the wall-clock time were proportional to the number of FLOPs, then $S = 1$ would be optimal.
- 100 updates with $S = 1$ requires the same FLOP count as 1 update with $S = 100$.
- Rewrite minibatch gradient descent as a for-loop:

S = 1

For $k = 1, \ldots, 100$:

$$\boldsymbol{\theta}_k \leftarrow \boldsymbol{\theta}_{k-1} - \alpha \nabla \mathcal{J}^{(k)}(\boldsymbol{\theta}_{k-1})$$

S = 100

For $k = 1, \ldots, 100$:

$$\boldsymbol{\theta}_k \leftarrow \boldsymbol{\theta}_{k-1} - \tfrac{\alpha}{100} \nabla \mathcal{J}^{(k)}(\boldsymbol{\theta}_0)$$

- All else being equal, you'd prefer to compute the gradient at a fresher value of $\boldsymbol{\theta}$. So $S = 1$ is better.

Stochastic Gradient Descent: Batch Size

- The mini-batch size S is a hyperparameter that needs to be set.
  - **Large batches:** converge in fewer weight updates because each stochastic gradient is less noisy.
  - **Small batches:** perform more weight updates per second because each one requires less computation.
- **Claim:** If the wall-clock time were proportional to the number of FLOPs, then $S = 1$ would be optimal.
  - 100 updates with $S = 1$ requires the same FLOP count as 1 update with $S = 100$.
- Rewrite minibatch gradient descent as a for-loop:

S = 1

For $k = 1, \ldots, 100$:

$$\boldsymbol{\theta}_k \leftarrow \boldsymbol{\theta}_{k-1} - \alpha \nabla \mathcal{J}^{(k)}(\boldsymbol{\theta}_{k-1})$$

S = 100

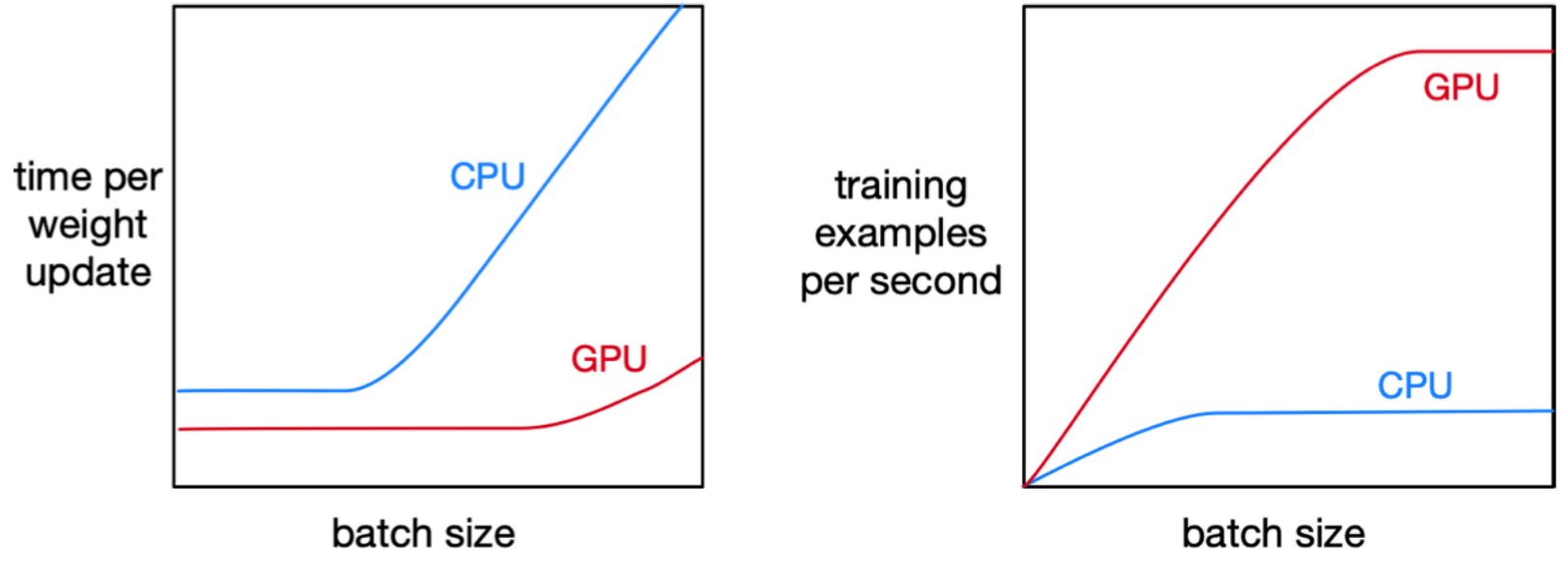
For $k = 1, \ldots, 100$:

$$\boldsymbol{\theta}_k \leftarrow \boldsymbol{\theta}_{k-1} - \tfrac{\alpha}{100} \nabla \mathcal{J}^{(k)}(\boldsymbol{\theta}_0)$$

- All else being equal, you'd prefer to compute the gradient at a fresher value of $\boldsymbol{\theta}$. So $S = 1$ is better.

Stochastic Gradient Descent: Batch Size

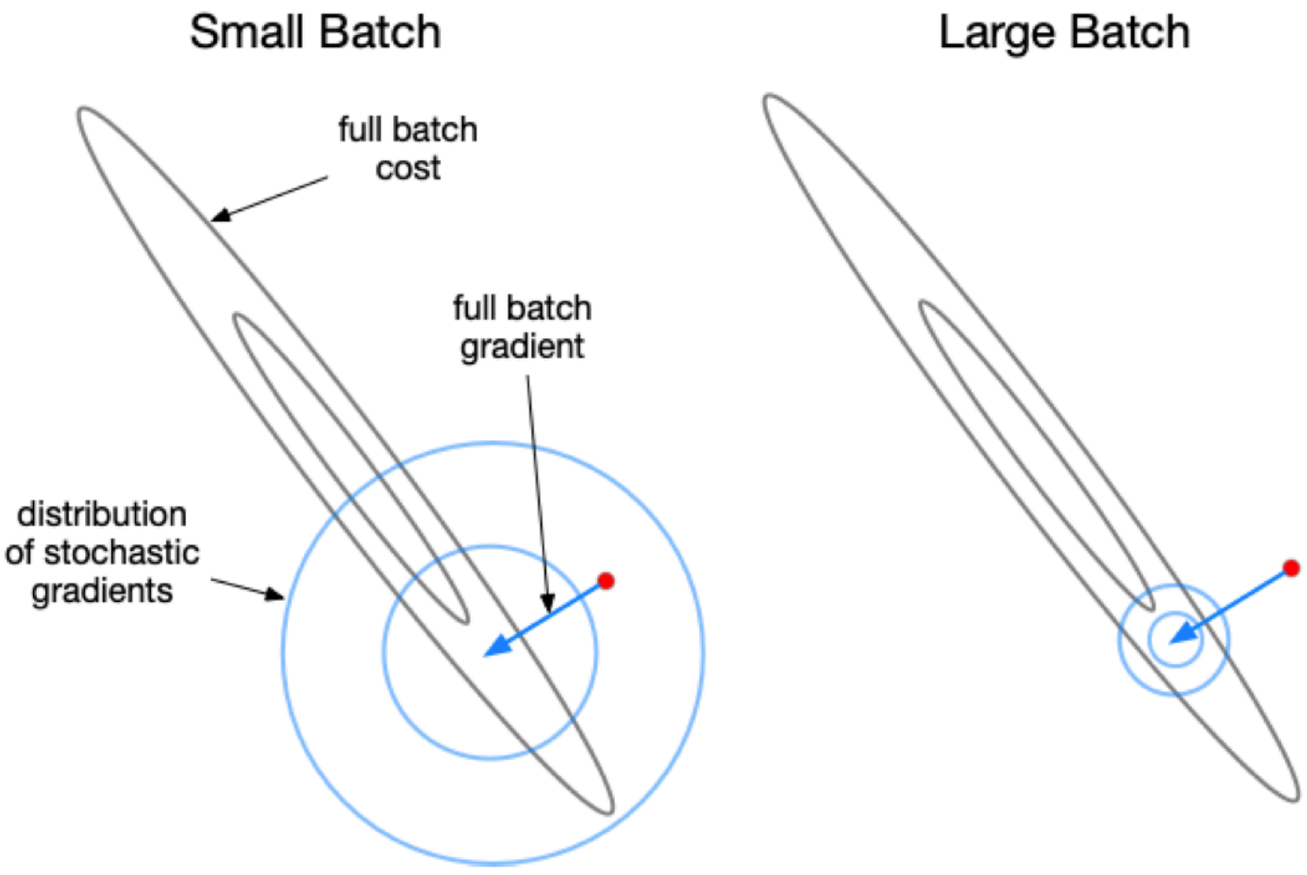
- The reason we don't use $S = 1$ is that larger batches can take advantage of fast matrix operations and parallelism.
- **Small batches:** An update with $S = 10$ isn't much more expensive than an update with $S = 1$.
- **Large batches:** Once S is large enough to saturate the hardware efficiencies, the cost becomes linear in S.
- Cartoon figure, not drawn to scale:

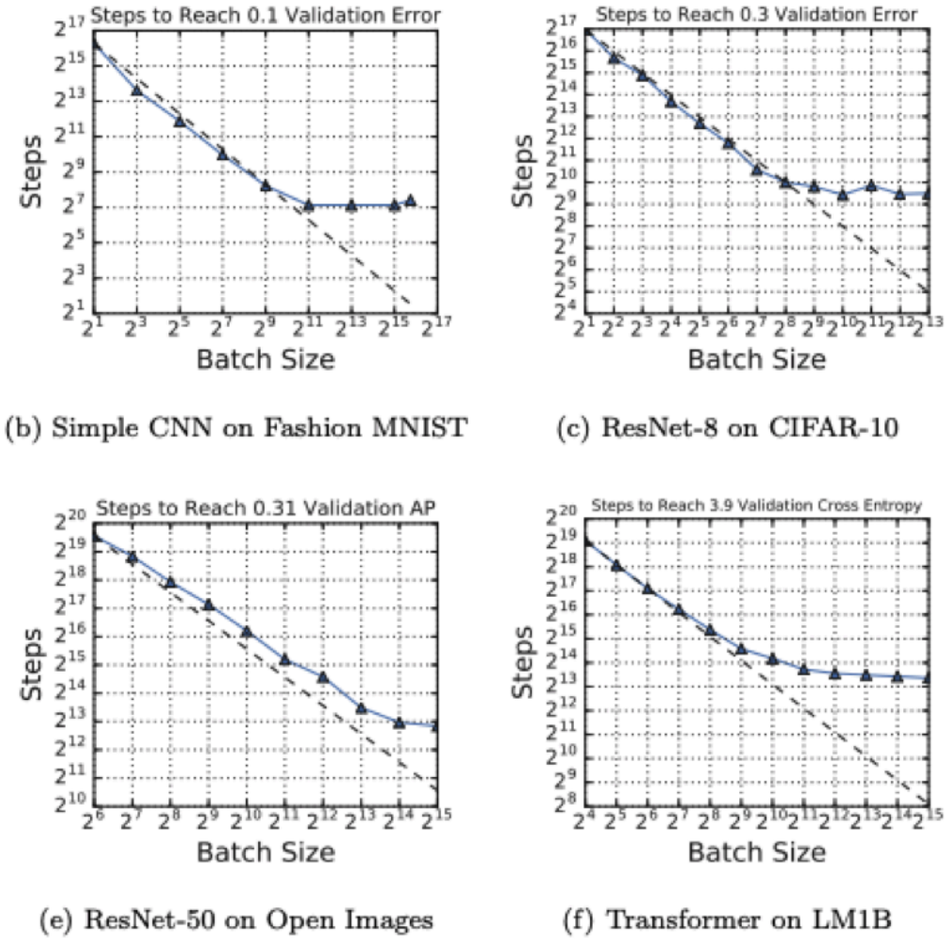


- Since GPUs afford more parallelism, they saturate at a larger batch size. Hence, GPUs tend to favor larger batch sizes.

Stochastic Gradient Descent: Batch Size

- The convergence benefits of larger batches also see diminishing returns.
- **Small batches:** large gradient noise, so large benefit from increased batch size
- **Large batches:** SGD approximates the batch gradient descent update, so no further benefit from variance reduction.





(b) Simple CNN on Fashion MNIST

(c) ResNet-8 on CIFAR-10

(e) ResNet-50 on Open Images

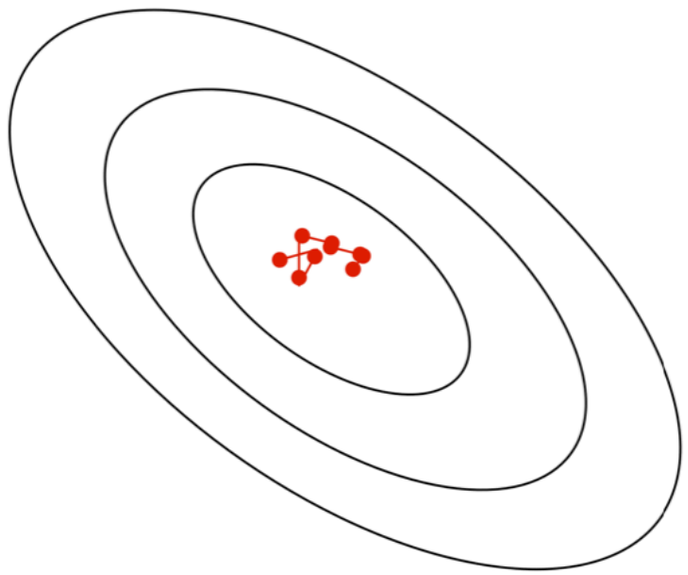
(f) Transformer on LM1B

- **Right:** # iterations to reach target validation error as a function of batch size. (Shallue et al., 2018)

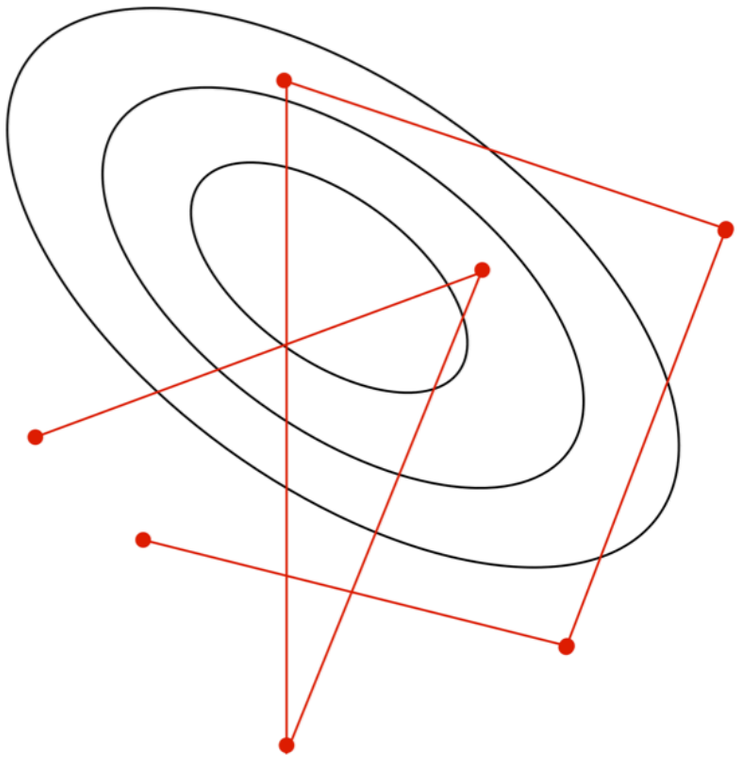
SGD Learning Rate

- In stochastic training, the learning rate also influences the fluctuations due to the stochasticity of the gradients.

small learning rate



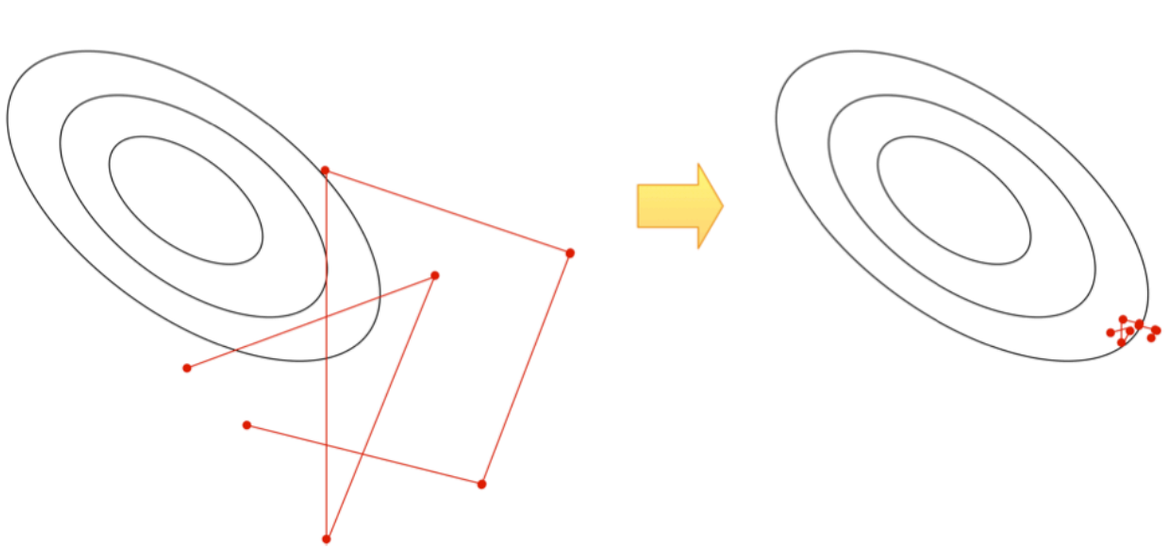
large learning rate

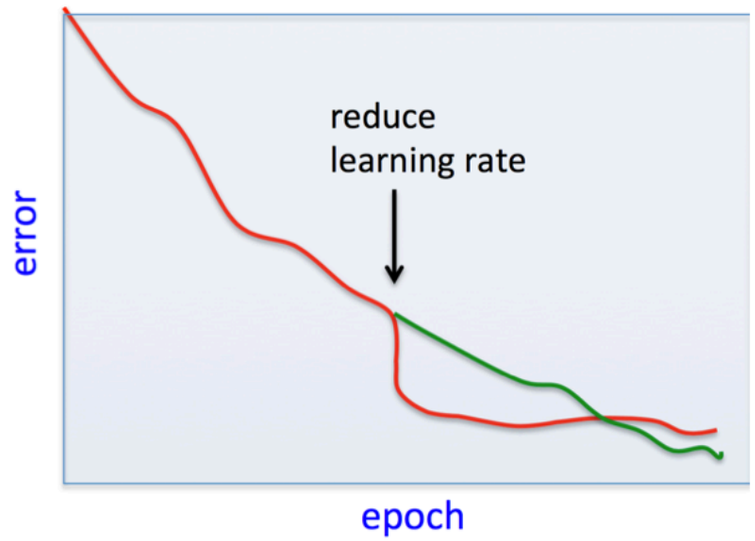


- Typical strategy:
  - Use a large learning rate early in training so you can get close to the optimum
  - Gradually decay the learning rate to reduce the fluctuations

SGD Learning Rate

- Warning: by reducing the learning rate, you reduce the fluctuations, which can appear to make the loss drop suddenly. But this can come at the expense of long-run performance.





Recap

Problem	Diagnostics	Workarounds
incorrect gradients	finite differences	fix them, or use autodiff
local optima	(hard)	random restarts
symmetries	visualize $\mathbf{W}$	initialize $\mathbf{W}$ randomly
slow progress	slow, linear training curve	increase α; momentum
instability	cost increases	decrease α
oscillations	fluctuations in training curve	decrease α; momentum
fluctuations	fluctuations in training curve	decay α; iterate averaging
dead/saturated units	activation histograms	initial scale of $\mathbf{W}$; ReLU
ill-conditioning	(hard)	normalization; momentum; Adam; second-order opt.

Recurrent Neural Networks

Overview

- Sometimes we're interested in predicting sequences
  - Speech-to-text and text-to-speech
  - Caption generation
  - Machine translation
- If the input is also a sequence, this setting is known as sequence-to-sequence prediction.
- We already saw one way of doing this: neural language models
  - But autoregressive models are memoryless, so they can't learn long-distance dependencies.
  - Recurrent neural networks (RNNs) are a kind of architecture which can remember things over time.

Overview

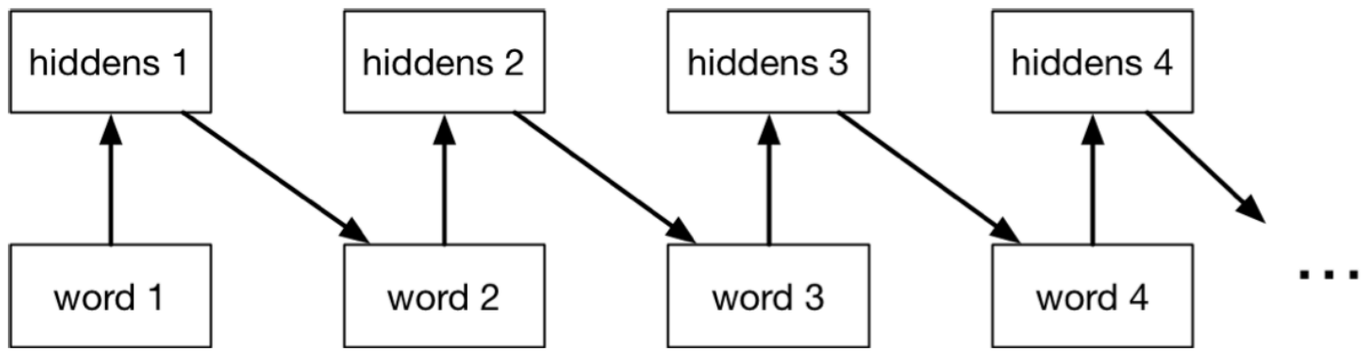
Recall that we made a Markov assumption:

$$p(w_i \mid w_1, \ldots, w_{i-1}) = p(w_i \mid w_{i-3}, w_{i-2}, w_{i-1}).$$

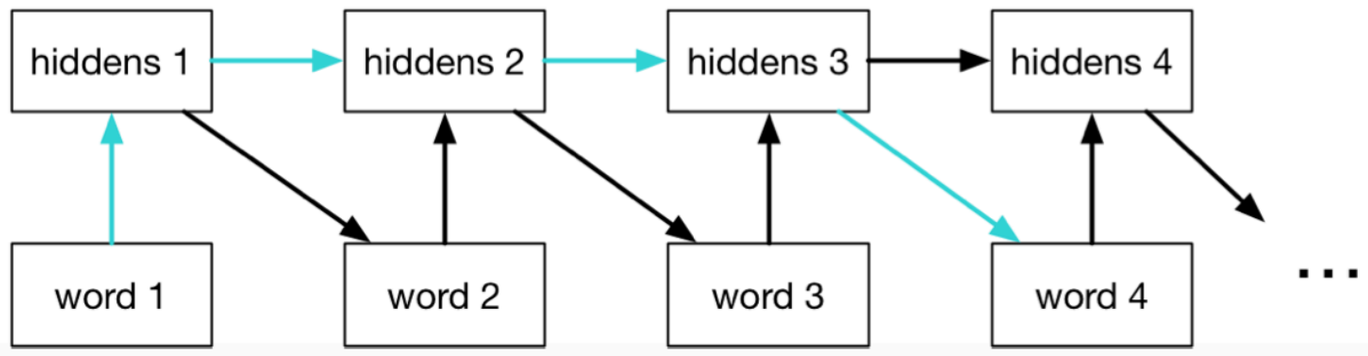
This means the model is memoryless, i.e. it has no memory of anything before the last few words. But sometimes long-distance context can be important.

Overview

- Autoregressive models such as the neural language model are memoryless, so they can only use information from their immediate context (in this figure, context length $= 1$):

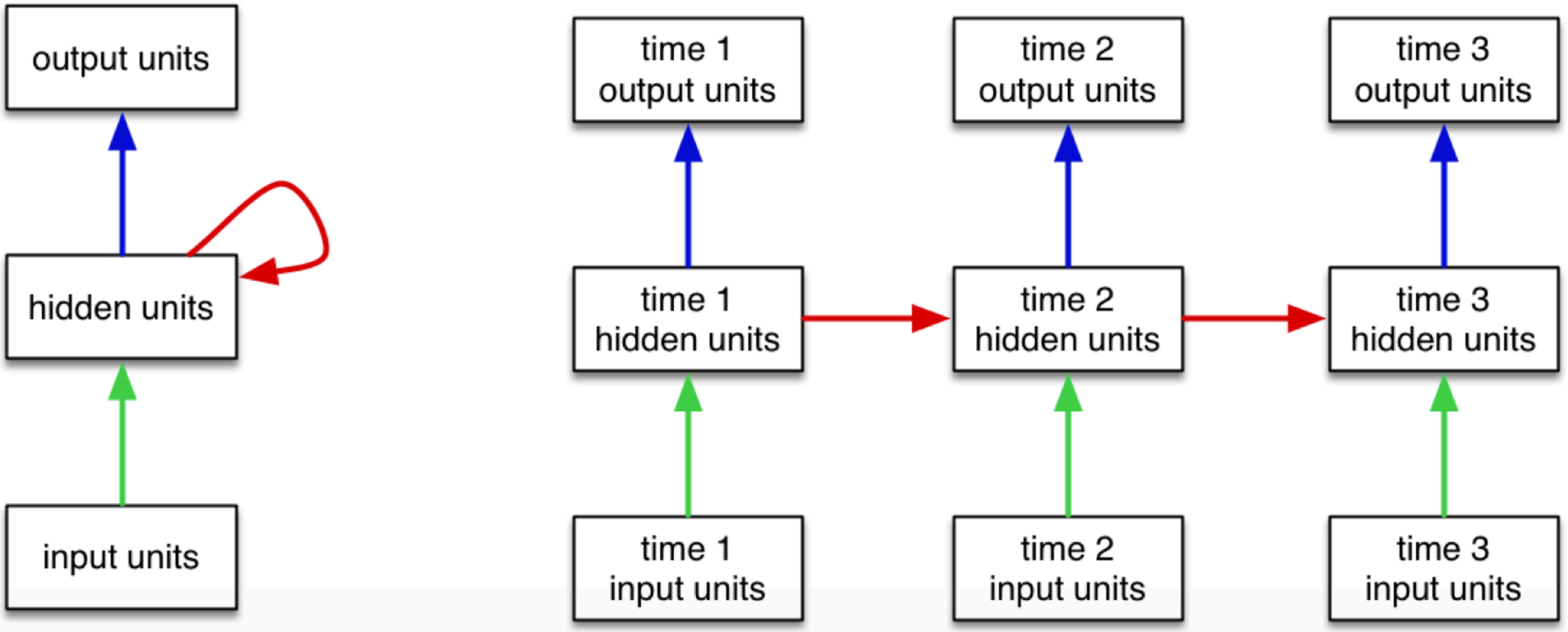


- If we add connections between the hidden units, it becomes a recurrent neural network (RNN). Having a memory lets an RNN use longer-term dependencies:



Recurrent neural nets

- We can think of an RNN as a dynamical system with one set of hidden units which feed into themselves. The network's graph would then have self-loops.
- We can unroll the RNN's graph by explicitly representing the units at all time steps. The weights and biases are shared between all time steps
    - Except there is typically a separate set of biases for the first time step.



Example: Parity

Assume we have a sequence of binary inputs. We'll consider how to determine the parity, i.e. whether the number of 1's is even or odd.

We can compute parity incrementally by keeping track of the parity of the input so far:

Parity bits: 0 1 1 0 1 1 $\longrightarrow$

Input: 0 1 0 1 1 0 1 0 1 1

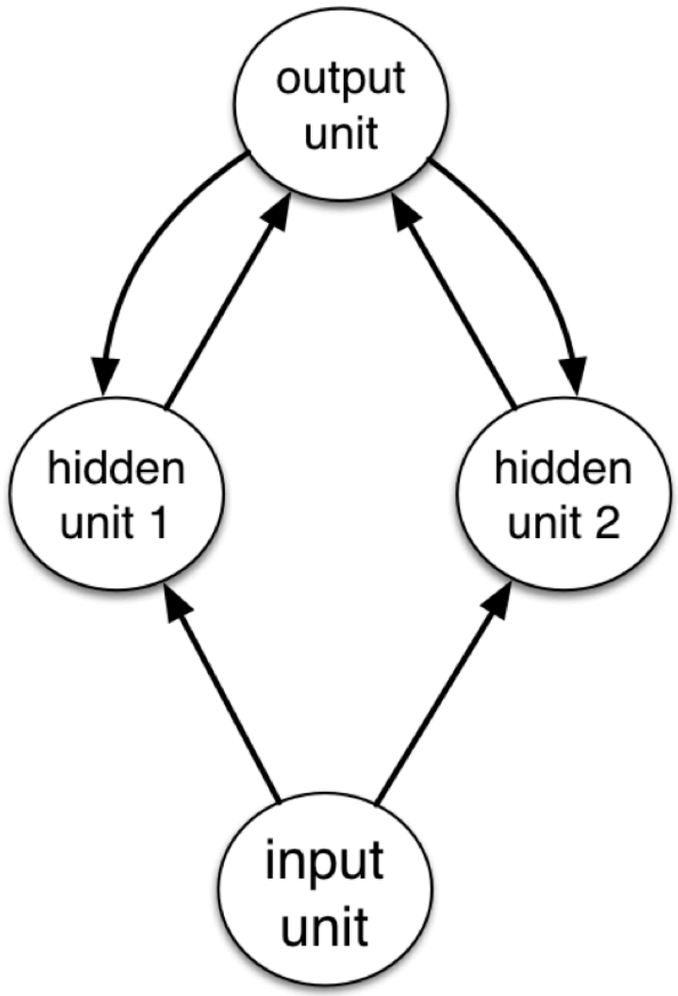
Each parity bit is the XOR of the input and the previous parity bit.

Parity is a classic example of a problem that's hard to solve with a shallow feed-forward net, but easy to solve with an RNN.

Example: Parity

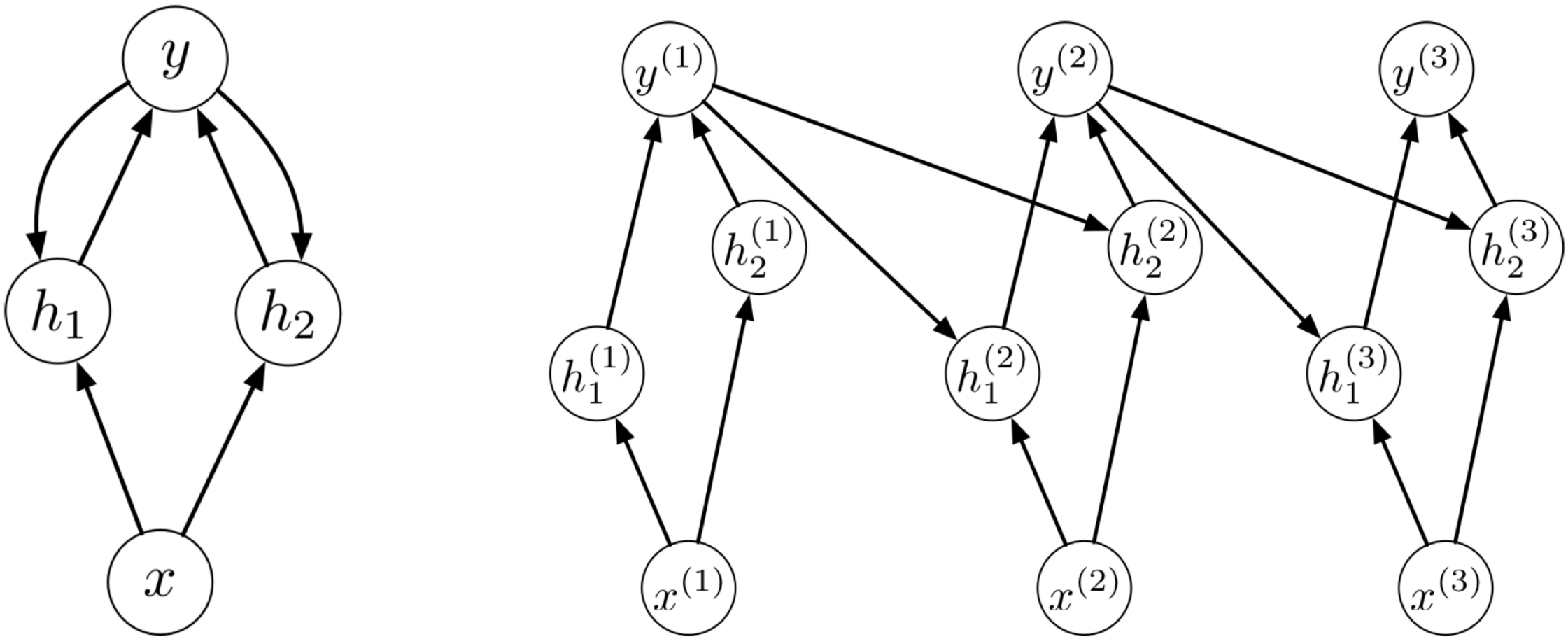
Assume we have a sequence of binary inputs. We'll consider how to determine the parity, i.e. whether the number of 1's is even or odd.

- Let's find weights and biases for the RNN on the right so that it computes the parity. All hidden and output units are **binary threshold units**.
- **Strategy:**
  - The output unit tracks the current parity, which is the XOR of the current input and previous output.
  - The hidden units help us compute the XOR.



Example: Parity

Unrolling the parity RNN:



Example: Parity

The output unit should compute the XOR of the current input and previous output:

$y^{(t-1)}$	$x^{(t)}$	$y^{(t)}$
0	0	0
0	1	1
1	0	1
1	1	0

Example: Parity

Let's use hidden units to help us compute XOR.

- Have one unit compute AND, and the other one compute OR.
- Then we can pick weights and biases just like we did for multilayer perceptrons.

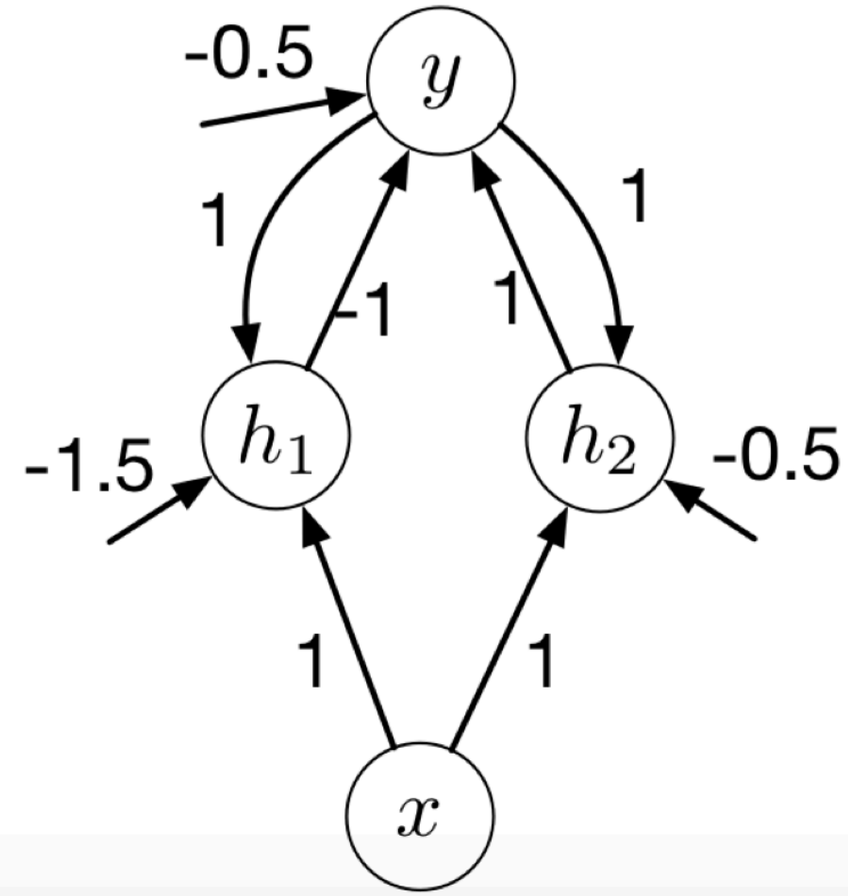
$y^{(t-1)}$	$x^{(t)}$	$h_1^{(t)}$	$h_2^{(t)}$	$y^{(t)}$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Example: Parity

Let's use hidden units to help us compute XOR.

- Have one unit compute AND, and the other one compute OR.
- Then we can pick weights and biases just like we did for multilayer perceptrons.

$y^{(t-1)}$	$x^{(t)}$	$h_1^{(t)}$	$h_2^{(t)}$	$y^{(t)}$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

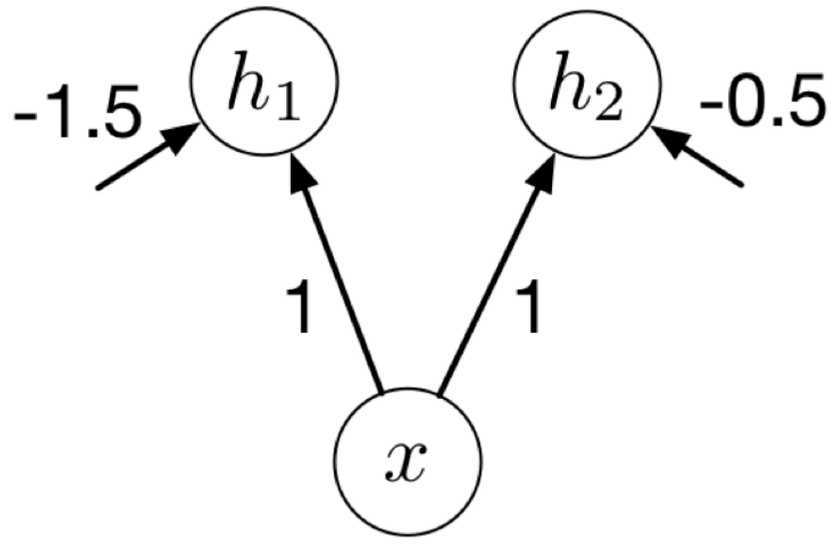


Example: Parity

We still need to determine the hidden biases for the first time step.

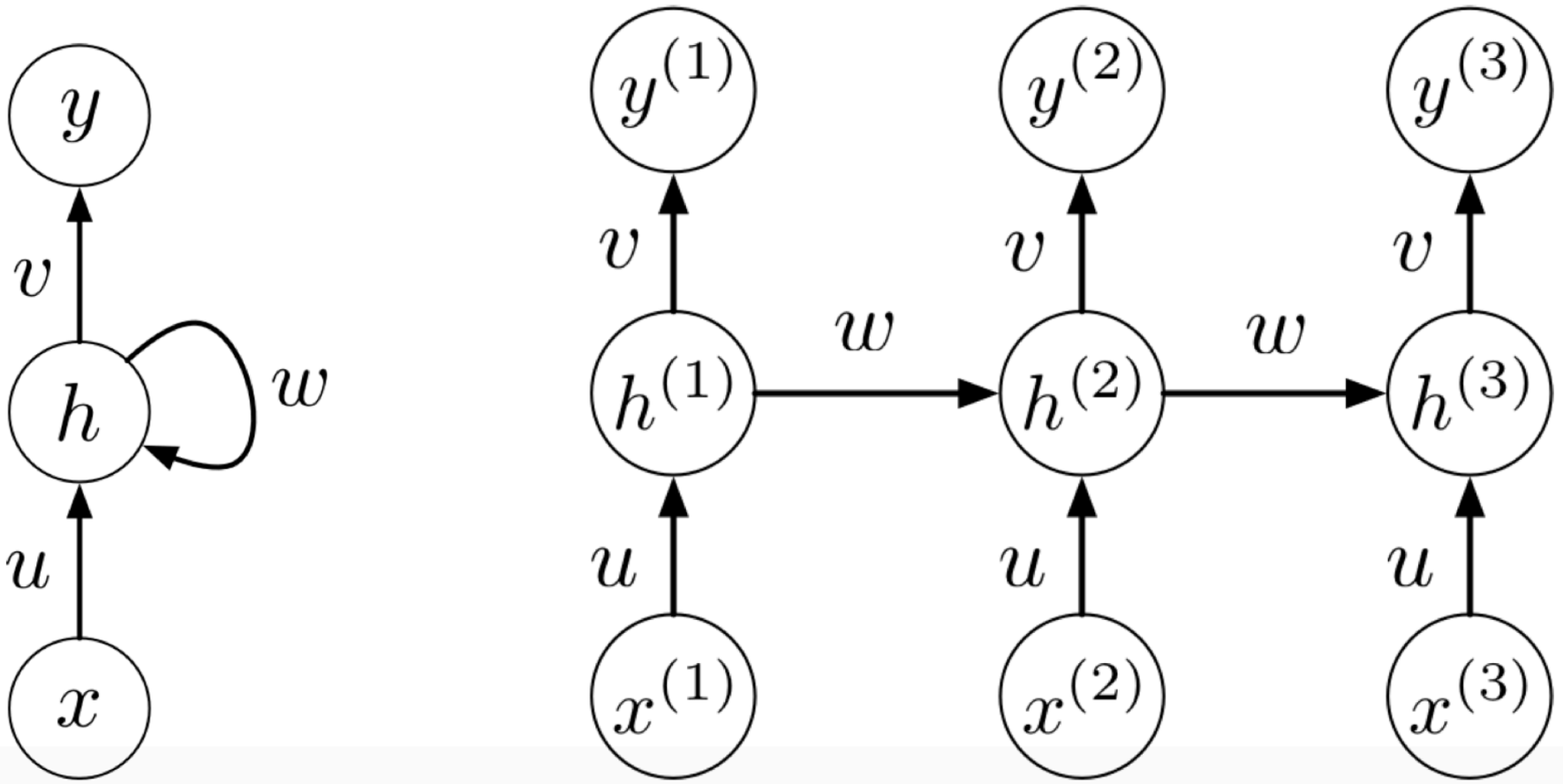
- The network should behave as if the previous output was 0. This is represented with the following table:

$x^{(1)}$	$h_1^{(1)}$	$h_2^{(1)}$
0	0	0
1	0	1



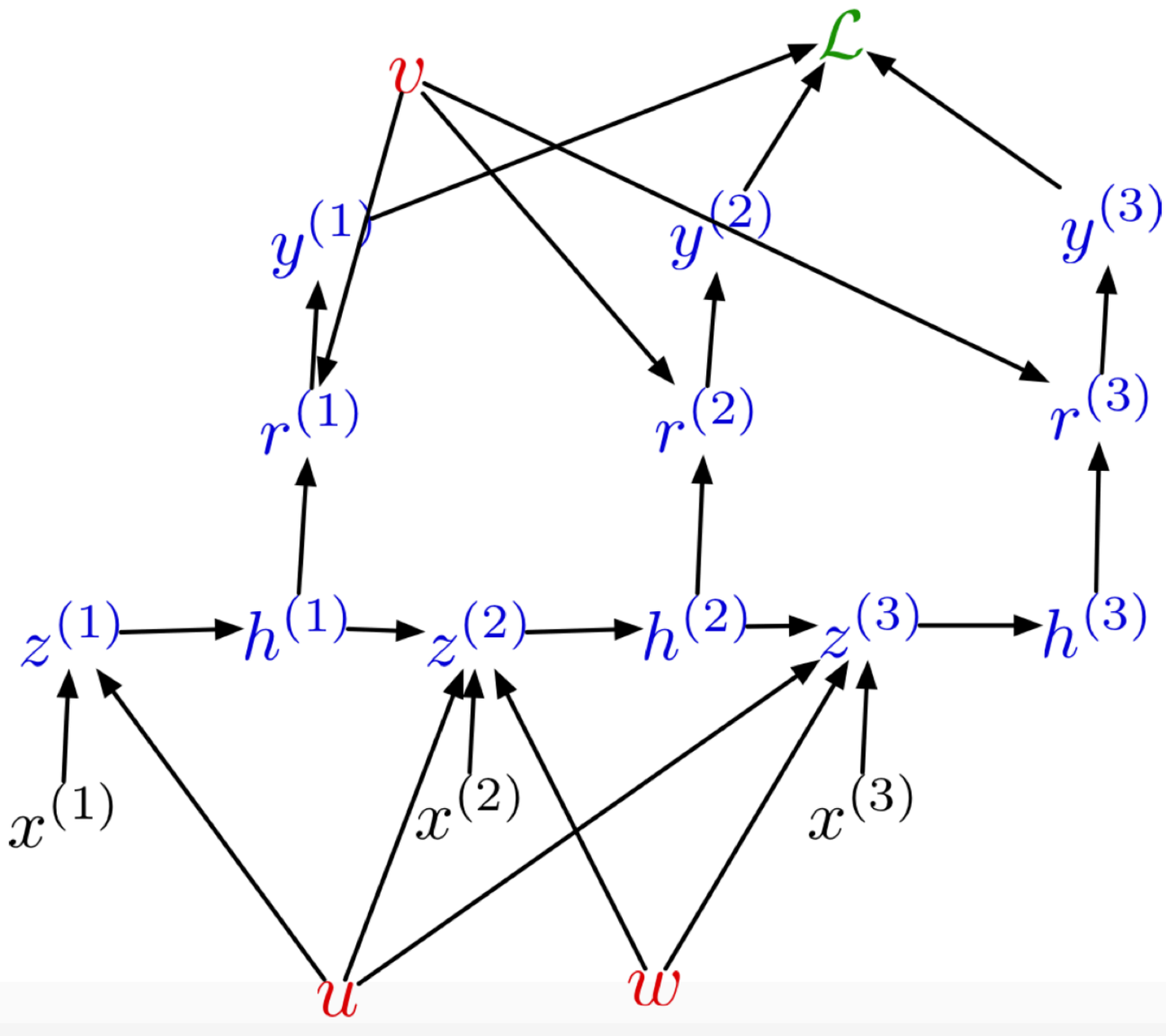
Backprop Through Time

- As you can guess, we don't usually set RNN weights by hand. Instead, we learn them using backprop.
- In particular, we do backprop on the unrolled network. This is known as backprop through time.

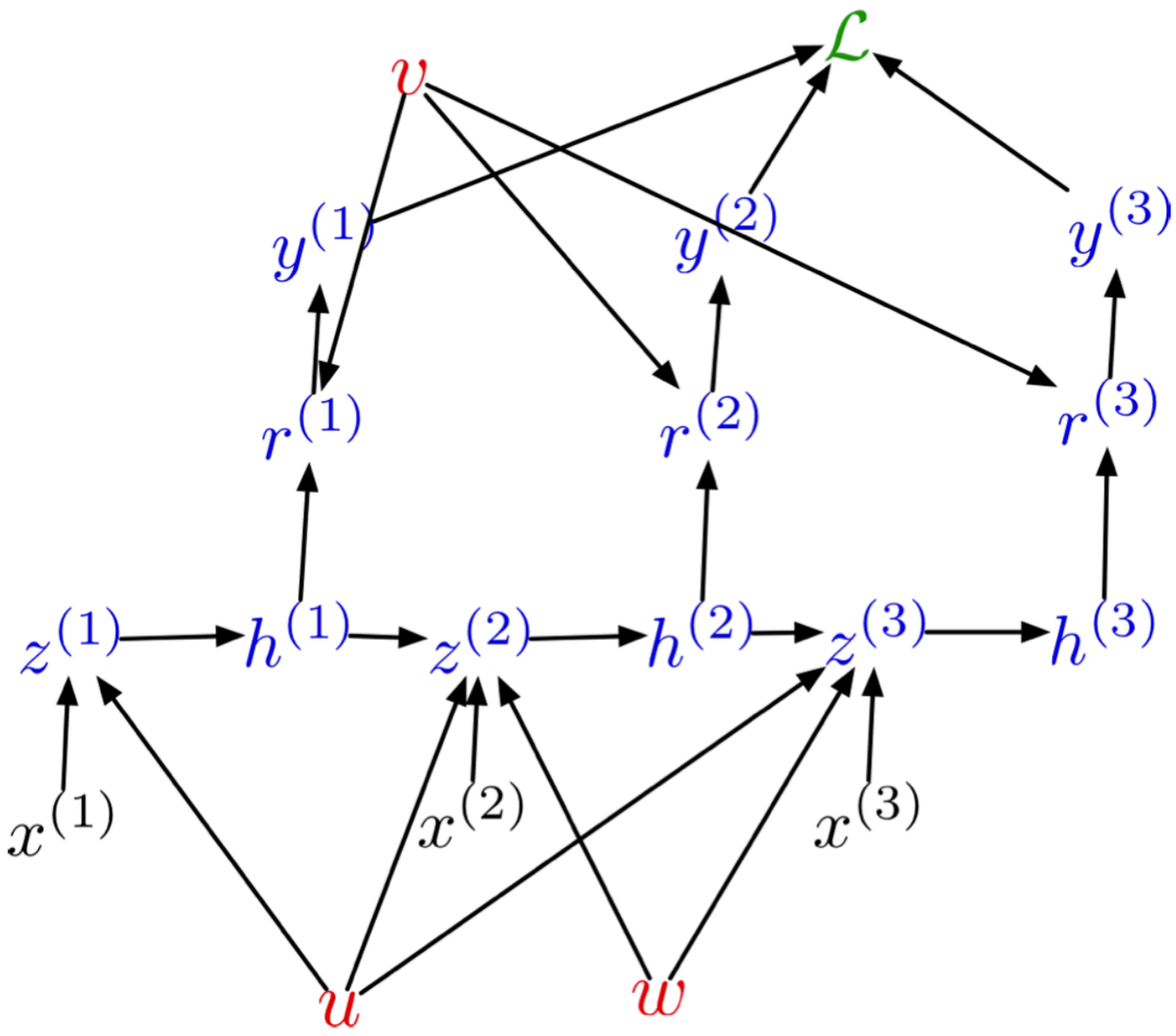


Backprop Through Time

Here's the unrolled computation graph. Notice the weight sharing.



Backprop Through Time



Activations:

$$\overline{\mathcal{L}} = 1$$

$$\overline{y^{(t)}} = \overline{\mathcal{L}} \frac{\partial \mathcal{L}}{\partial y^{(t)}}$$

$$\overline{r^{(t)}} = \overline{y^{(t)}}\, \phi'(r^{(t)})$$

$$\overline{h^{(t)}} = \overline{r^{(t)}}\, v + \overline{z^{(t+1)}}\, w$$

$$\overline{z^{(t)}} = \overline{h^{(t)}}\, \phi'(z^{(t)})$$

Parameters:

$$\overline{u} = \sum_t \overline{z^{(t)}}\, x^{(t)}$$

$$\overline{v} = \sum_t \overline{r^{(t)}}\, h^{(t)}$$

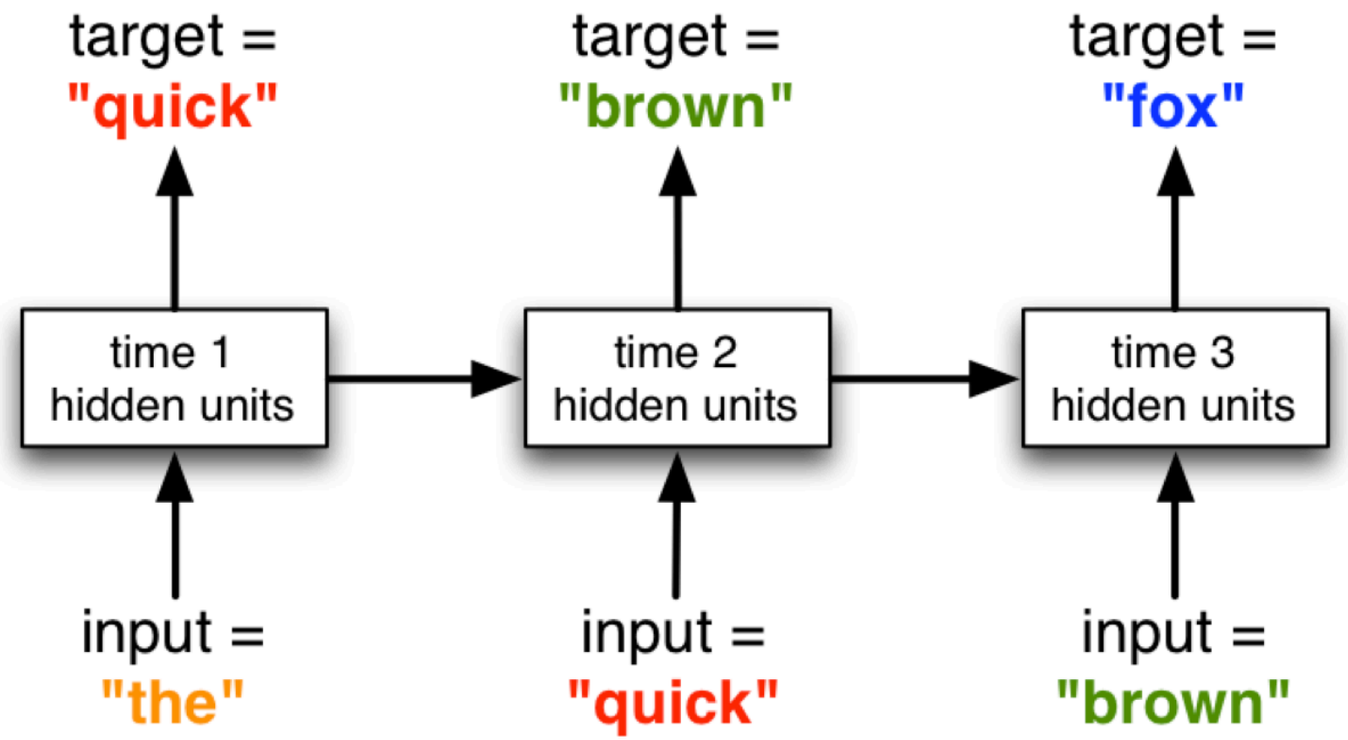
$$\overline{w} = \sum_t \overline{z^{(t+1)}}\, h^{(t)}$$

Backprop Through Time

- Now you know how to compute the derivatives using backprop through time.
- The hard part is using the derivatives in optimization. They can explode or vanish. Addressing this issue will take all of the next lecture.

Language Modeling

One way to use RNNs as a language model:

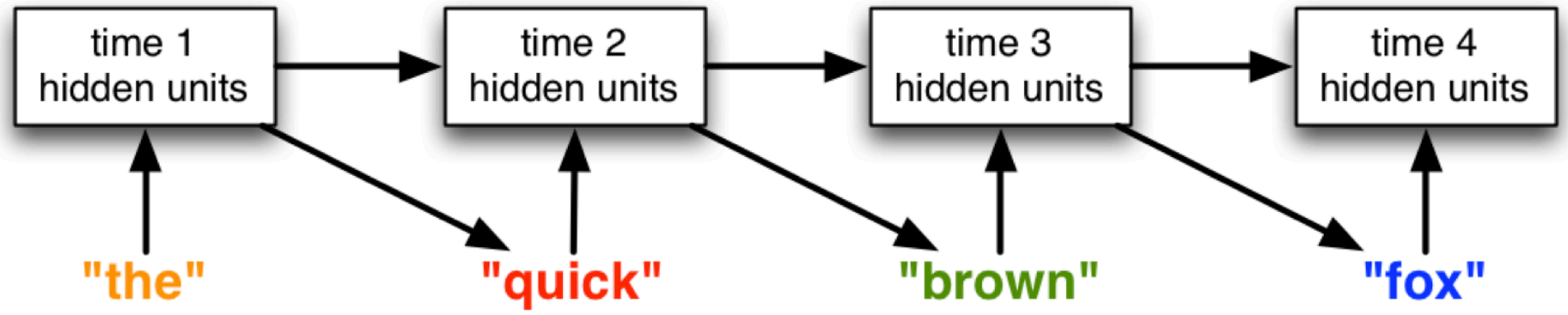


As with our language model, each word is represented as an indicator vector, the model predicts a distribution, and we can train it with cross-entropy loss.

This model can learn long-distance dependencies.

Language Modeling

When we generate from the model (i.e. compute samples from its distribution over sentences), the outputs feed back in to the network as inputs.



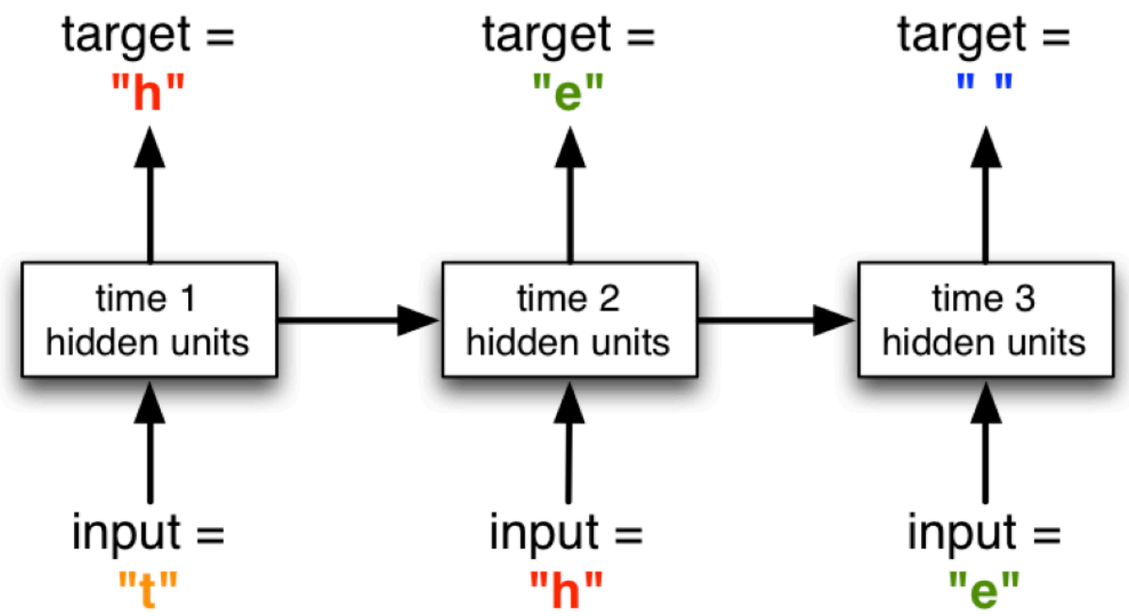
At training time, the inputs are the tokens from the training set (rather than the network's outputs). This is called teacher forcing.

Some remaining challenges:

- Vocabularies can be very large once you include people, places, etc. It's computationally difficult to predict distributions over millions of words.
- How do we deal with words we haven't seen before?
- In some languages (e.g. German), it's hard to define what should be considered a word.

Language Modeling

Another approach is to model text *one character at a time*!



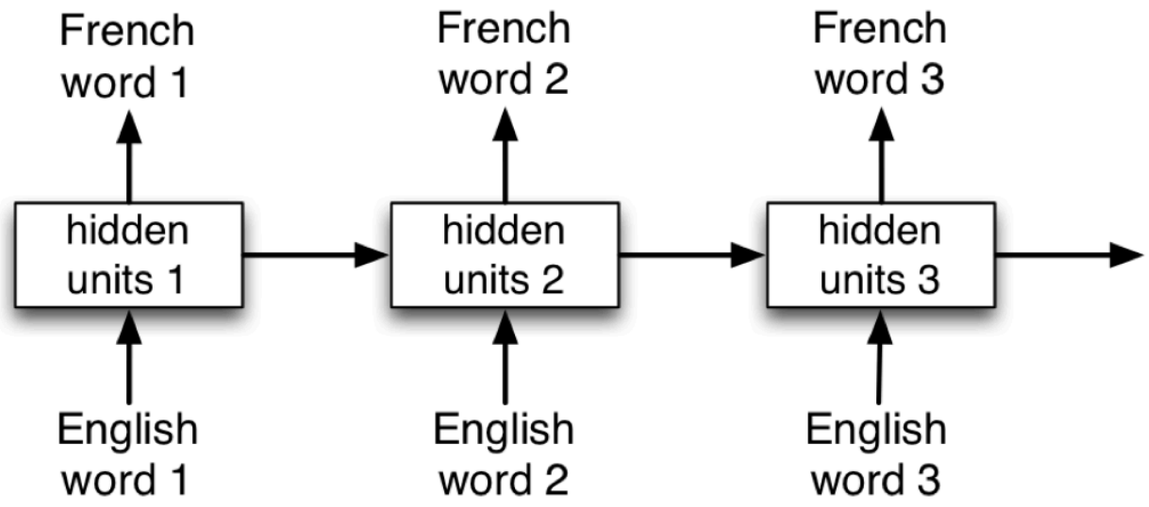
This solves the problem of what to do about previously unseen words.
Note that long-term memory is *essential* at the character level!

Note: modeling language well at the character level requires *multiplicative* interactions, which we're not going to talk about.

Neural Machine Translation

We'd like to translate, e.g., English to French sentences, and we have pairs of translated sentences to train on.

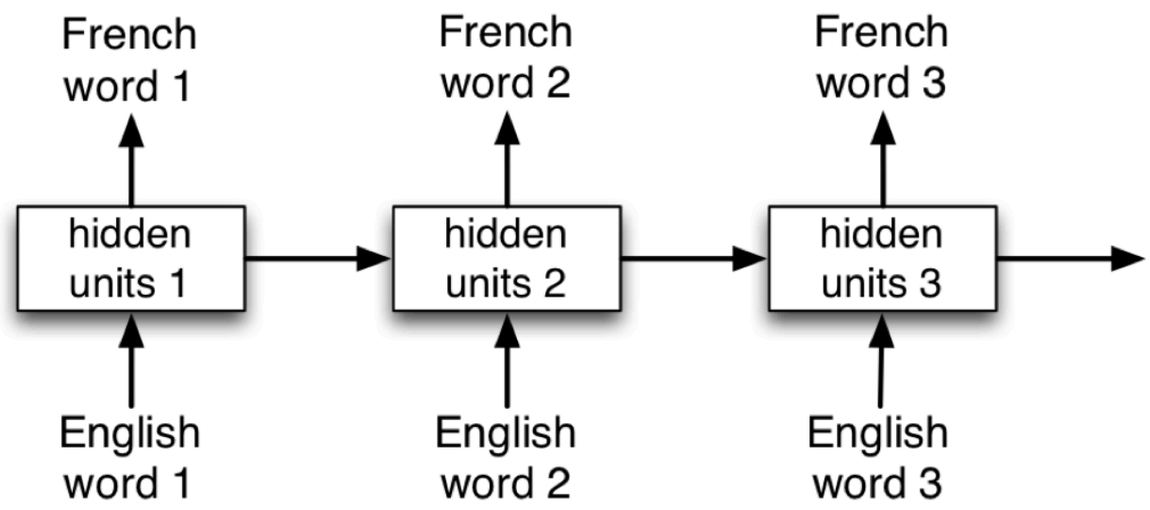
What's wrong with the following setup?



Neural Machine Translation

We'd like to translate, e.g., English to French sentences, and we have pairs of translated sentences to train on.

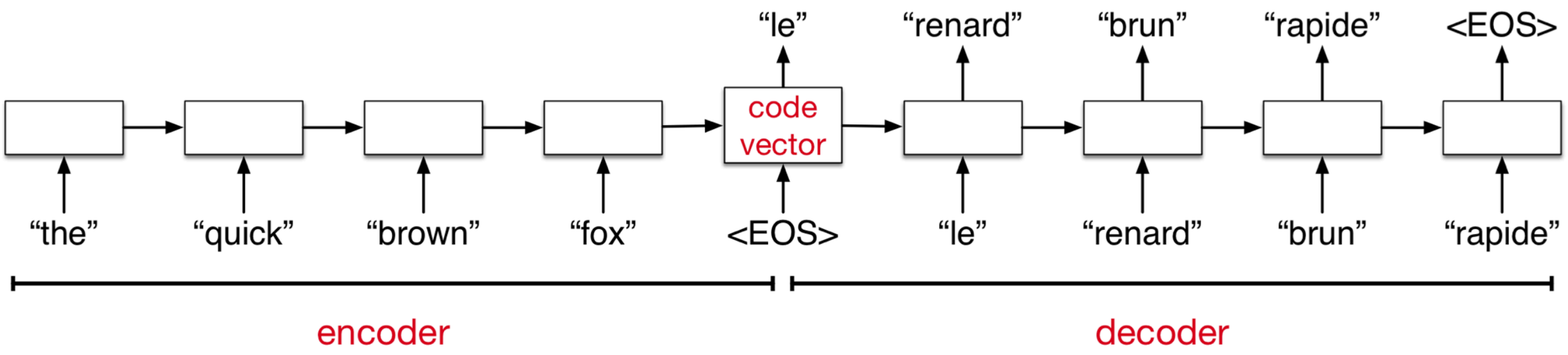
What's wrong with the following setup?



- The sentences might not be the same length, and the words might not align perfectly.
- You might need to resolve ambiguities using information from later in the sentence.

Neural Machine Translation

Sequence-to-sequence architecture: the network first reads and memorizes the sentence. When it sees the end token, it starts outputting the translation.



The encoder and decoder are two different networks with different weights.

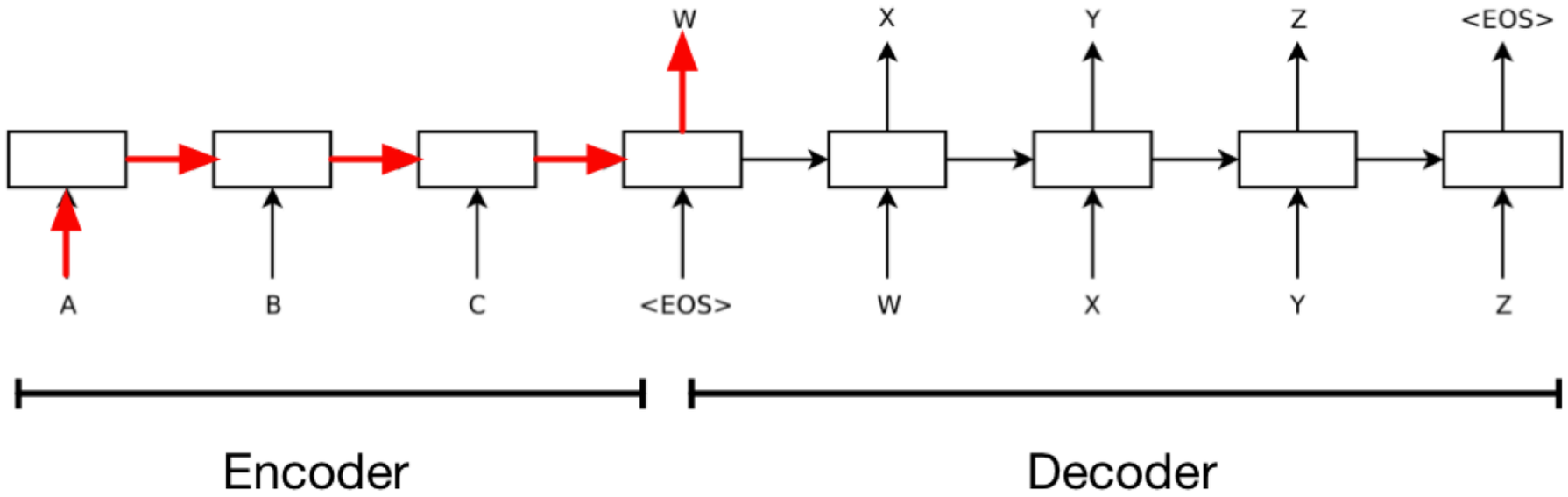
Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation, K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, Y. Bengio. EMNLP 2014.

Sequence to Sequence Learning with Neural Networks, Ilya Sutskever, Oriol Vinyals and Quoc Le, NIPS 2014.

Exploding and Vanishing Gradients

Why Gradients Explode or Vanish

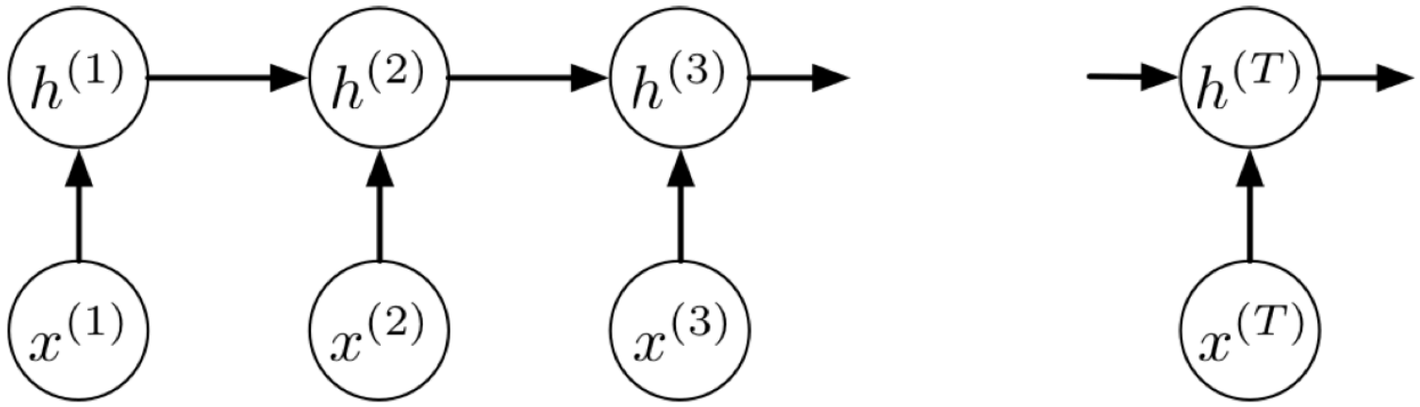
- Recall the RNN for machine translation. It reads an entire English sentence, and then has to output its French translation.



- A typical sentence length is 20 words. This means there's a gap of 20 time steps between when it sees information and when it needs it.
- The derivatives need to travel over this entire pathway.

Why Gradients Explode or Vanish

Consider a univariate version of the encoder network:



Backprop updates:

$$\overline{h^{(t)}} = \overline{z^{(t+1)}}\, w$$

$$\overline{z^{(t)}} = \overline{h^{(t)}}\, \phi'(z^{(t)})$$

Applying this recursively:

$$\overline{h^{(1)}} = \underbrace{w^{T-1}\phi'(z^{(2)}) \cdots \phi'(z^{(T)})}_{\text{the Jacobian } \partial h^{(T)}/\partial h^{(1)}}\, \overline{h^{(T)}}$$

With linear activations:

$$\partial h^{(T)}/\partial h^{(1)} = w^{T-1}$$

Exploding:

$$w = 1.1, T = 50 \quad \Rightarrow \quad \frac{\partial h^{(T)}}{\partial h^{(1)}} = 117.4$$

Vanishing:

$$w = 0.9, T = 50 \quad \Rightarrow \quad \frac{\partial h^{(T)}}{\partial h^{(1)}} = 0.00515$$

Why Gradients Explode or Vanish

- More generally, in the multivariate case, the Jacobians multiply:

$$\frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{h}^{(1)}} = \frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{h}^{(T-1)}} \cdots \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{h}^{(1)}}$$

- Matrices can explode or vanish just like scalar values, though it's slightly harder to make precise.
- Contrast this with the forward pass:
  - The forward pass has nonlinear activation functions which squash the activations, preventing them from blowing up.
  - The backward pass is linear, so it's hard to keep things stable. There's a thin line between exploding and vanishing.

Why Gradients Explode or Vanish

- We just looked at exploding/vanishing gradients in terms of the mechanics of backprop. Now let's think about it conceptually.
- The Jacobian $\partial \mathbf{h}^{(T)} / \partial \mathbf{h}^{(1)}$ means, how much does $h^{(T)}$ change when you change $\mathbf{h}^{(1)}$?
- Each hidden layer computes some function of the previous hiddens and the current input:

$$\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)})$$

- This function gets iterated:

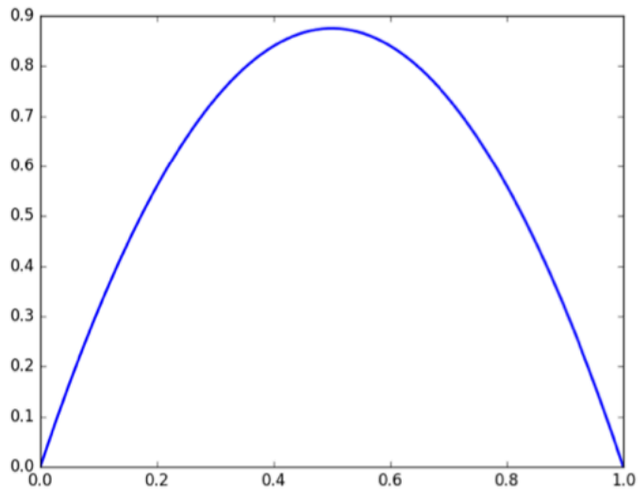
$$\mathbf{h}^{(4)} = f(f(f(\mathbf{h}^{(1)}, \mathbf{x}^{(2)}), \mathbf{x}^{(3)}), \mathbf{x}^{(4)}).$$

- Let's study iterated functions as a way of understanding what RNNs are computing.

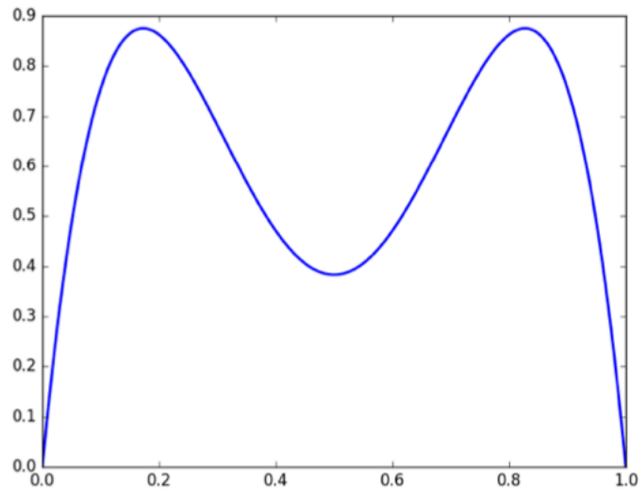
Iterated Functions

- Iterated functions are complicated. Consider:

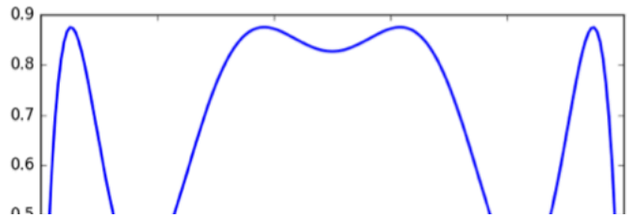
$$f(x) = 3.5\,x\,(1-x)$$

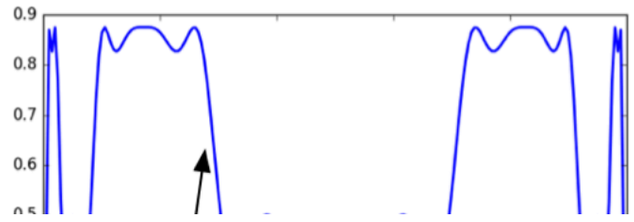


$y = f(x)$



$y = f(f(x))$

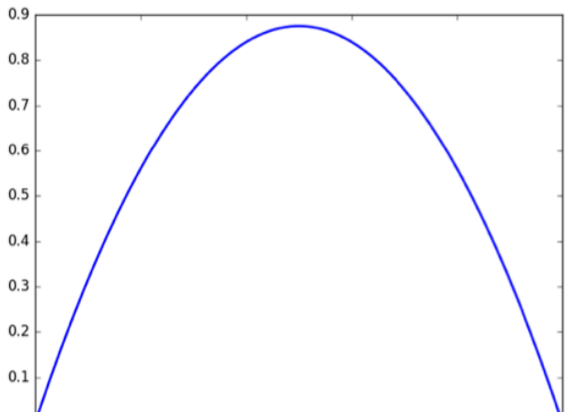




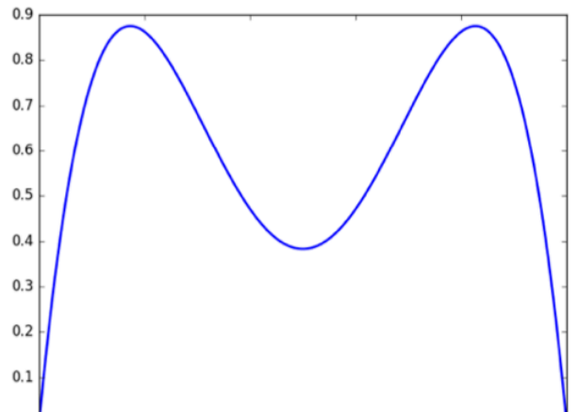
Iterated Functions

- Iterated functions are complicated. Consider:

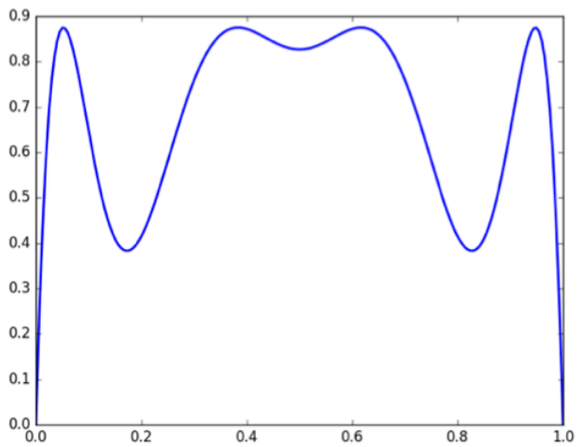
$$f(x) = 3.5\,x\,(1-x)$$



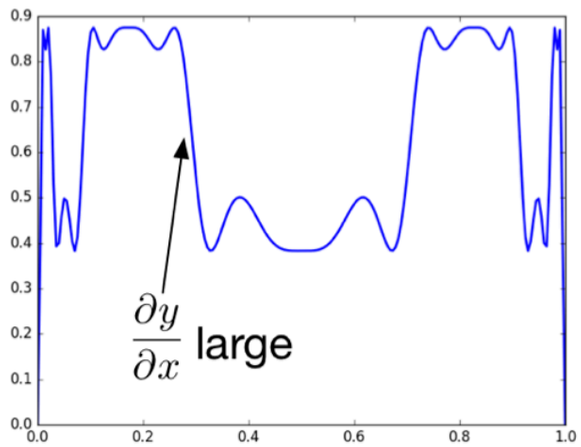
$y = f(x)$



$y = f(f(x))$



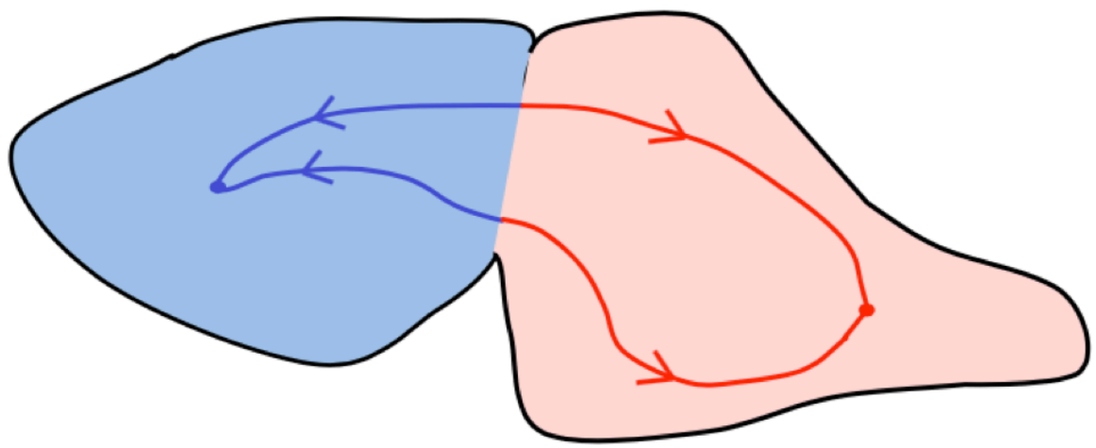
$y = f(f(f(x)))$



$y = \underbrace{f \circ \cdots \circ f}_{6 \text{ times}}(x)$

Why Gradients Explode or Vanish

- Let's imagine an RNN's behavior as a dynamical system, which has various attractors:

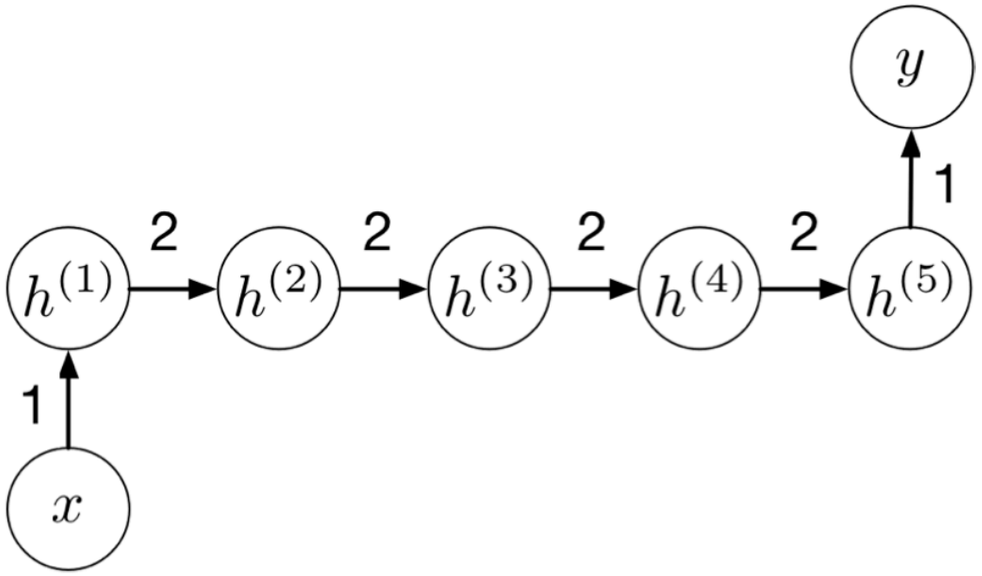


– Geoffrey Hinton, Coursera

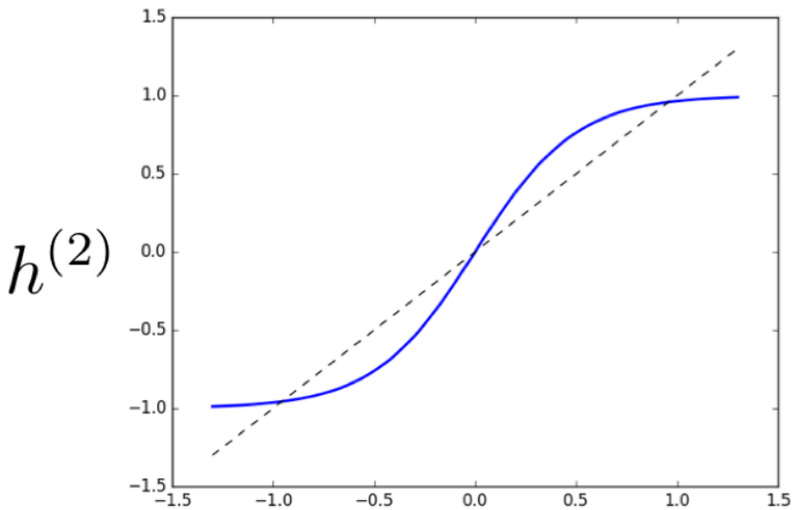
- Within one of the colored regions, the gradients vanish because even if you move a little, you still wind up at the same attractor.
- If you're on the boundary, the gradient blows up because moving slightly moves you from one attractor to the other.

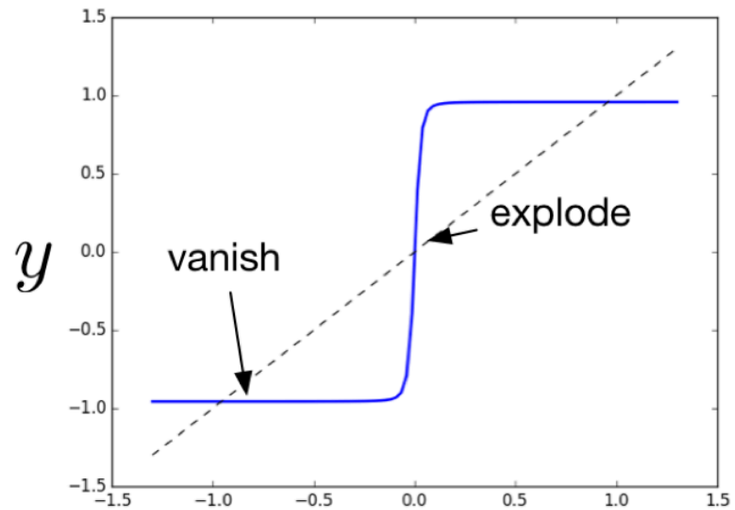
Why Gradients Explode or Vanish

- Consider an RNN with tanh activation function:



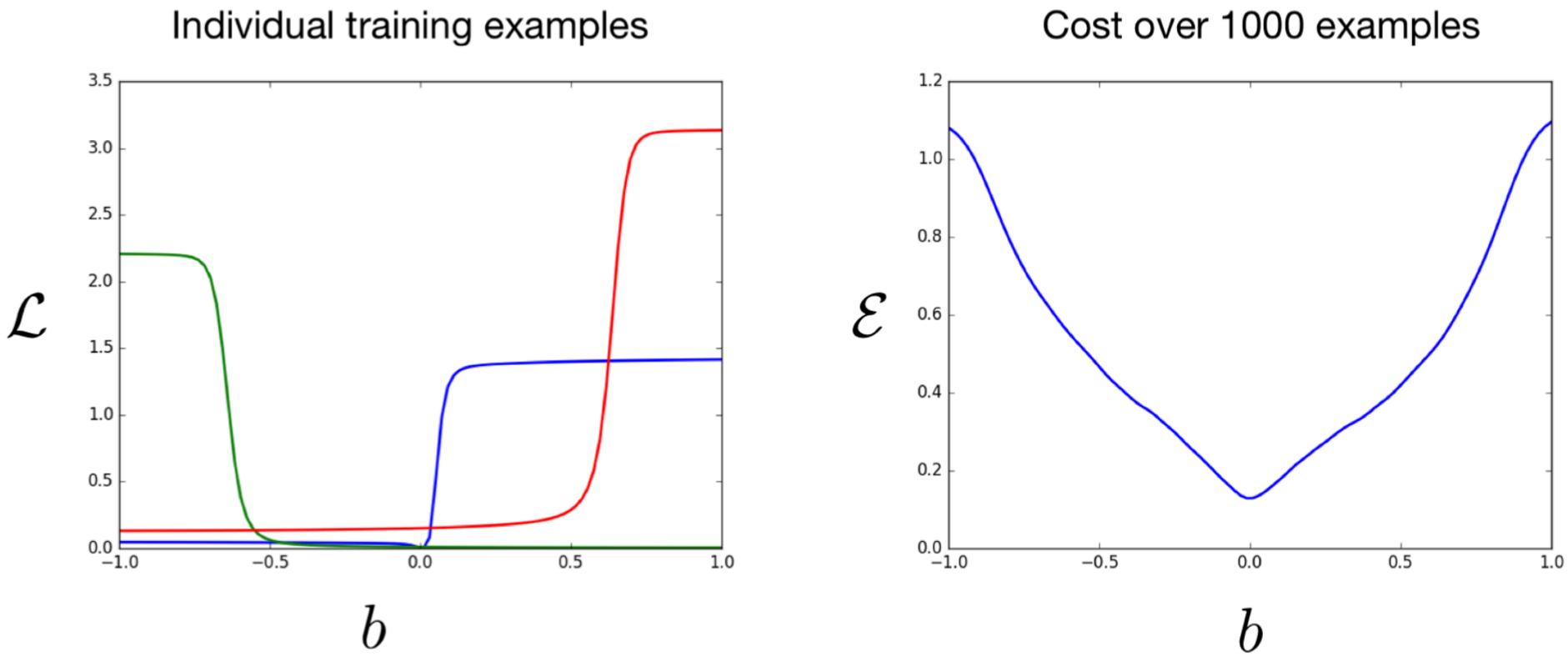
- The function computed by the network:





Why Gradients Explode or Vanish

- Cliffs make it hard to estimate the true cost gradient. Here are the loss and cost functions with respect to the bias parameter for the hidden units:



- Generally, the gradients will explode on some inputs and vanish on others. In expectation, the cost may be fairly smooth.

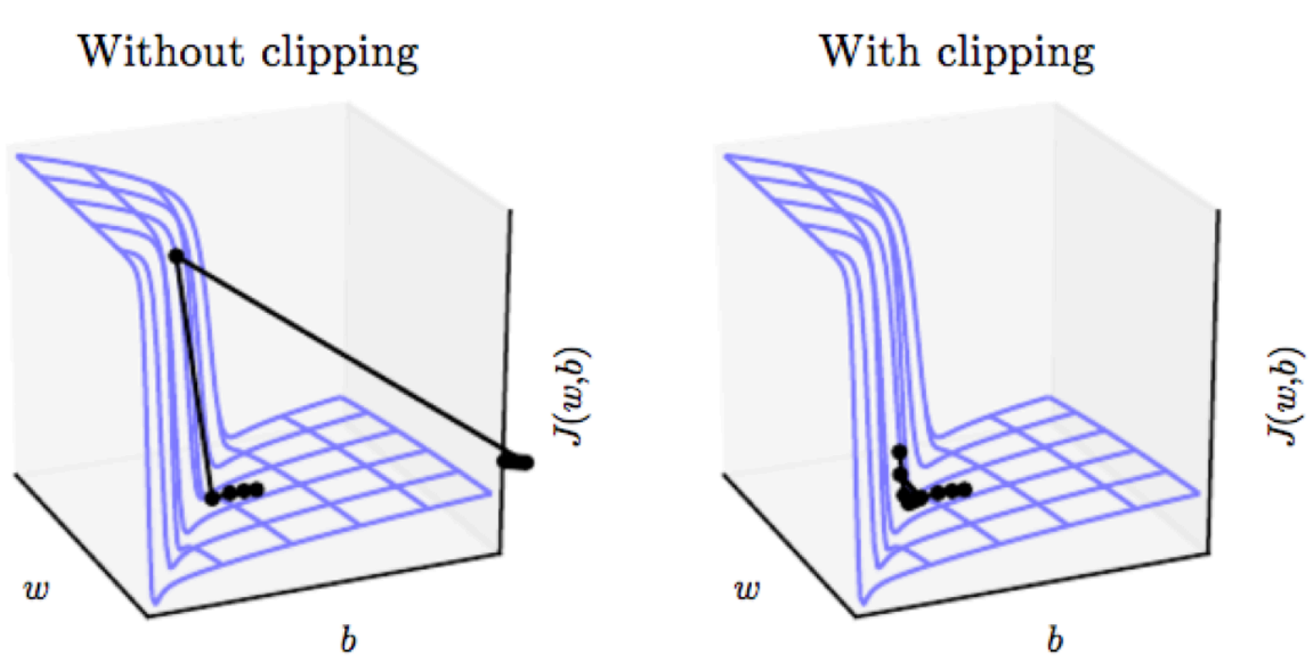
Keeping Things Stable

- One simple solution: gradient clipping
- Clip the gradient $\mathbf{g}$ so that it has a norm of at most η:

  if $\|\mathbf{g}\| > \eta$:

$$\mathbf{g} \leftarrow \frac{\eta \mathbf{g}}{\|\mathbf{g}\|}$$

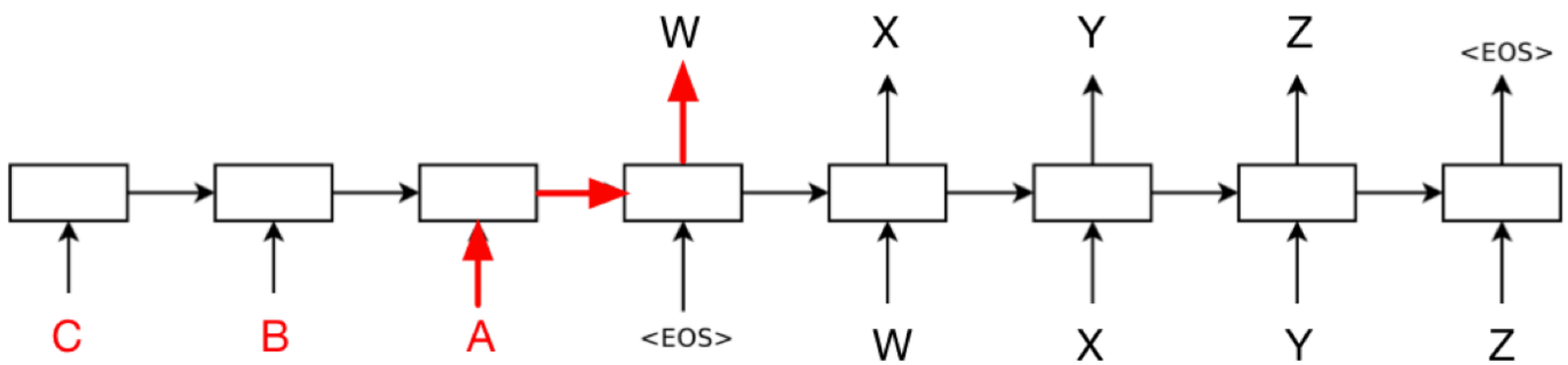
- The gradients are biased, but at least they don't blow up.



— Goodfellow et al., *Deep Learning*

Keeping Things Stable

- Another trick: reverse the input sequence.



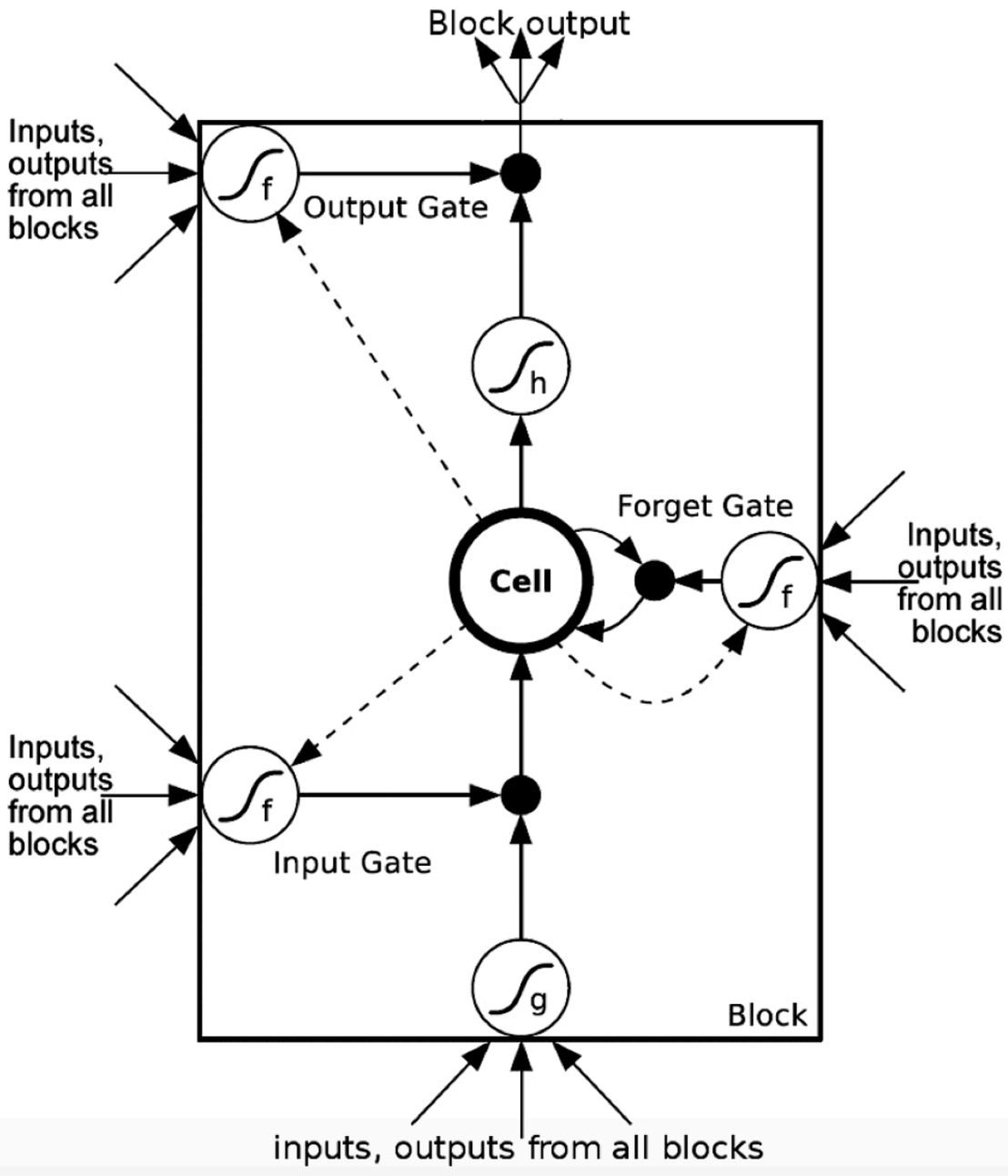
- This way, there's only one time step between the first word of the input and the first word of the output.
- The network can first learn short-term dependencies between early words in the sentence, and then long-term dependencies between later words.

Long-Term Short Term Memory

- Another architecture which makes it easy to remember information over long time periods is called Long-Term Short Term Memory (LSTM)
  - What's with the name? The idea is that a network's activations are its short-term memory and its weights are its long-term memory.
  - The LSTM architecture wants the short-term memory to last for a long time period.
- It's composed of memory cells which have controllers saying when to store or forget information.

Long-Term Short Term Memory

Replace each single unit in an RNN by a memory block -



$$c_{t+1} = c_t \cdot \text{forget gate} + \text{new input} \cdot \text{input gate}$$

- $i = 0, f = 1 \Rightarrow$ remember the previous value
- $i = 1, f = 1 \Rightarrow$ add to the previous value
- $i = 0, f = 0 \Rightarrow$ erase the value
- $i = 1, f = 0 \Rightarrow$ overwrite the value

Setting $i = 0, f = 1$ gives the reasonable "default" behavior of just remembering things.

Long-Term Short Term Memory

- In each step, we have a vector of memory cells $\mathbf{c}$, a vector of hidden units $\mathbf{h}$, and vectors of input, output, and forget gates $\mathbf{i}$, $\mathbf{o}$, and $\mathbf{f}$.
- There's a full set of connections from all the inputs and hiddens to the input and all of the gates:

$$\begin{pmatrix} \mathbf{i}_t \\ \mathbf{f}_t \\ \mathbf{o}_t \\ \mathbf{g}_t \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} \mathbf{W} \begin{pmatrix} \mathbf{y}_t \\ \mathbf{h}_{t-1} \end{pmatrix}$$

$$\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \mathbf{g}_t$$

$$\mathbf{h}_t = \mathbf{o}_t \circ \tanh(\mathbf{c}_t)$$

- Exercise: show that if $\mathbf{f}_{t+1} = 1$, $\mathbf{i}_{t+1} = 0$, and $\mathbf{o}_t = 0$, the gradients for the memory cell get passed through unmodified, i.e.

$$\overline{\mathbf{c}_t} = \overline{\mathbf{c}_{t+1}}.$$

Long-Term Short Term Memory

- Sound complicated? ML researchers thought so, so LSTMs were hardly used for about a decade after they were proposed.
- In 2013 and 2014, researchers used them to get impressive results on challenging and important problems like speech recognition and machine translation.
- Since then, they've been one of the most widely used RNN architectures.
- There have been many attempts to simplify the architecture, but nothing was conclusively shown to be simpler and better.
- You never have to think about the complexity, since frameworks like TensorFlow provide nice black box implementations.