CSC321 Neural Networks and Machine Learning

Lecture 5

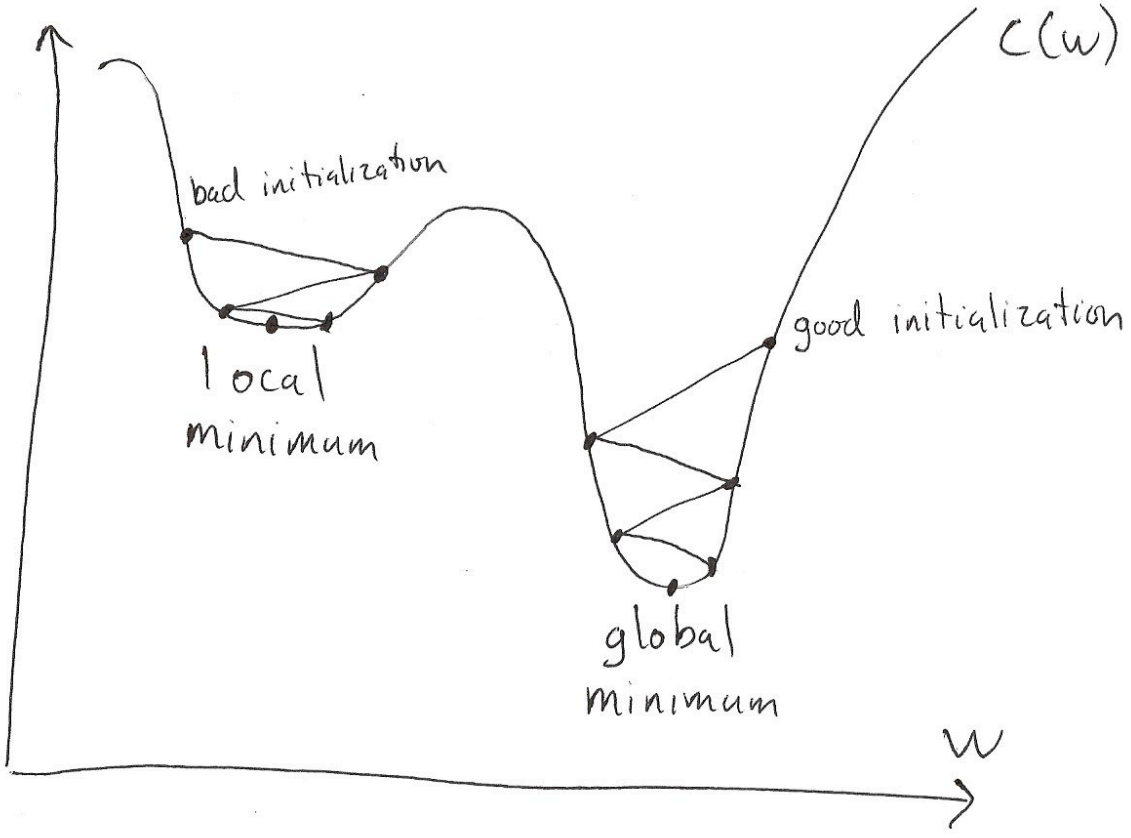
February 5, 2020

Agenda

- Optimization
- Bias-Variance Decomposition
- Distributed Representations (Project 2)

Optimization

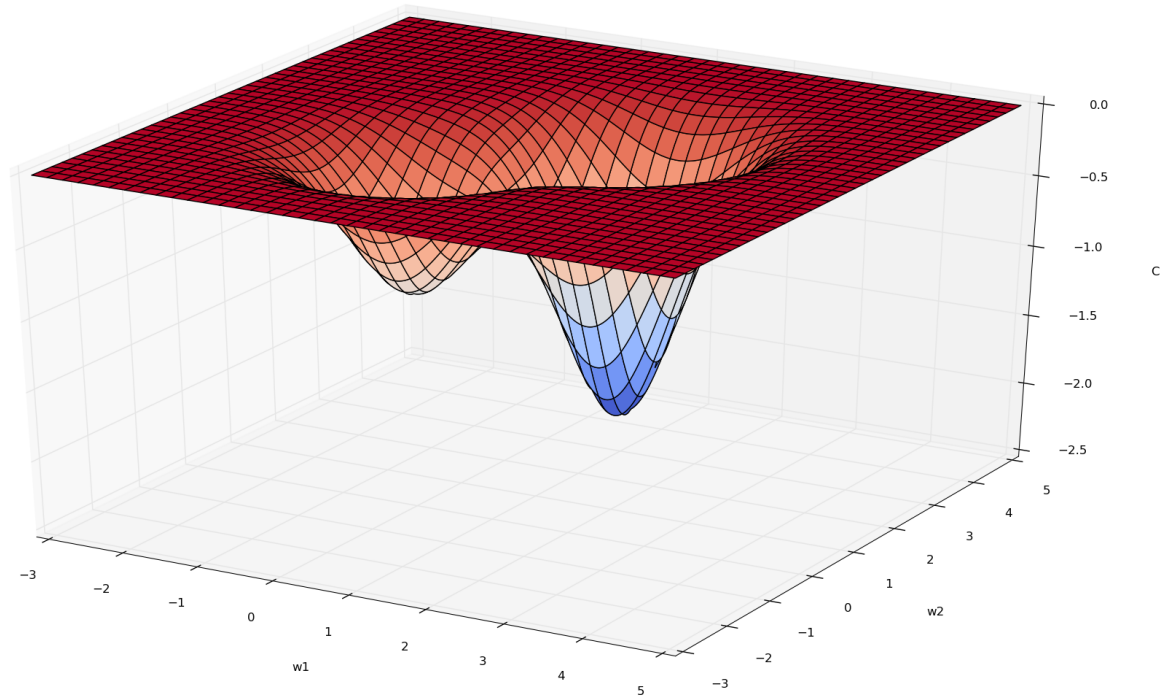
Gradient Descent (1D)



The regions where gradient descent converges to a particular local minimum are called **basins of attraction**.

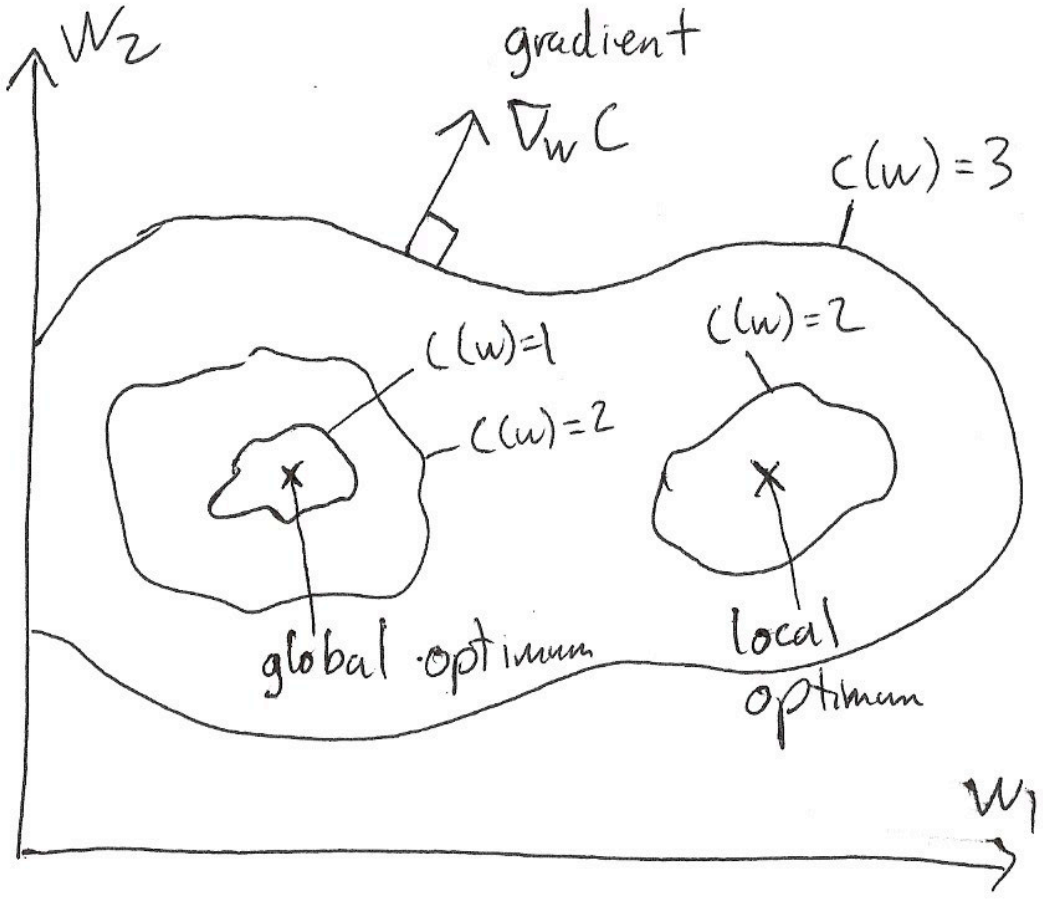
Optimization Problems in 2D

Visualizing two-dimensional optimization problems is trickier. Surface plots can be hard to interpret:



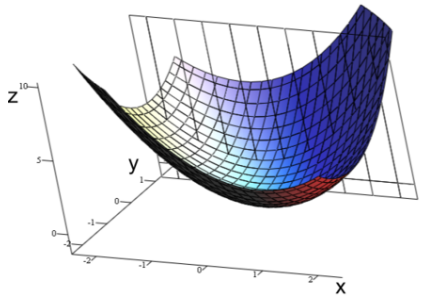
Let's group all the parameters (weights and biases) of a network into a single vector θ

Gradient Descent (2D)

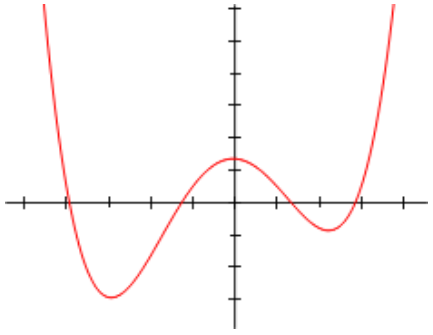


Contours: set of points where $\mathcal{E}(\theta)$ is constant.

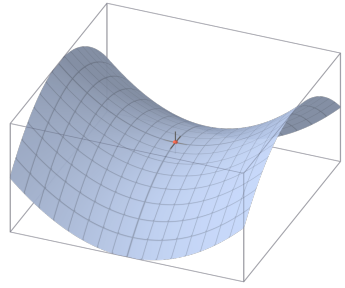
Features of the Optimization Landscape



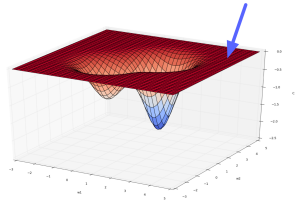
convex functions



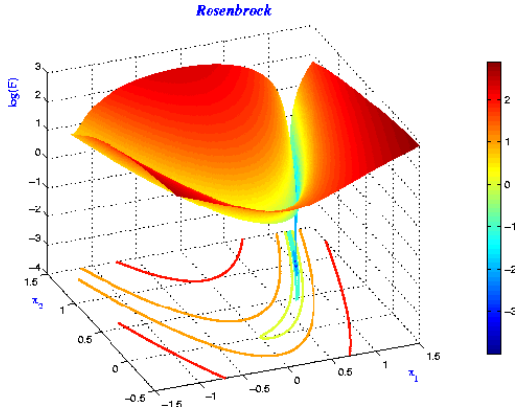
local minima



saddle points



plateaux



narrow ravines

Convexity of Linear Models

Linear regression and logistic regressions are **convex** problems—i.e. its loss function is convex.

A function f is convex if for any $a \in (0, 1)$

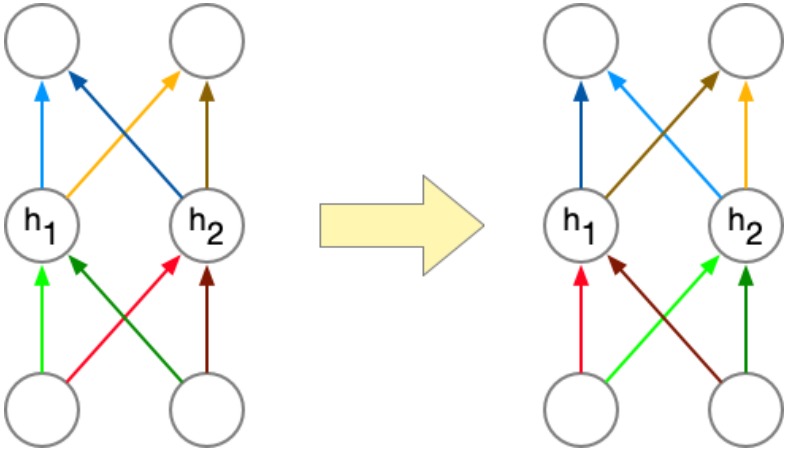
$$f(ax + (1 - a)y) < af(x) + (1 - a)f(y)$$

- The cost function only has one minima.
- There are no local minima that is not global minima.
- Intuitively: the cost function is "bowl-shaped".

Neural Networks are Not Convex

In general, neural networks are **not convex**.

One way to see this is that neural networks have **weight space symmetry**:



- Suppose you are at a local minima θ.
- You can swap two hidden units, and therefore swap the corresponding weights/biases, and get θ',
- then θ' must also be a local minima!

Local Minima in Neural Networks

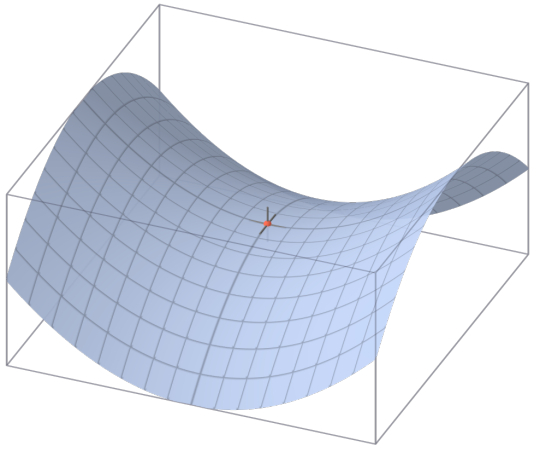
Even though any multilayer neural net can have local optima, we usually don't worry too much about them.

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.

Intuition pump: if you have enough randomly sampled hidden units, you can approximate any function just by adjusting the output layer.

- Then it's essentially a regression problem, which is convex.
- Hence, local optima can probably be fixed by adding more hidden units.
- Note: this argument hasn't been made rigorous.

Saddle Points



A saddle point has $\frac{\partial \mathcal{E}}{\partial \theta} = 0$, even though we are not at a minimum. (Minima with respect to some directions, maxima with respect to others.)

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.

Initialization

- If we initialize all weights/biases to the same value, (e.g. 0)
- . . . then all the hidden states in the same layer will have the same value, (e.g. $\mathbf{h}$ will be a vector containing the same value repeated)
- . . . then all of the error signals for weights in the same layer are the same. (e.g. each row of $\overline{W^{(2)}}$ will be identical)

$$
\begin{aligned}
\bar{\mathbf{y}} &= \overline{\mathcal{L}}(\mathbf{y} - \mathbf{t}) \\
\overline{W^{(2)}} &= \bar{\mathbf{y}}\mathbf{h}^T \\
\bar{\mathbf{h}} &= {W^{(2)}}^T \bar{\mathbf{y}} \\
\bar{\mathbf{z}} &= \bar{\mathbf{h}} \circ \sigma'(\mathbf{z}) \\
\overline{W^{(1)}} &= \bar{\mathbf{z}}\mathbf{x}^T
\end{aligned}
$$

Random Initialization

Solution: don't initialize all your weights to zero!

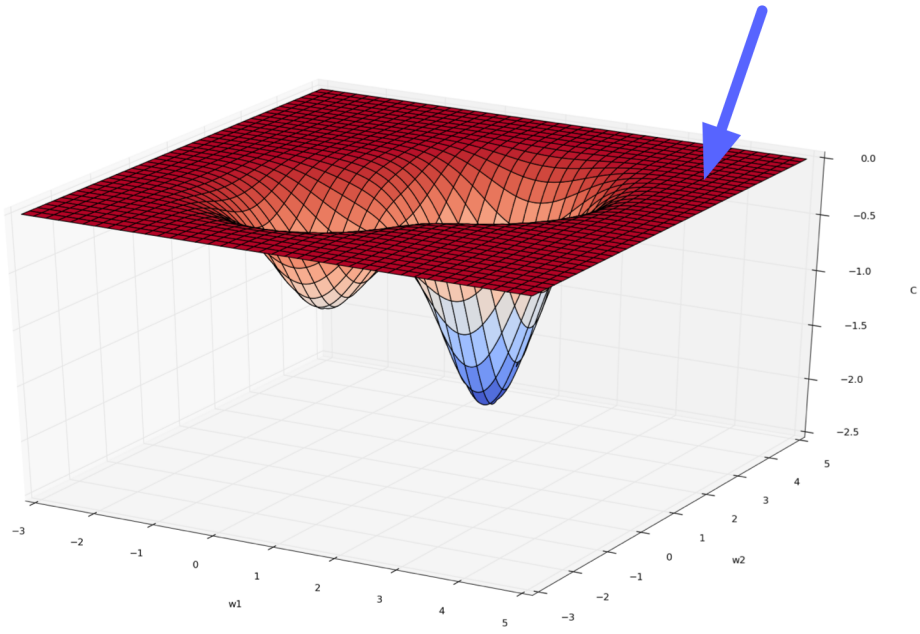
Instead, *break the symmetry* by using small random values.

In project 2, we initialize the weights by sampling from a random normal distribution with:

- Mean $= 0$
- Variance $= \frac{2}{\text{fan\_in}}$ where `fan_in` is the number of input neurons that feed into this feature. (He et al. 2015)

Plateaux

A flat region in the cost is called a **plateau**. (Plural: plateaux)



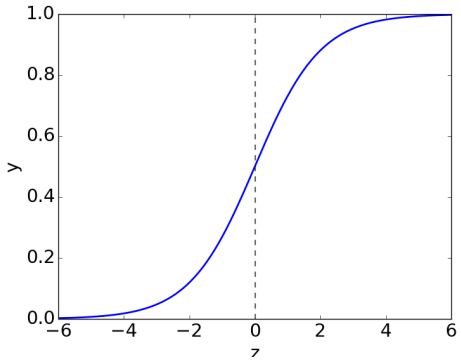
Can you think of examples?

- logistic activation with least squares
- 0-1 loss
- ReLU activation (potentially)

Plateaux and Saturated Units

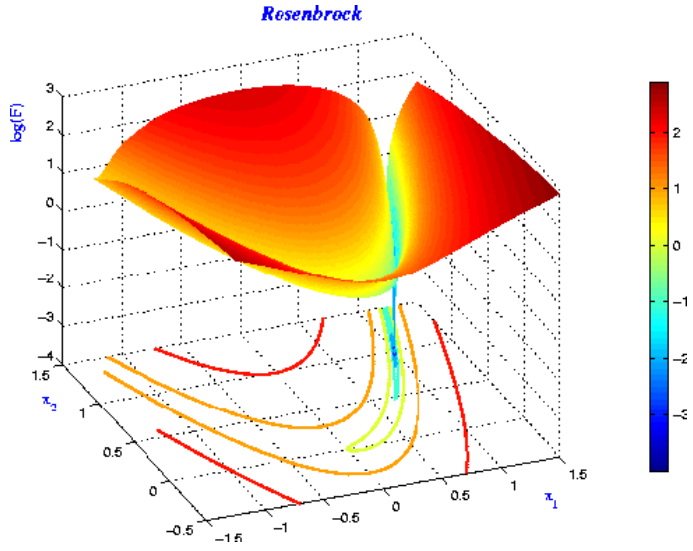
An important example of a plateau is a **saturated unit**. This is when activations always end up in the flat region of its activation function. Recall the backprop equation for the weight derivative:

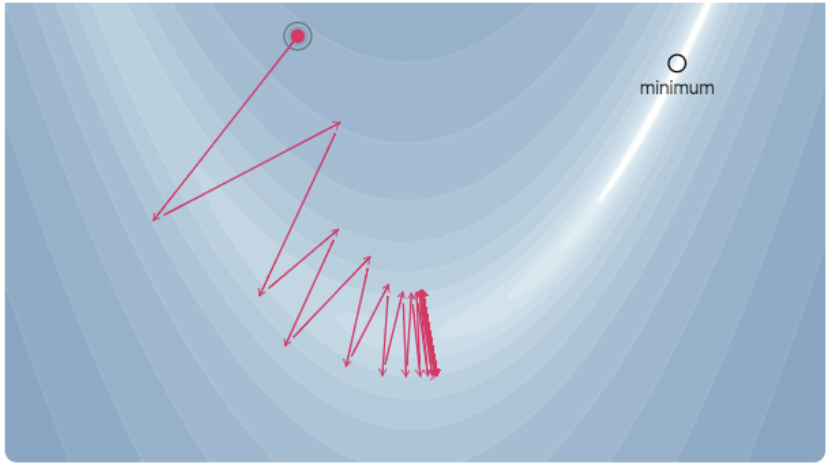
$$\overline{z_i} = \overline{h_i}\,\phi'(z)$$
$$\overline{w_{ij}} = \overline{z_i}\,x_j$$



- If $\phi'(z)$ 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 neuron a **dead unit**.

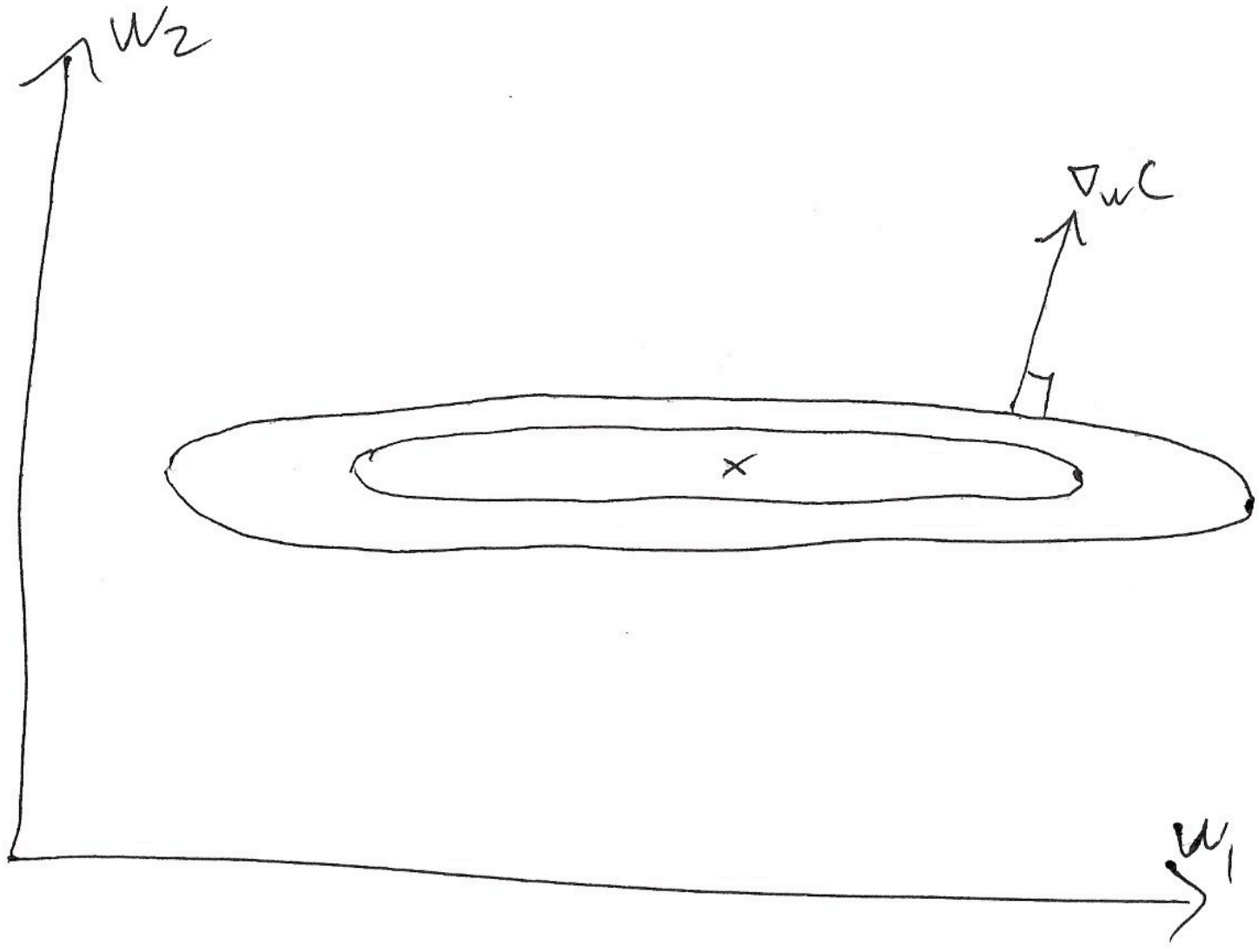
Ravines





Lots of sloshing around the walls, only a small derivative along the slope of the ravine's floor.

Ravines (2D Intuition)

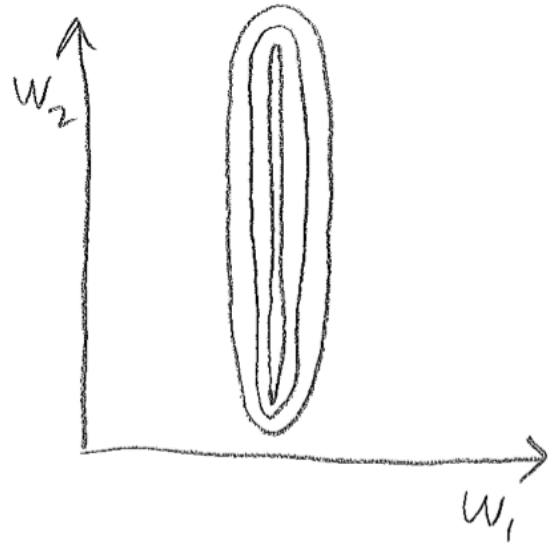


Ravines Example

Suppose we have the following dataset for linear regression.

x_1	x_2	t
114.8	0.00323	5.1
338.1	0.00183	3.2
98.8	0.00279	4.1
⋮	⋮	⋮

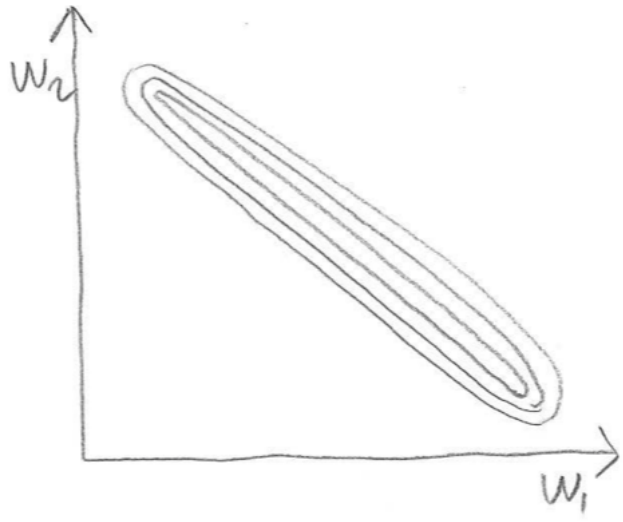
$$\overline{w_i} = \overline{y}\, x_i$$



- Which weight, w_1 or w_2, will receive a larger gradient descent update?
- Which one do you want to receive a larger update?
- Note: the figure vastly *understates* the narrowness of the ravine!

Ravines: another examples

x_1	x_2	t
1003.2	1005.1	3.3
1001.1	1008.2	4.8
998.3	1003.4	2.9
⋮	⋮	⋮



Avoiding Ravines

To avoid these problems, it's a good idea to **center** or **normalize** your inputs to zero mean and unit variance, especially when they're in arbitrary units (feet, seconds, etc.).

Hidden units may have non-centered activations, and this is harder to deal with.

A recent method called **batch normalization** explicitly centers each hidden activation. It often speeds up training by 1.5-2x.

Batch Normalization

Idea: Normalize the activations **per batch** during training, so that the activations have zero mean and unit variance.

What about during test time (i.e. during model evaluation)?

- Keep track of the activation mean μ and variance σ during training.
- Use that μ and σ at test time: $z' = \frac{z-\mu}{\sigma}$.

Batch Normalization in PyTorch (Project 2)

```
class PyTorchWordEmb(nn.Module):
  def __init__(self, emb_size=100, ...)
    super(PyTorchWordEmb, self).__init__()
    self.word_emb_layer = nn.Linear(vocab_size, ...)
    self.fc_layer1 = nn.Linear(emb_size * 3, num_hidden)
    self.bn = nn.BatchNorm1d(num_hidden)          # <----
    self.fc_layer2 = nn.Linear(num_hidden, 250)
    self.num_hidden = num_hidden
    self.emb_size = emb_size
  def forward(self, inp):
    embeddings = torch.relu(self.word_emb_layer(inp))
    embeddings = embeddings.reshape([-1, ...])
    hidden = torch.relu(self.fc_layer1(embeddings))
    hidden = self.bn(hidden)                      # <----
    return self.fc_layer2(hidden)
```

Differentiating model training vs evaluation

Since model behaviour is different during training vs evaluation, we need to annotate

`model.train()`

. . . before a forward pass during training, and. . .

`model.eval()`

. . . before a forward pass during evaluation.

(**Note**: You can use batch normalization for Project 2 if you would like. It should speed up training.)

Momentum

Momentum is a simple and highly effective method to deal with narrow ravines. Imagine a hockey puck on a frictionless surface (representing the cost function). It will accumulate momentum in the downhill direction:

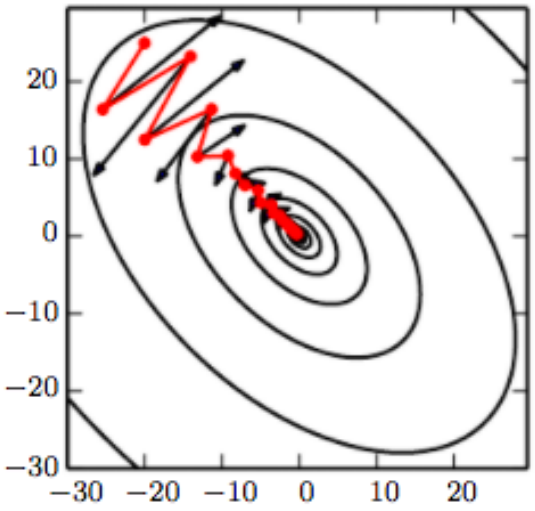
$$\mathbf{p} \leftarrow \mu\mathbf{p} - \alpha\frac{\partial \mathcal{E}}{\partial \theta}$$
$$\theta \leftarrow \theta + \mathbf{p}$$

- α is the learning rate, just like in gradient descent.
- μ is a damping parameter. It should be slightly less than 1 (e.g. 0.9 or 0.99).
  - If $\mu = 1$, conservation of energy implies it will never settle down.

(**Note**: You can use momentum for Project 2, numpy portion if you would like.)

Why Momentum Workds

- In the high curvature directions, the gradients cancel each other out, so momentum dampens the oscillations.
- In the low curvature directions, the gradients point in the same direction, allowing the parameters to pick up speed.



- If the gradient is constant (i.e. the cost surface is a plane), the parameters will reach a terminal velocity of $-\frac{\alpha}{1-\mu} \cdot \frac{\partial \mathcal{E}}{\partial \theta}$ This suggests if you increase μ, you should lower α to compensate.
- Momentum sometimes helps a lot, and almost never hurts.

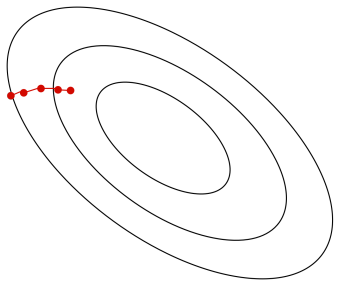
Second-Order Information

An area of research known as **second-order optimization** develops algorithms which explicitly use curvature information (second derivatives), but these are complicated and difficult to scale to large neural nets and large datasets.

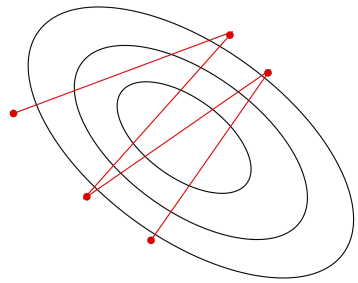
There is an optimization procedure called **Adam** which uses just a little bit of curvature information and often works much better than gradient descent. We will be using Adam in Project 2.

Learning Rate

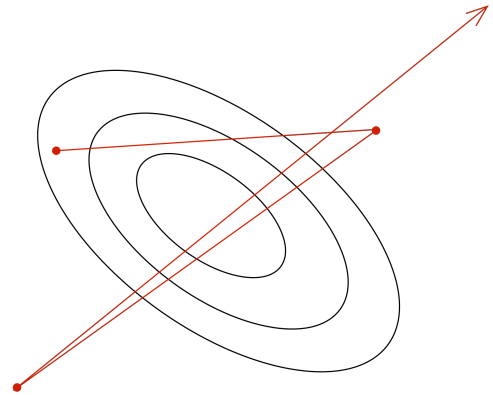
The learning rate α is a hyperparameter we need to tune. Here are the things that can go wrong in batch mode:



α too small:
slow progress



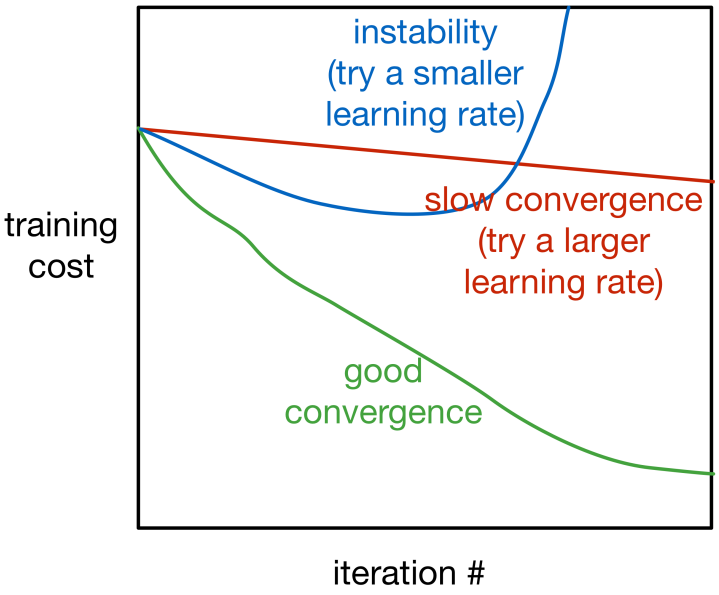
α too large:
oscillations



α much too large:
instability

Training Curve (or Learning Curve)

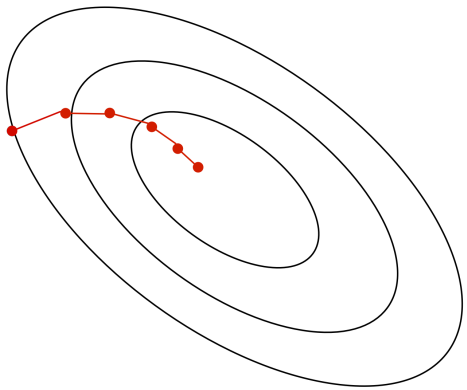
To diagnose optimization problems, it's useful to look at **learning curves**: plot the training cost as a function of iteration.



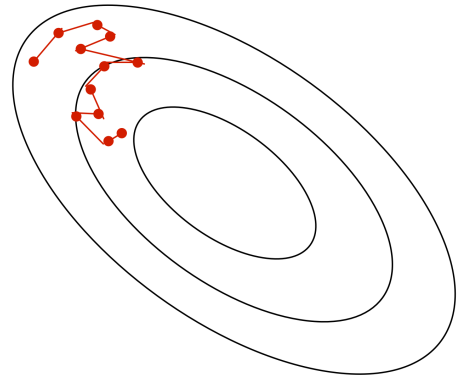
- **Note**: use a fixed subset of the training data to monitor the training error. Evaluating on a different batch (e.g. the current one) in each iteration adds a *lot* of noise to the curve!
- **Note**: it's very hard to tell from the training curves whether an optimizer has converged. They can reveal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

Stochastic Gradient Descent

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



batch gradient descent

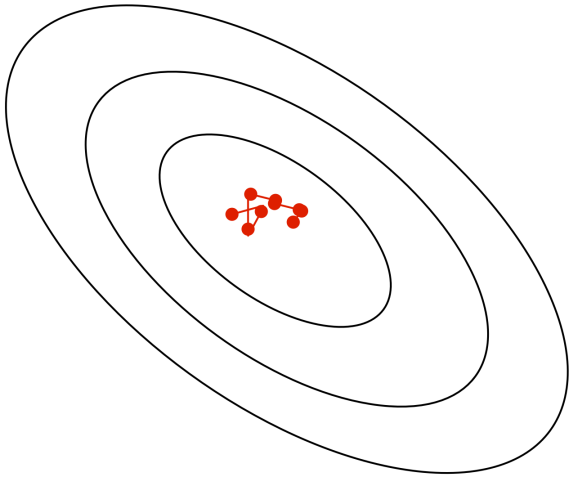


stochastic gradient descent

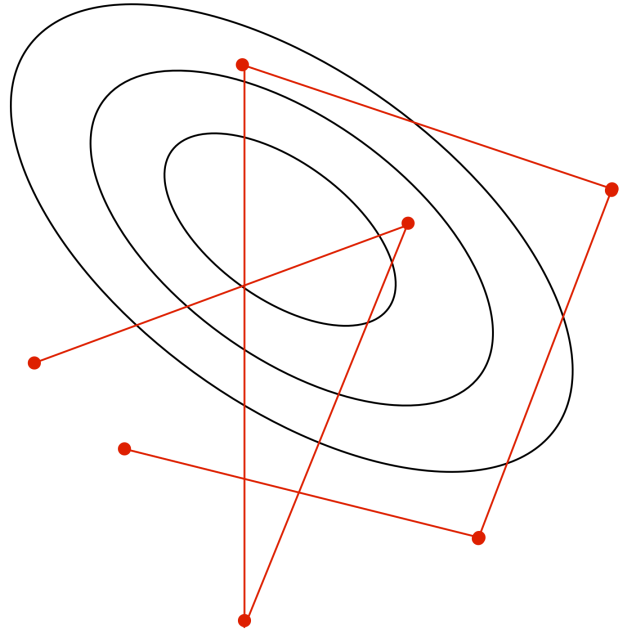
SGD Learning Rate

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

small learning rate



large learning rate

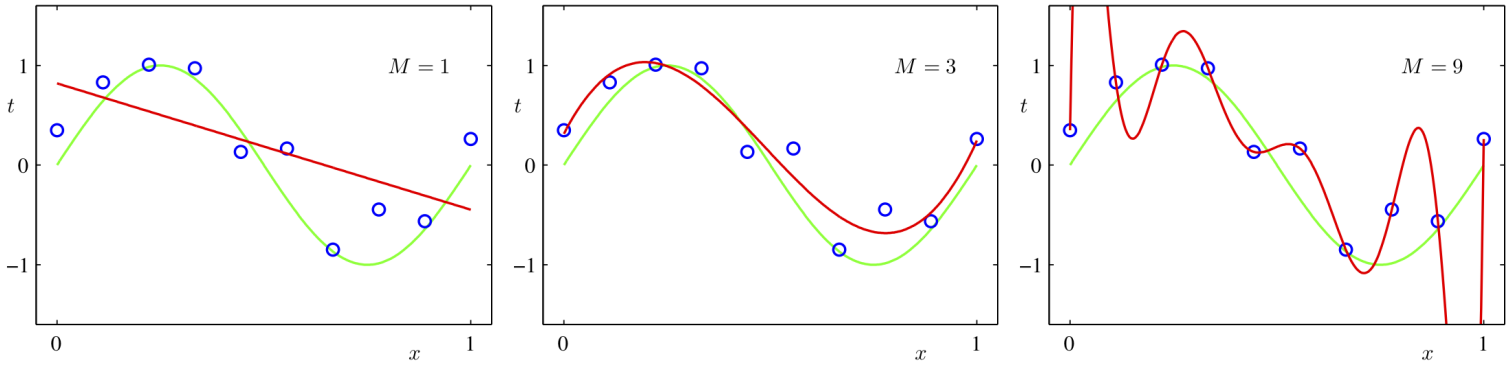


- Use a large learning rate early in training so you can get close to the optimum
- Gradually decay the learning rate to reduce the fluctuations

Bias-Variance Decomposition

Recall: Underfitting/Overfitting

We'd like to minimize the generalization error, i.e. error on novel examples



Let's systematically analyze **where error comes from**

Bias-Variance Decomposition (1)

Suppose our training and test data are sampled from a **data generating distribution** $p_{\mathcal{D}}(\mathbf{x}, t)$.

We are given an input $\mathbf{x}$ but not its corresponding t. We'd like to predict y to minimize the expected *square* loss: $\mathbb{E}_{p_{\mathcal{D}}}[(y-t)^2|\mathbf{x}]$

The best possible prediction we can make is the conditional expectation: $y_\star = \mathbb{E}_{p_{\mathcal{D}}}[t|\mathbf{x}]$

Proof:

$$\begin{aligned}
\mathbb{E}[(y-t)^2|\mathbf{x}] &= \mathbb{E}[y^2 - 2yt + t^2|\mathbf{x}] \\
&= y^2 - 2y\mathbb{E}[t|\mathbf{x}] + \mathbb{E}[t^2|\mathbf{x}] \\
&= y^2 - 2yy_\star + \mathbb{E}[t|\mathbf{x}]^2 + \mathrm{Var}[t|\mathbf{x}] \\
&= y^2 - 2yy_\star + y_\star^2 + \mathrm{Var}(t|\mathbf{x}) \\
&= (y-y_\star)^2 + \mathrm{Var}(t|\mathbf{x})
\end{aligned}$$

The term $\mathrm{Var}[t|\mathbf{x}]$ is called the **Bayes error**.

Bias-Variance Decomposition (2)

Now suppose we sample a training set D from the data generating distribution, and train a model $\hat{f} : \mathbf{x} \to y$ on the data, and use that model to make a prediction $y = \hat{f}(\mathbf{x})$ on test example $\mathbf{x}$.

Here, y is a random variable, since we get a different model $\hat{f}$ each time we sample a new training set.

We would like to minimize the **risk** $\mathbb{E}_{p_{\mathcal{D}}}[(y - t)^2 | \mathbf{x}]$. We can decompose this into **bias**, **variance**, and **Bayes error**.

We'll suppress the conditioning on $\mathbf{x}$ for clarity

Bias-Variance Decomposition (3)

$$\begin{aligned}
\mathbb{E}[(y - t)^2] &= \mathbb{E}[(y - y\star)^2] + \mathrm{Var}(t) \\
&= \mathbb{E}[y_\star^2 - 2yy_\star - y^2] + \mathrm{Var}(t) \\
&= y_\star^2 - 2y_\star\mathbb{E}[y] + \mathbb{E}[y^2] + \mathrm{Var}(t) \\
&= y_\star^2 - 2y_\star\mathbb{E}[y] + \mathbb{E}[y]^2 + \mathrm{Var}(y) + \mathrm{Var}(t) \\
&= (\mathbb{E}[y_\star - y])^2 + \mathrm{Var}(y) + \mathrm{Var}(t)
\end{aligned}$$

Or

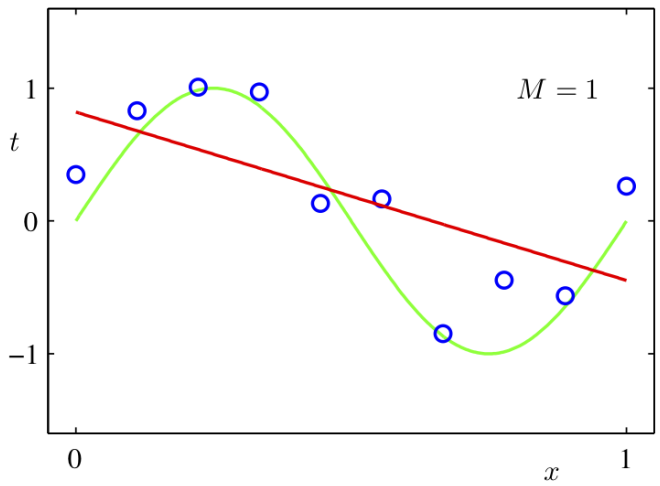
$$\mathbb{E}[(\hat{f}(\mathbf{x}) - t)^2] = \underbrace{(\mathbb{E}[y_\star - \hat{f}(\mathbf{x})])^2}_{\text{bias}} + \underbrace{\mathrm{Var}(\hat{f}(\mathbf{x}))}_{\text{variance}} + \underbrace{\mathrm{Var}(t)}_{\text{Bayes error}}$$

Terms in the Bias-Variance Decomposition

- $\mathbb{E}[(y - t)^2]$: the expected training error
- $\mathbb{E}[y_\star - \hat{f}(\mathbf{x})])^2$: the **bias**, or the squared average difference between the best possible prediction and the prediction given by $\hat{f}$
  - How badly will our model perform on average, across the possible datasets we could receive?
- $\mathrm{Var}(\hat{f}(\mathbf{x}))$
  - How the average prediction (across training sets) differs from the prediction we get from one particular training set.

High bias

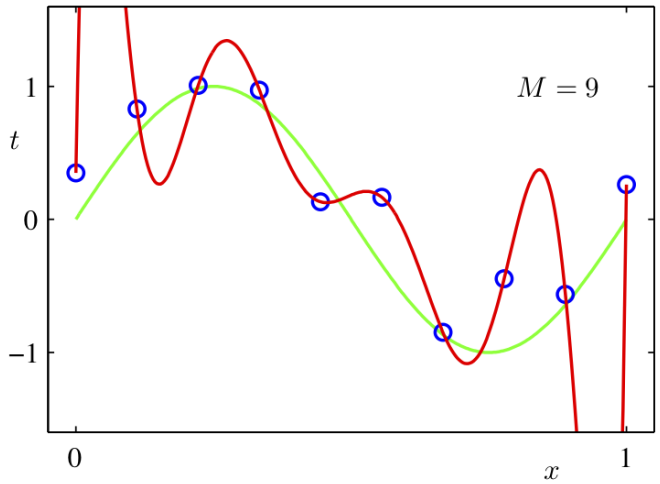
A **low-capacity** model has high bias



- bad performance on average, even across different training sets
- erroneous assumption in the model

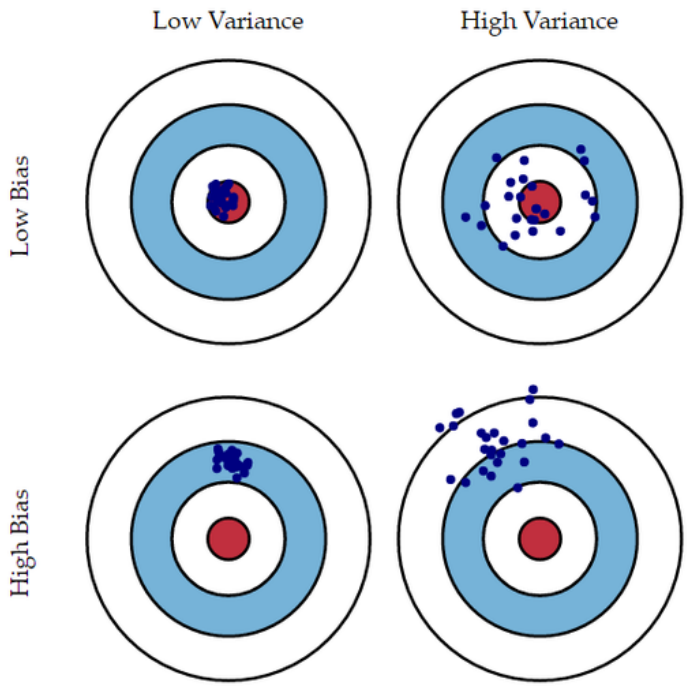
High variance

A **high-capacity** model can have high variance



- prediction differs widely across different training sets
- sensitive to small changes in training data, including accidentally regularities

Bias vs Variance



Distributed Representations

Feature Mapping

- Learning good representations is an important goal in machine learning
  - These representations are also called *feature mappings*, or *embeddings*
  - The representations we learn are often **reusable** for other tasks
  - Finding good representations is an **unsupervised learning** problem!
- Project 2:
  - Learn how to predict the next word in a sentence given the previous three (supervised learning)
  - Learn vector representations of *words* (unsupervised learning)

Language Modeling

A language model...

- Models the probability distribution of natural language text.
- Determine the **probability** $p(\mathbf{s})$ that a sequence of words (or a *sentence*) **s** occurs in text.

A language model gives us a way to compute $p(\mathbf{s})$

Why language models $p(\mathbf{s})$?

- Determine authorship:
  - build a language model $p(\mathbf{s})$ of Donald Trump's tweets
  - determine whether a new tweet is written by Trump
- Generate a machine learning paper (given a *corpus* of machine learning papers)
- Use as a *prior* for a speech recogntition system $p(\mathbf{s}|\mathbf{a})$, where $\mathbf{a}$ represents the observed speech signal.
  - An **observation model**, or likelihood, represented as $p(\mathbf{a}|\mathbf{s})$, 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. For example, "recognize speech" is more likely than "wreck a nice beach"
  - Use Bayes rule to infer a *posterior distribution* over sentences given the speech signal:

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

Training a Language Model

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 that our model assigns to the observed sentences. We assume the sentences are independent, so that their probabilities multiply.

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

Or minimize:

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

Since $p(\mathbf{s})$ is usually small, $-\log p(\mathbf{s})$ is reasonably sized, positive numbers

Probability of a sentence

A sentence is a sequence of words $w_1, w_2, \ldots, w_T$, so

$$\begin{aligned} p(\mathbf{s}) &= p(w_1, w_2, \ldots, w_T) \\ &= p(w_1)p(w_2|w_1)\ldots p(w_T|w_1, w_2, \ldots, w_{T-1}) \end{aligned}$$

We can make a simplifying **Markov assumption** that the distribution over the next word depends on the preceding few words.

In project 2, we use a context length of 3 and model:

$$p(w_t|w_1, w_2, \ldots, w_{t-1}) = p(w_t|w_{t-3}, w_{t-2}, w_{t-1})$$

This is a supervised learning problem!

N-Gram Language Model

A simple way of modeling $p(w_t|w_{t-2}, w_{t-1})$ is by constructing a table of conditional probabilities:

	cat	and	city	$\cdots$
the fat	0.21	0.003	0.01	
four score	0.0001	0.55	0.0001	$\cdots$
New York	0.002	0.0001	0.48	
$\vdots$		$\vdots$		

Where the probabilities come from the **empirical distribution**:

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

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 model!)

Example: Shakespeare N-Gram Language Model

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.

Figure 1: From https://lagunita.stanford.edu/c4x/Engineering/CS-224N/asset/slp4.pdf

Problems with N-Gram Language Model

- 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.

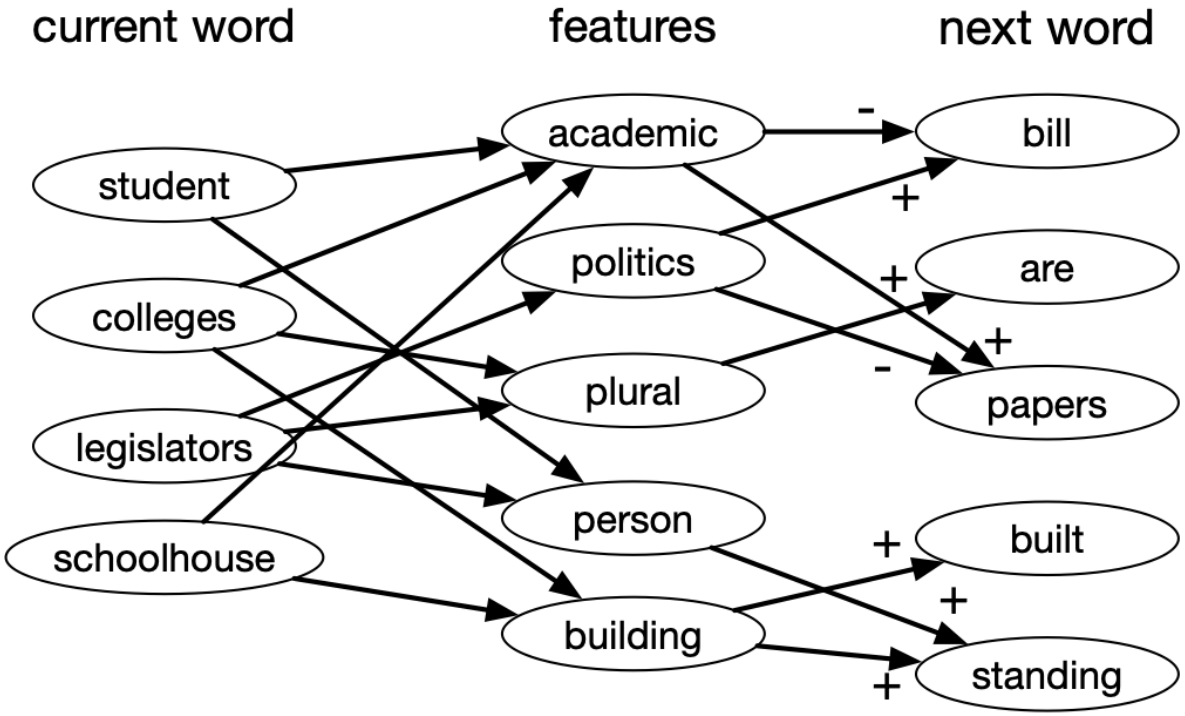
Ways to deal with data sparisty:

- 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 ns.

Localist vs Distributed Representations

Conditional probability tables are a kind of **localist representation**: all the information about a particular word is stored in one place: a column of the table.

But different words are related, so we ought to be able to share information between them.



Distributed Representations: 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

Idea:

1. use the **word attributes** to predict the next word.
2. learn the **word attributes** using an MLP with backpropagation

Sharing Information

Distributed representations allows us to share information between related words. E.g., suppose we've seen the sentence

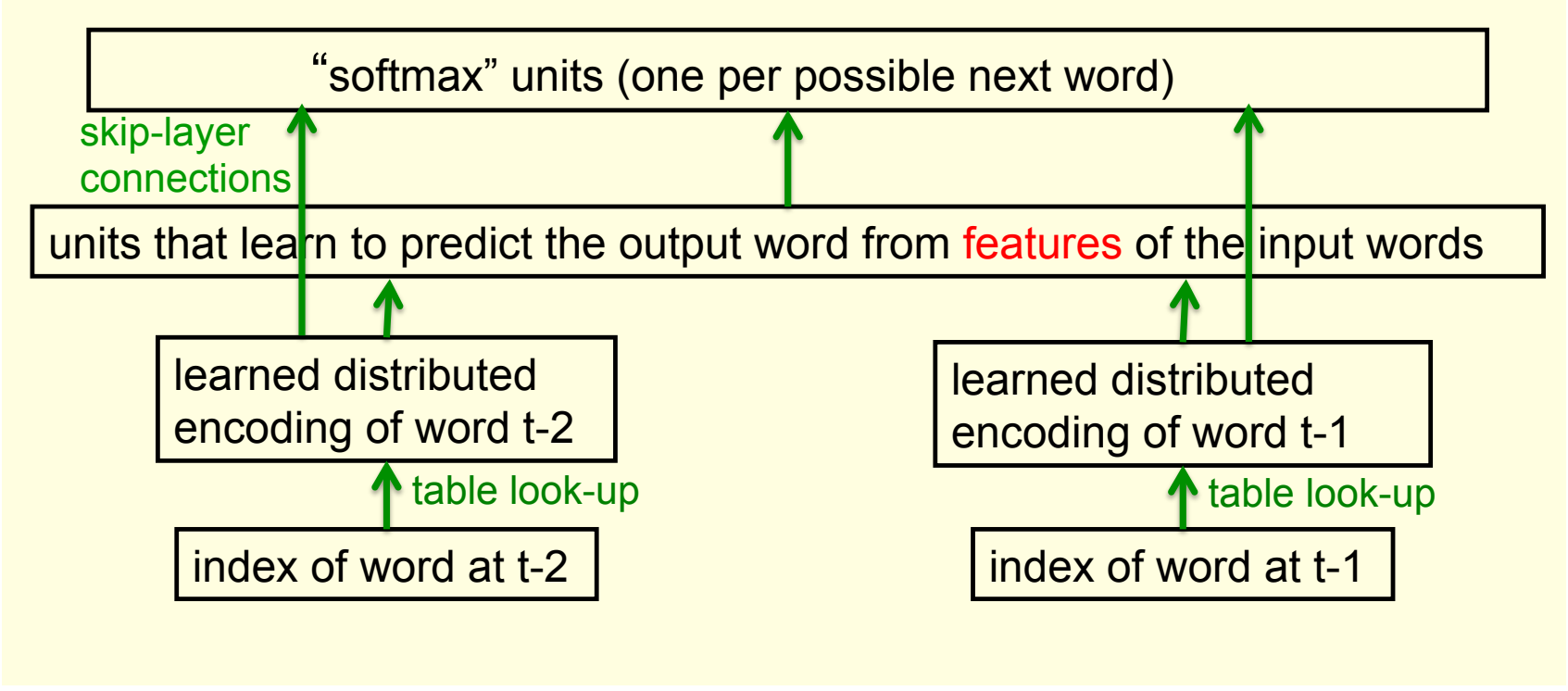
The cat got squashed in the garden on Friday.

This should help us predict the words in the sentence

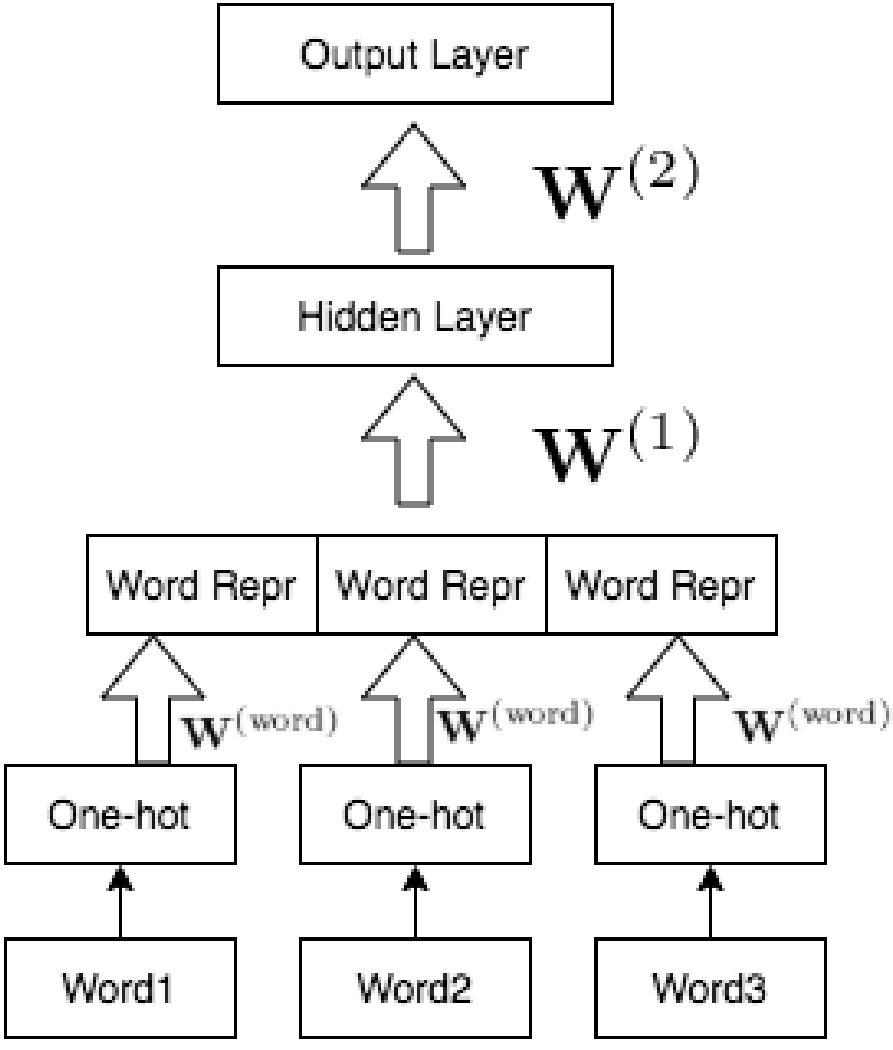
The dog got flattened in the yard on (???)

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

Neural Language Model

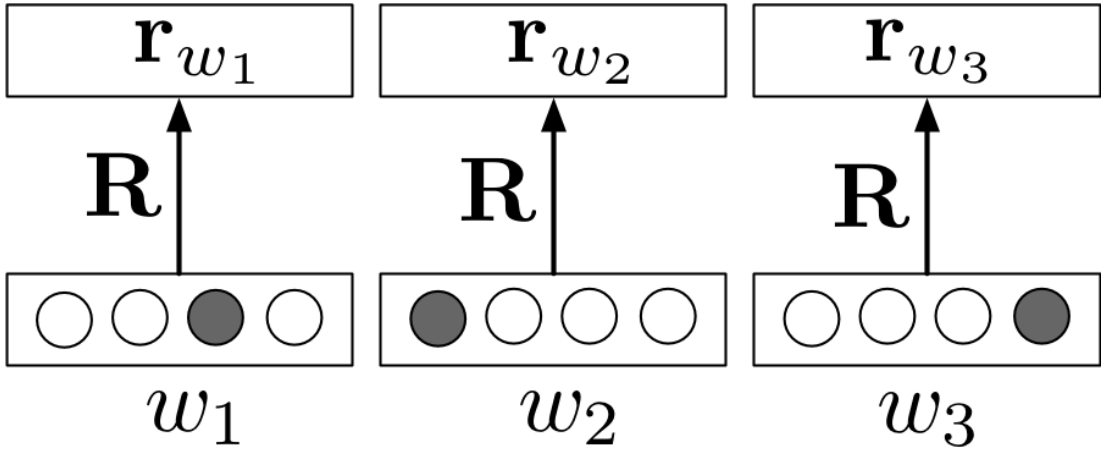


Neural Language Model (Project 2)



Word Representations

Since we are usign one-hot encodings for the words, the weight matrix of the word embedding layer acts like a lookup table.

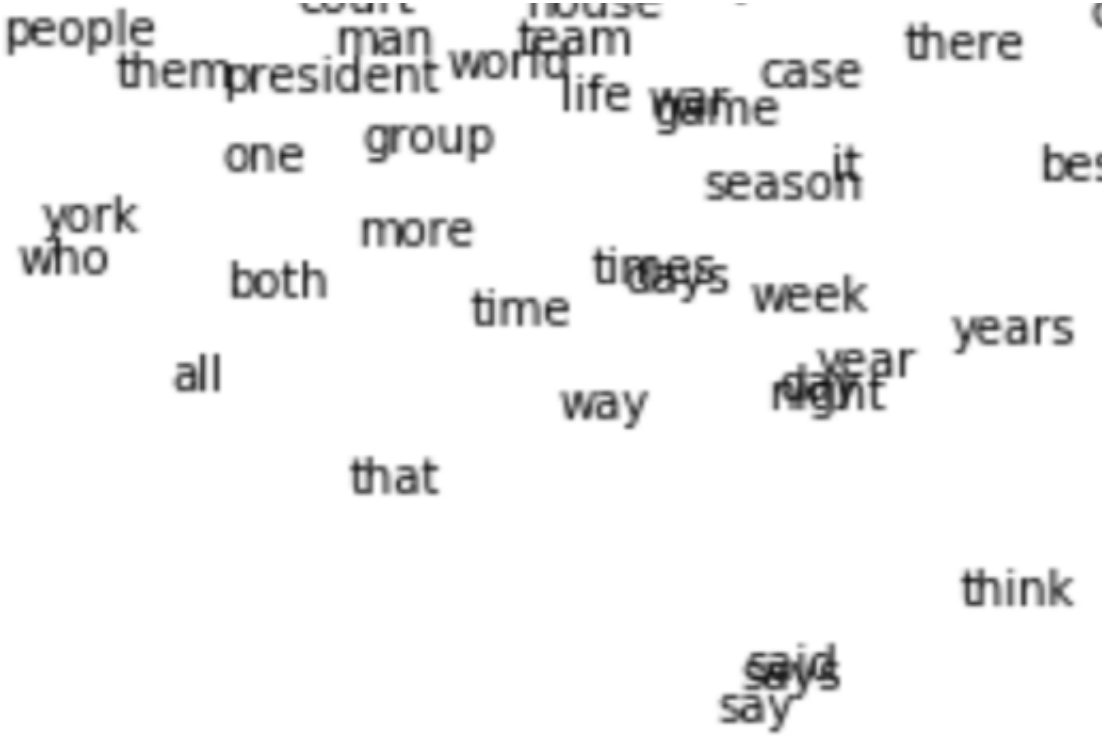


Terminology:

- "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).

What do word embeddings look like?

It's hard to visualize an n-dimensional space, but there are algorithms for mapping the embeddings to two dimensions.



In project 2, we use algorithm called tSNE, which tries to make distnaces in the 2-D embedding match the original high-dimensional distances as closely as possible.