CS 533: Natural Language Processing

From Log-Linear to Neural Language Models

Karl Stratos

Rutgers University

Agenda

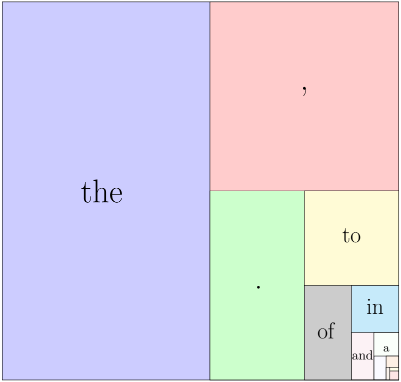
1. Loose ends (STOP symbol, Zipf's law)
2. Log-linear language models
   - Gradient descent
3. Neural language models
   - Feedforward
   - Recurrent

Zipf's Law

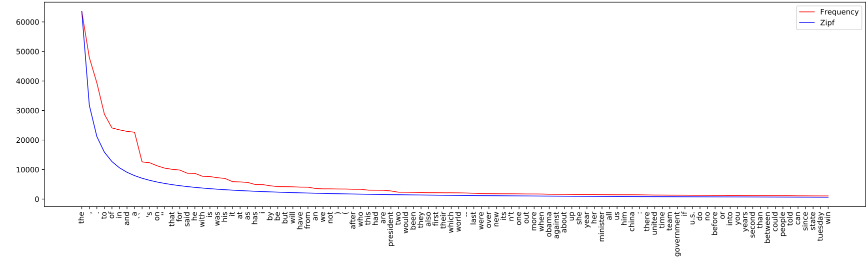
$w_1 \dots w_{|V|} \in V$ sorted in decreasing probability

$$p(w_i) = 2p(w_{i+1})$$

First four words: 93% of the unigram probability mass?



Zipf's Law: Empirical



Log-Linear Language Model

- Random variables: context x (e.g., previous n words), next word y
- Assumes a feature function $\phi(x, y) \in \{0, 1\}^d$
- Model parameter: weight vector $w \in \mathbb{R}^d$
- Model: for any (x, y)

$$q^{\phi,w}(y|x) = \frac{e^{w^\top \phi(x,y)}}{\sum_{y' \in V} e^{w^\top \phi(x,y')}}$$

- Model estimation: minimize cross entropy ($\equiv$ MLE)

$$w^* = \arg\min_{w \in \mathbb{R}^d} \mathop{\mathbf{E}}_{(x,y) \sim p_{XY}} \left[-\ln q^{\phi,w}(y|x) \right]$$

Example: Feature Extraction

Corpus:

- `the dog chased the cat`
- `the cat chased the mouse`
- `the mouse chased the dog`

Feature template

- $(x[-1], y)$
- $(x[-2], y)$
- $(x[-2], x[-1], y)$
- $(x[-1][: -2], y)$

How many features do we extract from the corpus (what is d)?

Example: Score of (x, y)

For any (x, y), its "score" given by parameter $w \in \mathbb{R}^d$ is

$$w^\top \phi(x, y) = \sum_{i=1:\ \phi_i(x,y)=1}^{d} w_i$$

Example: $x = \texttt{mouse chased}$

$$\begin{aligned} w^\top \phi(\texttt{mouse chased}, \texttt{the}) &= w_{\text{(-1)chased,the}} + w_{\text{(-2)mouse,the}} \\ &\quad + w_{\text{(-2)mouse(-1)chased,the}} + w_{\text{(-1:-2)ed,the}} \\ w^\top \phi(\texttt{mouse chased}, \color{red}{\texttt{chased}}) &= w_{\text{(-1)chased,}\color{red}{\text{chased}}} + w_{\text{(-2)mouse,}\color{red}{\text{chased}}} \\ &\quad + w_{\text{(-2)mouse(-1)chased,}\color{red}{\text{chased}}} + w_{\text{(-1:-2)ed,}\color{red}{\text{chased}}} \end{aligned}$$

Empirical Objective

$$\mathop{\mathbf{E}}_{(x,y)\sim p_{XY}}\left[-\ln q^{\phi,w}(y|x)\right]$$
$$\approx \frac{1}{N}\sum_{l=1}^{N} -\ln q^{\phi,w}(y^{(l)}|x^{(l)})$$
$$= \underbrace{\frac{1}{N}\sum_{l=1}^{N} \ln\left(\sum_{y\in V} e^{w^\top \phi(x^{(l)},y)}\right) - w^\top \phi(x^{(l)},y^{(l)})}_{J(w)}$$

When is $J(w)$ minimized?

Regularization

Ways to make sure w doesn't overfit training data

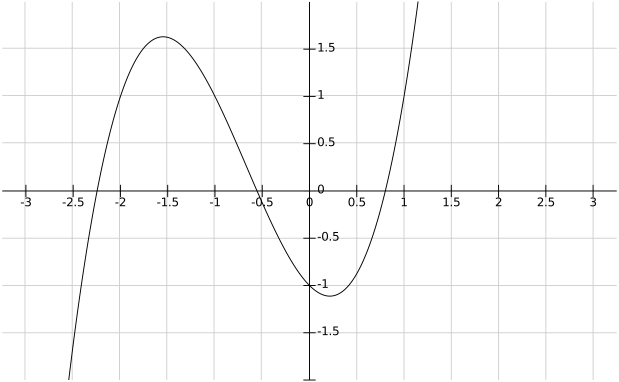
1. **Early stopping**: stop training when validation performance stops improving
2. Explicit regularization term

$$\min_{w \in \mathbb{R}^d} J(w) + \lambda \underbrace{\sum_{i=1}^{d} w_i^2}_{||w||_2^2} \quad \text{or} \quad \min_{w \in \mathbb{R}^d} J(w) + \lambda \underbrace{\sum_{i=1}^{d} |w_i|}_{||w||_1}$$

3. Other techniques (e.g., dropout)

Gradient Descent

Minimize $f(x) = x^3 + 2x^2 - x - 1$ over x



(Courtesy to FooPlot)

Local Search

Input: training objective $J(\theta) \in \mathbb{R}$, number of iterations T
Output: parameter $\hat{\theta} \in \mathbb{R}^d$ such that $J(\hat{\theta})$ is small

1. Initialize θ^0 (e.g., randomly).
2. For $t = 0 \ldots T - 1$,
   2.1 Obtain $\Delta^t \in \mathbb{R}^n$ such that $J(\theta^t + \Delta^t) \leq J(\theta^t)$.
   2.2 Choose some "step size" $\eta^t \in \mathbb{R}$.
   2.3 Set $\theta^{t+1} = \theta^t + \eta^t \Delta^t$.
3. Return θ^T.

What is a good Δ^t?

Gradient of the Objective at the Current Parameter

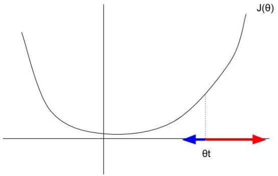
At $\theta^t \in \mathbb{R}^n$, the rate of increase (of the value of J) along a direction $u \in \mathbb{R}^n$ (i.e., $||u||_2 = 1$) is given by the **directional derivative**

$$\nabla_u J(\theta^t) := \lim_{\epsilon \to 0} \frac{J(\theta^t + \epsilon u) - J(\theta^t)}{\epsilon}$$

The **gradient** of J at θ^t is defined to be a vector $\nabla J(\theta^t)$ such that

$$\nabla_u J(\theta^t) = \nabla J(\theta^t) \cdot u \qquad\qquad \forall u \in \mathbb{R}^n$$

Therefore, the direction of the greatest rate of *decrease* is given by $-\nabla J(\theta^t) / \left|\left|\nabla J(\theta^t)\right|\right|_2$.



Gradient Descent

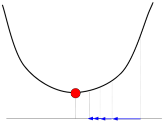
Input: training objective $J(\theta) \in \mathbb{R}$, number of iterations T
Output: parameter $\hat{\theta} \in \mathbb{R}^n$ such that $J(\hat{\theta})$ is small

1. Initialize θ^0 (e.g., randomly).
2. For $t = 0 \ldots T-1$,

$$\theta^{t+1} = \theta^t - \eta^t \nabla J(\theta^t)$$

3. Return θ^T.

When $J(\theta)$ is additionally *convex* (as in linear regression), gradient descent converges to an optimal solution (for appropriate step sizes).



Stochastic Gradient Descent for Log-Linear Model

Input: training objective

$$J(w) = \frac{1}{N} \sum_{l=1}^{N} J^{(l)}(w)$$

$$J^{(l)}(w) = \ln \left(\sum_{y \in V} e^{w^\top \phi(x^{(l)}, y)} \right) - w^\top \phi(x^{(l)}, y^{(l)})$$

number of iterations T ("epochs")

1. Initialize w^0 (e.g., randomly).
2. For $t = 0 \ldots T - 1$,
   2.1 For $l \in \text{shuffle}(\{1 \ldots N\})$,

$$w^{t+1} = w^t - \eta^t \nabla_w J^{(l)}(w^t)$$

3. Return w^T.

Gradient Derivation

Board

Summary of Gradient Descent

- **Gradient descent** is a local search algorithm that can be used to optimize *any* differentiable objective function.

- Stochastic gradient descent is the cornerstone of modern large-scale machine learning.

Word Vectors

- Instead of manually designing features ϕ, can we learn features themselves?
- Model parameter: now includes $E \in \mathbb{R}^{|V| \times d}$
  - $E_w \in \mathbb{R}^d$: **continuous dense representation** of word $w \in V$
- If we define $q(y|x)$ as a differentiable function of E, we learn E itself.

Simple Model?

- Parameters: $E \in \mathbb{R}^{|V| \times d}$, $W \in \mathbb{R}^{|V| \times 2d}$
- Model:

$$q^{E,W}(y|x) = \mathrm{softmax}_y \left(W \begin{bmatrix} E_{x[-1]} \\ E_{x[-2]} \end{bmatrix} \right)$$

- Model estimation: minimize cross entropy ($\equiv$ MLE)

$$E^*, W^* = \operatorname*{arg\,min}_{\substack{E \in \mathbb{R}^{|V| \times d} \\ W \in \mathbb{R}^{|V| \times 2d}}} \mathop{\mathbf{E}}_{(x,y) \sim p_{XY}} \left[-\ln q^{E,W}(y|x) \right]$$

Neural Network

Just a composition of linear/nonlinear functions.

$$f(x) = W^{(L)} \tanh\left(W^{(L-1)} \cdots \tanh\left(W^{(1)}x\right) \cdots\right)$$

More like a paradigm, not a specific model.

1. **Transform** your input $x \longrightarrow f(x)$.
2. Define **loss** between $f(x)$ and the target label y.
3. Train parameters by minimizing the loss.

You've Already Seen Some Neural Networks. . .

Log-linear model is a neural network with 0 hidden layer and a softmax output layer:

$$p(y|x) := \frac{\exp([Wx]_y)}{\sum_{y'} \exp([Wx]_{y'})} = \mathsf{softmax}_y(Wx)$$

Get W by minimizing $L(W) = -\sum_i \log p(y_i|x_i)$.

Linear regression is a neural network with 0 hidden layer and the identity output layer:

$$f(x) := Wx$$

Get W by minimizing $L(W) = \sum_i (y_i - f_i(x))^2$.

Feedforward Network

Think: log-linear with extra transformation

With 1 hidden layer:

$$h^{(1)} = \tanh(W^{(1)}x)$$
$$p(y|x) = \mathsf{softmax}_y(h^{(1)})$$

With 2 hidden layers:

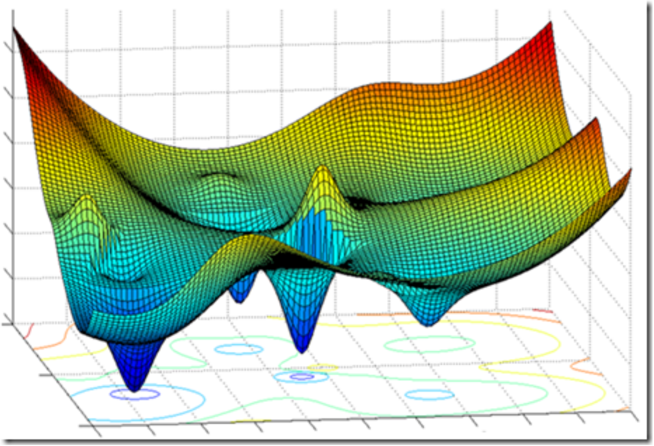
$$h^{(1)} = \tanh(W^{(1)}x)$$
$$h^{(2)} = \tanh(W^{(2)}h^{(1)})$$
$$p(y|x) = \mathsf{softmax}_y(h^{(2)})$$

Again, get parameters $W^{(l)}$ by minimizing $-\sum_i \log p(y_i|x_i)$.

- Q. What's the catch?

Training = Loss Minimization

We can decrease any continuous loss by following the gradient.



1. Differentiate the loss wrt. model parameters (backprop)
2. Take a gradient step

Backpropagation

- $J(\theta)$ any loss function differentiable with respect to $\theta \in \mathbb{R}^d$
- The gradient of J with respect to θ at some point $\theta' \in \mathbb{R}^d$

  $$\nabla_\theta J(\theta') \in \mathbb{R}^d$$

  can be calculated **automatically** by backpropagation.
- Note/code: http://karlstratos.com/notes/backprop.pdf

Bengio et al. (2003)

- Parameters: $E \in \mathbb{R}^{|V| \times d}$, $W \in \mathbb{R}^{d' \times nd}$, $V \in \mathbb{R}^{|V| \times d'}$

- Model:

$$q^{E,W,V}(y|x) = \operatorname{softmax}_y \left(V \tanh \left(W \begin{bmatrix} E_{x[-1]} \\ \vdots \\ E_{x[-n]} \end{bmatrix} \right) \right)$$

- Model estimation: minimize cross entropy ($\equiv$ MLE)

$$E^*, W^*, V^* = \operatorname*{arg\,min}_{\substack{E \in \mathbb{R}^{|V| \times d} \\ W \in \mathbb{R}^{d' \times nd} \\ V \in \mathbb{R}^{|V| \times d'}}} \mathop{\mathbf{E}}_{(x,y) \sim p_{XY}} \left[- \ln q^{E,W,V}(y|x) \right]$$

Bengio et al. (2003): Continued

	n	c	h	m	direct	mix	train.	valid.	test.
MLP1	5		50	60	yes	no	182	284	268
MLP2	5		50	60	yes	yes		275	257
MLP3	5		0	60	yes	no	201	327	310
MLP4	5		0	60	yes	yes		286	272
MLP5	5		50	30	yes	no	209	296	279
MLP6	5		50	30	yes	yes		273	259
MLP7	3		50	30	yes	no	210	309	293
MLP8	3		50	30	yes	yes		284	270
MLP9	5		100	30	no	no	175	280	276
MLP10	5		100	30	no	yes		265	**252**
Del. Int.	3						31	352	336
Kneser-Ney back-off	3							334	323
Kneser-Ney back-off	4							332	321
Kneser-Ney back-off	5							332	321
class-based back-off	3	150						348	334
class-based back-off	3	200						354	340
class-based back-off	3	500						326	**312**
class-based back-off	3	1000						335	319
class-based back-off	3	2000						343	326
class-based back-off	4	500						327	312
class-based back-off	5	500						327	312

Collobert and Weston (2008)

Nearest neighbors of trained word embeddings $E \in \mathbb{R}^{|V| \times d}$

FRANCE 454	JESUS 1973	XBOX 6909	REDDISH 11724	SCRATCHED 29869
SPAIN	CHRIST	PLAYSTATION	YELLOWISH	SMASHED
ITALY	GOD	DREAMCAST	GREENISH	RIPPED
RUSSIA	RESURRECTION	PSNUMBER	BROWNISH	BRUSHED
POLAND	PRAYER	SNES	BLUISH	HURLED
ENGLAND	YAHWEH	WII	CREAMY	GRABBED
DENMARK	JOSEPHUS	NES	WHITISH	TOSSED
GERMANY	MOSES	NINTENDO	BLACKISH	SQUEEZED
PORTUGAL	SIN	GAMECUBE	SILVERY	BLASTED
SWEDEN	HEAVEN	PSP	GREYISH	TANGLED
AUSTRIA	SALVATION	AMIGA	PALER	SLASHED

https://ronan.collobert.com/pub/matos/2008_nlp_icml.pdf

Neural Networks are (Finite-Sample) Universal Learners!

Theorem. (Zhang et al., 2016) Give me any

1. Set of n samples $S = \{\boldsymbol{x}^{(1)} \dots \boldsymbol{x}^{(n)}\} \subset \mathbb{R}^d$
2. Function $f : S \to \mathbb{R}$ that assigns some arbitrary value $f(\boldsymbol{x}^{(i)})$ to each $i = 1 \dots n$

Then I can specify a 1-hidden-layer feedforward network $C : S \to \mathbb{R}$ with $2n + d$ parameters such that $C(\boldsymbol{x}^{(i)}) = f(\boldsymbol{x}^{(i)})$ for all $i = 1 \dots n$.

Proof.

Define $C(\boldsymbol{x}) = \boldsymbol{w}^\top \text{relu}((\boldsymbol{a}^\top \boldsymbol{x} \dots \boldsymbol{a}^\top \boldsymbol{x}) + \boldsymbol{b})$ where $\boldsymbol{w}, \boldsymbol{b} \in \mathbb{R}^n$ and $\boldsymbol{a} \in \mathbb{R}^d$ are network parameters. Choose $\boldsymbol{a}, \boldsymbol{b}$ so that the matrix $A_{i,j} := [\max\{0, \boldsymbol{a}^\top \boldsymbol{x}^{(i)} - b_j\}]$ is triangular. Solve for $\boldsymbol{w}$ in

$$\begin{bmatrix} f(\boldsymbol{x}^{(1)}) \\ \vdots \\ f(\boldsymbol{x}^{(n)}) \end{bmatrix} = A\boldsymbol{w}$$

So Why Not Use a Simple Feedforward for Everything?

Computational reasons

- For example, using a giant feedforward to cover instances of different sizes is clearly inefficient.

Empirical reasons

- In principle, we can learn any function.
- This tells us nothing about *how to get there*. How many samples do we need? How can we find the right parameters?
- Specializing an architecture to a particular type of computation allows us to incorporate **inductive bias**.
- "Right" architecture is **absolutely critical** in practice.

Recurrent Neural Network (RNN)

Think: HMM (or Kalman filter) with extra transformation

Input: sequence $x_1 \dots x_N \in \mathbb{R}^d$

- For $i = 1 \dots N$,

$$h_i = \tanh\left(W x_i + V h_{i-1}\right)$$

Output: sequence $h_1 \dots h_N \in \mathbb{R}^{d'}$

RNN $\approx$ Deep Feedforward

Unroll the expression for the last output vector h_N:

$$h_N = \tanh\left(Wx_N + V\left(\cdots + V\tanh\left(Wx_1 + Vh_0\right)\cdots\right)\right)$$

It's just a deep "feedforward network" with one important difference: **parameters are reused**

- (V, W) are applied N times

Training: do backprop on this unrolled network, update parameters

LSTM

- RNN produces a sequence of output vectors

$$x_1 \dots x_N \quad \longrightarrow \quad h_1 \dots h_N$$

- LSTM produces "memory cell vectors" along with output

$$x_1 \dots x_N \quad \longrightarrow \quad c_1 \dots c_N, \; h_1 \dots h_N$$

- These $c_1 \dots c_N$ enable the network to keep or drop information from previous states.

LSTM: Details

At each time step i,

- Compute a *masking vector* for the memory cell:

$$q_i = \sigma\left(U^q x + V^q h_{i-1} + W^i c_{i-1}\right) \in [0,1]^{d'}$$

- Use q_i to keep/forget dimensions in previous memory cell:

$$c_i = (1 - q_i) \odot c_{i-1} + q_i \odot \tanh\left(U^c x + V^c h_{i-1}\right)$$

- Compute *another masking vector* for the output:

$$o_i = \sigma\left(U^o x + V^o h_{i-1} + W^o c_i\right) \in [0,1]^{d'}$$

- Use o_i to keep/forget dimensions in current memory cell:

$$h_i = o_i \odot \tanh(c_i)$$