Contents

Math and Machine Learning Basics

Contents

- For A^{-1} to exist, $Ax = b$ must have exactly one solution for every value of b. Determining whether a solution exists $\forall b \in \mathbb{R}^m$ means requiring that the **column space** (range) of A be all of $\mathbb{R}^m$. It is helpful to see Ax expanded out explicitly in this way:

  Unless stated otherwise, assume $A \in \mathbb{R}^{m \times n}$

$$\boldsymbol{A}\boldsymbol{x} = \sum_i x_i \boldsymbol{A}_{:,i} = x_1 \begin{pmatrix} A_{1,1} \\ \vdots \\ A_{m,1} \end{pmatrix} + \cdots + x_m \begin{pmatrix} A_{1,n} \\ \vdots \\ A_{m,n} \end{pmatrix} \tag{2.27}$$

  → Necessary: $\boldsymbol{A}$ must have at least m columns ($n \geq m$). ("wide").
  → Necessary *and* sufficient: matrix must contain at least one set of m linearly independent columns.
  → Invertibility: In addition to above, need matrix to be *square* (re: at most m columns $\wedge$ at least m columns).

- A square matrix with linearly dependent columns is known as **singular**. A (necessarily square) matrix is singular if and only if one or more eigenvalues are zero.

- A **norm** is any function f that satisfies the following properties:

  $||x||_\infty = \max_i |x_i|$

$$f(\boldsymbol{x}) = 0 \Rightarrow \boldsymbol{x} = \boldsymbol{0} \tag{1}$$
$$f(\boldsymbol{x} + \boldsymbol{y}) \leq f(\boldsymbol{x}) + f(\boldsymbol{y}) \tag{2}$$
$$\forall \alpha \in \mathbb{R},\ f(\alpha \boldsymbol{x}) = |\alpha| f(\boldsymbol{x}) \tag{3}$$

- An **orthogonal matrix** is a square matrix whose rows are mutually orthonormal and whose columns are mutually orthonormal:

  Note that orthonorm cols implies orthonorm rows (if square). To prove, consider the relationship between $\boldsymbol{A}^T\boldsymbol{A}$ and $\boldsymbol{A}\boldsymbol{A}^T$

$$\boldsymbol{A}^T\boldsymbol{A} = \boldsymbol{A}\boldsymbol{A}^T = \boldsymbol{I} \tag{2.37}$$
$$\boldsymbol{A}^{-1} = \boldsymbol{A}^T \tag{2.38}$$

- Suppose square matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ has n linearly independent eigenvectors $\{\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(n)}\}$. The **eigendecomposition** of $\boldsymbol{A}$ is then given by[1]

$$\boldsymbol{A} = \boldsymbol{V}\,\mathrm{diag}\,(\boldsymbol{\lambda})\ \boldsymbol{V}^{-1} \tag{2.40}$$

  In the special case where $\boldsymbol{A}$ is real-symmetric, $\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^T$. **Interpretation:** $\boldsymbol{A}\boldsymbol{x}$ can be decomposed into the following three steps:

  All real-symmetric $\boldsymbol{A}$ have an eigendecomposition, **but it might not be unique!**

[1] This appear to imply that unless the columns of V are also normalized, can't guarantee that its inverse equals its transpose? (since that is the only difference between it and an orthogonal matrix)

1) **Change of basis**: The vector $(\boldsymbol{Q}^T\boldsymbol{x})$ can be thought of as how $\boldsymbol{x}$ would appear in the basis of eigenvectors of $\boldsymbol{A}$.
2) **Scale**: Next, we scale each component $(\boldsymbol{Q}^T\boldsymbol{x})_i$ by an amount λ_i, yielding the new vector $(\boldsymbol{\Lambda}(\boldsymbol{Q}^T\boldsymbol{x}))$.
3) **Change of basis**: Finally, we rotate this new vector back from the eigen-basis into its original basis, yielding the transformed result of $\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^T\boldsymbol{x}$.

A common convention to sort the entries of Λ in descending order.

- **Positive definite**: all λ are positive; **positive semidefinite**: all λ are positive or zero.

  $\rightarrow$ PSD: $\forall \boldsymbol{x}, \quad \boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} \geq 0$
  $\rightarrow$ PD: $\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} = 0 \;\Rightarrow\; \boldsymbol{x} = \boldsymbol{0}$.[2]

- Any real matrix $\boldsymbol{A} \in \mathbb{R}^{m\times n}$ has a **singular value decomposition** of the form,

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}^T \tag{10}$$

  where both $\boldsymbol{U}$ and $\boldsymbol{V}$ are orthogonal matrices, and $\boldsymbol{D}$ is diagonal.

  – The **singular values** are the diagonal entries $\boldsymbol{D}_{ii}$.
  – The **left(right)-singular vectors** are the columns of $\boldsymbol{U}(\boldsymbol{V})$.
  – Eigenvectors of $\boldsymbol{A}\boldsymbol{A}^T$ are the L-S vectors. Eigenvectors of $\boldsymbol{A}^T\boldsymbol{A}$ are the R-S vectors. The eigenvalues of both $\boldsymbol{A}\boldsymbol{A}^T$ and $\boldsymbol{A}^T\boldsymbol{A}$ are given by the singular values squared.

$$\boldsymbol{U} \in \mathbb{R}^{m\times m} \tag{7}$$
$$\boldsymbol{D} \in \mathbb{R}^{m\times n} \tag{8}$$
$$\boldsymbol{V} \in \mathbb{R}^{n\times n} \tag{9}$$

- The Moore-Penrose **pseudoinverse**, denoted $\boldsymbol{A}^+$, enables us to find an "inverse" of sorts for a (possibly) non-square matrix $\boldsymbol{A}$. Most algorithms compute $\boldsymbol{A}^+$ via

$$\boldsymbol{A}^+ = \boldsymbol{V}\boldsymbol{D}^+\boldsymbol{U}^T \tag{11}$$

$\boldsymbol{A}^+$ is useful, e.g., when we want to solve $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{y}$ by left-multiplying each side to obtain $\boldsymbol{x} = \boldsymbol{B}\boldsymbol{y}$. It is far more likely for solution(s) to exist when $\boldsymbol{A}$ is wider than it is tall.

- The **determinant** of a matrix is $\det(\boldsymbol{A}) = \prod_i \lambda_i$. Conceptually, $|\det(\boldsymbol{A})|$ tells how much [multiplication by] $\boldsymbol{A}$ expands/contracts space. If $\det(\boldsymbol{A}) = 1$, the transformation preserves volume.

[2] I proved this and it made me happy inside. Check it out. Let $\boldsymbol{A}$ be positive definite. Then

$$\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} = \boldsymbol{x}^T\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^T\boldsymbol{x} \tag{4}$$
$$= \sum_i (\boldsymbol{Q}^T\boldsymbol{x})_i \lambda_i (\boldsymbol{Q}^T\boldsymbol{x})_i \tag{5}$$
$$= \sum_i \lambda_i (\boldsymbol{Q}^T\boldsymbol{x})_i^2 \tag{6}$$

Since all terms in the summation are non-negative and all $\lambda_i > 0$, we have that $\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} = 0$ if and only if $(\boldsymbol{Q}^T\boldsymbol{x})_i = 0 = \boldsymbol{q}^{(i)} \cdot \boldsymbol{x}$ for all i. Since the set of eigenvectors $\{\boldsymbol{q}^{(i)}\}$ form an orthonormal basis, we have that $\boldsymbol{x}$ must be the zero vector.

Example: Principal Component Analysis

Task. Say we want to apply lossy compression (less memory, but may lose precision) to a collection of m points $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$. We will do this by converting each $\boldsymbol{x}^{(i)} \in \mathbb{R}^n$ to some $\boldsymbol{c}^{(i)} \in \mathbb{R}^l$ $(l < n)$, i.e. finding functions f and g such that:

$$f(\boldsymbol{x}) = \boldsymbol{c} \quad \text{and} \quad \boldsymbol{x} \approx g(f(\boldsymbol{x})) \tag{12}$$

Decoding function (g). As is, we still have a rather general task to solve. *PCA* is defined by choosing $g(\boldsymbol{c}) = \boldsymbol{D}\boldsymbol{c}$, with $\boldsymbol{D} \in \mathbb{R}^{n \times l}$, where all columns of $\boldsymbol{D}$ are both (1) orthogonal and (2) unit norm.

Encoding function (f). Now we need a way of mapping $\boldsymbol{x}$ to $\boldsymbol{c}$ such that $g(\boldsymbol{c})$ will give us back a vector optimally close to $\boldsymbol{x}$. We've already defined g, so this amount to finding the optimal $\boldsymbol{c}*$ such that:

$$\boldsymbol{c}* = \arg\min_{\boldsymbol{c}} ||\boldsymbol{x} - g(\boldsymbol{c})||_2^2 \tag{13}$$

$$(\boldsymbol{x} - g(\boldsymbol{c}))^T(\boldsymbol{x} - g(\boldsymbol{c})) = \boldsymbol{x}^T\boldsymbol{x} - 2\boldsymbol{x}^T g(\boldsymbol{c}) + g(\boldsymbol{c})^T g(\boldsymbol{c}) \tag{14}$$

$$\boldsymbol{c}* = \arg\min_{\boldsymbol{c}} \left[-2\boldsymbol{x}^T\boldsymbol{D}\boldsymbol{c} + \boldsymbol{c}^T\boldsymbol{c}\right] \tag{15}$$

$$= \boldsymbol{D}^T\boldsymbol{x} = f(\boldsymbol{x}) \tag{16}$$

which means the PCA *reconstruction operation* is defined as $r(\boldsymbol{x}) = \boldsymbol{D}\boldsymbol{D}^T\boldsymbol{x}$.

Optimal $\boldsymbol{D}$. It is important to notice that we've been able to determine e.g. the optimal $\boldsymbol{c}*$ *for some* $\boldsymbol{x}$ because each $\boldsymbol{x}$ has a (allowably) different $\boldsymbol{c}*$. However, we use *the same* matrix $\boldsymbol{D}$ for all our samples $\boldsymbol{x}^{(i)}$, and thus must optimize it over all points in our collection. With that out of the way, we just do what we always do: minimize over the L^2 distance between points and their reconstruction. Formally, we minimize the Frobenius norm of the matrix of errors:

$$\boldsymbol{D}* = \arg\min_{\boldsymbol{D}} \sqrt{\sum_{i,j} \left(\boldsymbol{x}_j^{(i)} - r(\boldsymbol{x}^{(i)})_j\right)^2} \quad s.t. \quad \boldsymbol{D}^T\boldsymbol{D} = \boldsymbol{I} \tag{17}$$

Consider the case of $l = 1$ which means $\boldsymbol{D} = \boldsymbol{d} \in \mathbb{R}^n$. In this case, after [insert math here], we obtain

$$\boldsymbol{d}* = \arg\max_{\boldsymbol{d}} Tr\left(\boldsymbol{d}^T \boldsymbol{X}^T \boldsymbol{X} \boldsymbol{d}\right) \quad s.t. \quad \boldsymbol{d}^T \boldsymbol{d} = 1 \tag{18}$$

where, as usual, $\boldsymbol{X} \in \mathbb{R}^{m,n}$. It should be clear that the optimal $\boldsymbol{d}$ is just the largest eigenvector of $\boldsymbol{X}^T \boldsymbol{X}$.

Expectation. For some function $f(x)$, $\mathbb{E}_{x\sim P}[f(x)]$ is the mean value that f takes on when x is drawn from P. The formula for discrete and continuous variables, respectively is as follows:

$$\mathbb{E}_{x\sim P}[f(x)] = \sum_x P(x)f(x) \tag{3.9}$$

$$\mathbb{E}_{x\sim P}[f(x)] = \int p(x)f(x)\mathrm{d}x \tag{3.10}$$

Variance. A measure of how much the values of a function of a random variable x vary as we sample different values of x from its distribution.

$$\mathrm{Var}[f(x)] = \mathbb{E}\left[(f(x) - \mathbb{E}[f(x)])^2\right] \tag{3.11}$$

Covariance. Gives some sense of how much two values are *linearly* related to each other, as well as the *scale* of these variables.

$$\mathrm{Cov}[f(x),\ g(x)] = \mathbb{E}[\ (f(x) - \mathbb{E}[f(x)])\ \ (g(x) - \mathbb{E}[g(x)])\ ] \tag{3.13}$$

→ Large $|\mathrm{Cov}[f,\ g]|$ means the function values change a lot and both functions are far from their means at the same time.

→ **Correlation** normalizes the contribution of each variable in order to measure only how much the variables are related.

Covariance Matrix of a random vector $\mathbf{x} \in \mathbb{R}^n$ is an $n \times n$ matrix, such that

$$\mathrm{Cov}[\mathbf{x}]_{i,j} = \mathrm{Cov}[x_i,\ x_j] \tag{3.14}$$

and if we want the "sample" covariance matrix taken over m data point samples, then

$$\Sigma := \frac{1}{m}\sum_{k=1}^{m}(x_k - \bar{x})(x_k - \bar{x})^T \tag{19}$$

where m is the number of data points.

Measure Theory.

- A set of points that is negligibly small is said to have **measure zero**. In practical terms, think of such a set as occupying no volume in the space we are measuring (interested in).
- A property that holds **almost everywhere** holds throughout all space except for on a set of measure zero.

In $\mathbb{R}^2$, a line has measure zero.

Functions of RVs.

- **Common mistake:** Suppose $\mathbf{y} = g(\mathbf{x})$, and g is invertible/continuous/differentiable. It is NOT true that $p_y(\mathbf{y}) = p_x(g^{-1}(\mathbf{y}))$. This fails to account for the distortion of [probability] space introduced by g. Rather,

$$p_x(\mathbf{x}) = p_y(g(\mathbf{x}))\left|\frac{\partial g(\mathbf{x})}{\partial \mathbf{x}}\right| \tag{3.47}$$

Information Theory. Denote the **self-information** of an event $\mathbf{x} = x$ to be

$$I(x) \triangleq -\log P(x) \tag{20}$$

where log is always assumed to be the natural logarithm. We can quantify the amount of uncertainty in an entire probability distribution using the **Shannon entropy**,

$$H(\mathbf{x}) = \mathbb{E}_{\mathbf{x}\sim P}\left[I(x)\right] = -\mathbb{E}_{\mathbf{x}\sim P}\left[\log P(x)\right] \tag{21}$$

which gives the expected amount of information in an event drawn from that distribution. Taking it a step further, say we have two separate probability distributions $P(\mathbf{x})$ and $Q(\mathbf{x})$. We can measure how different these distributions are with the **Kullback-Leibler (KL) divergence**:

$$D_{KL}\left(P||Q\right) \triangleq \mathbb{E}_{\mathbf{x}\sim P}\left[\log \frac{P(x)}{Q(x)}\right] = \mathbb{E}_{\mathbf{x}\sim P}\left[\log P(x) - \log Q(x)\right] \tag{22}$$

Note that the expectation is taken over P, thus making D_{KL} not symmetric (and thus not a true distance measure), since $D_{KL}(P||Q) \neq D_{KL}(Q||P)$. Finally, a closely related quantity is the **cross-entropy**, $H(P, Q)$, defined as:

$$\begin{aligned} H(P,Q) &\triangleq H(P) + D_{KL}(P||Q) &(23)\\ &= -\mathbb{E}_{\mathbf{x}\sim P}\left[\log Q(x)\right] &(24)\end{aligned}$$

Some terminology. **Underflow** is when numbers near zero are rounded to zero. Similarly, **overflow** is when large [magnitude] numbers are approximated as $\pm\infty$. **Conditioning** refers to how rapidly a function changes w.r.t. small changes in its inputs. Consider the function $f(\boldsymbol{x}) = \boldsymbol{A}^{-1}\boldsymbol{x}$. When $\boldsymbol{A}$ has an eigenvalue decomposition, its *condition number* is

$$\max_{i,j} \left| \frac{\lambda_i}{\lambda_j} \right| \tag{4.2}$$

which is the ratio of the magnitude of the largest and smallest eigenvalue. When this is large, matrix inversion is sensitive to error in the input [of f(x)].

Gradient-Based Optimization (4.3)

Optimization algorithms that use only the gradient (e.g. SGD) are called 1st-order optimization algorithms. Likewise, ones using the Hessian matrix are 2nd-order.

.

Jacobian and Hessian Matrices. For when we want partial derivatives of some function f whose input and output are both vectors. The **Jacobian matrix** contains all such partial derivatives. Sometimes we want to know about second derivatives too, since this tells us whether a gradient step will cause as much of an improvement as we would expect based on the gradient alone. The **Hessian matrix** $\boldsymbol{H}(f)(\boldsymbol{x})$ is defined such that[3]

$f : \mathbb{R}^m \to \mathbb{R}^n$
$\boldsymbol{J} \in \mathbb{R}^{n \times m}$ where
$J_{i,j} = \frac{\partial}{\partial x_j} f(\boldsymbol{x})_i$

The Hessian is the Jacobian of the gradient.

$$\boldsymbol{H}(f)(\boldsymbol{x})_{i,j} = \frac{\partial^2}{\partial x_i \partial x_j} f(\boldsymbol{x}) \tag{4.6}$$

The second derivative in a specific direction $\hat{\boldsymbol{d}}$ is given by $\hat{\boldsymbol{d}}^T \boldsymbol{H} \hat{\boldsymbol{d}}$. It tells us how well we can expect a gradient descent step to perform. How so? Well, it shows up in the second-order approximation to the function $f(\boldsymbol{x})$ about our current spot, which we can denote $\boldsymbol{x}^{(0)}$. The standard gradient descent step will move us from $\boldsymbol{x}^{(0)} \to \boldsymbol{x}^{(0)} - \epsilon g$, where g is the gradient evaluated at $\boldsymbol{x}^{(0)}$. Plugging this in to the 2nd order approximation shows us how $\boldsymbol{H}$ can give information related to how "good" of a step that really was. Mathematically,

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}^{(0)}) + (\boldsymbol{x} - \boldsymbol{x}^{(0)})^T \boldsymbol{g} + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}^{(0)})^T \boldsymbol{H} (\boldsymbol{x} - \boldsymbol{x}^{(0)}) \tag{4.8}$$

$$f(\boldsymbol{x}^{(0)} - \epsilon \boldsymbol{g}) \approx f(\boldsymbol{x}^{(0)}) - \epsilon \boldsymbol{g}^T \boldsymbol{g} + \frac{1}{2}\epsilon^2 \boldsymbol{g}^T \boldsymbol{H} \boldsymbol{g} \tag{4.9}$$

[3] Recall that the directional derivative of $f(\boldsymbol{x})$ in direction $\boldsymbol{u}$ is $\boldsymbol{u}^T \nabla_x f(\boldsymbol{x})$

If $\boldsymbol{g}^T\boldsymbol{H}\boldsymbol{g}$ is positive, then we can easily solve for the optimal $\epsilon = \epsilon^*$ that decreases the Taylor series approximation as

$$\epsilon^* = \frac{\boldsymbol{g}^T\boldsymbol{g}}{\boldsymbol{g}^T\boldsymbol{H}\boldsymbol{g}} \tag{4.10}$$

The best (and perhaps only) way to take what we learned about the "second derivative test" in single-variable calculus and apply it to the multidimensional case with $\boldsymbol{H}$ is by using the *eigendecomposition of* $\boldsymbol{H}$. Why? Because we can examine the eigenvalues of the Hessian to determine whether the critical point $\boldsymbol{x}^{(0)}$ is a local maximum, local minimum, or saddle point[4]. If all eigenvalues are positive (remember that this is equivalent to saying that the Hessian is **positive definite**!), the point is a local minimum.

The condition number of the Hessian at a given point can give us an idea about how much the second derivatives (along different directions) differ from each other

[4]Emphasis on "values" in "eigenvalues" because it's important not to get tripped up here about what the eigenvectors of the Hessian mean. The reason for the decomposition is that it gives us an orthonormal basis (out of which we can get any direction) and therefore the magnitude of the second derivative along each of these directions as the eigenvalues.

Capacity, Overfitting, and Underfitting (5.2)

Difference between ML and optimization is that, in addition to wanting low training error, we want **generalization error** (test error) to be low as well. The ideal model is an oracle that simply knows the true probability distribution $p(\boldsymbol{x}, y)$ that generates the data. The error incurred by such an oracle, due things like inherently stochastic mappings from $\boldsymbol{x}$ to y or other variables, is called the **Bayes error**. The **no free lunch theorem** states that, averaged over all possible data-generating distributions, every classification algorithm has the same error rate when classifying previously unobserved points. Therefore, the goal of ML research is to understand what kinds of distributions are relevant to the "real world" that an AI agent experiences, and what kinds of ML algorithms perform well on data drawn from the relevant data-generating distributions.

Estimators, Bias and Variance (5.4)

Point Estimation: attempt to provide "best" prediction of some quantity, such as some parameter or even a whole function. Formally, a point estimator or *statistic* is any function of the data:

$$\hat{\theta}_m = g\left(x^{(1)}, \ldots, x^{(m)}\right) \tag{5.19}$$

where, since the data is drawn from a random process, $\hat{\theta}$ is a random variable. **Function estimation** is identical in form, where we want to estimate some $f(x)$ with $\hat{f}$, a point estimator in *function space*.

Bias. Defined below, where the expectation is taken over the data-generating distribution[5]. Bias measures the expected deviation from the true value of the func/param.

$$\text{bias}\left[\hat{\theta}_m\right] = \mathbb{E}\left[\hat{\theta}_m\right] - \theta \tag{5.20}$$

TODO: Figure out how to derive $\mathbb{E}\left[\hat{\theta}_m^2\right]$ for Gaussian distribution [helpful link].

Bias-Variance Tradeoff.

→ **Conceptual Info**. Two sources of error for an estimator are (1) bias and (2) variance, which are both defined as deviations from a certain value. Bias gives deviation from the *true* value, while variance gives the [expected] deviation from this *expected* value.

→ **Summary of main formulas**.

$$\text{bias}\left[\hat{\theta}_m\right] = \mathbb{E}\left[\hat{\theta}_m\right] - \theta \tag{25}$$

$$\text{Var}\left[\hat{\theta}_m\right] = \mathbb{E}\left[\left(\hat{\theta}_m - \mathbb{E}\left[\hat{\theta}_m\right]\right)^2\right] \tag{26}$$

→ **MSE decomposition**. The MSE of the estimates is given by[6]

$$\begin{aligned} \text{MSE} &= \mathbb{E}\left[(\hat{\theta}_m - \theta)^2\right] && (5.53)\\ &= \text{Bias}(\hat{\theta})^2 + \text{Var}\left[\hat{\theta}_m\right] && (5.54) \end{aligned}$$

and desirable estimators are those with low MSE.

[5] May want to double-check this, but I'm fairly certain this is what the book meant when it said "data," based on later examples.

[6] Derivation:

$$\begin{aligned} \text{MSE} &= \mathbb{E}\left[\hat{\theta}^2 + \theta^2 - 2\theta\hat{\theta}\right] && (27)\\ &= \mathbb{E}\left[\hat{\theta}^2\right] + \theta^2 - 2\theta\mathbb{E}\left[\hat{\theta}\right] && (28)\\ &= (\mathbb{E}\left[\hat{\theta}\right]^2 - \mathbb{E}\left[\hat{\theta}\right]^2) + \mathbb{E}\left[\hat{\theta}^2\right] + \theta^2 - 2\theta\mathbb{E}\left[\hat{\theta}\right] && (29)\\ &= \left(\mathbb{E}\left[\hat{\theta}\right]^2 + \theta^2 - 2\theta\mathbb{E}\left[\hat{\theta}\right]\right) + \left(\mathbb{E}\left[\hat{\theta}^2\right] - \mathbb{E}\left[\hat{\theta}\right]^2\right) && (30)\\ &= \text{Bias}(\hat{\theta})^2 + \text{Var}\left[\hat{\theta}_m\right] && (31) \end{aligned}$$

Consistency. As the number of training data points increases, we want the estimators to converge to the true values. Specifically, below are the definitions for *weak* and *strong* consistency, respectively.

$$\begin{aligned} \text{plim}_{m\to\infty}\hat{\theta}_m &= 0 \\ p\left(\lim_{m\to\infty}\hat{\theta}_m = \theta\right) &= 1 \end{aligned} \tag{5.55}$$

where the symbol plim means $P(|\hat{\theta}_m - \theta| > \epsilon) \to 0$ as $m \to \infty$.

Maximum Likelihood Estimation (5.5)

Consider set of m examples $\mathbb{X} = \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ drawn independently from the true (but unknown) $p_{data}(\mathbf{x})$. Let $p_{model}(\mathbf{x};\boldsymbol{\theta})$ be parametric family of probability distributions over the same space indexed by $\boldsymbol{\theta}$. The maximum likelihood estimator for $\boldsymbol{\theta}$ can be expressed as

$$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{x}\sim\hat{p}_{data}} \left[\log p_{model}(\boldsymbol{x};\boldsymbol{\theta})\right] \tag{5.59}$$

where we've chosen to express with log for underflow/gradient reasons. One interpretation of ML is to view it as minimizing the dissimilarity, as measured by the KL divergence[7], between $\hat{p}_{data}$ and p_{model}.

> Any loss consisting of a negative log-likelihood is a **cross-entropy** between the $\hat{p}_{data}$ distribution and the p_{model} distribution.

Thoughts: Let's look at D_{KL} in some more detail. First, I'll rewrite it with the explicit definition of $\mathbb{E}_{\boldsymbol{x}\sim\hat{p}_{data}}\left[\log\left(\hat{p}_{data}(\boldsymbol{x})\right)\right]$:

$$D_{KL}(\hat{p}_{data}||p_{model}) = \mathbb{E}_{\boldsymbol{x}\sim\hat{p}_{data}}\left[\log\left(\hat{p}_{data}(\boldsymbol{x})\right) - \log\left(p_{model}(\boldsymbol{x})\right)\right] \tag{32}$$

$$= \left(\frac{1}{N}\left(\sum_{i=1}^{N}\log\left(\text{Counts}(\boldsymbol{x}_i)\right)\right) - \log N\right) - \mathbb{E}_{\boldsymbol{x}\sim\hat{p}_{data}}\left[\log\left(p_{model}(\boldsymbol{x})\right)\right] \tag{33}$$

Note also that our goal is to find parameters $\boldsymbol{\theta}$ such that D_{KL} is minimized. It is for *this* reason, that we wish to optimize over $\boldsymbol{\theta}$, that minimizing D_{KL} amounts to maximizing the quantity, $\mathbb{E}_{\boldsymbol{x}\sim\hat{p}_{data}}\left[\log\left(p_{model}(\boldsymbol{x})\right)\right]$. Sure, I can agree this is *true*, but **why is our goal to minimize D_{KL}, as opposed to minimizing $|\ D_{KL}\ |$?** I'm assuming it is because optimizing w.r.t. an absolute value is challenging numerically.

---

[7] The KL divergence is given by

$$D_{KL}(\hat{p}_{data}||p_{model}) = \mathbb{E}_{\mathbf{x}\sim\hat{p}_{data}}\left[\log\hat{p}_{data}(\boldsymbol{x}) - \log p_{model}(\boldsymbol{x})\right] \tag{5.60}$$

Conditional Log-Likelihood and MSE. We can readily generalize $\boldsymbol{\theta}_{ML}$ to estimate a conditional probability $p(\mathbf{y} \mid \mathbf{x}; \boldsymbol{\theta})$ in order to predict $\mathbf{y}$ given $\mathbf{x}$, since

We are assuming the examples are i.i.d. here.

$$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log P(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \tag{5.63}$$

Consider linear regression as viewed with ML: Instead of a single prediction value $\hat{y}$ given input $\boldsymbol{x}$, think of the model as producing a *conditional distribution* $p(y \mid \boldsymbol{x})$[8]. To derive the standard linear regression algorithm, we *define*

$$p(y \mid \boldsymbol{x}) = \mathcal{N}\left(y;\ \hat{y}(\boldsymbol{x}; \boldsymbol{w}),\ \sigma^2\right)$$

Assume σ^2 is some fixed constant chosen by the user.

We can use this (and the i.i.d. assumption) to evaluate the conditional log-likelihood as

$$\sum_{i=1}^{m} \log p(y^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) = -m \log \sigma - \frac{m}{2} \log(2\pi) - \sum_{i=1}^{m} \frac{||\hat{y}^{(i)} - y^{(i)}||^2}{2\sigma^2} \tag{5.65}$$

and we see that finding the $\boldsymbol{w}$ that maximizes the conditional log-likelihood is equivalent to finding the $\boldsymbol{w}$ that minimizes the training MSE.

Recall that the training MSE is $\frac{1}{m} \sum_{i=1}^{m} ||\hat{y}^{(i)} - y^{(i)}||^2$

Bayesian Statistics (5.6)

Distinction between frequentist and bayesian approach:

- **Frequentist:** Estimate $\boldsymbol{\theta} \longrightarrow$ make predictions thereafter based on this estimate.
- **Bayesian:** Consider all possible values of $\boldsymbol{\theta}$ when making predictions.

The prior. Before observing the data, we represent our knowledge of $\boldsymbol{\theta}$ using the **prior probability distribution** $p(\boldsymbol{\theta})$. Unlike maximum likelihood, which makes predictions using a *point estimate* of $\boldsymbol{\theta}$ (a single value), the Bayesian approach uses Bayes' rule to make predictions using *full distribution* over $\boldsymbol{\theta}$. In other words, rather than focusing on the most accurate value estimate of $\boldsymbol{\theta}$, we instead focus on pinning down a range of possible $\boldsymbol{\theta}$ values and how likely we believe each of these values to be.

It is common to choose a high-entropy prior, e.g. uniform.

[8] After all, we might have several training points with the same value of $\boldsymbol{x}$ but different labels y.

Supervised Learning Algorithms (5.7)

Logistic Regression. We've already seen that linear regression corresponds to the family

$$p(y \mid \boldsymbol{x}) = \mathcal{N}\left(y;\ \boldsymbol{\theta}^T\boldsymbol{x},\ \boldsymbol{I}\right) \tag{5.80}$$

which we can generalize to the binary **classification** scenario by interpreting as the probability of class 1. One way of doing this while ensuring the output is between 0 and 1 is to use the logistic sigmoid function:

$$p(y = 1 \mid \boldsymbol{x}; \boldsymbol{\theta}) = \sigma(\boldsymbol{\theta}^T\boldsymbol{x}) \tag{5.81}$$

Equation 5.81 is the definition of logistic regression

Unfortunately, there is no closed-form solution for $\boldsymbol{\theta}$, so we must search via maximizing the log-likelihood.

Support Vector Machines. Driving by a linear function $\boldsymbol{w}^T\boldsymbol{x} + \boldsymbol{b}$ like logistic regression, but instead of outputting probabilities it outputs a class identity, which depends on the sign of $\boldsymbol{w}^T\boldsymbol{x} + \boldsymbol{b}$. SVMs make use of the **kernel trick**, the "trick" being that we can rewrite $\boldsymbol{w}^T\boldsymbol{x} + \boldsymbol{b}$ completely in terms of dot products between examples. The general form of our prediction function becomes

$$f(\boldsymbol{x}) = b + \sum_i \alpha_i k(\boldsymbol{x}, \boldsymbol{x}^{(i)}) \tag{5.83}$$

If our kernel function is just $k(x, x^{(i)}) = x^T x^{(i)}$ then we've just rewritten $\boldsymbol{w}$ in the form $\boldsymbol{w} \to \boldsymbol{X}^T\boldsymbol{\alpha}$

where the *kernel* [function] takes the general form $k(\boldsymbol{x}, \boldsymbol{x}^{(i)}) = \phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{x}^{(i)})$. A major drawback to kernel machines (methods) in general is that the cost of evaluating the decision function $f(\boldsymbol{x})$ is linear in the number of training examples. SVMs, however, are able to mitigate this by learning an α with mostly zeros. The training examples with *nonzero* α_i are known as **support vectors**.

Deep Networks: Modern Practices

Contents

Deep Feedforward Networks

 Written by Brandon McKinzie

The strategy/purpose of [feedforward] deep learning is to *learn the set of features/representation describing* $\boldsymbol{x}$ with a mapping ϕ before applying a linear model. In this approach, we have a model

$$y = f(\boldsymbol{x};\ \boldsymbol{\theta}, \boldsymbol{w}) = \phi(\boldsymbol{x}; \boldsymbol{\theta})^T \boldsymbol{w}$$

with ϕ defining a hidden layer.

ReLUs and their generalizations. Some nice properties of ReLUs are...

- Derivatives through a ReLU remain large and consistent whenever the unit is active.
- Second derivative is 0 a.e. and the derivative is 1 everywhere the unit is active, meaning the gradient direction is more useful for learning than it would be with activation functions that introduce 2nd-order effects (see equation 4.9)

Recall the ReLU activation function: $g(z) = \max\{0, z\}$.
a.e. is short for "almost everywhere"

Generalizing to aid gradients when $\boldsymbol{z} < 0$. Three such generalizations are based on using a nonzero slope α_i when $z_i < 0$:

$$h_i = g(\boldsymbol{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i) \tag{34}$$

→ Absolute value rectification: fix $\alpha_i = -1$ to obtain $g(z) = |z|$.
→ Leaky ReLU: fix α_i to a small value like 0.01.
→ Parametric ReLU (PReLU): treats α_i like a learnable parameter.

Logistic sigmoid and hyperbolic tangent. Sigmoid activations on hidden units is a bad idea, since they're only sensitive to their inputs near zero, with small gradients everywhere else. If sigmoid activations must be used, tanh is probably a better substitute, since it resembles the identity (i.e. a linear function) near zero.

Back-Propagation (6.5)

The chain rule. Suppose $\boldsymbol{z} = f(\boldsymbol{y})$ where $\boldsymbol{y} = g(\boldsymbol{x})$ (see margin for dimensions). Then[9],

$\boldsymbol{x} \in \mathbb{R}^m$

$\boldsymbol{y} \in \mathbb{R}^n$

$z : \mathbb{R}^n \to \mathbb{R}$

$g : \mathbb{R}^m \to \mathbb{R}^n$

$$\frac{\partial z}{\partial x_i} = (\nabla_{\boldsymbol{x}} z)_i = \sum_{j=1}^{n} \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i} = \sum_{j=1}^{n} (\nabla_{\boldsymbol{y}} z)_j \frac{\partial y_j}{\partial x_i} = \sum_{j=1}^{n} (\nabla_{\boldsymbol{y}} z)_j \ (\nabla_{\boldsymbol{x}} y_j)_i \tag{6.45}$$

$$\to \nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} \right)^T \nabla_{\boldsymbol{y}} z = \boldsymbol{J}^T_{y=g(\boldsymbol{x})} \nabla_{\boldsymbol{y}} z \tag{6.46}$$

From this we see that the gradient of a variable $\boldsymbol{x}$ can be obtained by multiplying a Jacobian matrix $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ by a gradient $\nabla_{\boldsymbol{y}} z$.

[9] Note that we can view $z = f(\boldsymbol{y})$ as a multi-variable function of the dimensions of $\boldsymbol{y}$,

$$z = f(y_1, y_2, \ldots, y_n)$$

Regularization for Deep Learning

Written by Brandon McKinzie

Recall the definition of regularization: "any modification we make to a learning algorithm that is intented to reduce its generalization error but not its training error."

Parameter Norm Penalties (7.1)

Limiting Model Capacity. Recall that **Capacity** [of a model] is the ability to fit a wide variety of functions. Low cap models may struggle to fit training set, while high cap models may overfit by simply memorizing the training set. We can limit model capacity by adding a parameter norm penalty $\Omega(\theta)$ to the objective function J:

$$\widetilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha\Omega(\theta) \quad \text{where} \quad \alpha \in [0, \infty) \tag{7.1}$$

where we typically choose Ω to only penalize the *weights* and leave biases unregularized.

L2-Regularization. Defined as setting $\Omega(\theta) = \frac{1}{2}||w||_2^2$. Assume that $J(w)$ is quadratic, with minimum at w^*. Since quadratic, we can approximate J with a second-order expansion about w^*.

$$\hat{J}(w) = J(w^*) + \frac{1}{2}(w - w^*)^T H(w - w^*) \tag{7.6}$$

$$\nabla_w \hat{J}(w) = H(w - w^*) \tag{7.7}$$

where $H_{ij} = \frac{\partial^2 J}{\partial w_i w_j}\big|_{w^*}$. If we add in the [derivative of] the weight decay and set to zero, we obtain the solution

$$\widetilde{w} = (H + \alpha I)^{-1} H w^* \tag{7.10}$$

$$= Q(\Lambda + \alpha I)^{-1} \Lambda Q^T w^* \tag{7.13}$$

which shows that the effect of regularization is to rescale the i eigenvectors of H by $\frac{\lambda_i}{\lambda_i + \alpha}$. This means that eigenvectors with $\lambda_i >> \alpha$ are relatively unchanged, but the eigenvectors with $\lambda_i << \alpha$ are shrunk to nearly zero. In other words, only directions along which the parameters contribute significantly to reducing the objective function are preserved relatively intact.

Sparse Representations (7.10)

Weight decay acts by placing a penalty directly on the model parameters. Another strategy is to place a penalty on the *activations* of the units, encouraging their activations to be sparse. It's important to distinguish the difference between sparse parameters and sparse *representations*. In the former, if we take the example of some $\boldsymbol{y} = \boldsymbol{B}\boldsymbol{h}$, there are many zero entries in some parameter matrix $\boldsymbol{B}$ while, in the latter, there are many zero entries in the representation vector $\boldsymbol{h}$. The modification to the loss function, analogous to 7.1, takes the form

$$\widetilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{h}) \quad \text{where} \quad \alpha \in [0, \infty) \tag{7.48}$$

Adversarial Training (7.13)

Even for networks that perform at human level accuracy have a nearly 100 percent error rate on examples that are intentionally constructed to search for an input $\boldsymbol{x}'$ near a data point $\boldsymbol{x}$ such that the model output for $\boldsymbol{x}'$ is very different than the output for $\boldsymbol{x}$.

In many cases, $\boldsymbol{x}'$ can be so similar to $\boldsymbol{x}$ that a human cannot tell the difference!

$$\boldsymbol{x}' \longleftarrow \boldsymbol{x} + \epsilon \cdot \text{sign}\left(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y})\right) \tag{35}$$

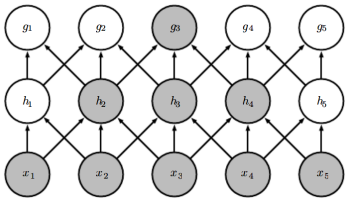
In the context of regularization, one can reduce the error rate on the original i.i.d. test set via **adversarial training** – training on adversarially perturbed training examples.

We use a 2-D image I as our input (and therefore require a 2-D kernel K). Note that most neural networks do not technically implement convolution[10], but instead implement a related function called the *cross-correlation*, defined as

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n) \tag{9.6}$$

Convolution leverages the following three important ideas:

- **Sparse interactions[/connectivity/weights]**. Individual input units only interact/-connect with a subset of the output units. Accomplished by making the kernel smaller than the input. It's important to recognize that the receptive field of the units in the deeper layers of a convolutional network is *larger* than the receptive field of the units in the shallow layers, as seen below.



- **Parameter sharing**.
- **Equivariance** (to translation). Changes in inputs [to a function] cause output to change in the same way. Specifically, f is equivariant to g if $f(g(x)) = g(f(x))$. For convolution, g would be some function that translates the input.

[10] Technically the convolution output is defined as

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n) K(i-m, j-n) \tag{9.4}$$

$$= (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n) K(m,n) \tag{9.5}$$

where 9.5 can be asserted due to commutativity of convolution.

Sequence Modeling (RNNs)

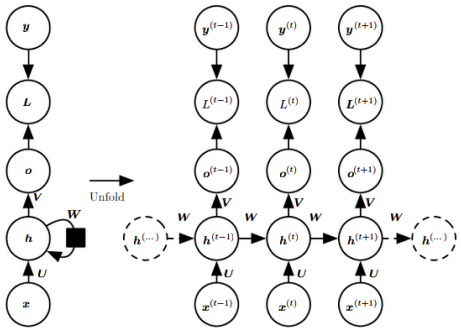
Written by Brandon McKinzie

Review: The Basics of RNNs

Notation/Architecture Used.

- **U**: input → hidden.
- **W**: hidden → hidden.
- **V**: hidden → output.
- **Activations**: tanh [hidden] and softmax [after output].
- **Misc. Details**: $\boldsymbol{x}^{(t)}$ is a *vector* of inputs fed at time t. Recall that RNNs can be unfolded for any desired number of steps τ. For example, if $\tau = 3$, the general functional representation output of an RNN is $\boldsymbol{s}^{(3)} = f(\boldsymbol{s}^{(2)};\ \boldsymbol{\theta}) = f(f(\boldsymbol{s}^{(1)};\boldsymbol{\theta});\ \boldsymbol{\theta})$. Typical RNNs read information out of the state $\boldsymbol{h}$ to make predictions.

Shape of $\boldsymbol{x}^{(t)}$ fixed, e.g. vocab length.



Black square on recurrent connection ≡ interaction w/delay of a single time step.

Forward Propagation & Loss. Specify initial state $\boldsymbol{h}^{(0)}$. Then, for each time step from $t = 1$ to $t = \tau$, feed input sequence $\boldsymbol{x}^{(t)}$ and compute the output sequence $\boldsymbol{o}^{(t)}$. To determine the loss at each time-step, $L^{(t)}$, we compare softmax($\boldsymbol{o}^{(t)}$) with (one-hot) $\boldsymbol{y}^{(t)}$.

$$\boldsymbol{h}^{(t)} = \tanh(\boldsymbol{a}^{(t)}) \qquad \text{where} \qquad \boldsymbol{a}^{(t)} = b + W\boldsymbol{h}^{(t-1)} + U\boldsymbol{x}^{(t)} \tag{10.9/8}$$

$$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}(\boldsymbol{o}^{(t)}) \qquad \text{where} \qquad \boldsymbol{o}^{(t)} = c + V\boldsymbol{h}^{(t)} \tag{10.11/10}$$

Note that this is an example of an RNN that maps input seqs to output seqs of the same length[11]. We can then compute, e.g., the log-likelihood loss $L = \sum_t L^{(t)}$ over all time steps as:

$$L = -\sum_t \log\left(p_{model}\left[y^{(t)} \mid \{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(t)}\}\right]\right) \tag{10.12/13/14}$$

Convince yourself this is identical to cross-entropy.

[11] Where "same length" is related to the number of timesteps (i.e. τ input steps means τ output steps), not anything about the actual shapes/sizes of each individual input/output.

where $y^{(t)}$ is the **ground-truth** (one-hot vector) at time t, whose probability of occurring is given by the corresponding element of $\hat{\boldsymbol{y}}^{(t)}$

Back-Propagation Through Time.

1. **Internal-Node Gradients**. In what follows, when considering what is included in the chain rule(s) for gradients with respect to a node $\boldsymbol{N}$, just need to consider paths from it [through its **descendents**] to loss node(s).
   - **Output nodes**. For any given time t, the node $\boldsymbol{o}^{(t)}$ has only one direct descendant, the loss node $L^{(t)}$. Since no other loss nodes can be reached from $\boldsymbol{o}^{(t)}$, it is the only one we need consider in the gradient.

$$
\begin{aligned}
\left(\nabla_{\boldsymbol{o}^{(t)}} L\right)_i &= \frac{\partial L}{\partial \boldsymbol{o}_i^{(t)}} \\
&= \frac{\partial L}{\partial L^{(t)}} \cdot \frac{\partial L^{(t)}}{\partial \boldsymbol{o}_i^{(t)}} \\
&= (1) \cdot \frac{\partial L^{(t)}}{\partial \boldsymbol{o}_i^{(t)}} \\
&= \frac{\partial}{\partial \boldsymbol{o}_i^{(t)}} \left\{ -\log\left(\hat{\boldsymbol{y}}_{y^{(t)}}^{(t)}\right) \right\} \\
&= -\frac{\partial}{\partial \boldsymbol{o}_i^{(t)}} \left\{ \log\left(\frac{e^{\boldsymbol{o}_{y^{(t)}}^{(t)}}}{\sum_j e^{\boldsymbol{o}_j^{(t)}}} \right) \right\} \\
&= -\frac{\partial}{\partial \boldsymbol{o}_i^{(t)}} \left\{ \boldsymbol{o}_{y^{(t)}}^{(t)} - \log\left(\sum_j e^{\boldsymbol{o}_j^{(t)}} \right) \right\} \\
&= -\left\{ \mathbf{1}_{i,y^{(t)}} - \frac{\partial}{\partial \boldsymbol{o}_i^{(t)}} \log\left(\sum_j e^{\boldsymbol{o}_j^{(t)}} \right) \right\}
\end{aligned}
\tag{36}
$$

Ground-truth $y^{(t)}$ here is a **scalar**, interpreted as the index of the correct label of output vector.

$$\mathbf{1}_{i,y^{(t)}} = \begin{cases} 1 & y^{(t)} = i \\ 0 & \text{otherwise} \end{cases}$$

$$
\begin{aligned}
&= -\left\{ \mathbf{1}_{i,y^{(t)}} - \frac{1}{\sum_j e^{\boldsymbol{o}_j^{(t)}}} \frac{\partial \sum_j e^{\boldsymbol{o}_j^{(t)}}}{\partial \boldsymbol{o}_i^{(t)}} \right\} \\
&= -\left\{ \mathbf{1}_{i,y^{(t)}} - \frac{e^{\boldsymbol{o}_i^{(t)}}}{\sum_j e^{\boldsymbol{o}_j^{(t)}}} \right\} \\
&= -\left\{ \mathbf{1}_{i,y^{(t)}} - \hat{\boldsymbol{y}}_i^{(t)} \right\} \\
&= \hat{\boldsymbol{y}}_i^{(t)} - \mathbf{1}_{i,y^{(t)}}
\end{aligned}
\tag{10.18}
$$

which leaves all entries of $\boldsymbol{o}^{(t)}$ unchanged *except* for the entry corresponding to the true label, which will become negative in the gradient. All this means is, since we

want to increase the probability of this entry, driving this value up will *decrease* the loss (hence negative) and driving any other entries up will *increase* the loss proportional to its current estimated probability (driving up an [incorrect] entry that is already high is "worse" than driving up a small [incorrect entry]).

- **Hidden nodes**. First, consider the simplest hidden node to take the gradient of, the last one, $\boldsymbol{h}^{(\tau)}$ (simplest because only one descendant [path] reaching any loss node(s)).

$$\begin{aligned}
\left(\nabla_{\boldsymbol{h}^{(\tau)}} L\right)_i &= \frac{\partial L}{\partial L^{(\tau)}} \sum_{k=1}^{n_{out}} \frac{\partial L^{(\tau)}}{\partial \boldsymbol{o}_k^{(\tau)}} \frac{\partial \boldsymbol{o}_k^{(\tau)}}{\partial \boldsymbol{h}_i^{(\tau)}} \\
&= \sum_{k=1}^{n_{out}} \left(\nabla_{\boldsymbol{o}^{(\tau)}} L\right)_k \frac{\partial \boldsymbol{o}_k^{(\tau)}}{\partial \boldsymbol{h}_i^{(\tau)}} \\
&= \sum_{k=1}^{n_{out}} \left(\nabla_{\boldsymbol{o}^{(\tau)}} L\right)_k \frac{\partial}{\partial \boldsymbol{h}_i^{(\tau)}} \left\{ c_k + \sum_{j=1}^{n_{hid}} V_{kj} \boldsymbol{h}_j^{(\tau)} \right\} \\
&= \sum_{k=1}^{n_{out}} \left(\nabla_{\boldsymbol{o}^{(\tau)}} L\right)_k V_{ki} \\
&= \sum_{k=1}^{n_{out}} (V^T)_{ik} \left(\nabla_{\boldsymbol{o}^{(\tau)}} L\right)_k \\
&= \left(V^T \nabla_{\boldsymbol{o}^{(t)}} L\right)_i
\end{aligned} \tag{10.19}$$

Before proceeding, **notice the following useful pattern**: If two nodes a and b, each containing n_a and n_b neurons, are fully connected by parameter matrix $W_{n_b \times n_a}$ and directed like $a \to b \to L$, then[12] $\nabla_a L = W^T \nabla_b L$. Using this result, we can then iterate and take gradients back in time from $t = \tau - 1$ to $t = 1$ as follows:

$$\nabla_{\boldsymbol{h}^{(t)}} L = \left(\frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{h}^{(t)}}\right)^T \left(\nabla_{\boldsymbol{h}^{(t+1)}} L\right) + \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}}\right)^T \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \tag{10.20}$$

$\frac{\mathrm{d}}{\mathrm{d}x} \tanh(x) = 1 - \tanh^2(x)$

$$= W^T \left(\nabla_{\boldsymbol{h}^{(t+1)}} L\right) \operatorname{diag}(1 - \tanh^2(\boldsymbol{a}^{(t+1)})) + V^T \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \tag{10.21}$$

$(\operatorname{diag}(\boldsymbol{a}))_{ii} \triangleq a_i$

$$= W^T \left(\nabla_{\boldsymbol{h}^{(t+1)}} L\right) \operatorname{diag}\left(1 - (\boldsymbol{h}^{(t+1)})^2\right) + V^T \left(\nabla_{\boldsymbol{o}^{(t)}} L\right)$$

2. **Parameter Gradients**. Now we can compute the gradients for the parameter matrices/vectors, where it is crucial to remember that a given parameter matrix (e.g. U) is shared across *all* time steps t. We can treat tensor derivatives in the same form as

[12] More generally,

$$\nabla_a L = (\frac{\partial b}{\partial a})^T \nabla_b L$$

which is a good example of how vector derivatives map into a matrix. For example, let $\boldsymbol{a} \in \mathbb{R}^{n_a}$ and $\boldsymbol{b} \in \mathbb{R}^{n_b}$. Then

$$\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \in \mathbb{R}^{n_b \times n_a}$$

previously done with vectors after a quick abstraction: For any tensor $\mathbf{X}$ of arbitrary rank (e.g. if rank-4 then index like $\mathbf{X}_{ijk\ell}$), use single variable (e.g. i) to represent the complete tuple of indices[13].

- **Bias parameters [vectors]**. These are nothing new, since just vectors.

$$\begin{aligned}(\nabla_{\boldsymbol{c}} L) &= \sum_t \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{c}^{(t)}}\right)^T \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \\ &= \sum_t \left(\nabla_{\boldsymbol{o}^{(t)}} L\right)\end{aligned} \tag{10.22}$$

$$\begin{aligned}(\nabla_{\boldsymbol{c}} L) &= \sum_t \left(\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{b}^{(t)}}\right)^T \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \\ &= \sum_t \operatorname{diag}\left(1 - (\boldsymbol{h}^{(t)})^2\right) \left(\nabla_{\boldsymbol{h}^{(t)}} L\right)\end{aligned} \tag{10.23}$$

- **$\mathbf{V}$ ($n_{out} \times n_{hid}$)**.

$$\nabla_{\mathbf{V}} L = \sum_t^{\tau} \nabla_{\mathbf{V}} L^{(t)} \tag{37a}$$

$$= \sum_t^{\tau} \nabla_{\mathbf{V}} L^{(t)}(\boldsymbol{o}_1^{(t)}, \ldots, \boldsymbol{o}_{n_{out}}^{(t)}) \tag{37b}$$

$$= \sum_t^{\tau} \sum_i^{n_{out}} \left(\nabla_{\boldsymbol{o}^{(t)}} L\right)_i \nabla_{\mathbf{V}} \boldsymbol{o}_i^{(t)} \tag{37c}$$

$$= \sum_t^{\tau} \sum_i^{n_{out}} \left(\nabla_{\boldsymbol{o}^{(t)}} L\right)_i \nabla_{\mathbf{V}} \left\{ c_i + \sum_{j=1}^{n_{hid}} V_{ij} \boldsymbol{h}_j^{(t)} \right\} \tag{37d}$$

$$= \sum_t^{\tau} \sum_i^{n_{out}} \left(\nabla_{\boldsymbol{o}^{(t)}} L\right)_i \begin{bmatrix} 0 & 0 & \ldots & 0 \\ \vdots & \vdots & \ldots & \vdots \\ \boldsymbol{h}_1^{(t)} & \boldsymbol{h}_2^{(t)} & \ldots & \boldsymbol{h}_{n_{hid}}^{(t)} \\ \vdots & \vdots & \ldots & \vdots \\ 0 & 0 & \ldots & 0 \end{bmatrix} \tag{37e}$$

$$= \sum_t^{\tau} \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) (\boldsymbol{h}^{(t)})^T \tag{37f}$$

[13] More details on tensor derivatives: Consider the chain defined by $\mathbf{Y} = g(\mathbf{X})$, and $z = f(\mathbf{Y})$, where z is some vector. Then

$$\nabla_{\mathbf{X}} z = \sum_j (\nabla_{\mathbf{X}} Y_j) \frac{\partial z}{\partial Y_j}$$

ith row, which is $(\boldsymbol{h}^{(t)})^T$.
(See footnote for more)

where if 37e confuses you, see the footnote[14].

- **W** ($n_{hid} \times n_{hid}$). This one is a bit odd, since $\boldsymbol{W}$ is, in a sense, even more "shared" across time steps than $\boldsymbol{V}$[15]. The authors here define/choose, when evaluating $\nabla_{\mathbf{W}} h_i^{(t)}$ to only concern themselves with $\boldsymbol{W} := \boldsymbol{W}^{(t)}$, i.e. the direct connections to $\boldsymbol{h}$ at time t.

$$\nabla_{\mathbf{W}} L = \sum_t^{\tau} \nabla_{\mathbf{W}} L^{(t)} \tag{39a}$$

$$= \sum_t^{\tau} \sum_i^{n_{hid}} \left(\nabla_{\boldsymbol{h}^{(t)}} L\right)_i \nabla_{\mathbf{W^{(t)}}} \boldsymbol{h}_i^{(t)} \tag{10.25}$$

$$= \sum_t^{\tau} \sum_i^{n_{hid}} \left(\nabla_{\boldsymbol{h}^{(t)}} L\right)_i \left(\operatorname{diag}\left(1 - (\boldsymbol{h}^{(t)})^2\right) \begin{bmatrix} 0 & 0 & \dots & 0 \\ \vdots & \vdots & \dots & \vdots \\ \boldsymbol{h}_1^{(t-1)} & \boldsymbol{h}_2^{(t-1)} & \dots & \boldsymbol{h}_{n_{hid}}^{(t-1)} \\ \vdots & \vdots & \dots & \vdots \\ 0 & 0 & \dots & 0 \end{bmatrix} \right) \tag{39b}$$

$$= \sum_t^{\tau} \operatorname{diag}\left(1 - (\boldsymbol{h}^{(t)})^2\right) \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) (\boldsymbol{h}^{(t-1)})^T \tag{10.26}$$

- **U** ($n_{hid} \times n_{in}$). Very similar to the previous calculation.

$$\nabla_{\mathbf{U}} L = \sum_t^{\tau} \nabla_{\mathbf{U}} L^{(t)} \tag{40a}$$

$$= \sum_t^{\tau} \sum_i^{n_{hid}} \left(\nabla_{\boldsymbol{h}^{(t)}} L\right)_i \nabla_{\mathbf{U^{(t)}}} \boldsymbol{h}_i^{(t)} \tag{10.27}$$

$$= \sum_t^{\tau} \operatorname{diag}\left(1 - (\boldsymbol{h}^{(t)})^2\right) \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) (\boldsymbol{x}^{(t)})^T \tag{10.28}$$

[14] The general lesson learned here is that, for some matrix $\boldsymbol{W} \in \mathbb{R}^{a \times b}$ and vector $\boldsymbol{x} \in \mathbb{R}^b$,

$$\sum_i \nabla_{\mathbf{W}} \left[(\boldsymbol{W}x)_i\right] = \begin{bmatrix} \boldsymbol{x}^T \\ \boldsymbol{x}^T \\ \vdots \\ \boldsymbol{x}^T \end{bmatrix} \tag{38}$$

where, of course, the output has the same dimensions as $\boldsymbol{W}$.

[15] Specifically, $\boldsymbol{h}^{(t)}$ is both

(a) An explicit function of the parameter matrix $\boldsymbol{W}^{(t)}$ directly feeding into it.

(b) An implicit function of all other $\boldsymbol{W}^{t=i}$ that came before.

This is different than before, where we had $\boldsymbol{o}^{(t)}$ not implicitly depending on earlier $\boldsymbol{V}^{(t=i)}$. In other words, $\boldsymbol{h}^{(t)}$ is a descendant of all earlier (and current) $\boldsymbol{W}$.

RNNs as Directed Graphical Models

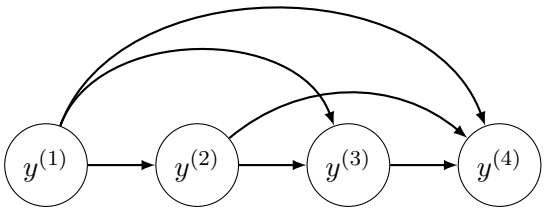
The advantage of RNNs is their efficient parameterization of the joint distribution over $\boldsymbol{y}^{(i)}$ via parameter sharing. This introduces a built-in assumption that we can model the effect of $y^{(i)}$ in the distant past on the current $y^{(t)}$ *via its effect on* $\boldsymbol{h}$. We are also assuming that the conditional probability distribution over the variables at $t+1$ given the variables at time t is **stationary**. Next, we want to know how to draw *samples* from such a model. Specifically, how to sample from the conditional distribution ($y^{(t)}$ given $y^{(t-1)}$) at each time step.

Say we want to model a sequence of scalar random variables $\mathbb{Y} \triangleq \{y^{(1)}, \ldots, y^{(\tau)}\}$ for some sequence length τ. Without making independence assumptions just yet, we can parameterize the joint distribution $P(\mathbb{Y})$ with basic definitions of probability:

$$P(\mathbb{Y}) \triangleq P(y^{(1)}, \ldots, y^{(\tau)}) = \prod_{t=1}^{\tau} P(y^{(t)} \mid y^{(t-1)}, \ldots, y^{(1)}) \tag{41}$$

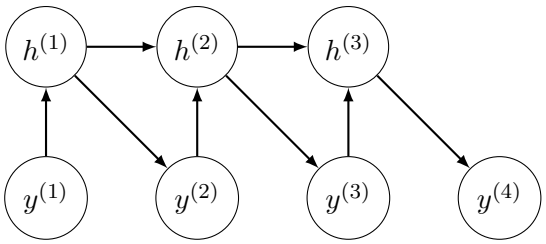
where I've drawn an example of the *complete graph* for $\tau = 4$ below.

The complete graph can represent the direct dependencies between any pairs of y values.



If each value y could take on the same fixed set of k values, we would need to learn k^4 parameters to represent the joint distribution $P(\mathbb{Y})$. This clearly inefficient, since the number of parameters needed scales like $\mathcal{O}(k^\tau)$. If we relax the restriction that each $y^{(i)}$ must depend *directly* on all past $y^{(j)}$, we can considerably reduce the number of parameters needed to compute the probability of some particular sequence.

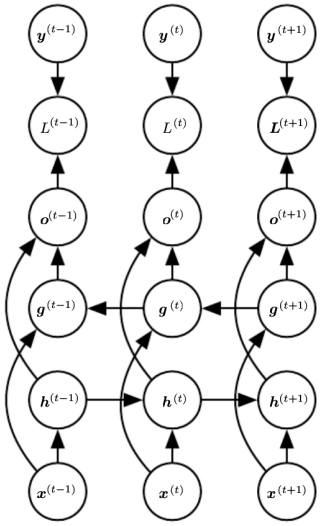
We could include latent variables h at each timestep that capture the dependencies, reminiscent of a classic RNN:



Since in the RNN case all factors $P(h^{(t)} \mid h^{(t-1)})$ are deterministic, we don't need any additional parameters to compute this probability[16], other than the single m^2 parameters needed to convert any $h^{(t)}$ to the next $h^{(t+1)}$ (which is shared across all transitions). Now, the number of parameters needed as a function of sequence length is constant, and as a function of k is just $\mathcal{O}(k)$.

Finally, to view the RNN as a graphical model, we must describe how to sample from it, namely how to sample a sequence $\boldsymbol{y}$ from $P(\mathbb{Y})$, if parameterized by our graphical model above. In the general case where we don't know the value of τ for our sequence $\boldsymbol{y}$, one approach is to have a EOS symbol that, if found during sampling, means we should stop there. Also, in the typical case where we actually want to model $P(y \mid x)$ for input sequence x, we can reinterpret the parameters $\boldsymbol{\theta}$ of our graphical model as a function of $\boldsymbol{x}$ the input sequence. In other words, the graphical model interpretation becomes a function of $\boldsymbol{x}$, where $\boldsymbol{x}$ determines the exact values of the probabilities the graphical model takes on – an "instance" of the graphical model.

Bidirectional RNNs. In many applications, it is desirable to output a prediction of $\boldsymbol{y}^{(t)}$ that may depend on *the whole sequence*. For example, in speech recognition, the interpretation of words/sentences can also depend on what is *about* to be said. Below is a typical bidirectional RNN, where the inputs $\boldsymbol{x}^(t)$ are fed both to a "forward" RNN ($\boldsymbol{h}$) and a "backward" RNN ($\boldsymbol{g}$).

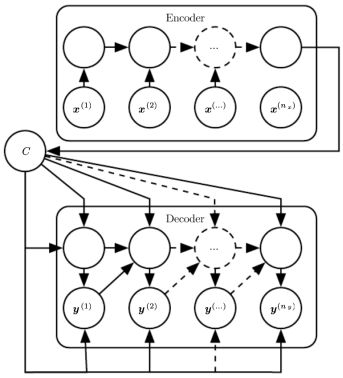


Notice how the output units $\boldsymbol{o}^{(t)}$ have the nice property of depending on both the past and future while being most sensitive to input values around time t.

[16] Don't forget that, in a neural net, a variable $y^{(t)}$ is represented by a *layer*, which itself is composed of k nodes, each associated with one of the k unique values that $y^{(t)}$ could be.

Encoder-Decoder Seq2Seq Architectures (10.4)

Here we discuss how an RNN can be trained to map an input sequence to output sequence which is not necessarily the same length. (Not really much of a discussion...figure below says everything.)



Challenge of Long-Term Deps. (10.7)

Gradients propagated over many stages either vanish (usually) or explode. We saw how this could occur when we took parameter gradients earlier, and for weight matrices $\boldsymbol{W}$ further along from the loss node, the expression for $\nabla_{\mathbf{W}} L$ contained multiplicative Jacobian factors. Consider the (linear activation) repeated function composition of an RNN's hidden state in 10.36. We can rewrite it as a power method (10.37), and if $\boldsymbol{W}$ admits an eigendecomposition (remember $\boldsymbol{W}$ is necessarily square here), we can further simplify as seen in 10.38.

$$\boldsymbol{h}^{(t)} = \boldsymbol{W}^T \boldsymbol{h}^{(t-1)} \tag{10.36}$$

$$= (\boldsymbol{W}^t)^T \boldsymbol{h}^{(0)} \tag{10.37}$$

$$= \boldsymbol{Q}^T \boldsymbol{\Lambda}^t \boldsymbol{Q} \boldsymbol{h}^{(0)} \tag{10.38}$$

Q: Explain interp. of mult. $\boldsymbol{h}$ by $\boldsymbol{Q}$ as opposed to the usual $\boldsymbol{Q}^T$ explained in the linear algebra review.

Any component of $\boldsymbol{h}^{(0)}$ that isn't aligned with the largest eigenvector will eventually be discarded.[17]

If, however, we have a non-recurrent network such that the state elements are repeatedly multiplied by different $w^{(t)}$ at each time step, the situation is different. Suppose the different

[17] Make sure to think about this from the right perspective. The largest value of $t = \tau$ in the RNNs we've seen would correspond with either (1) the largest output sequence or (2) the largest input sequence (if fixed-vector output). After we extract the output from a given forward pass, we reset the clock and either back-propagate errors (if training) or get ready to feed another sequence.

$w^{(t)}$ are i.i.d. with mean 0 and variance v. The variance of the product is easily seen to be $\mathcal{O}(v^n)$[18]. *To obtain some desired variance v^* we may choose the individual weights with variance $v = \sqrt[n]{v^*}$.*

LSTMs and Other Gated RNNs (10.10)

While leaky units have connection weights that are either manually chosen constants or are trainable parameters, gated RNNs generalize this to connection weights that may change *at each time step*. Furthermore, gated RNNs can learn to both accumulate and *forget*, while leaky units are designed for just accumulation[19]

LSTM (10.10.1). The idea is we want self-loops to produce paths where the gradient can flow for long durations. The self-loop weights are **gated**, meaning they are controlled by another hidden unit, interpreted as being conditioned on *context*. Listed below are the main components of the LSTM architecture.

- **Forget gate** $f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}\right)$.
- **Internal state** $s_i^{(t)} = f_i^{(t)} \odot s_i^{(t-1)} + g_i^{(t)} \odot \sigma\left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)}\right)$.
- **External input gate** $g_i^{(t)} = \sigma\left(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)}\right)$.
- **Output gate** $q_i^{(t)} = \sigma\left(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)}\right)$.

The subscript, i, identifies the cell. The superscript, t, denotes the time.

The final hidden state can then be computed via

$$h_i^{(t)} = \tanh(s_i^{(t)}) \odot q_i^{(t)} \tag{44}$$

[18] Quick sketch of (my) proof:

$$\text{Var}\left[w^{(i)}\right] = v = \mathbb{E}\left[(w^{(i)})^2\right] - \cancel{\mathbb{E}\left[w^{(i)}\right]^2} \tag{42}$$

$$\text{Var}\left[\prod_t^n w^{(t)}\right] = \mathbb{E}\left[\left(\prod_t^n w^{(t)}\right)^2\right] = \prod_t^n \mathbb{E}\left[(w^{(t)})^2\right] = v^n \tag{43}$$

[19] Q: Isn't choosing to update with higher relative weight on the present the same as forgetting? A: Sort of. It's like "soft forgetting" and will inevitably erase more/less than desired (smeary). In this context, "forget" means to set the weight of a specific past cell to zero.

Natural Language Processing (12.4)

Begins on pg. 448

n-grams. A **language model** defines a probability distribution over sequences of [discrete] tokens (words/characters/etc). Early models were based on the *n-gram*: a [fixed-length] sequence of n tokens. Such models define the conditional distribution for the nth token, given the $(n-1)$ previous tokens:

$$P(x_t \mid x_{t-(n-1)}, \dots, x_{t-1})$$

where x_i denotes the token at step/index/position i in the sequence.

To define distributions over longer sequences, we can just use Bayes rule over the shorter distributions, as usual. For example, say we want to find the [joint] distribution for some τ-gram ($\tau > n$), and we have access to an n-gram model and a [perhaps different] model for the initial sequence $P(x_1, \dots, x_{n-1})$. We compute the τ distribution simply as follows:

$$P(x_1, \dots, x_\tau) = P(x_1, \dots, x_{n-1}) \prod_{t=n}^{\tau} P(x_t \mid x_{t-1}, \dots, x_{t-(n-2)}, x_{t-(n-1)}) \tag{12.5}$$

where it's important to see that each factor in the product is a distribution over a length-n sequence. Since we need that initial factor, it is common to train both an n-gram model and an $n-1$-gram model simultaneously.

Let's do a specific example for a trigram (n = 3).

- **Assumptions [for this trigram model example]**:
  - For any $n \geq 3$, $P(x_n \mid x_1, \dots, x_{n-1}) = P(x_n \mid x_{n-2}, x_{n-1})$.
  - When we get to computing the full joint distribution over some sequence of arbitrary length, we assume we have access to both P_3 and P_2, the joint distributions over all subsequences of length 3 and 2, respectively.
- **Example sequence**: We want to know how to use a trigram model on the sequence ['THE', 'DOG', 'RAN', 'AWAY'].

- **Derivation**: We can use the built-in model assumption to derive the following formula.

$$\begin{aligned} P(\text{THE DOG RAN AWAY}) &= P_3(\text{AWAY} \mid \text{THE DOG RAN})\ P_3(\text{THE DOG RAN}) \\ &= P_3(\text{AWAY} \mid \text{DOG RAN})\ P_3(\text{THE DOG RAN}) \\ &= \frac{P_3(\text{DOG RAN AWAY})}{P_2(\text{DOG RAN})}\ P_3(\text{THE DOG RAN}) \\ &= P_3(\text{THE DOG RAN}) P_3(\text{DOG RAN AWAY}) / P_2(\text{DOG RAN}) \end{aligned} \tag{12.7}$$

Limitations of n-gram. The last example illustrates some potential problems one may encounter that arise [if using MLE] when the full joint we seek is nonzero, but (a) some P_n factor is zero, or (b) P_{n-1} is zero. Some methods of dealing with this are as follows.

Recall that, in MLE, the P_n and P_{n-1} are usually approximated via counting occurrences in the training set

- **Smoothing**: shifting probability mass from the observed tuples to unobserved ones that are similar.
- **Back-off methods**: look up the lower-order (lower values of n) n-grams if the frequency of the context $x_{t-1}, \ldots, x_{t-(n-1)}$ is too small to use the higher-order model.

In addition, n-gram models are vulnerable to the curse of dimensionality, since most n-grams won't occur in the training set[20], even for modest n.

Neural Language Models (12.4.2)

Designed to overcome curse of dimensionality by using a distributed representation of words. Recognize that any model trained on sentences of length n and then told to generalize to new sentences [also of length n] must deal with a space[21] of possible sentences that is exponential in n. Such word representations (i.e. viewing words as existing in some high-dimensional space) are often called **word embeddings**. The idea is to map the words (or sentences) from the raw high-dimensional [vocab sized] space to a smaller feature space, where similar words are closer to one another. Using distributed representations may also be used with graphical models (think Bayes' nets) in the form multiple *latent variables*.

[20] For a given vocabulary, which usually has much more than n possible words, consider how many possible sequences of length n.

[21] Ok I tried re-wording that from the book's confusing wording but that was also a bit confusing. Let me break it down. Say you train on a thousand sentences each of length 5. For a given vocabulary of size VOCAB_SIZE, the number of possible sequences of length 5 is (VOCAB_SIZE)^5, which can be quite a lot more than a thousand (not to mention the possibility of duplicate training examples). To the naive model, all points in this high-dimensional space are basically the same. A neural language model, however, tries to arrange the space of possibilities in a meaningful way, so that an unforeseen sample at test time can be said "similar" as some previously seen training example. It does this by *embedding* words/sentences in a lower-dimensional feature space.

Deep Learning Research

Contents

Linear Factor Models (Ch. 13)

Written by Brandon McKinzie

Overview. Much research is in building a *probabilistic model*[22] of the input, $p_{model}(x)$. Why? Because then we can perform *inference* to predict stuff about our environment given any of the other variables. We call the other variables **latent variables**, h, with

$$p_{model}(x) = \sum_h \Pr(h) \Pr(x \mid h) = \mathbb{E}_h \left[p_{model}(x \mid h) \right] \tag{45}$$

So what? Well, the latent variables provide another means of *data representation*, which can be useful. **Linear factor models** (LFM) are some of the simplest probabilistic models with latent variables.

> A linear factor model is defined by the use of a stochastic linear decoder function that generates $\boldsymbol{x}$ by adding noise to a linear transformation of $\boldsymbol{h}$.

Note that $\boldsymbol{h}$ is a *vector* of arbitrary size, where we assume $p(\boldsymbol{h})$ is a **factorial distribution**: $p(\boldsymbol{h}) = \prod_i p(h_i)$. This roughly means we assume the elements of $\boldsymbol{h}$ are mutually independent[23]. The LFM describes the data-generation process as follows:

1. Sample the explanatory factors: $\boldsymbol{h} \sim p(\boldsymbol{h})$.
2. Sample the real-valued observable variables given the factors:

$$\boldsymbol{x} = \boldsymbol{W}\boldsymbol{h} + \boldsymbol{b} + \text{noise} \tag{46}$$

Probabilistic PCA and Factor Analysis.

- **Factor analysis**:

$$\boldsymbol{h} \sim \mathcal{N}(\boldsymbol{h}; \boldsymbol{0}, \boldsymbol{I}) \tag{47}$$

$$\text{noise} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{\psi} \equiv \text{diag}(\boldsymbol{\sigma^2})) \tag{48}$$

$$\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{x};\ \boldsymbol{b}, \boldsymbol{W}\boldsymbol{W}^T + \boldsymbol{\psi}) \tag{49}$$

  where the last relation can be shown by recalling that a linear combination of Gaussian variables is itself Gaussian, and showing that $\mathbb{E}_h[\boldsymbol{x}] = \boldsymbol{b}$, and $\text{Cov}(\boldsymbol{x}) = \boldsymbol{W}\boldsymbol{W}^T + \boldsymbol{\psi}$.

[22] Whereas, before, we've been building *functions* of the input (deterministic).

[23] Note that, technically, this assumption isn't strictly the definition of mutual independence, which requires that every *subset* (i.e. not just the full set) of $\{h_i \in \boldsymbol{h}\}$ follow this factorial property.

It is worth emphasizing the interpretation of $\boldsymbol{\psi}$ as the matrix of **conditional variances** σ_i^2. *Huh?* Let's take a step back. The fact that we were able to separate the distributions in the above relations for $\boldsymbol{h}$ and noise is from a built-in assumption that $\Pr(x_i|\boldsymbol{h}, x_{j\neq i}) = \Pr(x_i|\boldsymbol{h})$[24].

The Big Idea

The latent variable $\boldsymbol{h}$ is a big deal because it **captures the dependencies** between the elements of $\boldsymbol{x}$. *How do I know?* Because of our assumption that the x_i are conditionally independent given $\boldsymbol{h}$. If, once we specify $\boldsymbol{h}$, all the elements of $\boldsymbol{x}$ become independent, then any information about their interrelationship is hiding somewhere in $\boldsymbol{h}$.

Detailed walkthrough of Factor Analysis (a.k.a me slowly reviewing, months after taking this note):

- **Goal**. Analyze and understand the motivations behind how Factor Analysis defines the data-generation process under the framework of LFMs (defined in steps 1 and 2 earlier). Assume $\boldsymbol{h}$ has dimension n.
- **Prior**. Defines $p(\boldsymbol{h}) := \mathcal{N}(\boldsymbol{h}; \boldsymbol{0}, \boldsymbol{I})$, the unit-variance Gaussian. Explicitly,

$$p(\boldsymbol{h}) := \frac{1}{(2\pi)^{n/2}} e^{-\frac{1}{2}\sum_i^n h_i^2}$$

- **Noise**. Assumed to be drawn from a Gaussian with diagonal covariance matrix $\boldsymbol{\psi} := \text{diag}\left(\boldsymbol{\sigma}^2\right)$. Explicitly,

$$p(\text{noise} = \boldsymbol{a}) := \frac{1}{(2\pi)^{n/2}\prod_i^n \sigma_i} e^{-\frac{1}{2}\sum_i^n a_i^2/\sigma_i^2}$$

- **Deriving distribution of $\boldsymbol{x}$**. We use the fact that any linear combination of Gaussians is itself Gaussian. Thus, deriving $p(\boldsymbol{x})$ is reduced to computing it's mean and covariance matrix.

$$\begin{aligned}
\boldsymbol{\mu}_x &= \mathbb{E}_{\boldsymbol{h}}\left[\boldsymbol{W}\boldsymbol{h} + \boldsymbol{b}\right] && (50)\\
&= \int p(\boldsymbol{h})(\boldsymbol{W}\boldsymbol{h} + \boldsymbol{b})\mathrm{d}\boldsymbol{h} && (51)\\
&= \boldsymbol{b} + \int \frac{1}{(2\pi)^{n/2}} e^{-\frac{1}{2}\sum_i^n h_i^2}\boldsymbol{W}\boldsymbol{h}\mathrm{d}\boldsymbol{h} && (52)\\
&= \boldsymbol{b} && (53)\\
\text{Cov}(\boldsymbol{x}) &= \mathbb{E}\left[(\boldsymbol{x} - \mathbb{E}\left[\boldsymbol{x}\right])(\boldsymbol{x} - \mathbb{E}\left[\boldsymbol{x}\right])^T\right] && (54)\\
&= \mathbb{E}\left[(\boldsymbol{W}\boldsymbol{h} + \text{noise})(\boldsymbol{h}^T\boldsymbol{W}^T + \text{noise}^T)\right] && (55)\\
&= \mathbb{E}\left[(\boldsymbol{W}\boldsymbol{h}\boldsymbol{h}^T\boldsymbol{W}^T)\right] + \boldsymbol{\psi} && (56)\\
&= \boldsymbol{W}\boldsymbol{W}^T + \boldsymbol{\psi} && (57)
\end{aligned}$$

  where we compute the expectation of $\boldsymbol{x}$ over $\boldsymbol{h}$ because $\boldsymbol{x}$ is defined as a function of $\boldsymbol{h}$, and noise is always expectation zero.

[24] Due to <MATH>, this introduces a constraint that knowing the value of some element x_j doesn't alter the probability $\Pr(x_i = W_i \cdot h + b_i + \text{noise})$. Given how we've defined the variable $\boldsymbol{h}$, this means that knowing noise_j provides no clues about noise_i. Mathematically, the noise must have a diagonal covariance matrix.

  - **Thoughts**. Not really seeing why this is useful/noteworthy. Feels very contrived (many assumptions) and restrictive – it only applies if the dependencies between each x_i can be modeled with a random variable $\boldsymbol{h}$ sampled from a unit variance Gaussian.

- **Probabilistic PCA**: Just factor analysis with $\boldsymbol{\psi} = \sigma^2 \boldsymbol{I}$. So zero-mean spherical Gaussian noise. It becomes regular PCA as $\sigma \to 0$. Here we can use an iterative EM algorithm for estimating the parameters $\boldsymbol{W}$.

Autoencoders (Ch. 14)

Written by Brandon McKinzie

Introduction. An autoencoder learns to copy its input to its output, via an encoder function $\boldsymbol{h} = f(\boldsymbol{x})$ and a decoder function $\boldsymbol{r} = g(\boldsymbol{h})$. Modern autoencoders generalize this to allow for stochastic mappings $p_{encoder}(\boldsymbol{h} \mid \boldsymbol{x})$ and $p_{decoder}(\boldsymbol{x} \mid \boldsymbol{h})$.

$\boldsymbol{r}$ for "reconstruction"

Undercomplete Autoencoders. Constrain dimension of $\boldsymbol{h}$ to be smaller than that of $\boldsymbol{x}$. The learning process minimizes some $L(\boldsymbol{x}, g(f(\boldsymbol{x})))$, where the loss function could be e.g. mean squared error. Be careful not to have too many learnable parameters in the functions g and f (thus increasing model capacity), since that defeats the purpose of using an undercomplete autoencoder in the first place.

Regularized Autoencoders. We can remove the undercomplete constraint/necessity by modifying our loss function. For example, a **sparse autoencoder** one that adds a penalty $\Omega(\boldsymbol{h})$ to the loss function that encourages the *activations on* (not connections to/from) the hidden layer to be sparse. One way to achieve *actual zeros* in $\boldsymbol{h}$ is to use rectified linear units for the activations.

Greedy Layer-Wise Unsupervised Pretraining. Given:

- Unsupervised learning algorithm $\mathcal{L}$ which accepts as input a training set of examples $\boldsymbol{X}$, and outputs an encoder/feature function f.
- $f^{(i)}(\tilde{\boldsymbol{X}})$ denotes the output of the ith layer of f, given as *immediate input* the (possibly transformed) set of examples $\tilde{\boldsymbol{X}}$.
- Let m denote the number of layers ("stages") in the encoder function (note that each layer/stage here *must* use a representation learning algorithm for its $\mathcal{L}$ e.g. an RBM, autoencoder, sparse coding model, etc.)

The procedure is as follows:

1. Initialize.

$$f(\cdot) \leftarrow I(\cdot) \tag{58}$$
$$\tilde{\boldsymbol{X}} = \boldsymbol{X} \tag{59}$$

2. For each layer (stage) i in range(m), do:

$$f^{(k)} = \mathcal{L}(\tilde{\boldsymbol{X}}) \tag{60}$$
$$f(\cdot) \leftarrow f^{(k)}\left(f\left(\cdot\right)\right) \tag{61}$$
$$\tilde{\boldsymbol{X}} \leftarrow f^{(k)}(\tilde{\boldsymbol{X}}) \tag{62}$$

   In English: just apply the regular learning/training process for each layer/stage **sequentially and individually**[25].

When this is complete, we can run **fine-tuning**: train all layers together (including any later layers that could not be pretrained) with a supervised learning algorithm. Note that we do indeed allow the pretrained encoding stages to be optimized here (i.e. not fixed).

[25] In other words, you proceed one layer at a time *in order*. You don't touch layer i until the weights in layer $i-1$ have been learned.

Motivation. In addition to classification, we can ask probabilistic models to perform other tasks such as density estimation ($\boldsymbol{x} \rightarrow p(\boldsymbol{x})$), denoising, missing value imputation, or sampling. What these [other] tasks have in common is they require a *complete understanding of the input*. Let's start with the most naive approach of modeling $p(\boldsymbol{x})$, where $\boldsymbol{x}$ contains n elements, each of which can take on k distinct values: we store a lookup table of all possible $\boldsymbol{x}$ and the corresponding probability value $p(\boldsymbol{x})$. This requires k^n parameters[26]. Instead, we use graphs to describe model structure (direct/indirect interactions) to drastically reduce the number of parameters.

Directed Models. Also called **belief networks** or **Bayesian networks**. Formally, a directed graphical model defined on a set of variables $\{\boldsymbol{x}\}$ is defined by a DAG, $\mathcal{G}$, whose vertices are the random variables in the model, and a set of **local conditional probability distributions**, $p(x_i \mid Pa_{\mathcal{G}}(x_i))$, where $Pa_{\mathcal{G}}(x_i)$ gives the parents of x_i in $\mathcal{G}$. The probability distribution over $\boldsymbol{x}$ is given by

$$p(\boldsymbol{x}) = \prod_i p(x_i \mid Pa_{\mathcal{G}}(x_i)) \tag{63}$$

Undirected Graphical Models. Also called **Markov Random Fields (MRFs)** or **Markov Networks**. Appropriate for situations where interactions do not have a well-defined direction. Each **clique** $\mathcal{C}$ (any set of nodes that are all [maximally] connected) in $\mathcal{G}$ is associated with a factor $\phi(\mathcal{C})$. The factor $\phi(\mathcal{C})$, also called a **clique potential**, is just a function (not necessarily a probability) that outputs a number when given a possible set of values over the nodes in $\mathcal{C}$. The output number measures the affinity of the variables in that clique for being in the states specified by the inputs. The set of all factors in $\mathcal{G}$ defines an **unnormalized probability distribution**:

Clique potentials are constrained to be nonnegative.

$$\tilde{p}(\boldsymbol{x}) = \prod_{\mathcal{C} \in \mathcal{G}} \phi(\mathcal{C}) \tag{64}$$

[26] Consider the common NLP case where our vector $\boldsymbol{x}$ contains n word tokens, each of which can take on any symbol in our vocabulary of size v. If we assign $n = 100$ and $v = 100,000$, which are relatively common values for this case, this amounts to $(1e5)^{1e2} = 10^{500}$ parameters.

The Partition Function. To obtain a valid probability distribution, we must normalize the probability distribution:

$$p(\boldsymbol{x}) = \frac{1}{Z}\widetilde{p}(\boldsymbol{x}) \tag{65}$$

$$Z = \int \widetilde{p}(\boldsymbol{x})\mathrm{d}\boldsymbol{x} \tag{66}$$

where the normalizing function $Z = Z(\{\phi\})$ is known as the **partition function** (physicz sh0ut0uT). It is typically intractable to compute, so we resort to approximations. Note that Z isn't even guaranteed to exist – it's only for those definitions of the clique potentials that cause the integral over $\widetilde{p}(\boldsymbol{x})$ to converge/be defined.

Energy-Based Models (EBMs). A convenient way to enforce $\forall \boldsymbol{x},\ \widetilde{p}(\boldsymbol{x}) > 0$ is to use EBMs, where

$$\widetilde{p}(\boldsymbol{x}) \triangleq \exp\left(-E(\boldsymbol{x})\right) \tag{67}$$

and $E(\boldsymbol{x})$ is known as the **energy function**[27]. Many algorithms need to compute not $p_{model}(\boldsymbol{x})$ but only $\log \widetilde{p}_{model}(\boldsymbol{x})$ (unnormalized log probabilities - logits!). For EBMs with latent variables $\boldsymbol{h}$, such algorithms are phrased in terms of the **free energy**:

$$\mathcal{F}(\boldsymbol{x} = x) = -\log \sum_h \exp\left(-E(\boldsymbol{x} = x,\ \boldsymbol{h} = h)\right) \tag{68}$$

where we sum over all possible assignments of the latent variables.

Separation and D-Separation. We want to know which subsets of variables are conditionally independent from each other, given the values of other subsets of variables. A set of variables $\mathbb{A}$ is **separated** (if undirected model)/**d-separated** (if directed model) from another set of variables $\mathbb{B}$ given a third set of variables $\mathbb{S}$ if the graph structure implies that $\mathbb{A}$ is independent from $\mathbb{B}$ given $\mathbb{S}$.

- **Separation**. For *undirected* models. If variables a and b are connected by a path involving only unobserved variables (an **active** path), then a and b are *not* separated. Otherwise, they are separated. Any paths containing at least one observed variable are called **inactive**.
- **D-Separation**[28]. For *directed* models. Although there are rules that help determine whether a path between a and b is d-separated, it is simplest to just determine whether a is independent from b given any observed variables along the path.

[27] Physics throwback: this mirrors the Boltzmann factor, $\exp(-\varepsilon/\tau)$, which is proportional to the probability of the system being in quantum energy state ε.

[28] The D stands for dependence.

Sampling from Graphical Models

For directed graphical models, we can do **ancestral sampling** to produce a sample $\boldsymbol{x}$ from the joint distribution represented by the model. Just sort the variables x_i into a topological ordering such that $\forall i, j : j > i \iff x_i \in Pa_{\mathcal{G}}(x_j)$. To produce the sample, just sequentially sample from the beginning, $x_1 \sim P(x_1)$, $x_2 \sim P(x_2 \mid Pa_{\mathcal{G}}(x_1))$, etc.

For undirected graphical models, one simple approach is **Gibbs sampling**. Essentially, this involves drawing a conditioned sample from $x_i \sim p(x_i \mid \text{neighbors}(x_i))$ for each x_i. This process is repeated many times, where each subsequent pass uses the previously sampled values in $\text{neighbors}(x_i)$ to obtain an asymptotically converging [to the correct distribution] estimate for a sample from $p(\boldsymbol{x})$.

Inference and Approximate Inference

One of the main tasks with graphical models is predicting the values of some subset of variables given another subset: inference. Although the graph structures we've discussed allow us to represent complicated, high-dimensional distributions with a reasonable number of parameters, the graphs used for deep learning are usually not restrictive enough to allow efficient inference. **Approximate inference** for deep learning usually refers to variational inference, in which we approximate the distribution $p(\boldsymbol{h} \mid \boldsymbol{v})$ by seeking an approximate distribution $q(\boldsymbol{h} \mid \boldsymbol{v})$ that is as close to the true one as possible.

Example: Restricted Boltzmann Machine. The quintessential example of how graphical models are used for deep learning. The canonical RBM is an energy-based model with **binary** visible and hidden units. Its energy function is

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\boldsymbol{b}^T\boldsymbol{v} - \boldsymbol{c}^T\boldsymbol{h} - \boldsymbol{v}^T\boldsymbol{W}\boldsymbol{h} \tag{69}$$

where $\boldsymbol{b}$, $\boldsymbol{c}$, and $\boldsymbol{W}$ are unconstrained, real-valued, learnable parameters. One could interpret the values of the bias parameters as the affinities for the associated variable being its given value, and the value $\boldsymbol{W}_{i,j}$ as the affinity of v_i being its value and h_j being its value at the same time[29].

The restrictions on the RBM structure, namely the fact that there are no intra-layer connections, yields nice properties. Since $\widetilde{p}(\boldsymbol{h}, \boldsymbol{v})$ can be factored into clique potentials, we can say

[29] More concretely, remember that $\boldsymbol{v}$ is a one-hot vector representing some state that can assume $\text{len}(\boldsymbol{v})$ unique values, and similarly for $\boldsymbol{h}$. Then $\boldsymbol{W}_{i,j}$ gives the affinity for the state associated with v being its ith value *and* the state associated with h being its jth value.

that:

$$p(\boldsymbol{h} \mid \boldsymbol{v}) = \prod_i p(h_i \mid \boldsymbol{v}) \tag{70}$$

$$p(\boldsymbol{v} \mid \boldsymbol{h}) = \prod_i p(v_i \mid \boldsymbol{h}) \tag{71}$$

Also, due to the restriction of binary variables, each of the conditionals is easy to compute, and can be quickly derived as

$$p(h_i = 1 \mid \boldsymbol{v}) = \sigma\left(c_i + \boldsymbol{v}^T \boldsymbol{W}_{:,i}\right) \tag{72}$$

allowing for efficient block Gibbs sampling.

Monte Carlo Sampling (Basics). We can approximate the value of a (usually prohibitively large) sum/integral by viewing it as an *expectation* under some distribution. We can then approximate its value by taking samples from the corresponding probability distribution and taking an empirical average. Mathematically, the basic idea is show below:

$$s = \int p(\boldsymbol{x}) f(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} = \mathbb{E}_p \left[f(\mathbf{x}) \right] \quad \rightarrow \quad \hat{s}_n = \frac{1}{n} \sum_{i=1,\ \mathbf{x}^{(i)} \sim p}^{n} f(\boldsymbol{x}^{(i)}) \tag{73}$$

As we've seen before, the empirical average is an unbiased[30] estimator. Furthermore, the central limit theorem tells us that the distribution of $\hat{s}_n$ converges to a normal distribution with mean s and variance $\mathrm{Var}\left[f(\boldsymbol{x}) \right] / n$.

Importance Sampling. What if it's not feasible for us to sample from p? We can approach this a couple ways, both of which will exploit the following identity:

$$p(\boldsymbol{x}) f(\boldsymbol{x}) = q(\boldsymbol{x}) \frac{p(\boldsymbol{x}) f(\boldsymbol{x})}{q(\boldsymbol{x})} \tag{77}$$

- **Optimal importance sampling**. We can use the aforementioned identity/decomposition to find the **optimal** $q*$ – optimal in terms of number of samples required to achieve a given level of accuracy. First, we rewrite our estimator $\hat{s}_p$ (they now use subscript to denote the sampling distribution) as $\hat{s}_q$:

$$\hat{s}_q = \frac{1}{n} \sum_{i=1,\ \mathbf{x}^{(i)} \sim q}^{n} \frac{p(\boldsymbol{x}^{(i)}) f(\boldsymbol{x}^{(i)})}{q(\boldsymbol{x}^{(i)})} \tag{78}$$

[30] Recall that expectations on such an average are still taken over the underlying (assumed) probability distribution:

$$\mathbb{E}_p \left[\hat{s}_n \right] = \frac{1}{n} \sum_{i=1}^{n} \mathbb{E}_p \left[f(\boldsymbol{x}^{(i)}) \right] \tag{74}$$

$$= \frac{1}{n} \sum_{i=1}^{n} s \tag{75}$$

$$= s \tag{76}$$

You should think of the expectation $\mathbb{E}_p \left[f(\boldsymbol{x}^{(i)}) \right]$ as the expected value of the *random sample* from the underlying distribution, which of course is s, because we defined it that way.

At first glance, it feels a little wonky, but recognize that we are *sampling from q instead of p* (i.e. if this were an integral, it would be over $q(\boldsymbol{x})\mathrm{d}\boldsymbol{x}$). The catch is that, now, the variance can be greatly sensitive to the choice of q:

$$\mathrm{Var}\left[\hat{s}_q\right] = \mathrm{Var}\left[\frac{p(\boldsymbol{x})f(\boldsymbol{x})}{q(\boldsymbol{x})}\right]/n \tag{79}$$

with the optimal (minimum) value of q at:

$$q* = \frac{p(\boldsymbol{x}) \mid f(\boldsymbol{x}) \mid}{Z} \tag{80}$$

- **Biased importance sampling**. Computing the optimal value of q can be as challenging/infeasible as sampling from p. Biased sampling does not require us to find a normalization constant for p or q. Instead, we compute:

$$\hat{s}_{BIS} = \frac{\sum_{i=1}^{n} \frac{\tilde{p}(\boldsymbol{x}^{(i)})}{\tilde{q}(\boldsymbol{x}^{(i)})} f(\boldsymbol{x}^{(i)})}{\sum_{i=1}^{n} \frac{\tilde{p}(\boldsymbol{x}^{(i)})}{\tilde{q}(\boldsymbol{x}^{(i)})}} \tag{81}$$

where $\tilde{p}$ and $\tilde{q}$ are the unnormalized forms of p and q, and the $\boldsymbol{x}^{(i)}$ samples are still drawn from [the original/unknown] q. $\mathbb{E}\left[\hat{s}_{BIS}\right] \neq s$ except asymptotically when $n \to \infty$.

Papers and Tutorials

Contents

Papers and Tutorials **January 15, 2017**

WaveNet

Table of Contents Local *Written by Brandon McKinzie*

Abstract.

- **What's WaveNet?** A deep neural network for generating raw audio waveforms.
- **How does it generate them? IDK**
- **What's it good for?** Text-to-speech, generating music, any waveform really.

Introduction.

- Inspired by recent advances in **neural autoregressive generative models**, and based on the PixelCNN architecture.
- Long-range dependencies dealt with via "dilated causal convolutions, which exhibit very large receptive fields."

WaveNet. The joint probability of a waveform $x = \{x_1, \ldots, x_T\}$ is factorised as a product of conditional probabilities,

$$p(x) = \prod_{t=1}^{T} p(x_t \mid x_1, \ldots, x_{t-1}) \tag{82}$$

which are modeled by a stack of convolutional layers (no pooling).

> The model outputs a categorical distribution over the next value xt with a softmax layer and it is optimized to maximize the log-likelihood of the data w.r.t. the parameters.

Main ingredient of WaveNet is *dilated* causal convolutions, illustrated below. Note the absence of recurrent connections, which makes them faster to train than RNNs, but at the cost of requiring many layers, or large filters to increase the receptive field[31].

[31] Loose interpretation of receptive fields here is that large fields can take into account more info (back in time) as opposed to smaller fields, which can be said to be "short-sighted"

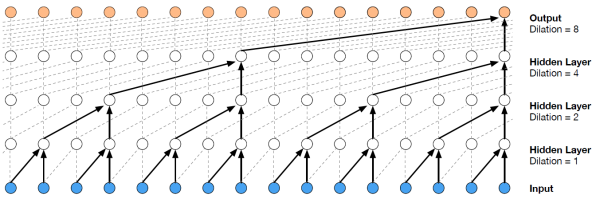


Figure 3: Visualization of a stack of *dilated* causal convolutional layers.

Excellent concise definition from paper:

> A dilated convolution (a convolution with holes) is a convolution where the filter is applied over an area larger than its length by skipping input values with a certain step. It is equivalent to a convolution with a larger filter derived from the original filter by dilating it with zeros, but is significantly more efficient. A dilated convolution effectively allows the network to operate on a coarser scale than with a normal convolution. This is similar to pooling or strided convolutions, but here the output has the same size as the input. As a special case, dilated convolution with dilation 1 yields the standard convolution.

Softmax distributions. Chose to model the conditional distributions $p(x_t \mid x_1, \dots, x_{t-1})$ with a softmax layer. To deal with the fact that there are 2^{16} possible values, first apply a "μ-law companding transformation" to data, and then quantize it to 256 possible values:

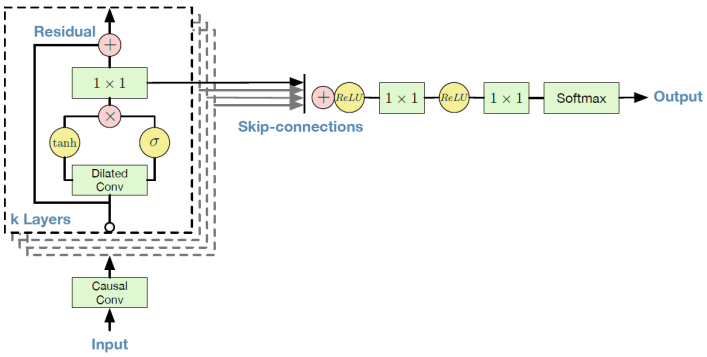
$$f(x_t) = \text{sign}(x_t)\frac{\ln(1+\mu|x_t|)}{\ln(1+\mu)} \tag{83}$$

which (after plotting in Wolfram) looks identical to the sigmoid function.

Gated activation and res/skip connections. Use the same gated activation unit as PixelCNN:

$$z = \tanh\left(W_{f,k} * x\right) \;\odot\; \sigma\left(W_{g,k} * x\right) \tag{84}$$

where $*$ denotes conv operator, $\odot$ denotes elem-wise mult., k is layer index, f, g denote filter/-gate, and W is learnable conv filter. This is illustrated below, along with the residual/skip connections used to speed up convergence/enable training deeper models.



Conditional Wavenets. Can also model conditional distribution of x given some additional h (e.g. speaker identity).

$$p(x \mid h) = \prod_{t=1}^{T} p(x_t \mid x_1, \ldots, x_{t-1}, h) \tag{85}$$

→ **Global conditioning**. Single h that influences output dist. accross all times. Activation becomes:

$$z = \tanh\left(W_{f,k} * x + V_{f,k}^T h\right) \odot \sigma\left(W_{g,k} * x + V_{g,k}^T h\right) \tag{86}$$

→ **Local conditioning** (confusing). Have a second time-series h_t. They first transform this h_t using a "transposed conv net (learned unsampling) that maps it to a new time-series $y = f(h)$ w/same resolution as x.

Experiments.

- **Multi-Speaker Speech Generation**. Dataset: multi-speaker corpus of 44 hours of data from 109 different speakers[32]. Receptive field of 300 milliseconds.
- **Text-to-Speech**. Single-speaker datasets of 24.6 hours (English) and 34.8 hours (Chinese) speech. Locally conditioned on *linguistic features*. Receptive field of 240 milliseconds. Outperformed both LSTM-RNN and HMM.
- **Music**. Trained the WaveNets to model two music datasets: (1) 200 hours of annotated music audio, and (2) 60 hours of solo piano music from youtube. Larger receptive fields sounded more musical.
- **Speech Recognition**. "With WaveNets we have shown that layers of dilated convolutions allow the receptive field to grow longer in a much cheaper way than using LSTM units."

[32]Speakers encoded as ID in form of a one-hot vector

Conclusion (verbatim): "This paper has presented WaveNet, a deep generative model of audio data that operates directly at the waveform level. WaveNets are autoregressive and combine causal filters with dilated convolutions to allow their receptive fields to grow exponentially with depth, which is important to model the long-range temporal dependencies in audio signals. We have shown how WaveNets can be conditioned on other inputs in a global (e.g. speaker identity) or local way (e.g. linguistic features). When applied to TTS, WaveNets produced samples that outperform the current best TTS systems in subjective naturalness. Finally, WaveNets showed very promising results when applied to music audio modeling and speech recognition."

Neural Style

Written by Brandon McKinzie

Notation.

- **Content image**: $\boldsymbol{p}$
- **Filter responses**: the matrix $P^l \in \mathcal{R}^{N_l \times M_l}$ contains the activations of the filters in layer l, where P^l_{ij} would give the activation of the ith filter at position j in layer l. N_l is number of feature maps, each of size M_l (height $\times$ width of the feature map)[33].
- **Reconstructed image**: $\boldsymbol{x}$ (initially random noise). Denote its corresponding filter response matrix at layer l as P^l.

Content Reconstruction.

1. Feed in **content image** $\boldsymbol{p}$ into pre-trained network, saving any desired filter responses during the forward pass. These are interpreted as the various "encodings" of the image done by the network. Think of them analogously to "ground-truth" labels.

2. Define $\boldsymbol{x}$ as the **generated image**, which we first initialize to random noise. We will be changing the pixels of $\boldsymbol{x}$ via gradient descent updates.

3. Define the **loss function**. After each forward pass, evaluate with squared-error loss between the two representations at the layer of interest:

$$\mathcal{L}_{content}(\boldsymbol{p}, \boldsymbol{x}, l) = \frac{1}{2}\sum_{i,j}(F^l_{ij} - P^l_{ij})^2 \tag{1}$$

$$\frac{\partial \mathcal{L}_{content}}{\partial F^l_{ij}} = \begin{cases} (F^l - P^l)_{ij} & F^l_{ij} > 0 \\ 0 & F^l_{ij} < 0 \end{cases} \tag{2}$$

   where it appears we are assuming ReLU activations (**?**).

4. Compute iterative updates to $\boldsymbol{x}$ via **gradient descent** until it generates the same response in a certain layer of the CNN as the original image $\boldsymbol{p}$.

[33] If not clear, M_l is a scalar, for any given value of l.

Style Representation. On top of the CNN responses in each layer, the authors built a style representation that computes the correlations between the different [aforementioned] filter responses. The correlation matrix for layer l is denoted in the standard way with a Gram matrix $G^l \in \mathcal{R}^{N_l \times N_l}$, with entries

$$G^l_{ij} = \langle F^l_i, F^l_j \rangle = \sum_k F^l_{ik} F^l_{jk} \tag{3}$$

To generate a texture that matches the style of a given image, do the following.

1. Let $\boldsymbol{a}$ denote the original [style] image, with corresponding A^l. Let $\boldsymbol{x}$, initialized to random noise, denote the generated [style] image, with corresponding G^l.
2. The contribution to the loss of layer l, denoted E_l, to the total loss, denoted $\mathcal{L}_{style}$, is given by

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{ij} (G^l_{ij} - A^l_{ij})^2 \tag{4}$$

$$\mathcal{L}_{style}(\boldsymbol{a}, \boldsymbol{x}) = \sum_{l=0}^{L} w_l E_l \tag{5}$$

$$\frac{\partial E_l}{\partial F^l_{ij}} = \begin{cases} \frac{1}{N_l^2 M_l^2} \left((F^l)^T (G^l - A^l) \right)_{ji} & F^l_{ij} > 0 \\ 0 & F^l_{ij} < 0 \end{cases} \tag{6}$$

   where w_l are [as of yet unspecified] weighting factors of the contribution of layer l to the total loss.

Mixing content with style. Essentially a joint minimization that combines the previous two main ideas.

1. Begin with the following images: white noise $\boldsymbol{x}$, content image $\boldsymbol{p}$, and style image $\boldsymbol{a}$.
2. The loss function to minimize is a linear combination of 1 and 5:

$$\mathcal{L}_{total}(\boldsymbol{p}, \boldsymbol{a}, \boldsymbol{x}, l) = \alpha \mathcal{L}_{content}(\boldsymbol{p}, \boldsymbol{x}, l) + \beta \mathcal{L}_{style}(\boldsymbol{a}, \boldsymbol{x}) \tag{7}$$

   Note that we can choose which layers L_{style} uses by tweaking the layer weights w_l. For example, the authors chose to set $w_l = 1/5$ for 'conv[1, 2, 4, 5]_1' and 0 otherwise. For the ratio α/β, they explored 1×10^{-3} and 1×10^{-4}.

Papers and Tutorials **February 8**

Neural Conversation Model

Table of Contents Local *Written by Brandon McKinzie*

[Reminder: **red text** means I need to come back and explain what is meant, once I understand it.]

Abstract. This paper presents a simple approach for conversational modeling which uses the sequence to sequence framework. It can be trained end-to-end, meaning fewer hand-crafted rules. The **lack of consistency** is a common failure of our model.

Introduction. Major advantage of the seq2seq model is it requires little feature engineering and domain specificity. Here, the model is tested on chat sessions from an IT helpdesk dataset of conversations, as well as movie subtitles.

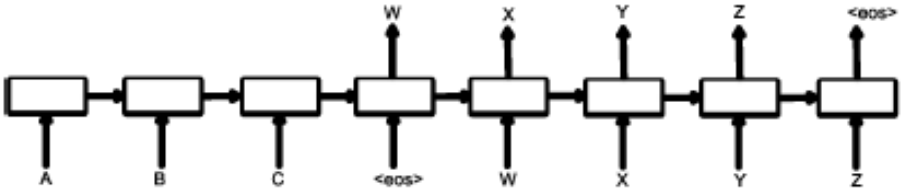
Related Work. The authors' approach is based on the following (linked and saved) papers on seq2seq:

- Kalchbrenner & Blunsom, 2013.
- Sutskever et al., 2014. (Describes Seq2Seq model)
- Bahdanau et al., 2014.

Model. Succinctly described by the authors:

> The model reads the input sequence one token at a time, and predicts the output sequence, also one token at a time. During training, the true output sequence is given to the model, so learning can be done by backpropagation. The model is trained to maximize the cross entropy of the correct sequence given its context. During inference, in which the true output sequence is not observed, we simply feed the predicted output token as input to predict the next output. This is a âĂIJgreedyâĂİ inference approach.

Example of less greedy approach: **beam search**.



The **thought vector** is the hidden state of the model when it receives [as input] the end of sequence symbol $\langle eos \rangle$, because it stores the info of the sentence, or *thought*, "ABC". The authors acknowledge that this model will *not* be able to "solve" the problem of modeling dialogue due to the objective function not capturing the actual objective achieved through human communication, which is typically longer term and based on exchange of information [rather than next step prediction][34].

Ponder: what *would* be a reasonable objective function & model for conversation?

IT Data & Experiment.

Reminder: Check out this git repo

- **Data Description**: Customers talking to IT support, where typical interactions are 400 words long and turn-taking is clearly signaled.
- **Training Data**: 30M tokens, 3M of which are used as validation. They built a vocabulary of the most common 20K words, and introduced special tokens indicating turn-taking and actor.
- **Model**: A single-layer LSTM with 1024 memory cells.
- **Optimization**: SGD with gradient clipping.
- **Perplexity**: At convergence, achieved **perplexity** of 8, whereas an n-gram model achieved 18.

[34] I'd imagine that, in order to model human conversation, one obvious element needed would be a *memory*. Reminds me of DeepMind's DNC. There would need to be some online filtering & output process to capture the crucial aspects/info to store in memory for later, and also some method of retrieving them when needed later. The method for retrieval would likely be some inference process where, given a sequence of inputs, the probability of them being related to some portion of memory could be trained. This would allow for conversations that stretch arbitrarily back in the past. Also, when storing the memories, I'd imagine a reasonable architecture would be some encoder-decoder for a sparse distributed representation of memory.

NMT By Jointly Learning to Align & Translate

Written by Brandon McKinzie

[Bahdanau et. al, 2014]. The primary motivation for me writing this is to better understand the **attention mechanism** in my sequence to sequence chatbot implementation.

Abstract. The authors claim that using a fixed-length vector [in the vanilla encoder-decoder for NMT] is a bottleneck. They propose allowing a model to (soft-)search for parts of a source sentence that are relevant to predicting a target word, without having to form these parts as a hard segment explicitly.

Learning to Align[35] and translate.

- **Decoder**. Their encoder defines the conditional output distribution as

$$p(y_i \mid y_1, \dots, y_{i-1}, \boldsymbol{x}) = g(y_{i-1}, s_i, c_i) \tag{87}$$

$$s_i = f(s_{i-1}, y_{i-1}, c_i) \tag{88}$$

  where s_i is the RNN [decoder] hidden state at time i.

  - NOTE: c_i is *not* the ith element of the standard context vector; rather, it is *itself* a distinct context vector that depends on a sequence of **annotations** $(h_1, \dots, h_{T_x})$. It seems that each annotation h_i is a hidden (encoder) state "that contains information about the whole input sequence with a strong focus on the parts surrounding the i-th word of the input sequence."
  - The context vector c_i is computed as follows:

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j \tag{89}$$

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})} \tag{90}$$

$$e_{ij} = a(s_{i-1}, h_j) \tag{91}$$

    where the function e_{ij} is given by an **alignment model** which scores how well the inputs around position j and the output at position i match.

- **Encoder**. It's just a bidirectional RNN. What they call "annotation h_j" is literally just a concatenated vector of $h_j^{forward}$ and $h_j^{backward}$

[35] By "align" the authors are referring to aligning the source-search to the relevant parts for prediction.

(Appendix A). Explained with the TensorFlow user in mind.

Decoder Internals. It's just a GRU. However, it will be helpful to detail how we format the inputs (given we now have attention). Wherever we'd usually pass the previous decoder state s_{i-1}, we now pass a *concatenated* state, $[s_{i-1},\ c_i]$, that also contains the ith context vector. Below I go over the flow of information from GRU input to output:

1. **Notation**: y_t is the loop-embedded output of the decoder (prediction) at time t, s_t is the internal hidden state of the decoder at time t, and c_t is the context vector at time t. $\tilde{s}_t$ is the proposed/proposal state at time t.
2. **Gates**:

$$z_t = \sigma\left(W_z y_{t-1} + U_z[s_{t-1},\ c_t]\right) \qquad \text{[update gate]} \tag{92}$$
$$r_t = \sigma\left(W_r y_{t-1} + U_r[s_{t-1},\ c_t]\right) \qquad \text{[reset gate]} \tag{93}$$
$$\tag{94}$$

3. **Proposal state**:

$$\tilde{s}_t = \tanh\left(W y_{t-1} + U[r_t \circ s_{t-1},\ c_t]\right) \tag{95}$$

4. **Hidden state**:

$$s_t = (1 - z_t) \circ s_{t-1} + z_t \circ \tilde{s}_t \tag{96}$$

Alignment Model. All equations enumerated below are for some timestep t during the decoding process.

1. **Score**: For all $j \in [0, L_{enc}-1]$ where L_{enc} is the number of words in the encoder sequence, compute:

$$a_j = a(s_{t-1}, h_j) = v_a^T \tanh\left(W_a s_{t-1} + U_a h_j\right) \tag{97}$$

2. **Alignments**: Feed the unnormalized alignments (scores) through a softmax so they represent a valid probability distribution.

$$a_j \leftarrow \frac{e^{a_j}}{\sum_{k=0}^{L_{enc}-1} e^{a_k}} \tag{98}$$

3. **Context**: The context vector input for our decoder at this timestep:

$$c = \sum_{j=1}^{L_{enc}} a_j h_j \tag{99}$$

Decoder Outputs. All below is for some timestep t during the decoding process. To find the probability of some (one-hot) word y [at timestep t]:

$$\Pr(y \mid s, c) \propto e^{y^T W_o u} \tag{100}$$

$$u = [\max\{\tilde{u}_{2j-1}, \tilde{u}_{2j}\}]_{j=1,\ldots,\ell}^{T} \tag{101}$$

$$\tilde{u} = U_o[s_{t-1},\ c] + V_o y_{t-1} \tag{102}$$

N.B.: From reading other (and more recent) papers, these last few equations do not appear to be the way it is usually done (thank the lord). See Luong's work for a much better approach.

[Luong et. al, 2015]

Attention-Based Models. For attention especially, the devil is in the details, so I'm going to go into somewhat excruciating detail here to ensure no ambiguities remain. For both global and local attention, the following information holds true:

- "At each time step t in the decoding phase, both approaches first take as input the hidden state $\boldsymbol{h}_t$ at the top layer of a stacking LSTM."
- Then, they derive [with different methods] a context vector $\boldsymbol{c}_t$ to capture source-side info.
- Given $\boldsymbol{h}_t$ and $\boldsymbol{c}_t$, they both compute the **attentional hidden state** as:

$$\tilde{\boldsymbol{h}}_t = \tanh\left(\boldsymbol{W}_c[\boldsymbol{c}_t;\ \boldsymbol{h}_t]\right) \tag{103}$$

- Finally, the predictive distribution (decoder outputs) is given by feeding this through a softmax:

$$p(y_t \mid y_{<t}, x) = \text{softmax}\left(\boldsymbol{W}_s \tilde{\boldsymbol{h}}_t\right) \tag{104}$$

Global Attention. Now I'll describe in detail the processes involved in $\boldsymbol{h}_t \to \boldsymbol{a}_t \to \boldsymbol{c}_t \to \tilde{\boldsymbol{h}}_t$.

1. $\boldsymbol{h}_t$: Compute the hidden state $\boldsymbol{h}_t$ in the normal way (not obvious if you've read e.g. Bahdanau's work...)
2. $\boldsymbol{a}_t$:
   (a) Compute the **scores** between $\boldsymbol{h}_t$ and each source $\bar{\boldsymbol{h}}_s$, where our options are:

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^T \bar{\boldsymbol{h}}_s & \textit{dot} \\ \boldsymbol{h}_t^T \boldsymbol{W}_a \bar{\boldsymbol{h}}_s & \textit{general} \\ \boldsymbol{v}_a^T \tanh\left(\boldsymbol{W}_a[\boldsymbol{h}_t;\ \bar{\boldsymbol{h}}_s]\right) & \textit{concat} \end{cases} \tag{105}$$

   (b) Compute the **alignment vector** $\boldsymbol{a}_t$ of length L_{enc} (number of words in the encoder sequence):

$$\boldsymbol{a}_t(s) = \text{align}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) \tag{106}$$

$$= \frac{\exp(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s))}{\sum_{s'} \exp(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_{s'}))} \tag{107}$$

3. $\boldsymbol{c}_t$: The weighted average over all source (encoder) hidden states[36]:

$$\boldsymbol{c}_t = \sum_{i=1}^{L_{enc}} \boldsymbol{a}_t(i)\bar{\boldsymbol{h}}_i \tag{108}$$

4. $\tilde{\boldsymbol{h}}_t$: For convenience, I'll copy the equation for $\tilde{\boldsymbol{h}}_t$ again here:

$$\tilde{\boldsymbol{h}}_t = \tanh\left(\boldsymbol{W}_c[\boldsymbol{c}_t;\ \boldsymbol{h}_t]\right) \tag{109}$$

Input-Feeding Approach. A copy of each output $\tilde{\boldsymbol{h}}_t$ is sent forward and concatenated with the inputs for the next timestep, i.e. the inputs go from $\boldsymbol{x}_{t+1}$ to $[\tilde{\boldsymbol{h}}_t; \boldsymbol{x}_{t+1}]$.

[36]NOTE: Right after mentioning the context vector, the authors have the following cryptic footnote that may be useful to ponder: *For short sentences, we only use the top part of a_t and for long sentences, we ignore words near the end.*

Using Large Vocabularies for NMT

 Written by Brandon McKinzie

Paper information:

- Full title: On Using Very Large Target Vocabulary for Neural Machine Translation.
- Authors: Jean, Cho, Memisevic, Bengio.
- Date: 18 Mar 2015.
- [arXiv link]

NMT Overview. Typical implementation is encoder-decoder network. Notation for inputs & encoder:

$$x = (x_1, \dots, x_T) \quad \text{[source sentence]} \tag{110}$$
$$h = (h_1, \dots, h_T) \quad \text{[encoder state seq]} \tag{111}$$
$$h_t = f(x_t, h_{t-1}) \tag{112}$$

where f is the function defined by the *cell state* (e.g. GRU/LSTM/etc.). Then the decoder generates the output sequence y, and with probability given below:

$$y = (y_1, \dots, y'_T) \quad [y_i \in \mathbb{Z}] \tag{113}$$
$$\Pr[y_t \mid y_{<t},\ x] \propto e^{q(y_{t-1},\ z_t,\ c_t)} \tag{114}$$
$$z_t = g(y_{t-1}, z_{t-1}, c_t) \quad \text{[decoder hidden?]} \tag{115}$$
$$c_t = r(z_{t-1}, h_1, \dots, h_T) \quad \text{[decoder inp?]} \tag{116}$$

The functions q, g, and r are just placeholders – "some function of [inputs]."

As usual, model is jointly trained to maximize the conditional log-likelihood of correct translation. For N training sample pairs (x^n, y^n), and denoting the length of the n-th target sentence as T_n, this can be written as,

$$\theta^* = \arg\max_\theta \sum_{n=1}^{N} \sum_{t=1}^{T_n} \log\left(\Pr[y_t^n \mid y_{<t}^n,\ x^n]\right) \tag{117}$$

Model Details. Above is the general structure. Here I'll summarize the specific model chosen by the authors.

- **Encoder**. Bi-directional, which just means $h_t = \left[h_t^{backward};\ h_t^{forward}\right]$. The chosen cell state (the function f) is GRU.
- **Decoder**. At each timestep, computes the following:
  → **Context vector** c_t.

$$c_t = \sum_{i=1}^{T} \alpha_i h_i \tag{118}$$

$$\alpha_t = \frac{e^{a(h_t, z_{t-1})}}{\sum_k e^{a(h_k, z_{t-1})}} \tag{119}$$

a is a standard single-hidden-layer NN.

  → **Decoder hidden state** z_t. Also a GRU cell. Computed based on the previous hidden state z_{t-1}, the previously generated symbol y_{t-1}, and also the computed context vector c_t.
- **Next-word probability**. They model equation 114 as[37] ,

$$\Pr[y_t \mid y_{<t},\ x] = \frac{1}{Z} e^{\mathbf{w}_t^T \phi(y_{t-1}, z_t, c_t) + b_t} \tag{120}$$

$$Z = \sum_{k:\ y_k \in V} e^{\mathbf{w}_k^T \phi(y_{t-1}, z_t, c_t) + b_k} \tag{121}$$

Reminder: y_i is an integer token, while $\mathbf{w}_i$ is the target vector of length vocab size

  where ϕ is affine transformation followed by a nonlinear activation, $\mathbf{w}_t$ and b_t are the **target word vector** and bias. V is the set of all target *vocabulary*.

Approximate Learning Approach. Main idea:

> "In order to avoid the growing complexity of computing the normalization constant, we propose here to use only a small subset V' of the target vocabulary at each update."

Consider the gradient of the log-likelihood[38], written in terms of the energy $\mathcal{E}$.

$$\nabla \log\left(\Pr[y_t \mid y_{<t},\ x]\right) = \nabla \mathcal{E}(y_t) - \sum_{k:\ y_k \in V} \Pr[y_k \mid y_{<t},\ x] \nabla \mathcal{E}(y_k) \tag{122}$$

$$\mathcal{E}(y_j) = \mathbf{w}_j^T \phi(y_{t-1}, z_t, c_t) + b_j \tag{123}$$

[37] Note: The formula for Z is correct. Notice that the only part of the RHS of $\Pr(y_t)$ with a t is as the subscript of w. To be clear, w_k is a full word vector and the sum is over all words in the output *vocabulary*, the index k has absolutely nothing to do with timestep. They use the word target but make sure not to misinterpret that as somehow meaning target words in the sentence or something.

[38] **NOTE TO SELF**: After long and careful consideration, I'm concluding that the authors made a typo when defining $\mathcal{E}(y_j)$, which they choose to subscript all parts of the RHS with j, but that is in direct contradiction with a step-by-step derivation, which is why I have written it the way it is. I'm pretty sure my version is right, but I know you'll have to re-derive it yourself next time you see this. And you'll somehow prove me wrong. Actually, after reading on further, I doubt you'll prove me wrong. Challenge accepted, me. Have fun!

The crux of the approach is interpreting the second term as $\mathbb{E}_P[\nabla\mathcal{E}(y)]$, where P denotes $Pr(y \mid y_{<t}, x)$. They approximate this expectation by taking it over a subset V' of the predefined proposal distribution Q. So Q is a p.d.f. over the possible y_i, and we sample *from* Q to generate the elements of the subset V'.

$$\mathbb{E}_P[\nabla\mathcal{E}(y)] \approx \sum_{k:\ y_k \in V'} \frac{\omega_k}{\sum_{k':y_{k'}\in V'} \omega_{k'}} \nabla\mathcal{E}(y_k) \tag{124}$$

$$\omega_k = e^{\mathcal{E}(y_k) - \log Q(y_k)} \tag{125}$$

Here is some math I did that was illuminating to me; I'm not sure why the authors didn't point out these relationships.

$$\omega_k = \frac{e^{\mathcal{E}(y_k)}}{Q(y_k)} \quad \text{thus} \quad p(y_k \mid y_{<t}, x) = \omega_k \frac{Q(y_k)}{Z} \tag{126}$$

$$\rightarrow \quad e^{\mathcal{E}(y_k)} = Z \cdot p(y_k \mid y_{<t}, x) = Q(y_k) \cdot \omega_k \tag{127}$$

Now *check this out*

Below are the exact and approximate formulas for $\mathbb{E}_P[\nabla\mathcal{E}(y)]$ written in a ~~seductive~~ suggestive manner. Pay careful attention to subscripts and primes.

$$\mathbb{E}_P[\nabla\mathcal{E}(y)] = \sum_{k:\ y_k \in V} \frac{\omega_k \cdot Q(y_k)}{\sum_{k':y_{k'}\in V} \omega_{k'} \cdot Q(y_{k'})} \nabla\mathcal{E}(y_k) \tag{128}$$

$$\mathbb{E}_P[\nabla\mathcal{E}(y)] = \sum_{k:\ y_k \in V'} \frac{\omega_k}{\sum_{k':y_{k'}\in V'} \omega_{k'}} \nabla\mathcal{E}(y_k) \tag{129}$$

They're almost the same! It's much easier to see why when written this way. I interpret the difference as follows: in the exact case, we explicitly attach the probabilities $Q(y_k)$ and sum over all values in V. In the second case, by sampling a subset V' from Q, we have encoded these probabilities implicitly as the relative frequency of elements y_k in V'

How to do in practice (very important).

> "In practice, we partition the training corpus and define a subset V' of the target vocabulary for each partition prior to training. Before training begins, we sequentially examine each target sentence in the training corpus and accumulate unique target words until the number of unique target words reaches the predefined threshold τ. The accumulated vocabulary will be used for this partition of the corpus during training. We repeat this until the end of the training set is reached. Let us refer to the subset of target words used for the i-th partition by V'_i.

Candidate Sampling – TensorFlow

Written by Brandon McKinzie

[Link to article]

What is Candidate Sampling The goal is to learn a compatibility function $F(x, y)$ which says something about the compatibility of a class y with a context x. Candidate sampling: for each training example (x_i, y_i), only need to evaluate $F(x, y)$ for a small set of classes $\{C_i\} \subset \{L\}$, where $\{L\}$ is the set of all possible classes (vocab size number of elements). We represent $F(x, y)$ as a *layer that is trained by back-prop from/within the loss function.*

C.S. for Sampled Softmax. I'll further narrow this down to my use case of having exactly 1 target class (word) at a given time. Any other classes are referred to as **negative** classes (for that example).

Sampling algorithm. For each training example (x_i, y_i), do:

- Sample the subset $S_i \subset L$. How? By sampling from $Q(y|x)$ which gives the probability of any particular y being included in S_i.
- Create the set of **candidates**, which is just $C_i := S_i \cup y_i$.

Training task. We are given this set C_i and want to find out which element of C_i is the target class y_i. In other words, we want the posterior probability that any of the y in C_i are the target class, given what we know about C_i and x_i. We can evaluate and rearrange as usual with Bayes' rule to get:

$$\Pr\left(y_i^{true} = y \mid C_i,\ x_i\right) = \frac{\Pr\left(y_i^{true} = y \mid x_i\right) \cdot \Pr\left(C_i \mid y_i^{true} = y,\ x_i\right)}{\Pr\left(C_i \mid x_i\right)} \tag{130}$$

$$= \frac{\Pr\left(y \mid x_i\right)}{Q\left(y \mid x_i\right)} \cdot \frac{1}{K(x_i, C_i)} \tag{131}$$

where they've just defined

$$K(x_i, C_i) \triangleq \frac{\Pr\left(C_i \mid x_i\right)}{\prod_{y' \in C_i} Q(y' \mid x_i) \prod_{y' \in (L - C_i)} \left(1 - Q(y' \mid x_i)\right)} \tag{132}$$

Clarifications.

- The learning function $F(x, y)$ is the *input* to our softmax. It is our neural network, excluding the softmax function.
- After training our network, it should have learned the general form

$$F(x, y) = \log(\Pr(y \mid x)) + K(x) \tag{133}$$

which is the general form because

$$\text{Softmax}(\log(\Pr(y \mid x)) + K(x)) = \frac{e^{\log(\Pr(y|x))+K(x)}}{\sum_{y'} e^{\log(\Pr(y'|x)+K(x)}} \tag{134}$$

$$= \Pr(y \mid x) \tag{135}$$

Note that I've been a little sloppy here, since $\Pr(y \mid x)$ up until the last line actually represented the (possibly) unnormalized/*relative* probabilities.

- [MAIN TAKEAWAY]. Time to bring it all together. Notice that we've only trained $F(x, y)$ to include *part* of what's needed to compute the probability of any y being the target given x_i and C_i ...equation 133 doesn't take into account C_i at all! Luckily we know the form of the full equation because it just the log of equation 131. We can easily satisfy that by subtracting $\log(Q(y \mid x))$ from $F(x, y)$ right before feeding into the softmax.

TL;DR. Train network to learn $F(x, y)$ before softmax, but instead of feeding $F(x, y)$ to softmax directly, feed

$$\text{Softmax Input: } F(x, y) - \log(Q(y \mid x)) \tag{136}$$

instead. That's it.

Attention Terminology

 Written by Brandon McKinzie

Generally useful info. Seems like there are a few notations floating around, and here I'll attempt to set the record straight. The order of notes here will loosely correspond with the order that they're encountered going from encoder output to decoder output.

Jargon. The people in the attention business *love* obscure names for things that don't need names at all. Terminology:

- **Attentions keys/values**: Encoder output sequence.
- **Query**: Decoder [cell] state. Typically the most recent one.
- **Scores**: Values of e_{ij}. For the Bahdanau version, in code this would be computed via

$$e_i = v^T \tanh(\text{FC}(s_{i-1}) + \text{FC}(h)) \tag{137}$$

  where we'd have FC be tf.layers.fully_connected with num_outputs equal to our attention size (up to us). Note that v is a vector.
- **Alignments**: output of the softmax layer on the attention scores.
- **Memory**: The α matrix in the equation $c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$.

When someone lazily calls some layer output the "attention", they are usually referring to the layer *just after* the linear combination/map of encoder hidden states. You'll often see this as some vague function of the previous decoder state, context vector, and possibly even decoder output (after project), like $f(s_{i-1}, y_{i-1}, c_i)$. In 99.9% of cases, this function is just a fully connected layer (if even needed) to map back to the state size for decoder input. That is it.

From encoder to decoder. The path of information flow from encoder outputs to decoder inputs, a non-trivial process that isn't given the *attention* (heh) it deserves[39]

1. **Encoder outputs**. Tensor of shape **[batch size, sequence length, state size]**. The state is typically some RNNCell state.
   - Note: TensorFlow's AttenntionMechanism classes will actually convert this to **[batch size, L_{enc}, attention size]**, and refer to it as the "memory". It is also what is returned when calling myAttentionMech.values.

[39] For some reason, the literature favors explaining the path "backwards", starting with the highly abstracted "decoder inputs as a weighted sum of encoder states" and then breaking down what the weights are. Unfortunately, the weights are computed via a multi-stage process so that becomes very confusing very quick.

2. **Compute the scores**. The attention scores are the computation described by Luong/Bahdanau techniques. They both take an inner product of sorts on *copies* of the encoder outputs and decoder previous state (query). The main choices are:

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^T \bar{\boldsymbol{h}}_s & \textit{dot} \\ \boldsymbol{h}_t^T \boldsymbol{W}_a \bar{\boldsymbol{h}}_s & \textit{general} \\ \boldsymbol{v}_a^T \tanh\left(\boldsymbol{W}_a[\boldsymbol{h}_t;\ \bar{\boldsymbol{h}}_s]\right) & \textit{concat} \end{cases} \tag{138}$$

Synonyms:
- scores
- unnormalized alignments

where the shapes are as follows (for single timestep during decoding process):

- $\bar{\boldsymbol{h}}_s$: **[batch size, 1, state size]**
- $\boldsymbol{h}_t$: **[batch size, 1, state size]**
- $\boldsymbol{W}_a$: **[batch size, state size, state size]**
- $\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s)$: **[batch size]**

Synonyms:
- softmax outputs
- attention dist.
- alignments

3. **Softmax the scores**. In the vast majority of cases, the attention scores are next fed through a softmax to convert them into a valid probability distribution. Most papers will call this some vague probability function, when in reality they are using softmaxonly.

$$\boldsymbol{a}_t(s) = \text{align}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) \tag{139}$$

$$= \frac{\exp(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s))}{\sum_{s'} \exp(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_{s'}))} \tag{140}$$

where the alignment vector $\boldsymbol{a}_t$ has shape **[batch size,** L_{enc}**]**

4. **Compute the context vector**. The inner product of the softmax outputs and the raw encoder outputs. This will have shape **[batch size, attention size]** in TensorFlow, where attention size is from the constructor for your AttentionMechanism.

Synonyms:
- context vector
- attention

5. **Combine context vector and decoder output**: Typically with a concat. *The result is what people mean when they say "attention".* Luong et al. denotes this as $\tilde{\boldsymbol{h}}_t$, the decoder output at timestep t. This is what TensorFlow means by "Luong-style mechanisms output the attention." And yes, these are used (at least for Luong) to compute the prediction:

$$\tilde{\boldsymbol{h}}_t = \tanh\left(\boldsymbol{W}_c\left[\boldsymbol{c}_t,\ \boldsymbol{h}_t\right]\right) \tag{141}$$

$$p(y_t \mid y_{<t}, x) = \text{softmax}(\boldsymbol{W}_s \tilde{\boldsymbol{h}}_t) \tag{142}$$

TextRank

Written by Brandon McKinzie

Introduction. A graph-based ranking algorithm is a way of deciding on the importance of a vertex within a graph, by taking into account global information recursively computed from the entire graph, rather than relying only on local vertex-specific information. **TextRank** is a graph-based ranking model for graphs extracted from natural language texts. The authors investigate/evaluate TextRank on unsupervised keyword and sentence extraction.

Semantic graph: one whose structure encodes meaning between the nodes (semantic elements).

The ~~TextRank~~ PageRank Model. In general [graph-based ranking], a vertex can be ranked based on certain properties such as the number of vertices pointing to it (in-degree), how highly-ranked *those* vertices are, etc. Formally, the authors [of *PageRank*] define the score of a vertex V_i as follows:

$$S(V_i) = (1-d) + d * \sum_{V_j \in \text{In}(V_i)} \frac{1}{|\ \text{Out}(V_i)\ |} S(V_j) \quad \text{where} \quad d \in \Re[0,1] \tag{143}$$

The factor d is usually set to 0.85.

and the damping factor d is interpreted as the probability of jumping from a given vertex[40] to another random vertex in the graph. In practice, the algorithm is implemented through the following steps:

(1) Initialize all vertices with arbitrary values.[41]
(2) Iterate over vertices, computing equation 4.9 until convergence [of the error rate] below a predefined threshold. The error-rate, defined as the difference between the "true score" and the score computed at iteration k, $S^k(V_i)$, is *approximated* as:

$$\text{Error}^k(V_i) \approx S^k(V_i) - S^{k-1}(V_i) \tag{144}$$

[40] Note that d is a single parameter for the graph, i.e. it is the same for all vertices.

[41] The authors do not specify what they mean by arbitrary. What range? What sampling distribution? Arbitrary as in uniformly random? **EDIT**: The authors claim that the vertex values upon completion are not affected by the choice of initial value. Investigate!

Weighted Graphs. In contrast with the PageRank model, here we are concerned with natural language texts, which may include multiple or partial links between the units (vertices). The authors hypothesize that modifying equation 4.9 to incorporate *weighted* connections may be useful for NLP applications.

$$WS(V_i) = (1-d) + d * \sum_{j \in \text{In}(V_i)} \frac{w_{ji}}{\sum V_k \in \text{Out}(V_j) w_{jk}} WS(V_j) \tag{145}$$

w_{ij} denotes the connection between vertices V_i and V_j.

where I've shown the modified part in green. The authors mention they set all weights to random values in the interval 0-10 (no explanation).

Text as a Graph. In general, the application of graph-based ranking algorithms to natural language texts consists of the following main steps:

(1) Identify text units that best define the task at hand, and add them as vertices in the graph.
(2) Identify relations that connect such text unit in order to draw edges between vertices in the graph. Edges can be directed or undirected, weighted or unweighted.
(3) Iterate the algorithm until convergence.
(4) Sort [in reversed order] vertices based on final score. Use the values attached to each vertex for ranking/selection decisions.

Keyword Extraction

Graph. The authors apply TextRank to extract words/phrases that are representative for a given document. The individual graph components are defined as follows:

- **Vertex**: sequence of one or more lexical units from the text.
  - In addition, we can restrict which vertices are added to the graph with syntactic filters.
  - Best filter [for the authors]: *nouns and adjectives only.*
- **Edge**: two vertices are connected if their corresponding lexical units co-occur within a window of N words[42].

Typically $N \in \mathbb{Z}[2, 10]$

Procedure:

(1) **Pre-Processing**: Tokenize and annotate [with POS] the text.

(2) **Build the Graph**: Add all [single] words to the graph that pass the syntactic filter, and connect [undirected/unweighted] edges as defined earlier (co-occurrence).

(3) **Run algorithm**: Initialize all scores to 1. For a convergence threshold of 0.0001, usually takes about 20-30 iterations.

(4) **Post-Processing**:

(i) Keep the top T vertices (by score), where the authors chose $T = |V|/3$.[43] Remember that vertices are still individual words.

(ii) From the new subset of T keywords, collapse any that were adjacent in the original text in a single lexical unit.

Evaluation. The data set used is a collection of 500 abstracts, each with a set of keywords. Results are evaluated using **precision**, **recall**, and **F-measure**[44]. The best results were obtained with a co-occurrence window of 2 [on an undirected graph], which yielded:

Precision: 31.2% Recall: 43.1% F-measure: 36.2

The authors found that larger window size corresponded with lower precision, and that directed graphs performed worse than undirected graphs.

[42] That is . . . simpler than expected. *Can we do better?*

[43] Another approach is to have T be a fixed value, where typically $5 < T < 20$.

[44] Brief terminology review:

- **Precision**: fraction of keywords extracted that are in the "true" set of keywords.
- **Recall**: fraction of "true" keywords that are in the extracted set of keywords.
- **F-score**: combining precision and recall to get a single number for evaluation:

$$F = \frac{2pr}{p + r}$$

A PR Curve plots precision as a function of recall.

Sentence Extraction

Graph. Now we move to "sentence extraction for automatic summarization."

- **Vertex**: a vertex is added to the graph for each sentence in the text.
- **Edge**: each weighted edge represents the similarity between two sentences. The authors use the following similarity measure between two sentences S_i and S_j:

$$\text{Similarity}(S_i, S_j) = \frac{\mid S_i \cap S_j \mid}{\log(\mid S_i \mid) + \log(\mid S_j \mid)} \tag{146}$$

  where the numerator is the number of words that occur in both S_i and S_j.

The **procedure** is identical to the algorithm described for keyword extraction, except we run it on full sentences.

Evaluation. The data set used is 567 news articles. For each article, TextRank generates a 100-word summary (i.e. they set $T = 100$). They evaluate with the Rouge evaluation toolkit (Ngram statistics).

Simple Baseline for Sentence Embeddings

Written by Brandon McKinzie

Overview. It turns out that simply taking a weighted average of word vectors and doing some PCA/SVD is a competitive way of getting unsupervised word embeddings. Apparently it *beats* supervised learning with LSTMs (?!). The authors claim the theoretical explanation for this method lies in a latent variable generative model for sentences (of course).

Discussion based on paper by Arora et al., (2017).

Algorithm.

1. Compute the weighted average of the word vectors in the sentence:

$$\frac{1}{N}\sum_{i}^{N}\frac{a}{a+p(\boldsymbol{w}_i)}\boldsymbol{w}_i \tag{147}$$

The authors call their weighted average the **Smooth Inverse Frequency (SIF)**.

   where $\boldsymbol{w}_i$ is the word vector for the ith word in the sentence, a is a parameter, and $p(\boldsymbol{w}_i)$ is the (estimated) word frequency [over the entire corpus].
2. Remove the projections of the average vectors on their first principal component ("common component removal") (y tho?).

Algorithm 1 Sentence Embedding

Input: Word embeddings $\{v_w : w \in \mathcal{V}\}$, a set of sentences $\mathcal{S}$, parameter a and estimated probabilities $\{p(w) : w \in \mathcal{V}\}$ of the words.
Output: Sentence embeddings $\{v_s : s \in \mathcal{S}\}$

for all sentence s in $\mathcal{S}$ **do**
  $v_s \leftarrow \frac{1}{|s|}\sum_{w\in s}\frac{a}{a+p(w)}v_w$
end for
Compute the first principal component u of $\{v_s : s \in \mathcal{S}\}$
for all sentence s in $\mathcal{S}$ **do**
  $v_s \leftarrow v_s - uu^\top v_s$
end for

Theory. Latent variable generative model. The model treats corpus generation as a dynamic process, where the t-th word is produced at time step t, driven by the random walk of a **discourse vector** $c_t \in \Re^d$ (d is size of the embedding dimension). The discourse vector is *not* pointing to a specific word; rather, it describes what is being talked about. We can tell how related (correlation) the discourse is to any word w and corresponding vector v_w by taking the inner product $c_t \cdot v_w$. Similarly, we model the probability of observing word w at time t, w_t, as:

$$\Pr[w_t \mid c_t] \propto e^{c_t \cdot v_w} \tag{148}$$

- **The Random Walk**. If we assume that c_t doesn't change much over the words in a single sentence, we can assume it stays at some c_s. The authors claim that in their previous paper they showed that the MAP[45] estimate of c_s is – up to multiplication by a scalar – the average of the embeddings of the words in the sentence.
- **Improvements/Modifications to 148**.
  1. Additive term $\alpha p(w)$ where α is a scalar. Allows words to occur even if $c_t \cdot v_w$ is very small.
  2. Common discourse vector $c_0 \in \Re^d$. Correction term for the most frequent discourse that is often related to syntax.
- **Model**. Given the discourse vector c_s for a sentence s, the probability that w is in the sentence (at all (?)):

$$\Pr[w \mid c_s] = \alpha p(w) + (1-\alpha)\frac{e^{\tilde{c}_s \cdot v_w}}{Z_{\tilde{c}_s}} \tag{149}$$

$$\tilde{c}_s = \beta c_0 + (1-\beta)c_s \tag{150}$$

with $c_0 \perp c_s$ and $Z_{\tilde{c}_s}$ is a normalization constant, taken over all $w \in V$.

Embedding Calculations. Here I'll go into more depth on how the algorithm described earlier is derived. **TODO**

[45] Review of MAP:

$$\theta_{MAP} = \arg\max_{\theta} \sum_i \log\left(p_X(x \mid \theta)p(\theta)\right)$$

Survey of Text Clustering Algorithms

 Written by Brandon McKinzie

Aggarwal et al., "A Survey of Text Clustering Algorithms," (2012).

Introduction. The unique characteristics for clustering *text*, as opposed to more traditional (numeric) clustering, are (1) large dimensionality but highly sparse data, (2) words are typically highly correlated, meaning the number of principal components is much smaller than the feature space, and (3) the number of words per document can vary, so we must normalize appropriately.

Common types of clustering algorithms include agglomerative clustering algorithms, partitioning algorithms, and standard parametric modeling based methods such as the EM-algorithm.

Feature Selection and Transformation Methods

Feature Selection.

- **Document Frequency-Based**. Using document frequency to filter *out* irrelevant features. Dealing with certain words, like "the", should probably be taken a step further and simply removed (stop words).
- **Term Strength**. A more aggressive technique for stop-word removal. It's used to measure how informative a word/term t is for identifying two related documents, x and y. Denoted $s(t)$, it is defined as:

  See ref 94 of the paper for more.

  $$s(t) = \Pr\left[t \in y \mid t \in x\right] \tag{151}$$

  So, how do we know x and y are related to begin with? One way is a user-defined cosine similarity threshold. Say we gather a set of such *pairs* and randomly identify one of the pair as the "first" document of the pair, then we can approximate $s(t)$ as

  $$s(t) = \frac{\text{Num pairs in which t occurs in both}}{\text{Num pairs in which t occurs in the first of the pair}} \tag{152}$$

  > *In order to prune features, the term strength may be compared to the expected strength of a term which is randomly distributed in the training documents with the same frequency. If the term strength of t is not at least two standard deviations greater than that of the random word, then it is removed from the collection.*

- **Entropy-Based Ranking**. The quality of a term is measured by the entropy reduction when it is removed [from the collection]. The entropy $E(t)$ of term t in a collection of n

documents is:

$$E(t) = -\sum_{i=1}^{n}\sum_{j=1}^{n}\left(S_{ij}\cdot\log(S_{ij}) + (1-S_{ij})\cdot\log(1-S_{ij})\right) \tag{153}$$

$$S_{ij} = 2^{-d_{ij}/\bar{d}} \tag{154}$$

where

- $S_{ij} \in (0,1)$ is the similarity between doc i and j.
- d_{ij} is the distance between i and j after the term t is removed
- $\bar{d}$ is the average distance between the documents after the term t is removed.

LSI-based Methods. Latent Semantic Indexing is based on dimensionality reduction where the new (transformed) features are a linear combination of the originals. This helps magnify the semantic effects in the underlying data. LSI is quite similar to PCA[46], except that we use an approximation of the covariance matrix C which is appropriate for the sparse and high-dimensional nature of text data.

See ref 28. of paper for more on LSI.

Let $\boldsymbol{A} \in \mathbb{R}^{n\times d}$ be term-document matrix, where $\boldsymbol{A}_{i,j}$ is the (normalized) frequency for term j in document i. Then $\boldsymbol{A}^T\boldsymbol{A} = n\cdot\Sigma$ is the (scaled) approximation to covariance matrix[47], assuming the data is mean-centered. Quick check/reminder:

$$(\boldsymbol{A}^T\boldsymbol{A})_{ij} = \boldsymbol{A}_{:,i}^T\boldsymbol{A}_{:,j} \triangleq \boldsymbol{a}_i^T\boldsymbol{a}_j \tag{155}$$

$$\approx n\cdot\mathbb{E}\left[\mathbf{a}_i\mathbf{a}_j\right] \tag{156}$$

where the expectation is technically over the underlying data distribution, which gives e.g. $P(a_i = x)$, the probability the ith word in our vocabulary having frequency x. Apparently, since the data is sparse, we don't have to worry much about it actually being mean-centered (**why?**).

As usual, we using the eigenvectors of $\boldsymbol{A}^T\boldsymbol{A}$ with the largest variance in order to represent the text[48]. In addition:

> *One excellent characteristic of LSI is that the truncation of the dimensions removes the noise effects of synonymy and polysemy, and the similarity computations are more closely affected by the semantic concepts in the data.*

[46] The difference between LSI and PCA is that PCA subtracts out the means, which destroys the sparseness of the design matrix.

[47] Approximation because it is based on our training data, not on true expectations over the underlying data-distribution.

[48] In typical collections, only about 300 to 400 eigenvectors are required for the representation.

Non-negative Matrix Factorization. Another latent-space method (like LSI), but particularly suitable for clustering. The main characteristics of the NMF scheme:

- In LSI, the new basis system consists of a set of orthonormal vectors. This is *not* the case for NMF.
- In NMF, the vectors in the basis system **directly correspond to cluster topics.** Therefore, the cluster membership for a document may be determined by examining the largest component of the document along any of the [basis] vectors.

Assume we want to create k clusters, using our n documents and vocabulary size d. The goal of NMF is to find matrices $\boldsymbol{U} \in \mathbb{R}^{n \times k}$ and $\boldsymbol{V} \in \mathbb{R}^{d \times k}$ that minimize:

$$J = \frac{1}{2}||\boldsymbol{A} - \boldsymbol{U}\boldsymbol{V}^T||_F^2 \tag{157}$$

$$= \frac{1}{2}\left(tr(\boldsymbol{A}\boldsymbol{A}^T) - 2tr(\boldsymbol{A}\boldsymbol{V}\boldsymbol{U}^T) + tr(\boldsymbol{U}\boldsymbol{V}^T\boldsymbol{V}\boldsymbol{U}^T)\right) \tag{158}$$

$u_{ij} \geq 0$

$v_{ij} \geq 0$

Note that the columns of $\boldsymbol{V}$ provide the k basis vectors which correspond to the k different clusters. We can interpret this as trying to factorize $\boldsymbol{A} \approx \boldsymbol{U}\boldsymbol{V}^T$. For each row, $\boldsymbol{a}$, of $\boldsymbol{A}$ (document vector), this is

$$\boldsymbol{a} \approx \boldsymbol{u} \cdot \boldsymbol{V}^T \tag{159}$$

$$= \sum_{i=1}^{k} \boldsymbol{u}_i \boldsymbol{V}_i^T \tag{160}$$

Therefore, the document vector $\boldsymbol{a}$ can be rewritten as an approximate linear (non-negative) combination of the basis vector which corresponds to the k columns of $\boldsymbol{V}^T$.

Langrange-multiplier stuff: Our optimization problem can be solved using the Lagrange method.

- Variables to optimize: All elements of both $\boldsymbol{U} = [u_{ij}]$ and $\boldsymbol{V} = [v_{ij}]$
- Constraint: non-negativity, i.e. $\forall i, j,\ u_{ij} \geq 0$ and $v_{ij} \geq 0$.
- Multipliers: Denote as matrices α and β, with same dimensions as $\boldsymbol{U}$ and $\boldsymbol{V}$, respectively.
- Lagrangian: I'll just show it here first, and then explain in this footnote[49]:

$$\mathcal{L} = J + tr(\alpha \cdot \boldsymbol{U}^T) + tr(\beta \cdot \boldsymbol{V}^T) \tag{161}$$

$$\text{where} \quad tr(\alpha \cdot \boldsymbol{U}^T) = \sum_{i=1}^{n} \alpha_i \cdot \boldsymbol{u}_i = \sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_{ij} u_{ij} \tag{162}$$

Any matrix multiplication with a $\cdot$ is just a reminder to think of the matrices as column vectors.

You should think of α as a column vector of length n, and $\boldsymbol{U}^T$ as a row vector of length n. The reason we prefer $\mathcal{L}$ over just J is because now we have an *unconstrained* optimization problem.

[49] Recall that in Lagrangian minimization, $\mathcal{L}$ takes the form of [the-function-to-be-minimized] + λ ([constraint-function] - [expected-value-of-constraint-at-optimum]). So the second term is expected to tend toward zero (i.e. critical point) at the optimal values. In our case, since our optimal value is sort-of (?) at 0 for any value of u_{ij} and/or v_{ij}, we just have a sum over [langrange-mult] $\times$ [variable].

- Optimization: Set the partials of $\mathcal{L}$ w.r.t both U and V (separately) to zero[50]:

$$\frac{\partial \mathcal{L}}{\partial U} = -\boldsymbol{A} \cdot \boldsymbol{V} + \boldsymbol{U} \cdot \boldsymbol{V}^T \cdot \boldsymbol{V} + \boldsymbol{\alpha} = 0 \tag{163}$$

$$\frac{\partial \mathcal{L}}{\partial V} = -\boldsymbol{A}^T \cdot \boldsymbol{U} + \boldsymbol{V} \cdot \boldsymbol{U}^T \cdot \boldsymbol{U} + \boldsymbol{\beta} = 0 \tag{164}$$

  Since, ultimately, these just say [some matrix] $= 0$, we can multiply both sides (element-wise) by a constant ($x \times 0 = 0$). *Using*[51] the **Kuhn-Tucker conditions** $\alpha_{ij} \cdot u_{ij} = 0$ and $\beta_{ij} \cdot v_{ij} = 0$, we get:

$$(\boldsymbol{A} \cdot \boldsymbol{V})_{ij} \cdot u_{ij} - (\boldsymbol{U} \cdot \boldsymbol{V}^T \cdot \boldsymbol{V})_{ij} \cdot u_{ij} = 0 \tag{165}$$

$$(\boldsymbol{A}^T \cdot \boldsymbol{U})_{ij} \cdot v_{ij} - (\boldsymbol{V} \cdot \boldsymbol{U}^T \cdot \boldsymbol{U})_{ij} \cdot v_{ij} = 0 \tag{166}$$

- Update rules:

$$u_{ij} = \frac{(\boldsymbol{A} \cdot \boldsymbol{V})_{ij} \cdot u_{ij}}{(\boldsymbol{U} \cdot \boldsymbol{V}^T \cdot \boldsymbol{V})_{ij}} \tag{167}$$

$$v_{ij} = \frac{(\boldsymbol{A}^T \cdot \boldsymbol{U})_{ij} \cdot v_{ij}}{(\boldsymbol{V} \cdot \boldsymbol{U}^T \cdot \boldsymbol{U})_{ij}} \tag{168}$$

Distance-based Clustering Algorithms

One challenge in clustering short segments of text (e.g., tweets) is that exact keyword matching may not work well. One general strategy for solving this problem is to expand text representation by exploiting related text documents, which is related to smoothing of a document language model in information retrieval.

See ref. 66 in the paper for computing similarities of short text segments.

Agglomerative and Hierarchical Clustering. "Agglomerative" refers to the process of bottom-up clustering to build a tree – at the bottom are leaves (documents) and internal nodes correspond to the merged groups of clusters. The different methods for merging groups of documents for the different agglomerative methods are as follows:

- **Single Linkage Clustering**. Defines similarity between two groups (clusters) of documents as the largest similarity between any pair of documents from these two groups. First, (1) compute all similarity pairs [between documents; ignore cluster labels], then (2) sort in decreasing order, and (3) walk through the list in that order, merging clusters if the pair belong to different clusters. One drawback is *chaining*: the resulting clusters assume transitivity of similarity[52].

[50] Recall that the Lagrangian consists entirely of traces (re: scalars). Derivatives of traces with respect to matrices output the same dimension as that matrix, and derivatives are taken element-wise as always.

[51] i.e. the equations that follow are *not* the KT conditions, they just *use*/exploit them...

[52] Here, transitivity of similarity means if A is similar to B, and B is similar to C, then A is similar to C. This is not guaranteed by any means for textual similarity, and so we can end up with A and Z in the same cluster, even though they aren't similar at all.

- **Group-Average Linkage Clustering**. Similarity between two clusters is the *average* similarity over all unique pairwise combinations of documents from one cluster to the other. One way to speed up this computation with an approximation is to just compute the similarity between the mean vector of either cluster.
- **Complete Linkage Clustering**. Similarity between two clusters is the *worst-case* similarity between any pair of documents.

Distance-Based Partitioning Algorithms.

- **K-Medoid Clustering**. Use a set of points from training data as anchors (medoids) around which the clusters are built. Key idea is we are using an optimal set of representative documents *from the original corpus*. The set of k reps is successively improved via randomized inter-changes. In each iteration, we replace a randomly picked rep in the current set of medoids with a randomly picked rep from the collection, if it improves the clustering objective function. This approach is applied until convergence is achieved.

  K-Medoid isn't great for clustering text, especially short texts.
- **K-Means Clustering**. Successively (1) assigning points to the nearest cluster centroid and then (2) re-computing the centroid of each cluster. Repeat until convergence. Requires typically few iterations (about 5 for many large data sets). Disadvantage: sensitive to initial set of seeds (initial cluster centroids). One method for improving the initial set of seeds is to use some supervision - e.g. initialize with k pre-defined topic vectors (see ref. 4 in paper for more).

Hybrid Approach: Scatter-Gather Method. Use a hierarchical clustering algorithm on a sample of the corpus in order to find a robust initial set of seeds. This robust set of seeds is used in conjunction with a standard k-means clustering algorithm in order to determine good clusters.**TODO:** resume note-taking; page 19/52 of PDF.

Scatter-Gather is discussed in detail in ref. 25 of the paper

---

Probabilistic Document Clustering and Topic Models

Overview. Primary assumptions in any topic modeling approach:

From pg. 31/52 of paper.

- The n documents in the corpus are assumed to each have a probability of belonging to one of k topics. Denote the probability of document D_i belonging to topic T_j as $\Pr[T_j \mid D_i]$. This allows for *soft cluster membership* in terms of probabilities.
- Each topic is associated with a probability vector, which quantifies the probability of the different terms in the lexicon for that topic. For example, consider a document that belongs completely to topic T_j. We denote the probability of term t_l occurring in that document as $\Pr[t_l \mid T_j]$.

The two main methods for topic modeling are **Probabilistic Latent Semantic Indexing** (PLSA) and **Latent Dirichlet Allocation** (LDA).

PLSA. We note that the two aforementioned probabilities, $\Pr[T_j \mid D_i]$ and $\Pr[t_l \mid T_j]$ allow us to calculate $\Pr[t_l \mid D_i]$: the probability that term t_l occurs in some document D_i:

$$\Pr[t_l \mid D_i] = \sum_{j=1}^{k} \Pr[t_l \mid T_j] \cdot \Pr[T_j \mid D_i] \tag{169}$$

which should be interpreted as a weighted average[53]. From here, we can generate a $n \times d$ matrix of probabilities.

Recall that we also have our $n \times d$ term-document matrix $\boldsymbol{X}$, where $\boldsymbol{X}_{i,l}$ gives the number of times term l occurred in document D_i. This allows us to do maximum likelihood! Our negative log-likelihood, J can be derived as follows:

Interpret $\Pr[\boldsymbol{X}]$ as the joint probability of observing the words in our data and with their assoc. frequencies.

$$\begin{aligned} J &= -\log(\Pr[\boldsymbol{X}]) && (170)\\ &= -\log\left(\prod_{i,l} \Pr[t_l \mid D_i]^{\boldsymbol{X}_{i,l}}\right) && (171)\\ &= -\sum_{i,l} \boldsymbol{X}_{i,l} \cdot \log(\Pr[t_l \mid D_i]) && (172) \end{aligned}$$

and we can plug-in eqn 169 to for evaluating $\Pr[t_l \mid D_i]$. We want to optimize the value of J, subject to the constraints:

$$(\forall T_j): \sum_{l} \Pr[t_l \mid T_j] = 1 \qquad (\forall D_i): \sum_{j} \Pr[T_j \mid D_i] = 1 \tag{173}$$

This can be solved with a Lagrangian method, similar to the process for NMF described earlier. See page 33/52 of the paper for details.

Latent Dirichlet Allocation (LDA). The term-topic probabilities and topic-document probabilities are modeled with a *Dirichlet distribution* as a prior[54]. Typically preferred over PLSI because PLSI more prone to overfitting.

[53] This is actually pretty bad notation, and borderline incorrect. $\Pr[T_j \mid D_i]$ is NOT a conditional probability! It is our prior! It is literally $\Pr[\text{ClusterOf}(D_i) = T_j]$.

[54] LDA is the Bayesian version of PLSI

Online Clustering with Text Streams

Reference List: [3]: A Framework for Clustering Massive Text and Categorical Data Streams; [112]: Efficient Streaming Text Clustering; [48]: Bursty feature representation for clustering text streams; [61]: Clustering Text Data Streams (Liu et al.)

Overview. Maintaining text clusters in real time. One method is the **Online Spherical K-Means Algorithm** (OSKM)[55].

See ref. 112 for more on OSKM

Condensed Droplets Algorithm. I'm calling it that because they don't call it anything – it is the algorithm in [3].

- **Fading function**: $f(t) = 2^{-\lambda \cdot t}$. A time-dependent weight for each data point (text stream). Non-monotonic decreasing; decays uniformly with time.
- **Decay rate**: $\lambda = 1/t_0$. Inverse of the half-life of the data stream.

When a cluster is created by a new point, it is allowed to remain as a trend-setting outlier for at least one half-life. During that period, if at least one more data point arrives, then the cluster becomes an active and mature cluster. If not, the trend-setting outlier is recognized as a true anomaly and is removed from the list of current clusters (*cluster death*). Specifically, this happens when the (weighted) number of points in the [single-point] cluster is 0.5. The same criterion is used to define the death of mature clusters. The statistics of the data points are referred to as **condensed droplets**, which represent the word distributions within a cluster, and can be used in order to compute the similarity of an incoming data point to the cluster. Main idea of algorithm is as follows:

1. Initialize empty set of clusters $\mathcal{C} = \{\}$. As new data points arrive, unit clusters containing individual data points are created. Once a maximum number k of such clusters have been created, we can begin the process of online cluster maintenance.
2. For a new data point X, compute its similarity to each cluster C_j, denoted as $S(X, C_j)$.
   - If $S(X, C_{best}) > \text{thresh}_{outlier}$, or if there are no inactive clusters left[56], insert X to the cluster with maximum similarity.
   - Otherwise, a new cluster is created[57] containing the solitary data point X.

[55] Authors only provide a very brief description, which I'll just copy here:
This technique divides up the incoming stream into small segments, each of which can be processed effectively in main memory. A set of k-means iterations are applied to each such data segment in order to cluster them. The advantage of using a segment-wise approach for clustering is that since each segment can be held in main memory, we can process each data point multiple times as long as it is held in main memory. In addition, the centroids from the previous segment are used in the next iteration for clustering purposes. A decay factor is introduced in order to age- out the old documents, so that the new documents are considered more important from a clustering perspective.

[56] We specify some max allowed number of clusters k.

[57] The new cluster replaces the least recently updated inactive cluster.

Misc.

- Mandatory read: reference [61]. Details phrase extraction/**topic signatures**. The use of using phrases instead of individual words is referred to as **semantic smoothing**.
- For *dynamic* (and more recent) topic modeling, see reference [107] of the paper, titled "A probabilistic model for online document clustering with application to novelty detection."

Semi-Supervised Clustering

Overview. Useful when we have any prior knowledge about the kinds of clusters available in the underlying data. Some approaches:

- Incorporate this knowledge when seeding the cluster centroids for k-means clustering.
- Iterative EM approach: unlabeled documents are assigned labels using a naive Bayes approach on the currently labeled documents. These newly labeled documents are then again used for re-training a Bayes classifier. Iterate to convergence.
- Graph-based approach: graph nodes are documents and the edges are similarities between the connected documents (nodes). We can incorporate prior knowledge by adding certain edges between nodes that we know are similar. A *normalized cut algorithm* is then applied to this graph in order to create the final clustering.

We can also use partially supervised methods in conjunction with pre-existing categorical *hierarchies*.

Deep Sentence Embedding Using LSTMs

Written by Brandon McKinzie

Palangi et al., "Deep Sentence Embeddings Using Long Short-Term Memory Networks: Analysis and Application to Information Retrieval," (2016).

Abstract. Sentence embeddings using LSTM cells, which automatically attenuate unimportant words and detect salient keywords. Main emphasis on applications for document retrieval (matching a query to a document[58]).

Introduction. Sentence embeddings are learned using a loss function defined on *sentence pairs*. For example, the well-known Paragraph Vector[59] is learned in an unsupervised manner as a distributed representation of sentences and documents, which are then used for sentiment analysis.

The authors appear to use a dataset of their own containing examples of (search-query, clicked-title) for a search engine. Their training objective is to maximize the similarity between the two vectors mapped by the LSTM-RNN from the query and the clicked document, respectively. One very interesting claim to pay close attention to:

> *We further show that different cells in the learned model indeed correspond to* ***different topics****, and the keywords associated with a similar topic activate the same cell unit in the model.*

Related Work. (Identified by reference number)

- [2] Good for sentiment, but doesn't capture fine-grained sentence structure.
- [6] Unsupervised embedding method trained on the BookCorpus [7]. Not good for document retrieval task.
- [9] Semi-supervised Recursive Autoencoder (RAE) for sentiment prediction.
- [3] DSSM (uses bag-of-words) and [10] CLSM (uses bag of n-grams) models for IR and also sentence embeddings.
- [12] Dynamic CNN for sentence embeddings. Good for sentiment prediction and question type classification. In [13], a CNN is proposed for **sentence matching**[60]

[58] Note that this similar to topic extraction.
[59] Q. V. Le and T. Mikolov, âĂIJDistributed representations of sentences and documents.âĂİ
[60] Might want to look into this.

Basic RNN. The information flow (sequence of operations) is enumerated below.

1. Encode tth word [of the given sentence] in one-hot vector $\boldsymbol{x}(t)$.
2. Convert $\boldsymbol{x}(t)$ to a letter tri-gram vector $\boldsymbol{l}(t)$ using fixed hashing operator[61] $\boldsymbol{H}$:

$$\boldsymbol{l}(t) = \boldsymbol{H}\boldsymbol{x}(t) \tag{174}$$

3. Compute the hidden state $\boldsymbol{h}(t)$, which is the sentence embedding for $t = T$, the length of the sentence.

$$\boldsymbol{h}(t) = \tanh\left(\boldsymbol{U}\boldsymbol{l}(t) + \boldsymbol{W}\boldsymbol{h}(t-1) + \boldsymbol{b}\right) \tag{175}$$

where $\boldsymbol{U}$ and $\boldsymbol{W}$ are the usual parameter matrices for the input/recurrent paths, respectively.

LSTM. With peephole connections that expose the internal cell state s to the sigmoid computations. I'll rewrite the standard LSTM equations from my textbook notes, but with the modifications for peephole connections:

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)} + \sum_j P_{i,j}^f s_j^{(t-1)}\right) \tag{176}$$

$$s_i^{(t)} = f_i^{(t)} \odot s_i^{(t-1)} + g_i^{(t)} \odot \sigma\left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)}\right) \tag{177}$$

$$g_i^{(t)} = \sigma\left(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)} + \sum_j P_{i,j}^g s_j^{(t-1)}\right) \tag{178}$$

$$q_i^{(t)} = \sigma\left(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)} + \sum_j P_{i,j}^o s_j^{(t)}\right) \tag{179}$$

The final hidden state can then be computed via

$$h_i^{(t)} = \tanh(s_i^{(t)}) \odot q_i^{(t)} \tag{180}$$

[61] Details aside, the hashing operator serves to lower the dimensionality of the inputs a bit. In particular we use it to convert one-hot word vectors into their letter tri-grams. For example, the word "good" gets surrounded by hashes, '#good#', and then hashed from the one-hot vector to vectorized tri-grams, "#go", "goo", "ood", "od#".

Learning Method

We want to maximize the likelihood of the clicked document given query, which can be formulated as the following optimization problem:

$$L(\mathbf{\Lambda}) = \min_{\mathbf{\Lambda}} \left\{ -\log \prod_{r=1}^{N} \Pr\left[D_r^+ \mid Q_r \right] \right\} = \min_{\mathbf{\Lambda}} \sum_{r=1}^{N} l_r(\mathbf{\Lambda}) \tag{181}$$

$$l_r(\mathbf{\Lambda}) = \log \left(1 + \sum_{j=1}^{n} e^{-\gamma \cdot \Delta_{r,j}} \right) \tag{182}$$

where

- N is the number of (query, clicked-doc) pairs in the corpus, while n is the number of negative samples used during training.
- D_r^+ is the clicked document for rth query.
- $\Delta_{r,j} = R(Q_r, D_r^+) - R(Q_r, D_{r,j}^-)$ (R is just cosine similarity)[62].
- $\mathbf{\Lambda}$ is all the parameter matrices (and biases) in the LSTM.

The authors then describe standard BPTT updates with momentum, which need not be detailed here. See the "Algorithm 1" figure in the paper for extremely detailed pseudo-code of the training procedure.

Analysis of the Sentence Embedding

TODO

[62] Note that $\Delta_{r,j} \in [-2, 2]$. We use γ as a scaling factor so as to expand this range.

Clustering Massive Text Streams

Written by Brandon McKinzie

Aggarwal et al., "A Framework for Clustering Massive Text and Categorical Data Streams," (2006).

Overview. Authors present an online approach for clustering massive text and categorical data streams with the use of a statistical summarization methodology. First, we will go over the process of storing and maintaining the data structures necessary for the clustering algorithm. Then, we will discuss the differences which arise from using different kinds of data, and the empirical results.

Maintaining Cluster Statistics. The data stream consists of d-dimensional records, where each dimension corresponds to the numeric frequency of a given word in the vector space representation. Each data point is weighted by the **fading function** $f(t)$, a non-monotonic decreasing function which decays uniformly with time t. The authors define the **half-life** of a data point (e.g. a tweet) as:

$$t_0 \quad \text{s.t.} \quad f(t_0) = \frac{1}{2} f(0) \tag{183}$$

and, similarly, the **decay-rate** as its inverse, $\lambda = 1/t_0$. Thus we have $f(t) = 2^{-\lambda \cdot t}$.

To achieve greater accuracy in the clustering process, we require a high level of granularity in the underlying data structures. To do this, we will use a process in which condensed clusters of data points are maintained, referred to as **cluster droplets**. We define them differently for the case of text and categorical data, beginning with categorical:

- **Categorical**. A cluster droplet $\mathcal{D}(t, \mathcal{C})$ for a set of categorical data points $\mathcal{C}$ at time t is defined as the tuple:

$$\mathcal{D}(t, \mathcal{C}) \triangleq (D\bar{F}2, D\bar{F}1, n, w(t), l) \tag{184}$$

  where
  - Entry k of the vector $D\bar{F}2$ is the (weighted) number of points in cluster $\mathcal{C}$ where the ith dimension had value x and the j dimension had value y. In other words, all pairwise combinations of values in the categorical vector[63]. $\sum_{i=1}^{d} \sum_{j \neq i}^{d} v_i v_j$ entries total[64].
  - Similarly, $D\bar{F}1$ consists of the (weighted) counts that some dimension i took on the value x. $\sum_{i=1}^{d} v_i$ entries total.

[63] This is intentionally written hand-wavy because I'm really concerned with *text* streams and don't want to give this much space.

[64] v_i is the number of values the ith categorical dimension can take on.

- $w(t)$ is the sum of the weights of the data points at time t.
- l is the time stamp of the last time a data point was added to the cluster.

- **Text**. Can be viewed as an example of a sparse numeric data set. A cluster droplet $\mathcal{D}(t, \mathcal{C})$ for a set of text data points $\mathcal{C}$ at time t is defined as the tuple:

$$\mathcal{D}(t, \mathcal{C}) \triangleq (D\bar{F}2, D\bar{F}1, n, w(t), l) \tag{185}$$

where
- $D\bar{F}2$ contains $3 \cdot wb \cdot (wb - 1)/2$ entries, where wb is the number of distinct words in the *cluster* $\mathcal{C}$.
- $D\bar{F}1$ contains $2 \cdot wb$ entries.
- n is the number of data points in the cluster $\mathcal{C}$.

Cluster Droplet Maintenance.

1. We first start of with k trivial clusters (the first k data points that arrived).
2. When a new point $\bar{X}$ arrives, the cosine similarity to each cluster's $D\bar{F}1$ is computed.
3. $\bar{X}$ is inserted into the cluster for which this is a maximum, so long as the associated $S(\bar{X}, D\bar{F}1) >$ thresh, a predefined threshold. If not above the threshold *and* some **inactive cluster** (wtf does this mean??) exists, a new cluster is created containing the solitary point $\bar{X}$, which replaces the inactive cluster. If not above threshold but no inactive clusters, then we just insert it into the max similarity cluster anyway.
4. If $\bar{X}$ was inserted (i.e. didn't replace an inactive cluster), then we need to:
   (a) Update the statistics to reflect the decay of the data points at the current moment in time[65]. This is done by multiplying entries in the droplet vectors by $2^{-\lambda \cdot (t-l)}$.
   (b) Add the statistics for each newly arriving data point to the cluster statistics.

[65] In other words, the statistics for a cluster do not decay, until a new point is added to it.

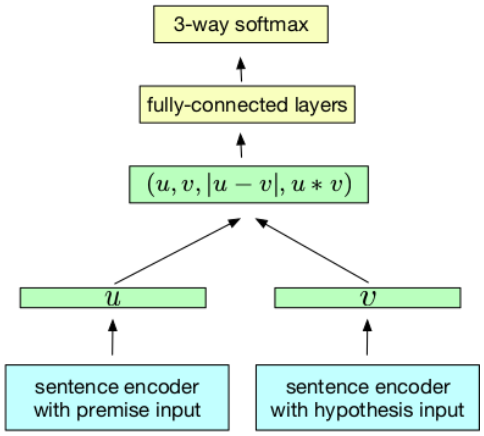
Supervised Universal Sentence Representations (InferSent)

Written by Brandon McKinzie

Conneau et al., "Supervised Learning of Universal Sentence Representations from Natural Language Inference Data," Facebook AI Research (2017).

Overview. Authors claim universal sentence representations trained using the supervised data of the Stanford Natural Language Inference (SNLI) dataset can consistently outperform unsupervised methods like SkipThought on a wide range of transfer tasks. They emphasize that training on NLI tasks in particular results in embeddings that perform well in transfer tasks. Their best encoder is a Bi-LSTM architecture with max pooling, which they claim is SOTA when trained on the SNLI data.

The Natural Language Inference Task. Also known as *Recognizing Textual Entailment* (RTE). The SNLI data consists of sentence pairs labeled as one of entailment, contradiction, or neutral. Below is a typical architecture for training on SNLI.



Note that the same sentence encoder is used for both u and v. To obtain a sentence vector from a BiLSTM encoder, they experiment with (1) the average h_t over all t (**mean pooling**), and (2) selecting the max value over each dimension of the hidden units [over all timesteps] (**max pooling**)[66].

[66] Since the authors have already mentioned that BiLSTM did the best, I won't go over the other architectures they tried: self-attentive networks, hierarchical convnet, vanilla LSTM/GRU.

Dist. Rep. of Sentences from Unlabeled Data (FastSent)

Written by Brandon McKinzie

Hill et al., "Learning Distributed Representations of Sentences from Unlabelled Data," (2016).

Overview. A systematic comparison of models that learn distributed representations of phrases/sentences from unlabeled data. Deeper, more complex models are preferable for representations to be used in supervised systems, but shallow log-linear models work best for building representation spaces that can be decoded with simple spatial distance metrics.

Authors propose two new phrase/sentence representation learning objectives:

1. **Sequential Denoising Autoencoders** (SDAEs)
2. **FastSent**: a sentence-level log-linear BOW model.

Distributed Sentence Representations. Existing models trained on text:

- **SkipThought Vectors** (Kiros et al., 2015). Predict target sentences $S_{i\pm1}$ given source sentence S_i. Sequence-to-sequence model.
- **ParagraphVector** (Le and Mikolov, 2014). Defines 2 log-linear models:
  1. **DBOW**: learns a vector $\boldsymbol{s}$ for every sentence S in the training corpus which, together with word embeddings v_w, define a softmax distribution to predict words $w \in S$ given S.
  2. **DM**: select k-grams of consecutive words $\{w_i \cdots w_{i+k} \in S\}$ and the sentence vector $\boldsymbol{s}$ to predict w_{i+k+1}.
- **Bottom-Up Methods**. Train **CBOW** and **Skip-Gram** word embeddings on the Books corpus.

The authors use **gensim** to implement ParagraphVector.

Models trained on *structured* (and freely-available) resources:

- **DictRep** (Hill et al., 2015a). Map dictionary definitions to pre-trained word embeddings, using either BOW or RNN-LSTM encoding.
- **NMT**. Consider sentence representations learned by sequence-to-sequence NMT models.

Novel Text-Based Methods.

- **Sequential (Denoising) Autoencoders**. To avoid needing coherent inter-sentence narrative, try this representation-learning objective based on DAEs. For a given sentence S and **noise function** $N(S \mid p_o, p_x)$ (where $p_0, p_x \in [0, 1]$),the approach is as follows:
  1. For each $w \in S$, N deletes w with probability p_o.
  2. For each non-overlapping bigram $w_i w_{i+1} \in S$, N swaps w_i and w_{i+1} with probability p_x.

     Authors recommend $p_o = p_x = 0.1$

     We then train the same LSTM-based encoder-decoder architecture as NMT, but with the denoising objective to predict (as target) the original source sentence S given a corrupted version $N(S \mid p_o, p_x)$ (as source).
- **FastSent**. Designed to be a more efficient/quicker to train version of SkipThought.

Latent Dirichlet Allocation

Written by Brandon McKinzie

Blei et al., "Latent Dirichlet Allocation," (2003).

Introduction. At minimum, one should be familiar with generative probabilistic models, mixture models, and the notion of latent variables before continuing. The "Dirichlet" in LDA of course refers to the **Dirichlet distribution**, which is a generalization of the beta distribution, B. It's PDF is defined as[67][68]:

$$\text{Dir}(\boldsymbol{x};\boldsymbol{\alpha}) = \frac{1}{B(\boldsymbol{\alpha})}\prod_{i=1}^{K}\boldsymbol{x}_i^{\boldsymbol{\alpha}_i-1} \quad \text{where} \quad B(\boldsymbol{\alpha}) = \frac{\prod_{i=1}^{K}\Gamma(\boldsymbol{\alpha}_i)}{\Gamma(\sum_{i=1}^{K}\boldsymbol{\alpha}_i)} \tag{186}$$

$$\sum_{i=1}^{K}\boldsymbol{x}_i = 1$$

$$(\forall i \in [1,K]) : \boldsymbol{x}_i \geq 0$$

Main things to remember about LDA:

- Generative probabilistic model for collections of discrete data such as text corpora.
- Three-level **hierarchical Bayesian model**. Each document is a mixture of topics, each topic is an infinite mixture over a set of topic probabilities.

Condensed comparisons/history of related models leading up to LDA:

- **TF-IDF**. Design matrix $\boldsymbol{X} \in \mathbb{R}^{V\times M}$, where M is the number of docs, and $\boldsymbol{X}_{i,j}$ gives the TF-IDF value for ith word in vocabulary and corresp. to document j.
- **LSI**:[69] Performs SVD on the TF-IDF design matrix $\boldsymbol{X}$ to identify a linear subspace in the space of tf-idf features that captures most of the variance in the collection.
- **pLSI**: **TODO**

> *pLSI is incomplete in that it provides no probabilistic model at the level of documents. In pLSI, each document is represented as a list of numbers (the mixing proportions for topics), and there is no generative probabilistic model for these numbers.*

[67] Recall that for positive integers n, $\Gamma(n) = (n-1)!$.

[68] The Dirichlet distribution is **conjugate** to the multinomial distribution. TODO: Review how to interpret this.

[69] Recall that LSI is basically PCA but without subtracting off the means

Model. LDA assumes the following generative process for each document (word sequence) $\boldsymbol{w}$:

1. $N \sim \textbf{Poisson}(\lambda)$: Sample N, the number of words (length of $\boldsymbol{w}$), from Poisson(λ) = $e^{-\lambda}\frac{\lambda^n}{n!}$. The parameter λ *should* represent the average number of words per document.
2. $\theta \sim \textbf{Dir}(\alpha)$: Sample k-dimensional vector θ from the Dirichlet distribution (eq. 186), Dir(α). k is the number of topics (pre-defined by us). Recall that this means θ lies in the (k-1) simplex. The Dirichlet distribution thus tells us the probability density of θ over this simplex – it defines the probability of θ being at a given position on the simplex.
3. Do the following N times to generate the words for this document.
   (a) $z_n \sim \textbf{Multinomial}(\theta)$. Sample a topic z_n.
   (b) $w_n \sim \Pr[w_n \mid z_n, \beta]$: Sample a word w_n from $\Pr[w_n \mid z_n, \beta]$, a "multinomial probability conditioned on topic z_n."[70] The parameter β gives the distribution of words given a topic:

$$\beta_{ij} = \Pr[w_j \mid z_i] \tag{187}$$

   In other words, we really sample $w_n \sim \beta_{i,:}$

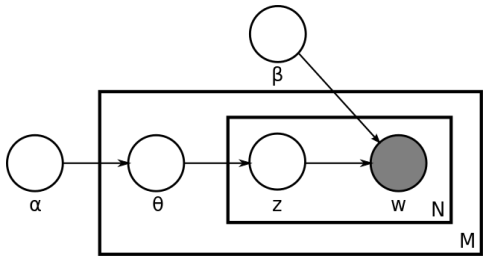
The defining equations for LDA are thus:

$$\Pr[\theta, \boldsymbol{z}, \boldsymbol{w} \mid \alpha, \beta] = \Pr[\theta \mid \alpha] \prod_{n=1}^{N} \Pr[z_n \mid \theta] \Pr[w_n \mid z_n, \beta] \tag{188}$$

$$\Pr[\boldsymbol{w} \mid \alpha, \beta] = \int \Pr[\theta' \mid \alpha] \left(\prod_{n=1}^{N} \sum_{z'_n} \Pr[z'_n \mid \theta'] \Pr[w_n \mid z'_n, \beta] \right) \mathrm{d}\theta' \tag{189}$$

$$\Pr\left[\mathcal{D} = \{\boldsymbol{w}^{(1)}, \ldots, \boldsymbol{w}^{(M)}\} \mid \alpha, \beta\right] = \prod_{d=1}^{M} \Pr\left[\boldsymbol{w}^{(d)} \mid \alpha, \beta\right] \tag{190}$$

Below is the plate notation for LDA, followed by an interpretation:



- **Outermost Variables**: α and β. Both represent a (Dirichlet) prior distribution: α parameterizes the probability of a given *topic*, while β a given *word*.
- Document Plate. M is the number of documents, $\boldsymbol{\theta}_m$ gives the true distribution of topics for document m[71].

[70] **TODO**: interpret meaning of the multinomial distributions here. Seems a bit different than standard interp...

[71] In other words, the meaning of $\boldsymbol{\theta}_{m,i} = x$ is "x percent of document m is about topic i."

- Topic/Word Place. $\boldsymbol{z}_{mn}$ is the topic for word n in doc m, and $\boldsymbol{w}_{mn}$ is the word. It is shaded gray to indicate it is the only **observed variable**, while all others are **latent variables**.

Theory. I'll quickly summarize and interpret the main theoretical points. Without having read all the details, this won't be of much use (i.e. it is for someone who has read the paper already).

- **LDA and Exchangeability**. We assume that each document is a bag of words (order doesn't matter; frequency sitll does) *and* a bag of topics. In other words, a document of N words *is* an unordered list of words and topics. De Finetti's theorem tells us that we can model the joint probability of the words and topics as if a random parameter θ were drawn from some distribution and then the variables within $\boldsymbol{w}, \boldsymbol{z}$ were **conditionally independent given** θ. LDA posits that a good distribution to sample θ from is a Dirichlet distribution.
- **Geometric Interpretation**: **TODO**

Inference and Parameter Estimation. As usual, we need to find a way to compute the posterior distribution of the hidden variables given a document $\boldsymbol{w}$:

$$\Pr\left[\boldsymbol{\theta}, \boldsymbol{z} \mid \boldsymbol{w}, \boldsymbol{\alpha}, \boldsymbol{\beta}\right] = \frac{\Pr\left[\boldsymbol{\theta}, \boldsymbol{z}, \boldsymbol{w} \mid \boldsymbol{\alpha}, \boldsymbol{\beta}\right]}{\Pr\left[\boldsymbol{w} \mid \boldsymbol{\alpha}, \boldsymbol{\beta}\right]} \tag{191}$$

Computing the denominator exactly is intractable. Common approximate inference algorithms for LDA include Laplace approximation, variational approximation, and Markov Chain Monte Carlo.

Conditional Random Fields

Written by Brandon McKinzie

Lafferty et al., "Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data," (2001).

Introduction. CRFs offer improvements to HMMs, MEMMs, and other discriminative Markov models. MEMMs and other non-generative models share a weakness called the **label bias problem**: the transitions leaving a given state compete only against each other, rather than against all other transitions in the model. The key difference between CRFs and MEMMs is that a CRF has a single exponential model for the joint probability of the entire sequence of labels given the observation sequence.

The Label Bias Problem. Recall that MEMMs are run left-to-right. One way of interpreting such a model is to consider how the probabilities (of state sequences) are distributed as we continue through the sequence of observations. The issue with MEMMs is that there's nothing we can do if, somewhere along the way, we observe something that makes one of these state paths extremely likely/unlikely; we can't redistribute the probability mass amongst the various allowed paths. The CRF solution:

> *Account for whole state sequences at once by letting some transitions "vote" more strongly than others depending on the corresponding observations. This implies that score mass will not be conserved, but instead individual transitions can "amplify" or "dampen" the mass they receive.*

Conditional Random Fields. Here we formalize the model and notation. Let $\mathbf{X}$ be a random variable over data sequences to be labeled (e.g. over all words/sentences), and let $\mathbf{Y}$ the random variable over corresponding label sequences[72]. Formal definition:

> *Let $G = (V, E)$ be a graph such that $Y = (Y_v)_{v \in V}$, so that Y is indexed by the vertices of G. Then (X, Y) is a CRF if, when conditioned on X, the random variables Y_v obey the Markov property with respect to the graph:*
>
> $$\Pr[Y_v | X, Y_w, w \neq v] = \Pr[Y_v | X, Y_w, w \sim v] \tag{192}$$
>
> *where $w \sim v$ means that w and v are neighbors in G.*

In human-speak, all this means is a CRF is a random field (discrete set of random-valued points in a space) where all points (i.e. globally) are conditioned on $\mathbf{X}$. If the graph $G = (V, E)$ of $\mathbf{Y}$ is a tree, its cliques[73] are the edges and vertices. By the fundamental theorem of random fields:

[72] We assume all components $\mathbf{Y}_i$ can only take on values in some finite label set $\mathcal{Y}$.

[73] A clique is a subset of vertices in an undirected graph such that every two distinct vertices in the clique are adjacent

$$p_\theta(\mathbf{y} \mid \mathbf{x}) \propto \exp\left(\sum_{e \in E,k} \lambda_k f_k(e, \mathbf{y}|_e, \mathbf{x}) + \sum_{v \in V,k} \mu_k g_k(v, \mathbf{y}|_v, \mathbf{x})\right) \tag{193}$$

where $\mathbf{y}|_S$ is the set of components of $\mathbf{y}$ associated with the vertices in subgraph S. We assume the K feature [functions] f_k and g_k are given and fixed.

Our estimation problem is thus to determine parameters $\theta = (\lambda_1, \lambda_2, \ldots; \mu_1, \mu_2, \ldots)$ from the labeled training data.

Attention Is All You Need

Written by Brandon McKinzie

Vaswani et al., "Attention Is All You Need," (2017)

Overview. Authors refer to sequence *transduction* models a lot – just a fancy way of referring to models that transform input sequences into output sequences. Authors propose new architecture, the **Transformer**, based solely on attention mechanisms (no recurrence!).

Model Architecture.

- **Encoder**. N = 6 identical layers, each with 2 sublayers: (1) a **multi-head** self-attention mechanism and (2) a position-wise FC feed-forward network. They employ a residual connection around each of the two, followed by layer normalization.
- **Decoder**. N = 6 with 3 sublayers each. In addition to the two sublayers described for the encoder, the decoder has a third sublayer, which performs **multi-head** attention over the output of the encoder stack. Same residual connections and layer norm.

Attention. An attention function can be described as a mapping:

$$\text{Attn}(\text{query}, \{(k_1, v_1), \dots, \}) \Rightarrow \sum_i fn(\text{query}, k_i)v_i \tag{194}$$

$$\Rightarrow \text{ output} \tag{195}$$

where the query, keys, values, and output are all vectors.

- **Scaled Dot-Product Attention**.
  1. **Inputs**: queries q, keys k of dimension d_k, values v of dimension d_v
  2. **Dot Products**: Compute $\forall k: \ (q \cdot k)/\sqrt{d_k}$.
  3. **Softmax**: on each dot product above. This gives the weights on the values shown earlier.

  Appears that $d_q \equiv d_k$.

  In practice, this is done simultaneously for all queries in a set via the following matrix equation:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{196}$$

  Note that this is identical to the standard dot-product attention mechanism, except for the *scaling* factor (hence the name) of $1/\sqrt{d_k}$. The scaling factor is motivated by the fact that additive attention outperforms dot-product attention for large d_k and the authors stipulate this is due to the softmax having small gradients in this case, due to the large dot products[74].

[74] Thus, they insert the $1/\sqrt{d_k}$ to counteract this effect.

- **Multi-Head Attention**. Basically just doing some number h of parallel attention computations. Before each of these, the queries, keys, and values are linearly projected with different, learned linear projections to d_k, d_k and d_v dimensions respectively (and then fed to their respective attention function). The h outputs are then concatenated and once again projected, resulting in the final values.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{197}$$

$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{198}$$

$W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$

$W_i^K \in \mathbb{R}^{d_{model} \times d_k}$

$W_i^V \in \mathbb{R}^{d_{model} \times d_v}$

$W^O \in \mathbb{R}^{hd_v \times d_{model}}$

  The authors employ $h = 8$, $d_k = d_v = d_{model}/h = 64$.

The Transformer uses multi-headed attention in 3 ways:

1. **Encoder-decoder attention**: the normal kind. Queries are previous decoder layer (state?), and memory keys and values come from output of the encoder.
2. **Encoder self-attention**: all of the keys, values, and queries come from the previous *layer* in the encoder. Each position in the encoder can attend to all positions in the previous layer of the encoder.
3. **Decoder self-attention**: Similarly, self-attention layers in the decoder allow each position in the decoder to attend to all positions in the decoder up to and including that position. Need to prevent leftward information flow in the decoder to preserve the auto-regressive property. This is implemented inside of a scaled dot-product attention by masking out (setting to $-\infty$) all values in the input of the softmax which correspond to illegal connections.

Other Components.

- **Position-wise Feed-Forward Networks (FFN)**: each layer of the encoder and decoder contains a FC FFN, applied to each position separately and identically:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{199}$$

The FFN is linear → ReLU → linear.

- **Embeddings and Softmax**: use learned embeddings to convert input/output tokens to vectors of dimension d_{model}, *and* for the pre-softmax layer at the output of the decoder[75].
- **Positional Encoding**: how the authors deal with the lack of recurrence (to make use of the sequence order). They add a sinusoid function of the position (timestep) *pos* and vector index i to the input embeddings for the encoder and decoder[76]:

For inputs to encoder/decoder, the embedding weights are multiplied by $\sqrt{d_{model}}$

$$\text{PE}(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{200}$$

$$\text{PE}(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{201}$$

  The authors justify this choice:

[75] In other words, they use the same weight matrix for all three of (1) encoder input embedding, (2) decoder input embedding, and (3) (opposite direction) from decoder output to pre-softmax.

[76] Note that the positional encodings must necessarily be of dimension d_{model} to be summed with the input embeddings.

We chose this function because we hypothesized it would allow the model to easily learn to attend by relative positions, since for any fixed offset k, PE_{pos+k} can be represented as a linear function of PE_{pos}.

Hierarchical Attention Networks

Written by Brandon McKinzie

Yang et al., "Hierarchical Attention Networks for Document Classification."

Overview. Authors introduce the Hierarchical Attention Network (HAN) that is designed to capture insights regarding (1) the hierarchical structure of documents (words -> sentences -> documents), and (2) the context dependence between words and sentences. The latter is implemented by including two levels of attention mechanisms, one at the word level and one at the sentence level.

Hierarchical Attention Networks. Below is an illustration of the network. The first stage is familiar to sequence to sequence models - a bidirectional encoder for outputting sentence-level representations of a sequence of words. The HAN goes a step further by feeding this another bidirectional encoder for outputting document-level representations for sequences of sentences.

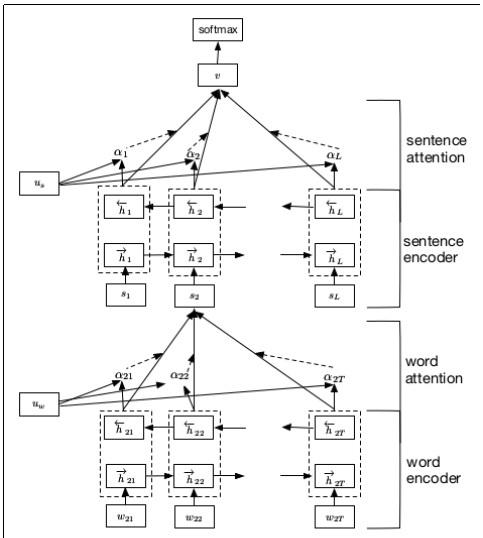


Figure 2: Hierarchical Attention Network.

The authors choose the GRU as their underlying RNN. For ease of reference, the defining equations of the GRU are shown below:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \tag{202}$$

$$z_t = \sigma\left(W_z x_t + U_z h_{t-1} + b_z\right) \tag{203}$$

$$\tilde{h}_t = \tanh\left(W_h x_t + r_t \odot \left(U_h h_{t-1}\right) + b_h\right) \tag{204}$$

$$r_t = \sigma\left(W_r x_t + U_r h_{t-1} + b_r\right) \tag{205}$$

Hierarchical Attention. Here I'll overview the main stages of information flow.

1. **Word Encoder**. Let the tth word in the ith sentence be denoted w_{it}. They embed the vectors with a word embedding matrix W_e, $x_{it} = W_e w_{it}$, and then feed x_{it} through a bidirectional GRU to ultimately obtain $h_{it} := [\overrightarrow{h}_{it}; \overleftarrow{h}_{it}]$.
2. **Word Attention**. Extracts words that are important to the meaning of the sentence and aggregates the representation of these informative words to form a sentence vector.

$$u_{it} = \tanh(W_w h_{it} + b_w) \tag{206}$$

$$\alpha_{it} = \frac{\exp(u_{it}^T u_w)}{\sum_t \exp(u_{it}^T u_w)} \tag{207}$$

$$s_i = \sum_t \alpha_{it} h_{it} \tag{208}$$

   Note the *context vector* u_w, which is shared for all words[77] and randomly initialized and jointly learned during the training process.
3. **Sentence Encoder**. Similar to the word encoder, but uses the sentence vectors s_i as the input for the ith sentence in the document. Note that the output of this encoder, h_i contains information from the neighboring sentences too (bidirectional) but focuses on sentence i.
4. **Sentence Attention**. For rewarding sentences that are clues to correctly classify a document. Similar to before, we now use a sentence level context vector u_s to measure the importance of the sentences.

$$u_i = \tanh(W_s h_i + b_s) \tag{209}$$

$$\alpha_i = \frac{\exp(u_i^T u_s)}{\sum_i \exp(u_i^T u_s)} \tag{210}$$

$$v = \sum_t \alpha_i h_i \tag{211}$$

   where v is the document vector that summarizes all the information of sentences in a document.

As usual, we convert v to a normalized probability vector by feeding through a softmax:

$$p = \text{softmax}(W_c v + b_c) \tag{212}$$

[77] To emphasize, there is only a single context vector u_w in the network, period. The subscript just tells us that it is the word-level context vector, to distinguish it from the sentence-level context vector in the later stage.

Configuration and Training. Quick overview of some parameters chosen by the authors:

- **Tokenization**: Stanford CoreNLP. Vocabulary consists of words occurring more than 5 times, all others are replaced with UNK token.
- **Word Embeddings**: train word2vec on the training and validation splits. Dimension of 200.
- **GRU**. Dimension of 50 (so 100 because bidirectional).
- **Context vectors**. Both u_w and u_s have dimension of 100.
- **Training**: batch size of 64, grouping documents of similar length into a batch. SGD with momentum of 0.9.

Joint Event Extraction via RNNs

Written by Brandon McKinzie

Nguyen, Cho, and Grishman, "Joint Event Extraction via Recurrent Neural Networks," (2016).

Event Extraction Task. Automatic Context Extraction (ACE) evaluation. Terminology:

- **Event**: something that happens or leads to some change of state.
- **Mention**: phrase or sentence in which an event occurs, including one trigger and an arbitrary number of arguments.
- **Trigger**: main word that most clearly expresses an event occurrence.
- **Argument**: an entity mention, temporal expression, or value that serves as a participant/attribute with a specific role in an event mention.

Example:

> In Baghdad, a **cameraman** **died**{*Die*} when an American tank **fired**{*Attack*} on the Palestine hotel.

TRIGGER **ARGUMENT**

Each event subtype has its own set of roles to be filled by the event arguments. For example, the roles for the *Die* event subtype include *Place*, *Victim*, and *Time*.

Model.

- **Sentence Encoding**. Let w_i denote the ith token in a sentence. It is transformed into a real-valued vector x_i, defined as

$$x_i := [\text{GloVe}(w_i);\ \text{Embed}(\text{EntityType}(w_i));\ \text{DepVec}(w_i)] \tag{213}$$

  where "Embed" is an embedding we learn, and "DepVec" is the binary vector whose dimensions correspond to the possible relations between words in the dependency trees.
- **RNN**. Bidirectional LSTM on the inputs x_i.
- **Prediction**. Binary memory vector G_i^{trg} for triggers; binary memory matrices G_i^{arg} and $G_i^{arg/trg}$ for arguments (at each timestep i). At each time step i, do the following in order:
  1. Predict trigger t_i for w_i. First compute the feature representation vector R_i^{trig}, defined as:

$$R_i^{trig} := \left[h_i;\ L_i^{trg};\ G_{i-1}^{trg}\right] \tag{214}$$

     where h_i is the RNN output, L_i^{trg} is the local context vector for w_i, and G_{i-1}^{trg} is the memory vector from the previous step. $L_i^{trg} := [\text{GloVe}(w_{i-d}); \ldots; \text{GloVe}(w_{i+d})]$ for

some predefined window size d. This is then fed to a fully-connected layer with softmax activation, $\boldsymbol{F}^{trg}$, to compute the probability over possible trigger subtypes:

$$P_{i;t}^{trg} := F_t^{trg}(R_i^{trg}) \tag{215}$$

As usual, the predicted trigger type for w_i is computed as $t_i = \arg\max_t \left(P_{i;t}^{trg}\right)$. If w_i is not a trigger, t_i should predict "*Other*."

2. Argument role predictions, $a_{i1}, \ldots, a_{ik}$, for all of the [already known] entity mentions in the sentence, $e_1, \ldots, e_k$ with respect to w_i. a_{ij} denotes the argument role of e_j with respect to [the predicted trigger of] w_i. If NOT(w_i is trigger AND e_j is one of its arguments), then a_{ij} is set to *Other*. For example, if w_i was the word "died" from our example sentence, we'd hope that its predicted trigger would be $t_i = Die$, and that **the entity associated with "cameraman" would get a predicted argument role of *Victim*.**

```
def getArgumentRoles(triggerType=t, entities=e):
   k = len(e)
   if isOther(t):
      return [Other] * k
   else:
      for e_j in e:
```

$$R_{ij}^{arg} := \left[h_i;\ h_{i_j};\ L_{ij}^{arg};\ B_{ij};\ G_{i-1}^{arg}[j]; G_{i-1}^{arg/trg}[j]\right] \tag{216}$$

3. Update memory. TO BE CONTINUED...(moving onto another paper because this model is getting a *bit* too contrived for my tastes. Also not a fan of the reliance on a dependency parse.)

Event Extraction via Bidi-LSTM Tensor NNs

 Written by Brandon McKinzie

Y. Chen, S. Liu, S. He, K. Liu, and J. Zhao, "Event Extraction via Bidirectional Long Short-Term Memory Tensor Neural Networks."

Overview. The task/goal is the event extraction task as defined in *Automatic Content Extraction* (ACE). Specifically, given a text document, our goal is to do the following in order *for each sentence*:

1. Identify any event triggers in the sentence.
2. If triggers found, predict their subtype. For example, given the trigger "fired," we may classify it as having the *Attack* subtype.
3. If triggers found, identify their candidate argument(s). ACE defines an event argument as "an entity mention, temporal expression, or value that is involved in an event."
4. For each candidate argument, predict its role: "the relationship between an argument to the event in which it participates."

Context-aware Word Representation. Use pre-trained word embeddings for the input word tokens, the predicted trigger, and the candidate argument. Note: *we assume we already have predictions for the event trigger t and are doing a pass for one of (possibly many) candidate arguments a.*

1. Embed each word in the sentence with pre-trained embeddings. Denote the embedding for ith word as $e(w_i)$.
2. Feed each $e(w_i)$ through a bidirectional LSTM. Denote the ith output of the forward LSTM as $c_l(w_{i+1})$ and the output of the backward LSTM at the same time step as $c_r(w_{i-1})$. As usual, they take the general functional form:

$$c_l(w_i) = \overrightarrow{LSTM}\left(c_l(w_{i-1}), e(w_{i-1})\right) \tag{217}$$

$$c_r(w_i) = \overleftarrow{LSTM}\left(c_r(w_{i+1}), e(w_{i+1})\right) \tag{218}$$

$$\tag{219}$$

3. Concatenate $e(w_i)$, $c_l(w_i)$, $c_r(w_i)$ together along with the embedding of the candidate argument $e(a)$ and predicted trigger $e(t)$. Also include the relative distance of w_i to t or (??) a, denoted as pi for position information, and the embedding of the predicted event type pe of the trigger. Denote this massive concatenation result as x_i:

$$x_i := c_l(w_i) \oplus e(w_i) \oplus c_r(w_i) \oplus pi \oplus pe \oplus e(a) \oplus e(t) \tag{220}$$

Dynamic Multi-Pooling. This is easiest shown by example. Continue with our example sentence:

In California, **Peterson** was arrested for the **murder** of his wife and unborn son.

where the colors are given for *this specific case where murder is our predicted trigger and we are considering the candidate argument Peterson*[78]. Given our n outputs from the previous stage, $y^{(1)} \in \mathbb{R}^{n \times m}$, where n is the length of the sentence and m is the size of that huge concatenation given in equation 220. We split our sentence by trigger and candidate argument, then (confusingly) redefine our notation as

$$y_{1j}^{(1)} \leftarrow \begin{bmatrix} y_{1j}^{(1)} & y_{2j}^{(1)} \end{bmatrix} \tag{221}$$

$$y_{2j}^{(1)} \leftarrow \begin{bmatrix} y_{3j}^{(1)} & \cdots & y_{7j}^{(1)} \end{bmatrix} \tag{222}$$

$$y_{3j}^{(1)} \leftarrow \begin{bmatrix} y_{8j}^{(1)} & \cdots & y_{nj}^{(1)} \end{bmatrix} \tag{223}$$

Peterson is the 3rd word, and murder is the 8th word.

where it's important to see that, for some $1 \leq j \leq m$, each new $y_{ij}^{(1)}$ is a *vector* of length equal to the number of words in segment i. Finally, the dynamic multi-pooling layer, $y^{(2)}$, can be expressed as

$$y_{i,j}^{(2)} := \max\left(y_{i,j}^{(1)}\right) \qquad 1 \leq i \leq 3,\ 1 \leq j \leq m \tag{224}$$

where the max is taken over each of the aforementioned vectors, leaving us with $3m$ values total. These are concatenated to form $y^{(2)} \in \mathbb{R}^{3m}$.

Output. To predict of each argument role [for the given argument candidate], $y^{(2)}$ is fed through a dense softmax layer,

$$O = W_2 y^{(2)} + b_2 \tag{225}$$

where $W_2 \in \mathbb{R}^{n_1 \times 3m}$ and n_1 is the number of possible argument roles (including "None"). The authors also use dropout on $y^{(2)}$.

[78] Yes, arrested could be another predicted trigger, but the network considers each possibility at separate times/locations in the architecture.

NLP with Deep Learning

Contents

NLP with Deep Learning **May 08**

Word Vector Representations (Lec 2)

Table of Contents Local *Written by Brandon McKinzie*

Meaning of a word. Common answer is to use a *taxonomy* like WordNet that has hypernyms (is-a) relationships. Problems with this discrete representation: misses nuances, e.g. the words in a set of synonyms can actually have quite different meanings/connotations. Furthermore, viewing words as atomic symbols is a bit like using one-hot vectors of words in a vocabulary space (inefficient).

Distributed representations. Want a way to encode word vectors such that two similar words have a similar structure/representation. The **distributional similarity-based**[79] approach represents words by means of its *neighbors* in the sentences in which it appears. You end up with a dense "vector for each word type, chosen so that it is good at predicting other words appearing in its context."

Skip-gram prediction. Given a word w_t at position t in a sentence, learn to predict [probability of] some number of surrounding words, given w_t. Standard minimization with negative log-likelihood:

$$J(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{-m\le j\le m}\log \Pr(w_{t+j}|w_t) \tag{226}$$

$$\Pr(o \mid c) = \frac{e^{\boldsymbol{u}_o^T \boldsymbol{v}_c}}{\sum_{w=1}^{\text{vocab size}} e^{\boldsymbol{u}_w^T \boldsymbol{v}_c}} \tag{227}$$

I cannot believe this actually works.

where

- The params θ are the vector representation of the words (they are the *only* learnable parameters here).
- m is our radius/window size.
- o and c are indices into our vocabulary (somewhat inconsistent notation).
- Yes, they are using different vector representations for $\boldsymbol{u}$ (context words) and $\boldsymbol{v}$ (center words). I'm assuming one reason this is done is because it makes the model architecture simpler/easier to build.

Some subtleties:

[79] Note that this is distinct from the way "distributed" is meant in "distributed representation." In contrast, distributional similarity-based representations refers to the notion that you can describe the meaning of words by understanding the context in which they appear.

- Looks like e.g. $Pr(w_{t+j} \mid w_t)$ doesn't really care what the value of j is, it is just modeling the probability that it is *somewhere* in the context window. The w_t are one-hot vectors into the vocabulary. Standard tricks for simplifying the cross-entropy loss apply.
- Equation 227 should be interpreted as the probability that the oth word in our vocabulary occurs in the context window of the cth word in our vocabulary.
- The model architecture is *identical* to an autoencoder. However, the (big) difference is the training procedure and interpretation of the model parameters going "in" versus the parameters going "out".

Sentence Embeddings. It turns out that simply taking a weighted average of word vectors and doing some PCA/SVD is a competitive way of getting unsupervised word embeddings. Apparently it *beats* supervised learning with LSTMs (?!). The authors claim the theoretical explanation for this method lies in a latent variable generative model for sentences (of course). Approach:

Discussion based on paper by Arora et al., (2017).

1. Compute the weighted average of the word vectors in the sentence:

$$\frac{1}{N}\sum_{i}^{N}\frac{a}{a+p(\boldsymbol{w}_i)}\boldsymbol{w}_i \tag{228}$$

The authors call their weighted average the **Smooth Inverse Frequency (SIF)**.

   where $\boldsymbol{w}_i$ is the word vector for the ith word in the sentence, a is a parameter, and $p(\boldsymbol{w}_i)$ is the (estimated) word frequency [over the entire corpus].
2. Remove the projections of the average vectors on their first principal component ("common component removal") (y tho?).

1

Algorithm 1 Sentence Embedding

Input: Word embeddings $\{v_w : w \in \mathcal{V}\}$, a set of sentences $\mathcal{S}$, parameter a and estimated probabilities $\{p(w) : w \in \mathcal{V}\}$ of the words.
Output: Sentence embeddings $\{v_s : s \in \mathcal{S}\}$
for all sentence s in $\mathcal{S}$ **do**
  $v_s \leftarrow \frac{1}{|s|}\sum_{w \in s}\frac{a}{a+p(w)}v_w$
end for
Compute the first principal component u of $\{v_s : s \in \mathcal{S}\}$
for all sentence s in $\mathcal{S}$ **do**
  $v_s \leftarrow v_s - uu^\top v_s$
end for

Further Reading.

- Learning representations by back-propagating errors (Rumelhard et al., 1986)
- A Neural Probabilistic Language Model (Bengio et al., 2003)
- NLP (almost) from Scratch (Collobert & Watson, 2008)
- Word2Vec (Miklov et al. 2013)

GloVe (Lec 3)

Written by Brandon McKinzie

Skip-gram and negative sampling. Main idea:

- Split the loss function from last lecture into two (additive) terms corresponding to the numerator and denominator respectively (you've done this a trillion times).
- The second term is an expectation over all the words in your vocab space. That is huge, so instead we only use a subsample of size k (the negative samples).
- Interpretation: First term is maximizing $\Pr(o \mid c)$, the probability that the true outside word (given by index o) occurs given context (index) c. Second term is minimizing the probability of random words (the negative samples) occurring around the center (context) word given by c.

To sample the negative samples, draw from $P(w) = U(w)^{3/4}/Z$, where U is the **unigram distribution**.

GloVe (Global Vectors). Given some co-occurrence matrix we computed with previous methods, we can use the following GloVe loss function over all pairs of co-occurring words in our matrix:

$$J(\theta) = \sum_{i,j=1}^{W} f(P_{ij})(u_i^T v_j - \log P_{ij})^2 \tag{229}$$

where P_{ij} is computed simply from the counts of words i and j co-occurring (empirical probability) and f is some squashing function that really isn't discussed in this lecture (**TODO**).

Evaluating word vectors.

- **Word Vector Analogies**: Basically, determining if we can do standard analogy fill-in-the-blank problems: "*man [a] is to woman [b] as king [c] is to <blank>*" (if you answered "queen", you'd make a good AI). We can determine this using a standard cosine distance measure:

$$d = \arg\max_i \frac{(x_b - x_a + x_c)^T x_i}{||x_b - x_a + x_c||} \tag{230}$$

  Woah that is pretty neat. The solution is x_i = queen. $x_b - x_a$ is the vector pointing from man to woman, which encodes the type of similarity we are looking for with the other pair. Therefore, we take the vector to "king" and *add* the aforementioned difference vector – the resultant vector should point to "queen". Neat!

Speech and Language Processing

Contents

Speech and Language Processing **July 10, 2017**

Introduction (Ch. 1 2nd Ed.)

Table of Contents Local *Written by Brandon McKinzie*

Overview. Going to rapidly jot down what seems most important from this chapter.

- **Morphology**: captures information about the shape and behavior of words in context (Ch. 2/3).
- **Syntax**: knowledge needed to order and group words together.
- **Lexical semantics**: knowledge of the meanings of individual words.
- **Compositional semantics**: knowledge of how these components (words) combine to form larger meanings.
- **Pragmatics**: the appropriate use of polite and indirect language.
- The knowledge of language needed to engage in complex language behavior can be separated into the following 6 distinct categories:
  1. Phonetics and Phonology – The study of linguistic sounds.
  2. Morphology – The study of the meaningful components of words.
  3. Syntax – The study of the structural relationships between words.
  4. Semantics – The study of meaning.
  5. Pragmatics – The study of how language is used to accomplish goals.
  6. Discourse – The study of linguistic units larger than a single utterance.
- Methods for **resolving ambiguity**: pos-tagging, word sense disambiguation, probabilistic parsing, and speech act interpretation.
- **Models and Algorithms**. Among the most important are **state space search** and **dynamic programming** algorithms.

Morphology (Ch. 3 2nd Ed.)

Written by Brandon McKinzie

English Morphology. Morphology is the study of the way words are built up from smaller meaning- bearing units, **morphemes**. A morpheme is often defined as the minimal meaning-bearing unit in a language[80]. The two classes of morphemes are **stems** (the "main" morpheme of the word) and **affixes** (the "additional" meanings of various kinds).

Affixes are further divided into prefixes (precede stem), suffixes (follow stem), circumfixes (do both), and infixes (inside the stem).

Two classes of ways to form words from morphemes: **inflection** and **derivation**. Inflection is the combination of a word stem with a grammatical morpheme, usually resulting in a word of the same class as the original stem, and usually filling some syntactic function like agreement. Derivation is the combination of a word stem with a grammatical morpheme, usually resulting in a word of a different class, often with a meaning hard to predict exactly.

[80] Examples: "fox" is its own morpheme, while "cats" consists of the morpheme "cat" and the morpheme "-s".

Counting Words. Most N-gram based systems deal with the *wordform*, meaning they treat words like "cats" and "cat" as distinct. However, we may want to treat the two as instances of a single abstract word, or **lemma**: a set of lexical forms having the same stem, the same major part of speech, and the same word-sense.

Simple (Unsmoothed) N-Grams. An N-gram is a N-1th order Markov model (because it looks N-1 steps in the past). Notation: the authors use the convention that $w_1^n \triangleq w_1, w_2, \dots, w_n$ to denote a sequence of n words. Given this, we can write the general equation for the N-gram approximation for the probability of the nth word ($n > N$) in a sentence:

$$\Pr\left[w_n \mid w_1^{n-1}\right] \approx \Pr\left[w_n \mid w_{n-N+1}^{n-1}\right] \tag{231}$$

for $N \geq 2$. We can compute these probabilities by simply counting:

$$\Pr\left[w_n \mid w_1^{n-1}\right] = \frac{C(w_{n-N+1}^{n-1} w_n)}{C(w_{n-N+1}^{n-1})} \tag{232}$$

where $C(\cdot)$ is the number of times the sequence, denoted as $\cdot$, occurred in the corpus.

Entropy. Denote the random variable of interest as $\mathbf{x}$ with probability function $p(\mathbf{x})$. The entropy of this random variable is:

$$H(\mathbf{x}) = -\sum_x p(\mathbf{x} = x) \log_2 p(\mathbf{x} = x) \tag{233}$$

which should be thought of as a lower bound on the number of bits it would take to encode a certain decision or piece of information in the optimal coding scheme. The value 2^H is the **perplexity**, which can be interpreted as the weighted average number of choices a random variable has to make.

Cross Entropy for Comparing Models. Useful when we don't know the actual probability distribution p that generated some data. Assume we have some model m that's an approximation of p. The cross-entropy of m on p is defined by:

$$H(p,m) = \lim_{n\to\infty} \frac{1}{n} \sum_{W\in L} p(w_1,\ldots,w_n) \log m(w_1,\ldots,w_n) \tag{234}$$

That is we draw sequences according to the probability distribution p, but sum the log of their probability according to m. **TODO**: continue off at pg 250/975 of textbook.

Naive Bayes and Sentiment (Ch. 6 3rd Ed.)

Written by Brandon McKinzie

[3rd Ed.] [Quick Review]

Overview. This chapter is concerned with **text categorization**, the task of classifying an entire text by assigning it a label drawn from some set of labels. *Generative classifiers* like naive Bayes build a model of each class. Given an observation, they return the class most likely to have generated the observation. *Discriminative classifiers* like logistic regression instead learn what features from the input are most useful to discriminate between the different possible classes. Notation: we will assume we have a training set of N documents, each hand-labeled with some class: $\{(d_1, c_1), \dots, (d_N, c_N)\}$.

Discriminative systems are often more accurate and hence more commonly used.

Naive Bayes. A multinomial[81] classifier with a naive assumption about how the features interact. We model a text document as a **bag of words**, meaning we store (1) the words that occurred and (2) their frequencies. It's a probabilistic classifier, meaning it estimates the label/class of a document d as

$$\hat{c} = \arg\max_{c \in C} \Pr[c \mid d] \tag{236}$$

$$= \arg\max_{c \in C} \frac{\Pr[d \mid c] \Pr[c]}{\Pr[d]} \tag{237}$$

$$= \arg\max_{c \in C} \Pr[d \mid c] \Pr[c] \tag{238}$$

Computing the class-conditional distribution (the likelihood) $\Pr[d \mid c]$ over all possible $d \in D$ is typically intractable; we must introduce some simplifying assumptions and use an approximation of it. Our assumptions in this section will be:

- **Bag-of-Words**: Assume that word position within a document is irrelevant. (Counts still matter)

[81] Rapid review: **multinomial distribution**. Let $\mathbf{x} = (x_1, \dots, x_k)$ denote the result of an experiment with n independent trials ($n = \sum_i^k x_i$) and k possible outcomes for any given trial. i.e. x_i is the number of trials that had outcome i ($1 \le i \le k$). The pmf of this multinomial distribution, over all possible $\mathbf{x}$ constrained by $n = \sum_i^k x_i$, is:

$$\Pr[\mathbf{x} = (x_1, \dots, x_k); n] = \frac{n!}{x_1! \cdots x_k!} p_1^{x_1} \times \cdots \times p_k^{x_k} \tag{235}$$

where p_i is the probability of outcome i for any single trial.

- **Naive Bayes Assumption**: First, recall that d is typically modeled as a (random) vector consisting of features $f_1, \ldots, f_n$, each of which has an associated probability distribution $\Pr[f_i \mid c]$. The NB assumption is that the features are mutually independent given the class c:

$$\Pr[f_1, \ldots, f_n \mid c] = \Pr[f_1 \mid c] \cdots \Pr[f_n \mid c] \tag{239}$$

$$c_{NB} = \arg\max_{c \in C} \Pr[c] \prod_{f \in F} \Pr[f \mid c] \tag{240}$$

  where 240 is the final equation for the class chosen by the naive Bayes classifier.

In text classification we typically use the word at position i in the document as f_i, and move to log space to avoid underflow/increase speed:

$$c_{NB} = \arg\max_{c \in C} \log \Pr[c] + \sum_{i}^{len(d)} \log \Pr[w_i \mid c] \tag{241}$$

Classifiers that use a linear combination of the inputs to make a classification decision (e.g. NB, logistic regression) are called linear classifiers.

Training the NB Classifier. No real "training" in my opinion, just simple counting from the data:

$$\hat{P}[c] = \frac{N_c}{N_{docs}} \tag{242}$$

$$\hat{P}[w_i \mid c] = \frac{\text{count}(w_i, c) + 1}{(\sum_{w \in V} \text{count}(w, c)) + |V|} \tag{243}$$

The Laplace smoothing is added to avoid the occurrence of zero-probabilities in equation 240.

Optimizing for Sentiment Analysis.

- It often improves performance [for sentiment] to clip word counts in each document to 1 ("binary NB").
- deal with *negations* in some way.
- Use sentiment lexicons, lists of words that are pre-annotated with positive or negative sentiment.

Hidden Markov Models (Ch. 9 3rd Ed.)

Written by Brandon McKinzie

Overview. Here we will first go over the math behind HMMs: the **Viterbi**, **Forward**, and **Baum-Welch** (EM) algorithms for unsupervised or semi-supervised learning. Recall that a HMM is defined by specifying the set of N states Q, transition matrix A, sequence of T observations O, sequence of observation likelihoods B, and the initial/final states. They can be characterized by three fundamental problems:

1. **Likelihood**. Given an HMM $\lambda = (A, B)$ and observation sequence, compute the likelihood (prob. of the observations given the model). (**Forward**)
2. **Decoding**. Given an HMM $\lambda = (A, B)$ and observation sequence, discover the best hidden state sequence. (**Viterbi**)
3. **Learning**. Given an observation sequence and the set of states in the HMM, learn the HMM parameters A and B. (**Baum-Welch/Forward-Backward/EM**)

The Forward Algorithm. For likelihood computation. We want to compute the probability of some sequence of observations O, without knowing the sequence of hidden states (that emitted the observations) Q. In general, this can be expressed by summing over all possible hidden state sequences:

$$\Pr[O] = \sum_Q \Pr[Q] \Pr[O \mid Q] \tag{244}$$

However, for N hidden states and T observations, this summation involves N^T terms, which becomes intractable rather quickly. Instead, we can use the $\mathcal{O}(N^2T)$ **Forward Algorithm**. The forward algorithm can be defined by initialization, recursion definition, and termination, shown respectively as follows:

$$\alpha_1(j) = a_{0j} b_j(o_1) \qquad 1 \leq j \leq N \tag{245}$$

$$\alpha_t(j) = \sum_{i=1}^{N} \alpha_{t-1}(i) a_{ij} b_j(o_t) \quad 1 \leq j \leq N, 1 \leq t \leq T \tag{246}$$

$$\Pr[O \mid \lambda] = \alpha_T(q_F) = \sum_{i=1}^{N} \alpha_T(i) a_{iF} \tag{247}$$

Viterbi Algorithm. For decoding. Want the most likely hidden state sequence given observations. Let $v_t(j)$ represent the probability that we are in state j after t observations and passing through the most probable state sequence $q_0, q_1, \ldots, q_{t-1}$. Similar to the forward algorithm, we show the defining equations for the Viterbi algorithm below:

$$v_1(j) = a_{0j} b_j(o_1) \quad 1 \leq j \leq N \tag{248}$$

$$v_t(j) = \max_{i=1}^{N} v_{t-1}(j) a_{ij} b_j(o_t) \tag{249}$$

$$P^* = v_T(q_F) = \max_{i=1}^{N} v_T(i) a_{iF} \tag{250}$$

N.B.: At each step, the best path up to that point can be found by taking the *argmax* instead of max.

Baum-Welch Algorithm. AKA forward-backward algorithm, a special case of the EM algorithm. Given an observation sequence O and the set of best possible states in the HMM, learn the HMM parameters A and B. First, we must define some new notation. The **backward probability** β is defined as:

$$\beta_t(i) \triangleq \Pr\left[o_{t+1}, \ldots, o_T \mid q_t = i, \lambda\right] \tag{251}$$

Remember $\lambda \equiv (A, B)$

As usual, we can compute its values inductively as follows:

$$\beta_T(i) = a_{iF} \qquad 1 \leq i \leq N \tag{252}$$

$$\beta_t(i) = \sum_{j=1}^{N} a_{ij} b_j(o_{t+1}) \beta_{t+1}(j) \quad 1 \leq i \leq N, 1 \leq t \leq T \tag{253}$$

$$\Pr\left[O \mid \lambda\right] = \alpha_T(q_F) = \beta_1(q_0) \tag{254}$$

$$= \sum_{j=1}^{N} a_{0j} b_j(o_1) \beta_1(j) \tag{255}$$

We can use the forward and backward probabilities α and β to compute the transition probabilities a_{ij} and observation probabilities $b_i(o_t)$ from an observation sequence. The derivation steps are as follows:

1. **Estimating** $\hat{a}_{ij}$. Begin by defining quantities that will prove useful:

Remember, knowing the observation sequence does NOT give us the sequence of hidden states.

$$\xi_t(i,j) \triangleq \Pr[q_t = i, q_{t+1} = j \mid O, \lambda] \tag{256}$$

$$\widetilde{\xi}_t(i,j) \triangleq \Pr[q_t = i, q_{t+1} = j, O \mid \lambda] \tag{257}$$

$$= \alpha_t(i) a_{ij} b_j(t+1) \beta_{t+1}(j) \tag{258}$$

where you should be able to derive eq. 258 in your head using just logic. If you cannot, review before continuing. We can then derive $\xi_t(i,j)$ using basic definitions of conditional probability, combined with eq. 254. Finally, we estimate $\hat{a}_{ij}$ as the expected number of transitions $q_i \to q_j$ divided by the expected number of transitions from q_i total:

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \sum_{k=1}^{N} \xi_t(i,k)} \tag{259}$$

2. **Estimating** $\hat{b}_j(v_k)$. We define our estimate as the expected number of times we are in q_j and emit observation v_k, divided by the expected number of times we are in state j. Similar to our approach for $\hat{a}_{ij}$ we define helper quantities for these values at a given timestep, then sum over them (all t) to obtain our estimate.

$$\gamma_t(j) \triangleq \Pr[q_t = j \mid O, \lambda] \tag{260}$$

$$= \frac{\Pr[q_t = j, O \mid \lambda]}{\Pr[O \mid \lambda]} \tag{261}$$

$$= \frac{\alpha_t(j) \beta_t(j)}{\Pr[O \mid \lambda]} \tag{262}$$

Thus, we obtain $\hat{b}_j(v_k)$ by summing over all timesteps where $o_t = v_k$, denoted as the set T_{v_k}, divided by the summation over all t regadless of o_t:

$$\hat{b}_j(v_k) = \frac{\sum_{t \in T_{v_k}} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)} \tag{263}$$

At last we can finally define the **Forward-Backward Algorithm** as follows:

1. Initialize A and B.
2. **E-step**. Compute $\gamma_t(j)$ $(\forall t, j)$, and compute $\xi_t(i,j)$ $(\forall t, i, j)$.
3. **M-step**. Update all $\hat{a}_{ij}$ and $\hat{b}_j(v_k)$.
4. Upon convergence, return A and B.

English Parts-of-Speech. POS are traditionally defined based on syntactic and morphological function, grouping words that have similar neighboring words or take similar affixes.

	Part of Speech	Definition	Properties
Open classes:	noun	people, places, things	occur with determiners, take possessives,
	verb	actions, processes	3rd-person-sg, progressive, past participle
	adjective	properties, qualities	
	adverb	modify verbs, adverbs, verb phrases	directional, locative, degree, manner, tem

Common nouns can be divided into *count* (e.g. goat/goats) and *mass* (e.g. snow) nouns.

Closed classes. POS with relatively fixed membership. Some of the most important in English are:

prepositions: on, under, over, near, by, at, from, to, with
determiners: a, an, the
pronouns: she, who, I, others
conjunctions: and, but, or, as, if, when
auxiliary verbs: can, may, should, are
particles: up, down, on, off, in, out, at, by
numerals: one, two, three, first, second, third

Some subtleties: the **particle** resembles a preposition or an adverb and is used in combination with a verb. An example case where "over" is a particle: "she turned the paper over." When a verb and a particle behave as a single syntactic and/or semantic unit, we call the combination a **phrasal verb**. Phrasal verbs cause widespread problems with NLP because they often behave as a semantic unit with a noncompositional meaning – one that is not predictable from the distinct meanings of the verb and the particle. Thus, "turn down" means something like "reject", "rule out" means "eliminate", "find out" is "discover", and "go on" is "continue".

HMM POS Tagging. Since we typically train on labeled data, we need only use the Viterbi algorithm for decoding[82]. In the POS case, we wish to find the sequence of n tags, $\hat{t}_1^n$, given the observation sequence of n words w_1^n.

$$\hat{t}_1^n = \arg\max_{t_1^n} \Pr\left[t_1^n \mid w_1^n\right] = \arg\max_{t_1^n} \Pr\left[w_1^n \mid t_1^n\right] \Pr\left[t_1^n\right] \tag{264}$$

where we've dropped the denominator after using Bayes' rule (since argmax is the same).

[82] Recall that decoding is the problem of finding the best hidden state sequence, given $\lambda = (A, B)$ and observation sequence O.

HMM taggers made two further simplifying assumptions:

$$\Pr\left[w_1^n \mid t_1^n\right] \approx \prod_{i=1}^{n} \Pr\left[w_i \mid t_i\right] \tag{265}$$

$$\Pr\left[t_1^n\right] \approx \prod_{i=1}^{n} \Pr\left[t_i \mid t_{i-1}\right] \tag{266}$$

We can thus plug-in these values into eq. 264 to obtain the equation for $\hat{t}_1^n$.

$$\hat{t}_1^n = \arg\max_{t_1^n} \prod_{i=1}^{n} \Pr\left[w_i \mid t_i\right] \Pr\left[t_i \mid t_{i-1}\right] \tag{267}$$

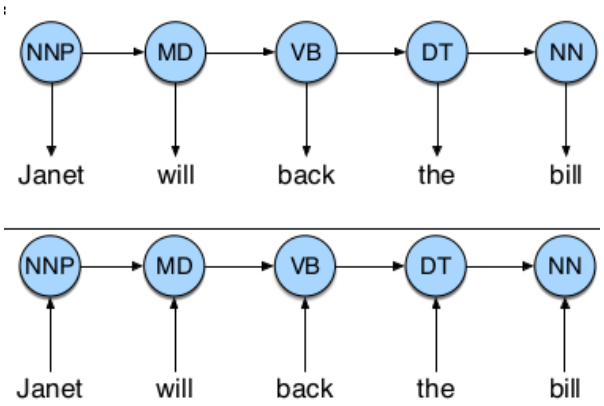
$$= \arg\max_{t_1^n} \prod_{i=1}^{n} b_i(w_i) a_{i-1,i} \tag{268}$$

where I've written a "translated" version on the second line using the familiar syntax from the previous chapter. In practice, we can obtain quick estimates for the two probabilities on the RHS by taking counts/averages over our tagged training data. We then run through the Viterbi algorithm to find all the argmaxes over states for the most likely hidden state sequence.

Maximum Entropy Markov Models (MEMMs). A sequence model adaptation of the MaxEnt (multinomial logistic regression) classifier[83]. Since HMMs are generative models, they decompose $\Pr\left[T \mid W\right]$ into $\Pr\left[W \mid T\right] \Pr\left[T\right]$ when computing the best tag sequence $\hat{T}$. Since MEMMs are discriminative, they compute/model $\Pr\left[T \mid W\right]$ directly:

$$\hat{T} = \arg\max_{T} \Pr\left[T \mid W\right] = \arg\max_{T} \prod_{i} \Pr\left[t_i \mid w_i, t_{i-1}\right] \tag{269}$$

Visually, we can think of the difference between HMMs and MEMMs via the direction of arrows, as illustrated below.



[83] Because it is based on logistic regression, the MEMM is a **discriminative sequence model**. By contrast, the HMM is a **generative sequence model**.

The top shows the HMM representation, while the bottom is MEMM.

> *The reason to use a discriminative sequence model is that discriminative models make it easier to incorporate a much wider variety of features.*

Bidirectionality. The one problem with the MEMM and HMM models as presented is that they are exclusively run left-to-right. MEMMs[84] have a weakness known as the **label bias problem**. Consider the tagged fragment: "*will/NN to/TO fight/VB*[85]." Even though the word "*will*" is followed by "*to*", which strongly suggests "*will*" is a NN, a MEMM will incorrectly label "*will*" as MD (modal verb). The culprit lies in the fact that $\Pr[\text{TO} \mid to, t_{will}]$ is essentially 1 regardless of t_{will}; i.e. the fact that "*to*" must have the tag TO has **explained away** the presence of TO and so the model doesn't learn the importance of the previous NN tag for predicting TO.

One way to implement bidirectionality (and thus allowing e.g. the link between TO being available when tagging the NN) is to use a **Conditional Random Field** (CRF) model. However, CRFs are much more computationally expensive than MEMMs and don't work better for tagging.

[84] And other non-generative finite-state models based on next-state classifiers

[85] Note on the tag meanings: TO literally means "to". MD means "modal" and refers to modal verbs such as *will*, *shall*, etc.

Constituency and CFGs. Discovering the inventory of constituents present in the language. Groups of words like *noun phrases* or *prepositional phrases* can be thought of as single units which can only be placed within certain parts of a sentence.

The most widely used formal system for modeling constituent structure in English is the **Context-Free Grammar**[86]. A CFG consists of a set of **productions** (rules), e.g.

$$\text{NP} \longrightarrow \text{Det Nominal} \tag{270}$$

$$\text{NP} \longrightarrow \text{ProperNoun} \tag{271}$$

$$\text{Nominal} \longrightarrow \text{Noun} \mid \text{Nominal Noun} \tag{272}$$

where the arrow is to be read "is composed of" or "consists of."

The sequence of rule expansions going from left to right is called a **derivation** of the string of words, commonly represented by a **parse tree**. The formal language defined by a CFG is the set of strings that are derivable from the designated **start symbol**.

[86] Also called Phrase-Structure Grammars. Equiv formalism as Backus-Naur Form (BNF)

Vector Semantics (Ch. 15)

 Written by Brandon McKinzie

Words and Vectors. Vector models are generally based on a **co-occurrence**, an example of which is a **term-document matrix**: Each row is identified by a word, and each column a document. A given cell value is the number of times the assoc. word occurred in the assoc. document. Can also view each column as a document vector.

Information Retrieval: task of finding document d, from D docs total, that best matches a query q.

For individual word vectors, however, it is most common to instead use a **term-term** matrix[87], in which columns are also identified by individual words. Now, cell values are the number of times the row (target) word and the column (context) word co-occur in some context in some training corpus. The context is most commonly a window around the row/target word, meaning a cell gives the number of times the column word occurs in a window of $\pm N$ words from the row word.

- **Q**: What about the co-occurrence of a word with itself (row i, col i)?
  - **A**: It is included, yes. Source: "The size of the window ... is generally between 1 and 8 words on each side of the target word (*for a total context of 3-17 words*)."
- **Q**: Why is the size of each vector generally $|V|$ (vocab size)? Shouldn't this vary substantially with window and corpus size?
  - **A**: idk

(**TODO**: revisit end of page 5 in my pdf of this)

Pointwise Mutual Information (PMI). Motivation: raw frequencies in a co-occurrence matrix aren't that great, and words like "the" (which aren't useful and occur everywhere) can really skew things. *The best weighting or measure of association between words should tell us how much more often than chance the two words co-occur.* PMI is such a measure.

[Mutual Information]
$$I(X,Y) = \sum_x \sum_y P(X=x, Y=y)\text{PMI}(x,y) \tag{273}$$

[PMI]
$$\text{PMI}(x,y) = \ln \frac{P(x,y)}{P(x)P(y)} \tag{274}$$

which can be applied for our specific use case as $\text{PMI}(w,c) = \ln \frac{P(w,c)}{P(w)P(c)}$. The interpretation is simple: the denominator tells the joint probability of the given target word w occurring with context word c if they were independent of each other, while the numerator tells us how often we observed the two words together (assuming we compute probability by using the MLE).

[87] Also called the word-word or term-context matrix

Therefore, the ratio gives us how an estimate of how much more the target and feature co-occur than we expect by chance[88]. Most people use **Positive PMI**, which is just $\max(0, \text{PMI})$. We can compute a **PPMI matrix** (to replace our co-occurrence matrix), where PPMI_{ij} gives the PPMI value of word w_i with context c_j. The authors show a few formulas which is really distracting since all we actually need is the counts $f_{ij} = \text{counts}(w_i, c_j)$, and from there we can use basic probability and Bayes rule to get the PPMI formula.

- **Q**: Explain why the following is true: very rare words tend to have very high PMI values.
  - **A**: hi
- **Q**: What is the range of α used in PPMI_α? What is the intuition behind doing this?
  - **A**: For reference:

$$\text{PPMI}_\alpha(w, c) = \max\left(\ln \frac{P(w, c)}{P(w)P_\alpha(c)}, 0\right) \tag{275}$$

$$P_\alpha(c) = \frac{\text{count}(c)^\alpha}{\sum_{c'} \text{count}(c')^\alpha} \tag{276}$$

Although there are better methods than PPMI for weighted co-occurrence matrices, most notably **TF-IDF**, things like tf-idf are <u>not</u> generally used for measuring *word similarity*. For that, PPMI and significance-testing metrics like t-test and likelihood-ratio are more common. The **t-test** statistic, like PMI, measures how much more frequent the association is than chance.

$$t = \frac{\bar{x} - \mu}{\sqrt{s^2/N}} \tag{277}$$

$$\text{t-test}(a, b) = \frac{P(a, b) - P(a)P(b)}{\sqrt{P(a)P(b)}} \tag{278}$$

where $\bar{x}$ is the observed mean, while μ is the expected mean [under our null-hypothesis of independence].

Measuring Similarity. By far the most common similarity metric is the **cosine** of the angle between the vectors:

$$\text{cosine}(\boldsymbol{v}, \boldsymbol{w}) = \frac{\boldsymbol{v} \cdot \boldsymbol{w}}{|\boldsymbol{v}|\ |\boldsymbol{w}|} \tag{279}$$

Note that, since we've been defining vector elements as frequencies/PPMI values, they won't have negative elements, and thus our cosine similarities will be between 0 and 1 (not -1 and 1).

[88] Computing PMI this way can be problematic for word pairs with small probability, especially if we have a small corpus. Recognize that PMI should never really be negative, but in practice this happens for such cases

Alternatives to cosine:

- **Jaccard measure**: Described as "weighted number of overlapping features, normalized", but looks like a silly hack in my opinion:

$$\text{sim}_{Jac}(\boldsymbol{v}, \boldsymbol{w}) = \frac{\sum_{i=1}^{N} \min(\boldsymbol{v}_i, \boldsymbol{w}_i)}{\sum_{i=1}^{N} \max(\boldsymbol{v}_i, \boldsymbol{w}_i)} \tag{280}$$

- **Dice measure**: Another hack. This displeases me.

$$\frac{2 \times \sum_{i=1}^{N} \min(\boldsymbol{v}_i, \boldsymbol{w}_i)}{\sum_{i=1}^{N} (\boldsymbol{v}_i + \boldsymbol{w}_i)} \tag{281}$$

- **Jensen-Shannon Divergence**: An alternative to the KL-divergence[89], which represents the divergence of each distribution from the mean of the two:

$$\text{sim}_{JS}(\boldsymbol{v}||\boldsymbol{w}) = D\left(\boldsymbol{v}\middle|\middle|\frac{\boldsymbol{v}+\boldsymbol{w}}{2}\right) + D\left(\boldsymbol{w}\middle|\middle|\frac{\boldsymbol{v}+\boldsymbol{w}}{2}\right) \tag{283}$$

[89] Idea:if two vectors, $\boldsymbol{v}$ and $\boldsymbol{w}$, each express a probability distribution (their values sum to one), then they are are similar to the extent that these probability distributions are similar. The basis of comparing two probability distributions P and Q is the **Kullback-Leibler** divergence or relative entropy, defined as:

$$D(P||Q) = \sum_{x} P(x) \log \frac{P(x)}{Q(x)} \tag{282}$$

Semantics with Dense Vectors (Ch. 16)

Written by Brandon McKinzie

Overview. This chapter introduces three methods for generating short, dense vectors: (1) dimensionality reduction like SVD, (2) neural networks like skip-gram or CBOW, and (3) Brown clustering.

Dense Vectors via SVD. Method for finding more important dimensions of a dataset, "important" defined as dimensions wherein the data most varies. First applied (for language) for generating embeddings from term-document matrices in a model called **Latent Semantic Analysis** (LSA). LSA is just SVD on a $|V| \times c$ term-document matrix $\boldsymbol{X}$, factorized into $\boldsymbol{W\Sigma C}^T$. By using only the top $k < m$ dimensions of these three matrices, the product becomes a least-squares approx. to the original X. It also gives us the reduced $|V| \times k$ matrix $\boldsymbol{W}_k$, where each row (word) is a k-dimensional vector (embedding). VoilÃă, we have our dense vectors!

$\boldsymbol{W} \in \mathbb{R}^{|V| \times m}$
$\boldsymbol{\Sigma} \in \mathbb{R}^{m \times m}$
$\boldsymbol{C}^T \in \mathbb{R}^{m \times c}$

Note that LSA implementations typically use a particular weighting of each cell in the term-document matrix called the **local** and **global** weights.

$$\text{[local]} \qquad \log \mathrm{f}(i,j) + 1 \tag{284}$$

$$\text{[global]} \qquad 1 + \frac{\sum_j p(i,j) \log p(i,j)}{\log D} \tag{285}$$

$f(i,j)$ is the raw frequency of word i in context j. D is number of docs.

For the case of a word-word matrix, it is common to use PPMI weighting.

Skip-Gram and CBOW. Neural models learn an embedding by starting with a random vector and then iteratively shifting a word's embeddings to be more like the embeddings of neighboring words, and less like the embeddings of words that don't occur nearby[90] Word2vec, for example, learns embeddings by training to predict neighboring words[91].

- **Skip-Gram**: Learns two embeddings for each word w: the **word embedding** v (within matrix $\boldsymbol{W}$) and **context embedding** c (within matrix $\boldsymbol{C}$). Visually:

$$\boldsymbol{W} = \begin{pmatrix} \boldsymbol{v}_0^T \\ \boldsymbol{v}_1^T \\ \vdots \\ \boldsymbol{v}_{|V|}^T \end{pmatrix} \qquad \boldsymbol{C} = \begin{pmatrix} \boldsymbol{c}_0 & \boldsymbol{c}_1 & \cdots & \boldsymbol{c}_{|V|} \end{pmatrix} \tag{286}$$

  For a context window of $L = 2$, and at a given word $\boldsymbol{v}^{(t)}$ inside the corpus[92], our goal is to predict the context [words] denoted as $[\,\boldsymbol{c}^{(t-2)}, \boldsymbol{c}^{(t-1)}, \boldsymbol{c}^{(t+1)}, \boldsymbol{c}^{(t+2)}\,]$.
  - Example: Consider one of the context words, say $\boldsymbol{c}^{(t+1)} \triangleq \boldsymbol{c}_k$, where we also assume it's the kth word in our vocab. Also assume that our target word $\boldsymbol{v}^{(t)} \triangleq \boldsymbol{v}_j$ is the jth word in our vocab.
  - Our task is to compute $\Pr[\boldsymbol{c}_k \mid \boldsymbol{v}_j]$. We do this with a softmax:

$$\Pr[\boldsymbol{c}_k \mid \boldsymbol{v}_j] = \frac{e^{\boldsymbol{c}_k^T \boldsymbol{v}_j}}{\sum_{i \in |V|} e^{\boldsymbol{c}_i^T \boldsymbol{v}_j}} \tag{287}$$

- **CBOW**: Continuous bag of words. Basically the mirror-image of skip-gram. Goal is to predict current word $\boldsymbol{v}^{(t)}$ from the context window of 2L words $[\,\boldsymbol{c}^{(t-2)}, \boldsymbol{c}^{(t-1)}, \boldsymbol{c}^{(t+1)}, \boldsymbol{c}^{(t+2)}\,]$.

As usual, the denominator of the softmax is computationally expensive, and usually we approximate it with **negative sampling**.

> *In the training phase, the algorithm walks through the corpus, at each target word choosing the surrounding context words as positive examples, and for each positive example also choosing k noise samples or negative samples: non-neighbor words. The goal will be to move the embeddings toward the neighbor words and away from the noise words.*

[90] Why? Why is this a sensible assumption? I see no reason a priori why it ought to be true.

[91] Note that the prediction task is not the goal – it just happens to result in good word embeddings. Hacky.

[92] Note that, technically, the position t of $\boldsymbol{v}^{(t)}$ is irrelevant for our computation; we are predicting those words based on which word $\boldsymbol{v}^{(t)}$ is in the vocabulary, not it's position in the corpus.

Example: Suppose we come along the following window (in "[]") (L=2) in our corpus:

```
lemon, a [tablespoon of apricot preserves or] jam
```

Ultimately, we want dot products, $\boldsymbol{c}_i \cdot \textit{vector}(\text{“apricot”})$, to be high for all four of the context words $\boldsymbol{c}_i$. We do negative sampling by sampling k random noise words according to their [unigram] frequency. So here, for e.g. $k = 2$, this would amount to 8 noise words, 2 for each context word. We want the dot products between "apricot" and these noise words to be low. For a given single context-word pair (w, c), our training objective is to maximize:

$$\log \sigma(c \cdot w) + \sum_{i=1}^{k} \mathbb{E}_{w_i \sim p(w)} \left[\log \sigma(-w_i \cdot w)\right] \tag{288}$$

In practice, common to use $p^{3/4}(w)$ instead of $p(w)$

Again, the above is for a single context-target word-pair and, accordingly, the summation is only over $k = 2$ (for our example). Don't try to split the expectation into a summation or anything – just view it as an expected value. To iteratively shift parameters, we use an optimizer like SGD.

The actual model architecture is a typical neural net, progressing as follows:

$$\begin{aligned}
\text{“apricot”} &\rightarrow \boldsymbol{w}^{\text{one-hot}} = \left[\,0\ 0\ \cdots\ 1\ \cdots\ 0\,\right] && (289)\\
&\rightarrow \boldsymbol{h} = \boldsymbol{W}^T \boldsymbol{w}^{\text{one-hot}} && (290)\\
&\rightarrow \boldsymbol{o} = \boldsymbol{C}^T \boldsymbol{h} = \left[\,\boldsymbol{c}_0^T \boldsymbol{h},\, \boldsymbol{c}_1^T \boldsymbol{h},\, \cdots\ \boldsymbol{c}_{|V|}^T \boldsymbol{h}\,\right]^T && (291)\\
&\rightarrow \boldsymbol{y} = \text{softmax}(\boldsymbol{o}) = \left[\,\Pr[\boldsymbol{c}_0|\boldsymbol{h}],\, \Pr[\boldsymbol{c}_2|\boldsymbol{h}],\, \cdots\ \Pr[\boldsymbol{c}_{|V|}|\boldsymbol{h}]\,\right]^T && (292)\\
& && (293)
\end{aligned}$$

Brown Clustering. An agglomerative clustering algorithm for deriving vector representations of words by clustering words based on their associations with the preceding or following words. Makes use of the **class-based language model** (CBLM), wherein each word w belongs to some class $c \in C$ via the probability $P(w \mid c)$. CBLMs define

$$P(w_i \mid w_{i-1}) = P(c_i \mid c_{i-1}) P(w_i \mid c_i) \tag{294}$$

$$P(\text{corpus} \mid C) = \prod_{i-1}^{n} P(w_i \mid w_{i-1}) \tag{295}$$

A naive and extremely inefficient version of Brown clustering, a hierarchical clustering algorithm, is as follows:

1. Each word is initially assigned to its own cluster.
2. For each cluster pair $(c_i, c_{j \neq i})$, compute the value of eq 295 that would result from merging c_i and c_j into a single cluster. The pair whose merger results in the smallest decrease in eq 295 is merged.
3. Clustering proceeds until all words are in one big cluster.

This process builds a binary tree from the bottom-up, and the binary string corresp. to a word's traversal from leaf-to-root is its representation.

Overview. The first step in most IE tasks is **named entity recognition** (NER). Next we can do **relation extraction**: finding and classifying semantic relations among the entities, e.g. "spouse-of." **Event extraction** is finding the events in which the entities participate, and **event coreference** for figuring out which event mentions actually refer to the same event. It's also common to extract dates/times (temporal expression) and perform **temporal expression normalization** to map them onto specific calendar dates. Finally, we can do **template filling**: finding recurring/stereotypical situations in documents and filling the template slots with appropriate material.

Named Entity Recognition. Standard algorithm is word-by-word sequence labeling task by a MEMM or CRF, trained to label tokens with tags indicating the presence of particular kinds of NEs. It is common to label with **BIO tagging**, for beginning, inside, and outside of entities. If we have n unique entity types, then we'd have $2n + 1$ BIO tags[93]. A helpful illustration is shown below:

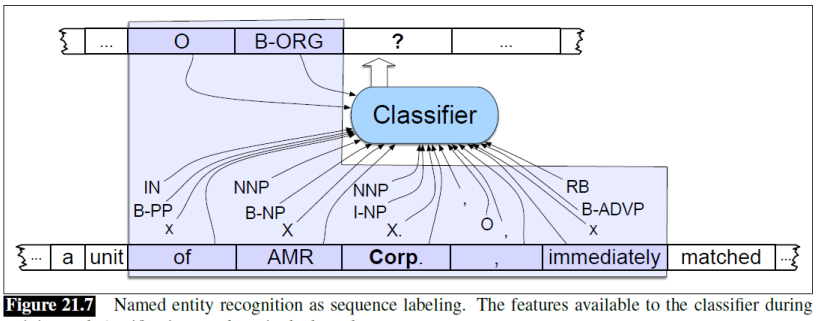


Figure 21.7 Named entity recognition as sequence labeling. The features available to the classifier during training and classification are those in the boxed area.

Here we see a classifier determining the label for *Corp.* with a context window of size 2 and various features shown in the boxed region. For evaluation of NER, we typically use the familiar **recall**, **precision**, and **F1 measure**.

[93] $2n$ for B-<NE> and I-<NE>, with +1 for the blanket O tag (not any of our NEs)

Relation Extraction. The four main algorithm classes used are (1) hand-written patterns, (2) supervised ML, (3) semi-supervised, and (4) unsupervised. Terminology:

- **Infobox**: structured tables associated with certain articles/topics/etc. For example, the Wikipedia infobox for Stanford includes structured facts like state = 'California'.
- **Resource Description Framework (RDF)**: a metalanguage of RDF triples, tuples of (entity, relation, entity), called a subject-predicate-object expression. For example: (Golden Gate Park, location, San Francisco).
- **hypernym**: the "is-a" relation.
- **hyponym**: the "kind-of" relation. *Gelidium is a kind of red algae.*

Overview of the four algorithm classes:

1. **Patterns**. Consider a sentence that has the following form:

   NP_0 *such as* $NP_1\{, NP_2, \ldots, (and|or)NP_i\}, i \geq 1$

   also known as a *lexico-syntactic pattern*, which implies $\forall NP_i, i \geq 1$,hyponym(NP_i, NP_0)[94]. Patterns typically have high precision but low-recall.
2. **Supervised**. The general approach for finding relations in a given sequence of words is the following:
   (a) Find all pairs of named entities in the sequence (typically a single sentence).
   (b) For each pair, use a trained binary classifier to predict whether or not the entities in the pair are indeed related.
   (c) If related, use a classifier trained to predict the relation given the entity-pair.

   As with NER, **the most important step** in this process is to identify useful surface features that will be useful for relation classification, including word features, NER features, syntactic paths (chunk seqs, constituent paths, dependency-tree paths), and more.
3. **Semi-supervised** via bootstrapping. Suppose we have a few high-precision **seed patterns** (or seed tuples)[95]. **Bootstrapping** proceeds by taking the entities in the seed pair, and then finding sentences (on the web, or whatever dataset we are using) that contain both entities. From all such sentences, we extract and generalize the context around the entities to learn new patterns.

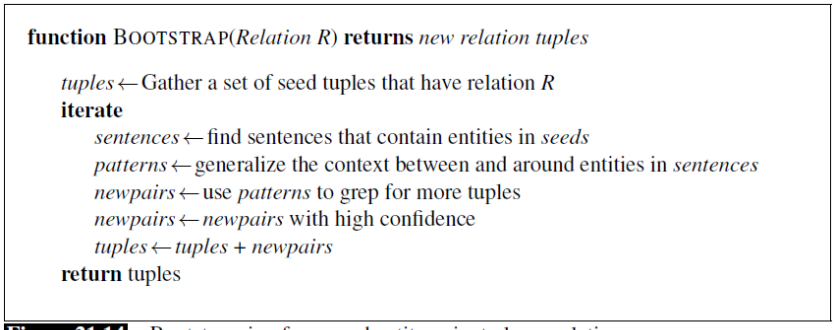


Figure 21.14 Bootstrapping from seed entity pairs to learn relations.

[94] Here, hyponym(A, B) means "A is a kind-of (hyponym) of B."

[95] seed tuples are tuples of the general form (M1, M2) where M1 and M2 are each specific named entities we know have the relation of interest R.

4. **Unsupervised**. The Re Verb system extracts a relation from a sentence s in 4 steps:

    1. Run a part-of-speech tagger and entity chunker over s
    2. For each verb in s, find the longest sequence of words w that start with a verb and satisfy syntactic and lexical constraints, merging adjacent matches.
    3. For each phrase w, find the nearest noun phrase x to the left which is not a relative pronoun, wh-word or existential "there". Find the nearest noun phrase y to the right.
    4. Assign confidence c to the relation $r = (x, w, y)$ using a confidence classifier and return it.

Event Extraction. An event mention is any expression denoting an event or state that can be assigned to a particular point, or interval, in time. Note that this is quite different than the colloquial usage of the word "event," you should think of the two as distinct. Here, most event mentions correspond to verbs, and most verbs introduce events. Event extraction is typically modeled via ML, detecting events via sequence models with BIO tagging, and assigning event classes/attributes with multi-class classifiers.

Template Filling. The task is creation of one template for each event in the input documents, with the slots filled with text from the document. For example, an event could be "Fare-Raise Attempt" with corresponding template (slots to be filled) "(<Lead Airline>, <Amount>, <Effective Date>, <Follower>)". This is generally modeled by training two separate supervised systems:

1. **Template recognition**. Trained to determine if template T is present in sentence S. Here, "present" means there is a sequence within the sentence that could be used to fill a slot within template T.
2. **Role-filler extraction**. Trained to detect each role (slot-name), e.g. "Lead Airline".

Blogs

Contents

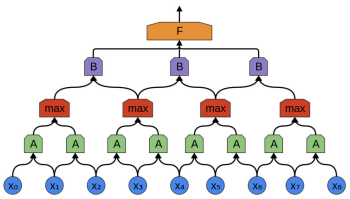
Blogs **December 21, 2016**

Conv Nets: A Modular Perspective

Table of Contents Local *Written by Brandon McKinzie*

From this post on Colah's Blog.

The title is inspired by the following figure. Colah mentions how groups of neurons, like A, that appear in multiple places are sometimes called **modules**, and networks that use them are sometimes called modular neural networks. You can feed the output of one convolutional layer into another. With each layer, the network can detect higher-level, more abstract features.



⟶ Function of the A neurons: compute certain *features*.
⟶ Max pooling layers: kind of "zoom out". They allow later convolutional layers to work on larger sections of the data. They also make us invariant to some very small transformations of the data.

Blogs **December 21, 2016**

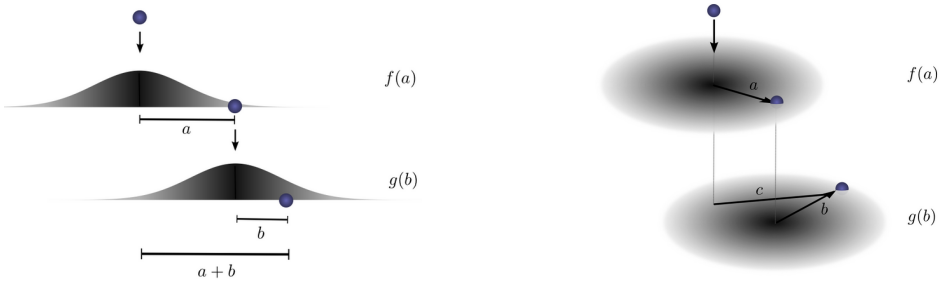
Understanding Convolutions

Table of Contents Local *Written by Brandon McKinzie*

From Colah's Blog.

Ball-Dropping Example. The posed problem:

> Imagine we drop a ball from some height onto the ground, where it only has one dimension of motion. How likely is it that a ball will go a distance c if you drop it and then drop it again from above the point at which it landed?



From basic probability, we know the result is a sum over possible outcomes, constrained by $a + b = c$. It turns out this is actually the definition of the convolution of f and g.

$$\Pr(\text{a} + \text{b} = \text{c}) = \sum_{a+b=c} f(a) \cdot g(b) \tag{296}$$

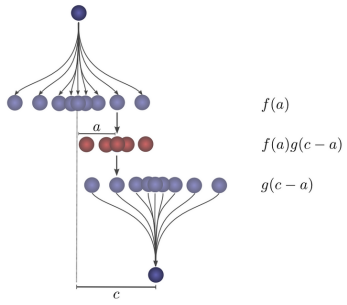
$$(f * g)(c) = \sum_{a+b=c} f(a) \cdot g(b) \tag{297}$$

$$= \sum_{a} f(a) \cdot g(c - a) \tag{298}$$

Visualizing Convolutions. Keeping the same example in the back of our heads, consider a few interesting facts.

- **Flipping directions**. If $f(x)$ yields the probability of landing a distance x away from where it was dropped, what about the probability that it was dropped a distance x from where it *landed*? Apparently[96] it is $f(-x)$.
- Above is a visualization of one term in the summation of $(f * g)(c)$. It is meant to show how we can move the bottom around to think about evaluating the convolution for different c values.

[96] Not entirely sold on the generalization of this, or even how true it is here.



We can relate these ideas to image recognition. Below are two common kernels used to convolve images with.

0	0	0	0	0
0	1/9	1/9	1/9	0
0	1/9	1/9	1/9	0
0	1/9	1/9	1/9	0
0	0	0	0	0

0	0	0	0	0
0	0	0	0	0
0	-1	1	0	0
0	0	0	0	0
0	0	0	0	0

On the left is a kernel for *blurring* images, accomplished by taking simple averages. On the right is a kernel for *edge detection*, accomplished by taking the difference between two pixels, which will be largest at edges, and essentially zero for similar pixels.

Blogs **December 23, 2016**

Deep Reinforcement Learning

Table of Contents Local *Written by Brandon McKinzie*

Link to tutorial – Part I of "Demystifying deep reinforcement learning."

Reinforcement Learning. Vulnerable to the *credit assignment problem* - i.e. unsure which of the preceding actions was responsible for getting some reward and to what extent. Also need to address the famous *explore-exploit dilemma* when deciding what strategies to use.

Markov Decision Process. Most common method for representing a reinforcement problem. MDPs consist of states, actions, and rewards. Total reward is sum of current (includes previous) and *discounted* future rewards:

$$R_t = r_t\gamma(r_{t+1} + \gamma(r_{t+2} + \ldots)) = r_t + \gamma R_{t+1} \tag{299}$$

Q - learning. Define function $Q(s, a)$ to be best possible score at end of game after performing action a in state s; the "quality" of an action from a given state. The recursive definition of Q (for one transition) is given below in the *Bellman equation*.

$$Q(s, a) = r + \gamma \max_{a'} Q(s', a')$$

and updates are computed with a learning rate α as

$$Q(s_t, a_t) = (1 - \alpha) \cdot Q(s_{t-1}, a_{t-1}) + \alpha \cdot (r + \gamma \max_{a'} Q(s'_{t+1}, a'_{t+1}))$$

Deep Q Network. Deep learning can take deal with issues related to prohibitively large state spaces. The implementation chosen by DeepMind was to represent the Q-function with a neural network, with the states (pixels) as the input and Q-values as output, where the number of output neurons is the number of possible actions from the input state. We can optimize with simple squared loss:

$$L = \tfrac{1}{2}[\underbrace{r + max_{a'} Q(s', a')}_{\text{target}} - \underbrace{Q(s, a)}_{\text{prediction}}]^2$$

and our algorithm from some state s becomes

1. **First forward pass** from s to get all predicted Q-values for each possible action. Choose action corresponding to max output, leading to next s'.

2. **Second forward pass** from s' and again compute $\max_{a'} Q(s', a')$.
3. **Set target output** for each action a' from s'. For the action corresponding to max (from step 2) set its target as $r + \gamma \max_{a'} Q(s', a')$, and for all other actions set target to same as originally returned from step 1, making the error 0 for those outputs. (Interpret as update to our guess for the best Q-value, and keep the others the same.)
4. **Update weights** using backprop.

Experience Replay. This the most important trick for helping convergence of Q-values when approximating with non-linear functions. During gameplay all the experience $< s, a, r, s' >$ are stored in a replay memory. When training the network, random minibatches from the replay memory are used instead of the most recent transition.

Exploration. One could say that initializing the Q-values randomly and then picking the max is essentially a form of exploitation. However, this type of exploration is *greedy*, which can be tamed/fixed with **ε-greedy exploration**. This incorporates a degree of randomness when choosing next action at *all* time-steps, determined by probability ε that we choose the next action randomly. For example, DeepMind decreases ε over time from 1 to 0.1.

Deep Q-Learning Algorithm.

```
initialize replay memory D
initialize action-value function Q with random weights
observe initial state s
repeat
      select an action a
            with probability ε select a random action
            otherwise select a = argmax_a' Q(s,a')
      carry out action a
      observe reward r and new state s'
      store experience <s, a, r, s'> in replay memory D

      sample random transitions <ss, aa, rr, ss'> from replay memory D
      calculate target for each minibatch transition
            if ss' is terminal state then tt = rr
            otherwise tt = rr + γmax_a' Q(ss', aa')
      train the Q network using (tt - Q(ss, aa))^2 as loss

      s = s'
until terminated
```

Blogs **January 15, 2017**

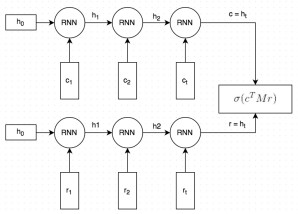
Deep Learning for Chatbots (WildML)

Table of Contents Local *Written by Brandon McKinzie*

Overview.

- **Model**. Implementing a retrieval-based model. Input: conversation/context c. Output: response r.
- **Data**. Ubuntu Dialog Corpus (UDC). 1 million examples of form (context, utterance, label). The label can be 1 (utterance was actual response to the context) or a 0 (utterance chosen randomly). Using NLTK, the data has been . . .
  → **Tokenized**: dividing strings into lists of substrings.
  → **Stemmed**. **IDK**
  → **Lemmatized**. **IDK**
  The test/validation set consists (context, ground-truth utterance, [9 distractors (incorrect utterances)]). The distractors are picked at random[97]

Dual-Encoder LSTM.



1. **Inputs**. Both the context and the response text are split by words, and each word is embedded into a vector and fed into the same RNN.
2. **Prediction**. Multiply the [vector representation ("meaning")] c with param matrix M to predict some response r'.
3. **Evaluation**. Measure similarity of predicted r' to actual r via simple dot product. Feed this into sigmoid to obtain a probability [of r' being the correct response]. Use (binary) cross-entropy for loss function:

$$L = -y \cdot \ln(y') - (1 - y) \cdot \ln(1 - y') \tag{300}$$

where y' is the predicted probability that r' is correct response r, and $y \in \{0, 1\}$ is the true label for the context-response pair (c, r).

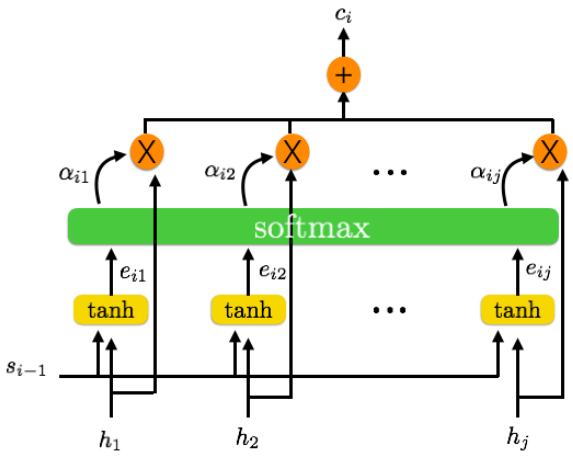
[97] Better example/approach: Google's Smart Reply uses clustering techniques to come up with a set of possible responses.

Data Pre-Processing. Courtesy of WildML, we are given 3 files after preprocessing: train.tfrecords, validation.tfrecords, and test.tfrecords, which use TensorFlow's 'Example' format. Each Example consists of . . .

- context: Sequence of word ids.
- context_len: length of the aforementioned sequence.
- utterance: seq of word ids representing utterance (response).
- utterance_len.
- label: only in training data. 0 or 1.
- distractor_[N]: Only in test/validation. N ranges from 0 to 8. Seq of word ids reppin the distractor utterance.
- distractor_[N]_len.

[Link to article]

Attention Mechanism. Below is a close-up view/diagram of an attention layer. Technically, it only corresponds to a single time step i; we are using the previous decoder state s_{i-1} to compute the ith context vector c_i which will be fed as an input to the decoder for step i.



For convenience, I'll rewrite the familiar equations for computing quantities at some step i.

$$\text{[decoder state]} \qquad s_i = f(s_{i-1}, y_{i-1}, c_i) \tag{301}$$

$$\text{[context vect]} \qquad c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j \tag{302}$$

$$\text{[annotation weights]} \qquad \alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})} \tag{303}$$

$$e_{ij} = a(s_{i-1}, h_j) \tag{304}$$

Now we can see just howthe simple this really is. Recall that Bahdanau *et al.*, 2015 use the wording: "e_{ij} *is an alignment model which scores how well the inputs around position* j *and the output at position* i *match.*" But we can see an example implementation of an alignment model above: the *tanh* function (that's it).

APPENDIX A - Synthesizing What I've Learned

TODO...

APPENDIX B - Questions and Stuff I Always Forget

Questions:

- **Q:** In general, how can one tell if a matrix $\boldsymbol{A}$ has an eigenvalue decomposition? [insert more conceptual matrix-related questions here . . .]
- **Q:** Let $\boldsymbol{A}$ be real-symmetric. What can we say about $\boldsymbol{A}$?
  - Proof that eigendecomposition $A = Q\Lambda Q^T$ exists: Wow this is apparently quite hard to prove according to many online sources. Guess I don't feel so bad now that it wasn't (and still isn't) obvious.
  - Eigendecomposition not unique. This is apparently because two or more eigenvectors may have same eigenvalue.

This is the principal axis theorem: if A symmetric, then orthonorm basis of e-vects exists.

Stuff I Forget:

- Existence of eigenvalues/eigenvectors. Let $\boldsymbol{A} \in \mathbb{R}^{n \times n}$.
  - λ is an eigenvalue of A iff it satisfies $\det(\lambda \boldsymbol{I} - \boldsymbol{A}) = 0$. Why? Because it is an equivalent statement as requiring that $(\lambda \boldsymbol{I} - \boldsymbol{A})\boldsymbol{x} = 0$ has a nonzero solution for $\boldsymbol{x}$.
  - The following statements are equivalent:
    - A is diagonalizable.
    - A has n linearly independent eigenvectors.
  - The **eigenspace** of $\boldsymbol{A}$ corresponding to λ is the solution space of the homogeneous system $(\lambda I - A)x = 0$.
  - $\boldsymbol{A}$ has at most n distinct eigenvalues.

Most info here comes from chapter 5 of your "Elementary Linear Algebra" textbook (around pg305)

- Diagonalizability notes from 5.2 of advanced linear alg. book (261). Recall that $\boldsymbol{A}$ is defined to be diagonalizable if and only if there exists and ordered basis β for the space consisting of eigenvectors of $\boldsymbol{A}$.
  - If the standard way of finding eigenvalues leads to k distinct λ_i, then the corresponding set of k eigenvectors v_i are guaranteed to be linearly independent (but might not span the full space).
  - If $\boldsymbol{A}$ has n linearly independent eigenvectors, then $\boldsymbol{A}$ is diagonalizable.
  - The characteristic polynomial of any diagonalizable linear operator splits (can be factored into product of linear factors). The **algebraic multiplicity** of an eigenvalue λ is the largest positive integer k for which $(t - \lambda)^k$ is a factor of $f(t)$.

Recall that a linear operator is a special case of a linear map where the input space is the same as the output space.

- Logistic regression is called a *linear model* because it classifies based on the linear $\boldsymbol{\theta}^T \boldsymbol{x}$ before feeding that to the logistic sigmoid function.

Coreference Resolution. The task of finding all expressions that refer to the same entity in a text. It is an important step for a lot of higher level NLP tasks that involve natural language understanding such as document summarization, question answering, and information extraction.

Conditional Random Field (CRF). A type of discriminative undirected probabilistic graphical model[98]. The CRF defines the probability, $p(\ell \mid s)$, of a given sentence s being described with the [vector] label ℓ as follows:

See this post by Edwin Chen for a great CRF intro.

$$\text{score}(\ell \mid s) = \sum_{j=1}^{m} \sum_{i=1}^{n} \lambda_j f_j(s, i, \ell_i, \ell_{i-1}) \tag{305}$$

$$p(\ell \mid s) = \frac{e^{\text{score}(\ell|s)}}{\sum_{\ell'} e^{\text{score}(\ell'|s)}} \tag{306}$$

A label element, ℓ_i, could be the POS label for the ith word. The sum in the denominator of (2) is over all possible label combinations for s.

where m is the number of features (e.g. different POS labels) and n is the number of words/tokens in sentence s. So, where's the graph part, you may ask? It can be inferred from the above equation. I like to imagine a picture where I have my set of label nodes $\boldsymbol{Y}$ and set of word nodes $\boldsymbol{X}$ for the sentence s. Note that all nodes are modeled as random variables. The built-in assumption of the CRF is that the value of $\boldsymbol{Y}_i$, the label for the ith word in s, is conditionally independent of the other labels $\boldsymbol{Y}_j$ *given* the previous label $\boldsymbol{Y}_{i-1}$.

- Often used for labeling or parsing of sequential data, e.g. POS tagging, NER.

Question/Answer.

- **Q**: How is this different from logistic regression?
  - **A**: Indeed, this would be identical to logistic regression if we viewed all possible label [vector] combinations ℓ as black boxes with no internal structure (i.e. replace their vector representations with integer ids). The difference is that we are exploiting the sequential interdependencies of the structure of ℓ – we are explicitly modeling the labels as *sequential*.
- **Q**: How is this different from HMMs?
  - **A**: Basically, we can build a CRF equivalent to any HMM, but CRFs are able to define more features and they can have arbitrary weights. Let's look into this in more detail (TODO).
- **Q**: How is this different from basically any neural network-based text classifier? They all accept sequences and output a label prediction.
  - **A**: .

[98] From browsing the literature, it appears that *by far* the most important CRF for sequential labels with NLP is the **linear-chain CRF**, so much so that often authors will say CRF when they really mean linear-chain CRF (and henceforth so will I)