Contents

Math and Machine Learning

Basics

Contents

Linear Algebra (Quick Review): January 23

Written by Brandon McKinzie

- 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]

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

[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)

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

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:

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 eigenbasis 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,

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

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

  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.

- 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

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

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.

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

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

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

$$\Sigma := \frac{1}{m}\sum_{k=1}^{m}(x_k - \bar{x})(x_k - \bar{x})^T \tag{12}$$

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,

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

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

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

shows us how $\boldsymbol{H}$ can give information related to how "good" of a step that really was. Mathematically,

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

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

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**.

$$\text{bias}\left[\hat{\theta}_m\right] = \mathbb{E}\left[\hat{\theta}_m\right] - \theta \tag{13}$$

$$\text{Var}\left[\hat{\theta}_m\right] = \mathbb{E}\left[\left(\hat{\theta}_m - \mathbb{E}\left[\hat{\theta}_m\right]\right)^2\right] \tag{14}$$

→ **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:

$$\begin{aligned} \text{MSE} &= \mathbb{E}\left[\hat{\theta}^2 + \theta^2 - 2\theta\hat{\theta}\right] && (15)\\ &= \mathbb{E}\left[\hat{\theta}^2\right] + \theta^2 - 2\theta\mathbb{E}\left[\hat{\theta}\right] && (16)\\ &= (\mathbb{E}\left[\hat{\theta}\right]^2 - \mathbb{E}\left[\hat{\theta}\right]^2) + \mathbb{E}\left[\hat{\theta}^2\right] + \theta^2 - 2\theta\mathbb{E}\left[\hat{\theta}\right] && (17)\\ &= \left(\mathbb{E}\left[\hat{\theta}\right]^2 + \theta^2 - 2\theta\mathbb{E}\left[\hat{\theta}\right]\right) + \left(\mathbb{E}\left[\hat{\theta}^2\right] - \mathbb{E}\left[\hat{\theta}\right]^2\right) && (18)\\ &= \text{Bias}(\hat{\theta})^2 + \text{Var}\left[\hat{\theta}_m\right] && (19) \end{aligned}$$

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 $m \to \infty$.

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.

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}.

[7] The KL divergence is given by

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

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

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 $\frac{1}{m}\sum_{i=1}^{m} ||\hat{y}^{(i)} - y^{(i)}||^2$

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.

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

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

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

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

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

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...

Recall the ReLU activation function: $g(z) = \max\{0, z\}$

- 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)

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 $z_i < 0$:

$$h_i = g(\boldsymbol{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i) \tag{20}$$

→ 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: January 12

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!

$$\boldsymbol{x}' \longleftarrow \boldsymbol{x} + \epsilon \cdot \text{sign}\left(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y})\right) \tag{21}$$

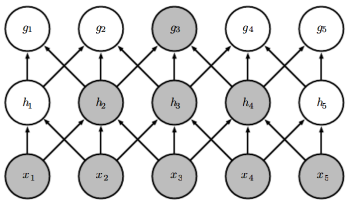
In the context of regularization, one can reduce the error rate on the original i.i.d. test set via **adversarial training** – training on adversarially perturbed training examples.

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.

Modern Practices **Fall 2016**

Sequence Modeling (RNNs): January 15

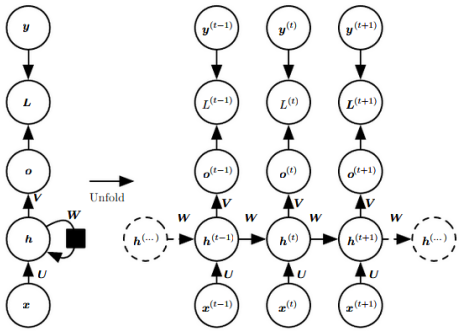
Table of Contents Local *Written by Brandon McKinzie*

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) $\boldsymbol{y}^{(t)}$.

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

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

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

[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.

steps as:

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

Convince yourself this is identical to cross-entropy.

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.

$$\begin{aligned}
\left(\nabla_{\boldsymbol{o}^{(t)}} L\right)_i &= \frac{\partial L}{\partial \boldsymbol{o}_i^{(t)}} \\
&= \frac{\partial L}{\partial L^{(t)}} \cdot \frac{\partial L^{(t)}}{\partial \boldsymbol{o}_i^{(t)}} \\
&= (1) \cdot \frac{\partial L^{(t)}}{\partial \boldsymbol{o}_i^{(t)}} \\
&= \frac{\partial}{\partial \boldsymbol{o}_i^{(t)}} \left\{ -\log\left(\hat{\boldsymbol{y}}_{y^{(t)}}^{(t)}\right) \right\} \\
&= -\frac{\partial}{\partial \boldsymbol{o}_i^{(t)}} \left\{ \log\left(\frac{e^{\boldsymbol{o}_{y^{(t)}}^{(t)}}}{\sum_j e^{\boldsymbol{o}_j^{(t)}}}\right) \right\} \\
&= -\frac{\partial}{\partial \boldsymbol{o}_i^{(t)}} \left\{ \boldsymbol{o}_{y^{(t)}}^{(t)} - \log\left(\sum_j e^{\boldsymbol{o}_j^{(t)}}\right) \right\} \\
&= -\left\{ \mathbf{1}_{i,y^{(t)}} - \frac{\partial}{\partial \boldsymbol{o}_i^{(t)}} \log\left(\sum_j e^{\boldsymbol{o}_j^{(t)}}\right) \right\}
\end{aligned} \tag{22}$$

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

[12] More generally,

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

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 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.

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

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

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

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

[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

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

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

$$\nabla_{\mathbf{V}} L = \sum_t^\tau \nabla_{\mathbf{V}} L^{(t)} \tag{23a}$$

$$= \sum_t^\tau \nabla_{\mathbf{V}} L^{(t)}(\boldsymbol{o}_1^{(t)}, \ldots, \boldsymbol{o}_{n_{out}}^{(t)}) \tag{23b}$$

$$= \sum_t^\tau \sum_i^{n_{out}} \left(\nabla_{\boldsymbol{o}^{(t)}} L\right)_i \nabla_{\mathbf{V}} \boldsymbol{o}_i^{(t)} \tag{23c}$$

$$= \sum_t^\tau \sum_i^{n_{out}} \left(\nabla_{\boldsymbol{o}^{(t)}} L\right)_i \nabla_{\mathbf{V}} \left\{ c_i + \sum_{j=1}^{n_{hid}} V_{ij} \boldsymbol{h}_j^{(t)} \right\} \tag{23d}$$

$$= \sum_t^\tau \sum_i^{n_{out}} \left(\nabla_{\boldsymbol{o}^{(t)}} L\right)_i \begin{bmatrix} 0 & 0 & \ldots & 0 \\ \vdots & \vdots & \ldots & \vdots \\ \boldsymbol{h}_1^{(t)} & \boldsymbol{h}_2^{(t)} & \ldots & \boldsymbol{h}_{n_{hid}}^{(t)} \\ \vdots & \vdots & \ldots & \vdots \\ 0 & 0 & \ldots & 0 \end{bmatrix} \tag{23e}$$

NOTE: In equation 23e, all rows are 0 except the ith row, which is $(\boldsymbol{h}^{(t)})^T$. (See footnote for more)

$$= \sum_t^\tau \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) (\boldsymbol{h}^{(t)})^T \tag{23f}$$

where if 23e confuses you, see the footnote[14].

- $\mathbf{W}$ $(n_{hid} \times n_{hid})$. 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

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

$$\sum_i \nabla_{\mathbf{W}} \left[(\boldsymbol{W}x)_i\right] = \begin{bmatrix} \boldsymbol{x}^T \\ \boldsymbol{x}^T \\ \vdots \\ \boldsymbol{x}^T \end{bmatrix} \tag{24}$$

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) $\boldsymbol{W}$.

connections to $\boldsymbol{h}$ at time t.

$$\nabla_{\mathbf{W}} L = \sum_t^{\tau} \nabla_{\mathbf{W}} L^{(t)} \tag{25a}$$

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

$$= \sum_t^{\tau} \sum_i^{n_{hid}} \left(\nabla_{\boldsymbol{h}^{(t)}} L\right)_i \left(\operatorname{diag}\left(1 - (\boldsymbol{h}^{(t)})^2\right) \begin{bmatrix} 0 & 0 & \dots & 0 \\ \vdots & \vdots & \dots & \vdots \\ \boldsymbol{h}_1^{(t-1)} & \boldsymbol{h}_2^{(t-1)} & \dots & \boldsymbol{h}_{n_{hid}}^{(t-1)} \\ \vdots & \vdots & \dots & \vdots \\ 0 & 0 & \dots & 0 \end{bmatrix} \right) \tag{25b}$$

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

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

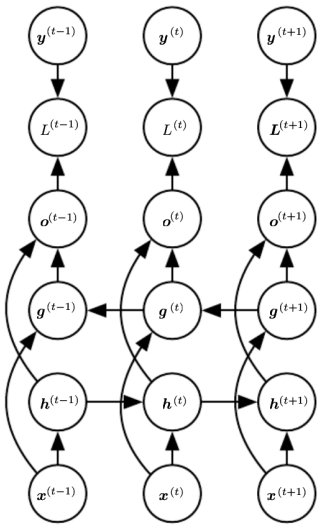
$$\nabla_{\mathbf{U}} L = \sum_t^{\tau} \nabla_{\mathbf{U}} L^{(t)} \tag{26a}$$

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

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

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. The author then only discusses methods of determining stopping conditions for sampling (pg. 380), and not really "how to draw samples," which I suppose is self-explanatory.

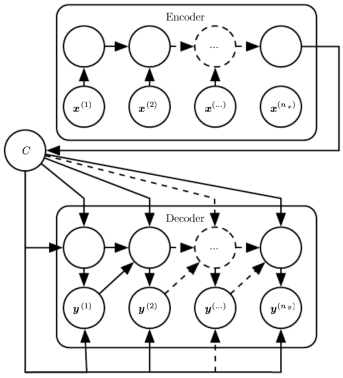
Bidirectional RNNs. In many applications, it is desirable to output a prediction of $\boldsymbol{y}^{(t)}$ that may depend on *the whole sequence*. For example, in speech recognition, the interpretation of words/sentences can also depend on what is *about* to be said. Below is a typical bidirectional 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.

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.[16]

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 $w^{(t)}$ are i.i.d. with mean 0 and variance v. The variance of the product is easily seen to be $\mathcal{O}(v^n)$[17]. *To obtain some desired variance v^* we may choose the individual weights with variance $v = \sqrt[n]{v^*}$.*

[16] 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.

[17] Quick sketch of (my) proof:

$$\text{Var}\left[w^{(i)}\right] = v = \mathbb{E}\left[(w^{(i)})^2\right] - \cancel{\mathbb{E}\left[w^{(i)}\right]^2} \tag{27}$$
$$\text{Var}\left[\prod_t^n w^{(t)}\right] = \mathbb{E}\left[\left(\prod_t^n w^{(t)}\right)^2\right] = \prod_t^n \mathbb{E}\left[(w^{(t)})^2\right] = v^n \tag{28}$$

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[18]

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** $f_i^{(t)}$.
- **Internal state** $s_i^{(t)}$.
- **External input gate** $g_i^{(t)}$.
- **Output gate** $q_i^{(t)}$.

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

[18] 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 $n \geq 3$, $P(x_n \mid x_1, \ldots, x_{n-1}) = P(x_n \mid x_{n-2}, x_{n-1})$.
  - 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[19], even for modest n.

[19] For a given vocabulary, which usually has much more than n possible words, consider how many possible sequences of length 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[20] 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] 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 $(\text{VOCAB\_SIZE})^5$, 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): January 12

Written by Brandon McKinzie

Overview. Much research is in building a *probabilistic model*[21] 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

$$p_{model}(x) = \sum_h \Pr(h) \Pr(x \mid h) = \mathbb{E}_h \left[p_{model}(x \mid h) \right] \tag{29}$$

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 just means that the elements of $\boldsymbol{h}$ are mutually independent[22]. 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:

$$\boldsymbol{x} = \boldsymbol{W}\boldsymbol{h} + \boldsymbol{b} + \text{noise} \tag{30}$$

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

[22] 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.

Probabilistic PCA and Factor Analysis.

- **Factor analysis**:

$$\boldsymbol{h} \sim \mathcal{N}(\boldsymbol{h}; \mathbf{0}, \boldsymbol{I}) \tag{31}$$

$$\text{noise} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\psi} \equiv \text{diag}(\boldsymbol{\sigma^2})) \tag{32}$$

$$\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{x};\ \boldsymbol{b}, \boldsymbol{W}\boldsymbol{W}^T + \boldsymbol{\psi}) \tag{33}$$

  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 $\text{Cov}(\boldsymbol{x}) = \boldsymbol{W}\boldsymbol{W}^T + \boldsymbol{\psi}$.

  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 $\Pr(x_i|\boldsymbol{h}, x_{j\neq i}) = \Pr(x_i|\boldsymbol{h})$[23].

  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 $\boldsymbol{h}$.

- **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 $\boldsymbol{W}$.

[23] 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.

Autoencoders (Ch. 14): May 07

Table of Contents Local

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.

$$f(\cdot) \leftarrow I(\cdot) \tag{34}$$
$$\tilde{\boldsymbol{X}} = \boldsymbol{X} \tag{35}$$

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

$$f^{(k)} = \mathcal{L}(\tilde{\boldsymbol{X}}) \tag{36}$$
$$f(\cdot) \leftarrow f^{(k)}\left(f\left(\cdot\right)\right) \tag{37}$$
$$\tilde{\boldsymbol{X}} \leftarrow f^{(k)}(\tilde{\boldsymbol{X}}) \tag{38}$$

   In English: just apply the regular learning/training process for each layer/stage **sequentially and individually**[24].

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).

[24] 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.

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:

$$s = \int p(\boldsymbol{x}) f(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} = \mathbb{E}_p \left[f(\mathbf{x}) \right] \quad \rightarrow \quad \hat{s}_n = \frac{1}{n} \sum_{i=1,\ \mathbf{x}^{(i)} \sim p}^{n} f(\boldsymbol{x}^{(i)}) \tag{39}$$

As we've seen before, the empirical average is an unbiased[25] 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:

$$p(\boldsymbol{x}) f(\boldsymbol{x}) = q(\boldsymbol{x}) \frac{p(\boldsymbol{x}) f(\boldsymbol{x})}{q(\boldsymbol{x})} \tag{43}$$

- **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

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

$$\mathbb{E}_p \left[\hat{s}_n \right] = \frac{1}{n} \sum_{i=1}^{n} \mathbb{E}_p \left[f(\boldsymbol{x}^{(i)}) \right] \tag{40}$$

$$= \frac{1}{n} \sum_{i=1}^{n} s \tag{41}$$

$$= s \tag{42}$$

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.

to denote the sampling distribution) as $\hat{s}_q$:

$$\hat{s}_q = \frac{1}{n} \sum_{i=1,\ \mathbf{x}^{(i)} \sim q}^{n} \frac{p(\boldsymbol{x}^{(i)}) f(\boldsymbol{x}^{(i)})}{q(\boldsymbol{x}^{(i)})} \tag{44}$$

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:

$$\mathrm{Var}\left[\hat{s}_q\right] = \mathrm{Var}\left[\frac{p(\boldsymbol{x}) f(\boldsymbol{x})}{q(\boldsymbol{x})}\right] / n \tag{45}$$

with the optimal (minimum) value of q at:

$$q* = \frac{p(\boldsymbol{x}) \mid f(\boldsymbol{x}) \mid}{Z} \tag{46}$$

- **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:

$$\hat{s}_{BIS} = \frac{\sum_{i=1}^{n} \frac{\tilde{p}(\boldsymbol{x}^{(i)})}{\tilde{q}(\boldsymbol{x}^{(i)})} f(\boldsymbol{x}^{(i)})}{\sum_{i=1}^{n} \frac{\tilde{p}(\boldsymbol{x}^{(i)})}{\tilde{q}(\boldsymbol{x}^{(i)})}} \tag{47}$$

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

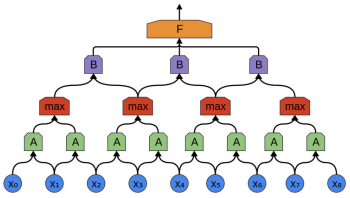
Contents

Conv Nets: A Modular Perspective: December 21

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.



$\longrightarrow$ Function of the A neurons: compute certain *features*.

$\longrightarrow$ 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.

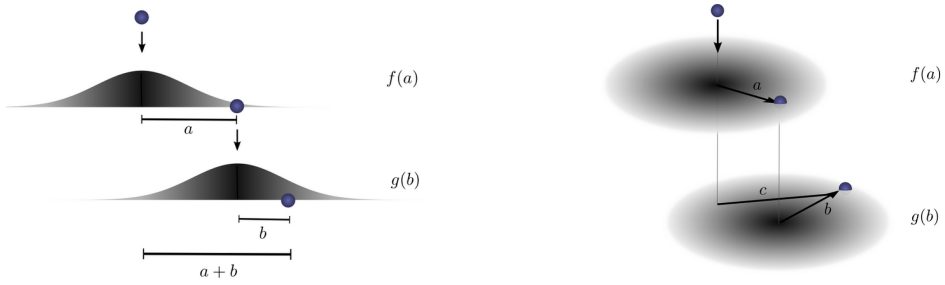
Understanding Convolutions: December 21

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.

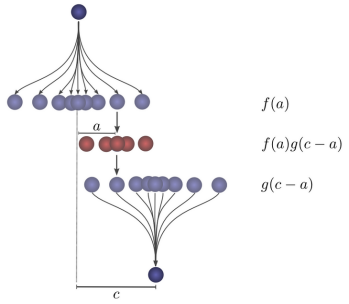
$$\Pr(\text{a} + \text{b} = \text{c}) = \sum_{a+b=c} f(a) \cdot g(b) \tag{48}$$

$$(f * g)(c) = \sum_{a+b=c} f(a) \cdot g(b) \tag{49}$$

$$= \sum_{a} f(a) \cdot g(c - a) \tag{50}$$

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[26] 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.

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.

[26] Not entirely sold on the generalization of this, or even how true it is here.

Deep Reinforcement Learning: December 23

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:

$$R_t = r_t \gamma(r_{t+1} + \gamma(r_{t+2} + \ldots)) = r_t + \gamma R_{t+1} \tag{51}$$

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:

$$L = \frac{1}{2}[\underbrace{r + max_{a'}Q(s', a')}_{\text{target}} - \underbrace{Q(s, a)}_{\text{prediction}}]^2$$

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
```

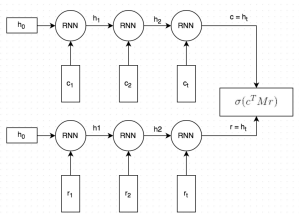
Deep Learning for Chatbots (WildML): January 15

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[27]

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:

$$L = -y \cdot \ln(y') - (1 - y) \cdot \ln(1 - y') \tag{52}$$

[27] Better example/approach: Google's Smart Reply uses clustering techniques to come up with a set of possible responses.

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).

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.

WaveNet (Paper): January 15

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 $x = \{x_1, \dots, x_T\}$ is factorised as a product of conditional probabilities,

$$p(x) = \prod_{t=1}^{T} p(x_t \mid x_1, \dots, x_{t-1}) \tag{53}$$

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[28].

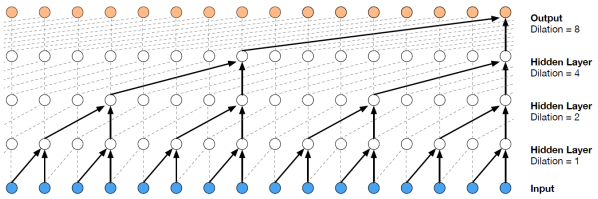


Figure 3: Visualization of a stack of *dilated* causal convolutional layers.

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 $p(x_t \mid x_1, \ldots, x_{t-1})$ 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:

$$f(x_t) = \text{sign}(x_t)\frac{\ln(1+\mu|x_t|)}{\ln(1+\mu)} \tag{54}$$

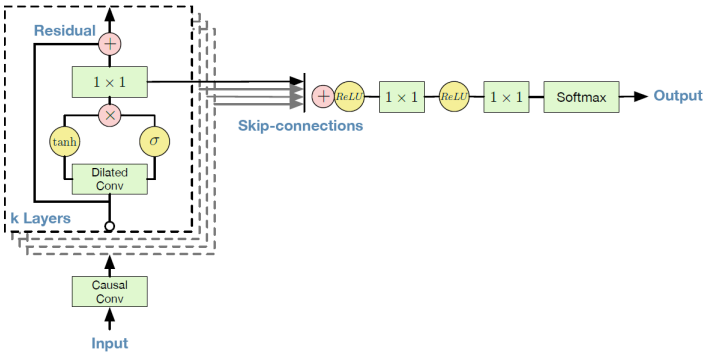
which (after plotting in Wolfram) looks identical to the sigmoid function.

[28]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"

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

$$z = \tanh\left(W_{f,k} * x\right) \odot \sigma\left(W_{g,k} * x\right) \tag{55}$$

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).

$$p(x \mid h) = \prod_{t=1}^{T} p(x_t \mid x_1, \ldots, x_{t-1}, h) \tag{56}$$

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

$$z = \tanh\left(W_{f,k} * x + V_{f,k}^T h\right) \odot \sigma\left(W_{g,k} * x + V_{g,k}^T h\right) \tag{57}$$

→ **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[29]. 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."

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."

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

Neural Style: January 22

 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)[30].
- **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 $\boldsymbol{p}$.

[30] 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 **Fall 2016**

Neural Conversation Model: February 8

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.

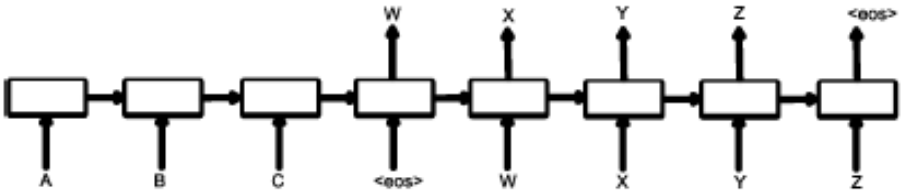
Related Work. The authors' approach is based on the following (linked and saved) papers on seq2seq:

- 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 greedy 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][31].

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.

[31] 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.

[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[32] and translate.

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

$$p(y_i \mid y_1, \dots, y_{i-1}, \boldsymbol{x}) = g(y_{i-1}, s_i, c_i) \tag{58}$$
$$s_i = f(s_{i-1}, y_{i-1}, c_i) \tag{59}$$

  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:

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j \tag{60}$$
$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})} \tag{61}$$
$$e_{ij} = a(s_{i-1}, h_j) \tag{62}$$

    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.

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

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

Detailed Model Architecture

(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**:

$$z_t = \sigma\left(W_z y_{t-1} + U_z[s_{t-1},\ c_t]\right) \qquad \text{[update gate]} \tag{63}$$
$$r_t = \sigma\left(W_r y_{t-1} + U_r[s_{t-1},\ c_t]\right) \qquad \text{[reset gate]} \tag{64}$$
$$\tag{65}$$

3. **Proposal state**:

$$\tilde{s}_t = \tanh\left(W y_{t-1} + U[r_t \circ s_{t-1},\ c_t]\right) \tag{66}$$

4. **Hidden state**:

$$s_t = (1 - z_t) \circ s_{t-1} + z_t \circ \tilde{s}_t \tag{67}$$

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:

$$a_j = a(s_{t-1}, h_j) = v_a^T \tanh\left(W_a s_{t-1} + U_a h_j\right) \tag{68}$$

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

$$a_j \leftarrow \frac{e^{a_j}}{\sum_{k=0}^{L_{enc}-1} e^{a_k}} \tag{69}$$

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

$$c = \sum_{j=1}^{L_{enc}} a_j h_j \tag{70}$$

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]:

$$\Pr(y \mid s, c) \propto e^{y^T W_o u} \tag{71}$$

$$u = [\max\{\tilde{u}_{2j-1}, \tilde{u}_{2j}\}]_{j=1,\ldots,\ell}^{T} \tag{72}$$

$$\tilde{u} = U_o[s_{t-1},\ c] + V_o y_{t-1} \tag{73}$$

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:

$$\tilde{\boldsymbol{h}}_t = \tanh\left(\boldsymbol{W}_c[\boldsymbol{c}_t;\ \boldsymbol{h}_t]\right) \tag{74}$$

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

$$p(y_t \mid y_{<t}, x) = \text{softmax}\left(\boldsymbol{W}_s \tilde{\boldsymbol{h}}_t\right) \tag{75}$$

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:

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^T \bar{\boldsymbol{h}}_s & \textit{dot} \\ \boldsymbol{h}_t^T \boldsymbol{W}_a \bar{\boldsymbol{h}}_s & \textit{general} \\ \boldsymbol{v}_a^T \tanh\left(\boldsymbol{W}_a[\boldsymbol{h}_t;\ \bar{\boldsymbol{h}}_s]\right) & \textit{concat} \end{cases} \tag{76}$$

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

$$\boldsymbol{a}_t(s) = \text{align}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) \tag{77}$$

$$= \frac{\exp(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s))}{\sum_{s'} \exp(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_{s'}))} \tag{78}$$

3. $\boldsymbol{c}_t$: The weighted average over all source (encoder) hidden states[33]:

$$\boldsymbol{c}_t = \sum_{i=1}^{L_{enc}} \boldsymbol{a}_t(i)\bar{\boldsymbol{h}}_i \tag{79}$$

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

$$\tilde{\boldsymbol{h}}_t = \tanh\left(\boldsymbol{W}_c[\boldsymbol{c}_t;\ \boldsymbol{h}_t]\right) \tag{80}$$

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 $[\tilde{\boldsymbol{h}}_t; \boldsymbol{x}_{t+1}]$.

[33]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: March 11

 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:

$$x = (x_1, \dots, x_T) \qquad \textbf{[source sentence]} \tag{81}$$
$$h = (h_1, \dots, h_T) \qquad \textbf{[encoder state seq]} \tag{82}$$
$$h_t = f(x_t, h_{t-1}) \tag{83}$$

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:

$$y = (y_1, \dots, y'_T) \qquad [y_i \in \mathbb{Z}] \tag{84}$$
$$\Pr[y_t \mid y_{<t},\ x] \propto e^{q(y_{t-1},\ z_t,\ c_t)} \tag{85}$$
$$z_t = g(y_{t-1}, z_{t-1}, c_t) \qquad \textbf{[decoder hidden?]} \tag{86}$$
$$c_t = r(z_{t-1}, h_1, \dots, h_T) \qquad \textbf{[decoder inp?]} \tag{87}$$

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,

$$\theta^* = \arg\max_\theta \sum_{n=1}^{N} \sum_{t=1}^{T_n} \log\left(\Pr[y_t^n \mid y_{<t}^n,\ x^n]\right) \tag{88}$$

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.

$$c_t = \sum_{i=1}^{T} \alpha_i h_i \tag{89}$$

$$\alpha_t = \frac{e^{a(h_t, z_{t-1})}}{\sum_k e^{a(h_k, z_{t-1})}} \tag{90}$$

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 85 as[34] ,

$$\Pr[y_t \mid y_{<t},\ x] = \frac{1}{Z} e^{\mathbf{w}_t^T \phi(y_{t-1}, z_t, c_t) + b_t} \tag{91}$$

$$Z = \sum_{k:\ y_k \in V} e^{\mathbf{w}_k^T \phi(y_{t-1}, z_t, c_t) + b_k} \tag{92}$$

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[35], written in terms of the energy $\mathcal{E}$.

$$\nabla \log\left(\Pr[y_t \mid y_{<t},\ x]\right) = \nabla \mathcal{E}(y_t) - \sum_{k:\ y_k \in V} \Pr[y_k \mid y_{<t},\ x] \nabla \mathcal{E}(y_k) \tag{93}$$

$$\mathcal{E}(y_j) = \mathbf{w}_j^T \phi(y_{t-1}, z_t, c_t) + b_j \tag{94}$$

[34]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.

[35]**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'.

$$\mathbb{E}_P[\nabla\mathcal{E}(y)] \approx \sum_{k:\ y_k \in V'} \frac{\omega_k}{\sum_{k':y_{k'}\in V'} \omega_{k'}} \nabla\mathcal{E}(y_k) \tag{95}$$

$$\omega_k = e^{\mathcal{E}(y_k) - \log Q(y_k)} \tag{96}$$

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

$$\omega_k = \frac{e^{\mathcal{E}(y_k)}}{Q(y_k)} \quad \text{thus} \quad p(y_k \mid y_{<t}, x) = \omega_k \frac{Q(y_k)}{Z} \tag{97}$$

$$\rightarrow \quad e^{\mathcal{E}(y_k)} = Z \cdot p(y_k \mid y_{<t}, x) = Q(y_k) \cdot \omega_k \tag{98}$$

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.

$$\mathbb{E}_P[\nabla\mathcal{E}(y)] = \sum_{k:\ y_k \in V} \frac{\omega_k \cdot Q(y_k)}{\sum_{k':y_{k'}\in V} \omega_{k'} \cdot Q(y_{k'})} \nabla\mathcal{E}(y_k) \tag{99}$$

$$\mathbb{E}_P[\nabla\mathcal{E}(y)] = \sum_{k:\ y_k \in V'} \frac{\omega_k}{\sum_{k':y_{k'}\in V'} \omega_{k'}} \nabla\mathcal{E}(y_k) \tag{100}$$

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 . 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: March 19

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:

$$\Pr\left(y_i^{true} = y \mid C_i,\ x_i\right) = \frac{\Pr\left(y_i^{true} = y \mid x_i\right) \cdot \Pr\left(C_i \mid y_i^{true} = y,\ x_i\right)}{\Pr\left(C_i \mid x_i\right)} \tag{101}$$

$$= \frac{\Pr\left(y \mid x_i\right)}{Q\left(y \mid x_i\right)} \cdot \frac{1}{K(x_i, C_i)} \tag{102}$$

where they've just defined

$$K(x_i, C_i) \triangleq \frac{\Pr\left(C_i \mid x_i\right)}{\prod_{y' \in C_i} Q(y' \mid x_i) \prod_{y' \in (L - C_i)} \left(1 - Q(y' \mid x_i)\right)} \tag{103}$$

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

$$F(x, y) = \log(\Pr(y \mid x)) + K(x) \tag{104}$$

which is the general form because

$$\text{Softmax}(\log(\Pr(y \mid x)) + K(x)) = \frac{e^{\log(\Pr(y|x))+K(x)}}{\sum_{y'} e^{\log(\Pr(y'|x)+K(x)}} \tag{105}$$

$$= \Pr(y \mid x) \tag{106}$$

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 104 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 102. We can easily satisfy that by subtracting $\log(Q(y \mid x))$ from $F(x, y)$ right before feeding into the softmax.

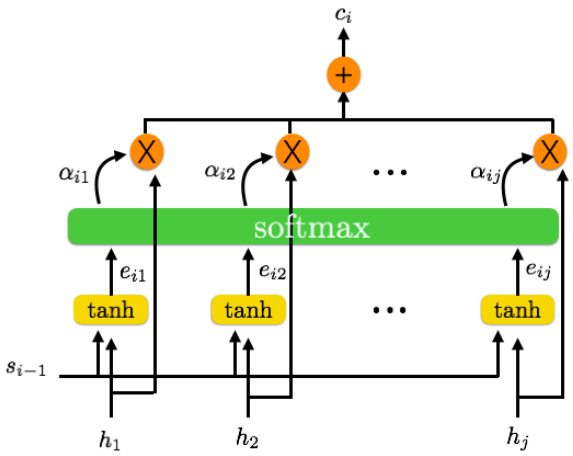
  TL;DR. Train network to learn $F(x, y)$ before softmax, but instead of feeding $F(x, y)$ to softmax directly, feed

$$\text{Softmax Input: } F(x, y) - \log(Q(y \mid x)) \tag{107}$$

  instead. That's it.

[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.

$$\textbf{[decoder state]} \qquad s_i = f(s_{i-1}, y_{i-1}, c_i) \tag{108}$$

$$\textbf{[context vect]} \qquad c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j \tag{109}$$

$$\textbf{[annotation weights]} \qquad \alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})} \tag{110}$$

$$e_{ij} = a(s_{i-1}, h_j) \tag{111}$$

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).

Attention Terminology: April 04

 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

$$e_i = v^T \tanh(\text{FC}(s_{i-1}) + \text{FC}(h)) \tag{112}$$

  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[36]

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, sequence length, attention size]**, and refer to it as the "memory". It is also what is returned when calling myAttentionMech.values.

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). It is important to note that they are copies, since the raw encoder outputs and decoder previous states will still be used later on, as we will see.

   Synonyms:
   - scores
   - unnormalized 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.

   Synonyms:
   - softmax outputs
   - attention dist.
   - alignments

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:

$$\tilde{\boldsymbol{h}}_t = \tanh\left(\boldsymbol{W}_c\left[\boldsymbol{c}_t,\ \boldsymbol{h}_t\right]\right) \tag{113}$$

$$p(y_t \mid y_{<t}, x) = \text{softmax}(\boldsymbol{W}_s\tilde{\boldsymbol{h}}_t) \tag{114}$$

[36]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.

TextRank: May 03

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:

$$S(V_i) = (1-d) + d * \sum_{V_j \in \text{In}(V_i)} \frac{1}{|\text{ Out}(V_i)|} S(V_j) \quad \text{where} \quad d \in \Re[0,1] \tag{115}$$

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[37] to another random vertex in the graph. In practice, the algorithm is implemented through the following steps:

(1) Initialize all vertices with arbitrary values.[38]
(2) Iterate over vertices, computing equation 4.14 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:

$$\text{Error}^k(V_i) \approx S^k(V_i) - S^{k-1}(V_i) \tag{116}$$

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

[38] 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 4.14 to incorporate *weighted* connections may be useful for NLP applications.

$$WS(V_i) = (1-d) + d * \sum_{j \in \text{In}(V_i)} \frac{w_{ji}}{\sum V_k \in \text{Out}(V_j) w_{jk}} WS(V_j) \qquad (117)$$

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[39].

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 $T = |V|/3$.[40] 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**[41]. 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.

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

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

[41] 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:

$$\text{Similarity}(S_i, S_j) = \frac{\mid S_i \cap S_j \mid}{\log(\mid S_i \mid) + \log(\mid S_j \mid)} \tag{118}$$

  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).

NLP with Deep Learning

Contents

NLP with Deep Learning **Fall 2016**

Word Vector Representations (Lec 2): May 08

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 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 similarity-based 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:

$$J(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{-m\leq j\leq m}\log \Pr(w_{t+j}|w_t) \tag{119}$$

$$\Pr(o \mid c) = \frac{e^{\boldsymbol{u}_o^T \boldsymbol{v}_c}}{\sum_{w=1}^{\text{vocab size}} e^{\boldsymbol{u}_w^T \boldsymbol{v}_c}} \tag{120}$$

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 find this extremely sloppy.

GloVe (Lec 3): May 08

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:

$$J(\theta) = \sum_{i,j=1}^{W} f(P_{ij})(u_i^T v_j - \log P_{ij})^2 \tag{121}$$

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:

$$d = \arg\max_i \frac{(x_b - x_a + x_c)^T x_i}{||x_b - x_a + x_c||} \tag{122}$$

  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!

APPENDIX A - Questions and Stuff I Always Forget

Questions:

- **Q:** What's the deal with **mixture models**? Why use them? [*Context: Wavenet paper]*
- **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.