Contents

Math and Machine Learning Basics

Contents

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

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

  → 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:

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

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

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

- 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]

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

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

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

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

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

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

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

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

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

- 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

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

- 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.

1.1.1 Example: Principal Component Analysis

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

where m is the number of data points.

Measure Theory.

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

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

- A property that holds **almost everywhere** holds throughout all space except for on a set of measure zero.

Functions of RVs.

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

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

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

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

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

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

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

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

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

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

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

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

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

Gradient-based optimization. Recall from basic calculus that the **directional derivative** of $f(\boldsymbol{x})$ in direction $\hat{\boldsymbol{u}}$ (a unit vector) is defined as the slope of the function f in direction $\hat{\boldsymbol{u}}$. By definition of the derivative, this is given by (with $\boldsymbol{v} := \boldsymbol{x} + \alpha\hat{\boldsymbol{u}}$)

$$\begin{aligned}
\lim_{\alpha\to 0} \frac{f(\boldsymbol{x}+\alpha\hat{\boldsymbol{u}}) - f(\boldsymbol{x})}{\alpha} &= \frac{\partial f(\boldsymbol{x}+\alpha\hat{\boldsymbol{u}})}{\partial\alpha}\Big|_{\alpha=0} && (25)\\
&= \sum_i \frac{\partial f(\boldsymbol{v})}{\partial v_i}\frac{\partial v_i}{\partial\alpha}\Big|_{\alpha=0} && (26)\\
&= \sum_i (\nabla_{\boldsymbol{v}} f(\boldsymbol{v}))_i \; u_i \Big|_{\alpha=0} && (27)\\
&= \hat{\boldsymbol{u}}^T \nabla_{\boldsymbol{v}} f(\boldsymbol{v})\Big|_{\alpha=0} && (28)\\
&= \hat{\boldsymbol{u}}^T \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) && (29)
\end{aligned}$$

where it's important to recognize the distinction between $\lim_{\alpha\to 0}$ and *setting* α to zero, which is denoted by $\big|_{\alpha=0}$. If we want to *find* the direction $\hat{\boldsymbol{u}}$ such that this directional derivative is a minimum, i.e.

$$\begin{aligned}
\hat{\boldsymbol{u}}^* &= \arg\min_{\hat{\boldsymbol{u}}, \hat{\boldsymbol{u}}^T\hat{\boldsymbol{u}}=1} \hat{\boldsymbol{u}}^T \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) && (30)\\
&= \arg\min_{\hat{\boldsymbol{u}}, \hat{\boldsymbol{u}}^T\hat{\boldsymbol{u}}=1} ||\hat{\boldsymbol{u}}||_2 \; ||\nabla_{\boldsymbol{x}} f(\boldsymbol{x})||_2 \; \cos(\boldsymbol{\theta}) && (31)\\
&= \cos(\boldsymbol{\theta}) && (32)
\end{aligned}$$

and we see that $\hat{\boldsymbol{u}}$ points in the opposite direction as the gradient.

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

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

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

The second derivative in a specific direction $\hat{\boldsymbol{d}}$ is given by $\hat{\boldsymbol{d}}^T \boldsymbol{H} \hat{\boldsymbol{d}}$[3]. It tells us how well we can expect a gradient descent step to perform. How so? Well, it shows up in the second-order approximation to the function $f(\boldsymbol{x})$ about our current spot, which we can denote $\boldsymbol{x}^{(0)}$. The standard gradient descent step will move us from $\boldsymbol{x}^{(0)} \to \boldsymbol{x}^{(0)} - \epsilon g$, where g is the gradient evaluated at $\boldsymbol{x}^{(0)}$. Plugging this in to the 2nd order approximation shows us how $\boldsymbol{H}$ can give information related to how "good" of a step that really was. Mathematically,

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

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

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

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

which can be as low as $1/\lambda_{max}$ (the worst case), and as high as $1/\lambda_{min}$ with the λ being the eigenvalues of the Hessian. 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.

---

[3] In the same manner that I derived equation 29, we can derive the second derivative in a specified direction $\hat{\boldsymbol{d}}$:

$$\frac{\partial^2}{\partial \alpha^2} f(\boldsymbol{x} + \alpha \hat{\boldsymbol{d}})\big|_{\alpha=0} = \frac{\partial}{\partial \alpha} \hat{\boldsymbol{d}}^T \nabla_{\boldsymbol{v}} f(\boldsymbol{v})\big|_{\alpha=0} \tag{33}$$

$$= \sum_i d_i \frac{\partial}{\partial \alpha} \frac{\partial f(\boldsymbol{v})}{\partial v_i} \Big|_{\alpha=0} \tag{34}$$

$$= \sum_i d_i \frac{\partial}{\partial v_i} \frac{\partial f(\boldsymbol{v})}{\partial \alpha} \Big|_{\alpha=0} \tag{35}$$

$$= \sum_i \sum_j d_i \frac{\partial^2 f(\boldsymbol{v})}{\partial v_i v_j} d_j \Big|_{\alpha=0} \tag{36}$$

$$= \hat{\boldsymbol{d}}^T \boldsymbol{H} \hat{\boldsymbol{d}} \tag{37}$$

[4] Emphasis on "values" in "eigenvalues" because it's important not to get tripped up here about what the

Constrained optimization: minimizing/maximizing a function $f(\boldsymbol{x})$ constrained to only values of $\boldsymbol{x}$ in some set $\mathbb{S}$. One way of approaching such a problem is to re-design the unconstrained optimization problem such that the re-designed problem's solution satisfies the constraints. For example, to minimize $f(\boldsymbol{x})$ for $\boldsymbol{x} \in \mathbb{R}^2$ with constraint $||\boldsymbol{x}||_2 = 1$, we can minimize $g(\theta) = f([\cos\theta, \sin\theta]^T)$ wrt θ, then return $[\cos\theta, \sin\theta]^T$ as the solution to the original problem.

The **Karush-Kuhn-Tucker** (KKT) approach, a generalization of **Lagrange multipliers**, provides a general approach for re-designing the optimization problem, with procedure as follows:

1. Find m functions $g^{(i)}$ and n functions $h^{(j)}$ such that your set of allowed values $\mathbb{S}$ can be written

$$\mathbb{S} = \{\boldsymbol{x} \mid \forall i, g^{(i)}(\boldsymbol{x}) = 0 \text{ and } \forall j, h^{(j)}(\boldsymbol{x}) \leq 0\} \tag{38}$$

   The equations involving $g^{(i)}$ are called the **equality constraints** and the inequalities involving $h^{(j)}$ are called the **inequality constraints**.
2. Introduce new variables λ_i (for the equality constraints) and α_j (for the inequality constraints). These are called the KKT multipliers. The generalized Lagrangian is then defined as

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = f(\boldsymbol{x}) + \sum_i \lambda_i g^{(i)}(\boldsymbol{x}) + \sum_j \alpha_j h^{(j)}(\boldsymbol{x}) \tag{39}$$

3. Solve the re-designed unconstrained optimization problem:

$$\min_{\boldsymbol{x}} \max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) \tag{40}$$

   which has the same optimal objective function value and set of optimal points $\boldsymbol{x}$ as the original constrained problem, $\min_{\boldsymbol{x} \in \mathbb{S}} f(\boldsymbol{x})$. Any time the constraints are satisfied, the expression $\max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha})$ evaluates to $f(\boldsymbol{x})$, and any time a constraint is violated, the same expression evaluates to ∞.

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. 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.

1.4.1 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].

[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.

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**.

$$\text{bias}\left[\hat{\theta}_m\right] = \mathbb{E}\left[\hat{\theta}_m\right] - \theta \tag{41}$$

$$\text{Var}\left[\hat{\theta}_m\right] = \mathbb{E}\left[\left(\hat{\theta}_m - \mathbb{E}\left[\hat{\theta}_m\right]\right)^2\right] \tag{42}$$

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

$$\text{MSE} = \mathbb{E}\left[(\hat{\theta}_m - \theta)^2\right] \tag{5.53}$$

$$= \text{Bias}(\hat{\theta})^2 + \text{Var}\left[\hat{\theta}_m\right] \tag{5.54}$$

and desirable estimators are those with low MSE.

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 $m \to \infty$.

[6] Derivation:

$$\text{MSE} = \mathbb{E}\left[\hat{\theta}^2 + \theta^2 - 2\theta\hat{\theta}\right] \tag{43}$$

$$= \mathbb{E}\left[\hat{\theta}^2\right] + \theta^2 - 2\theta\mathbb{E}\left[\hat{\theta}\right] \tag{44}$$

$$= (\mathbb{E}\left[\hat{\theta}\right]^2 - \mathbb{E}\left[\hat{\theta}\right]^2) + \mathbb{E}\left[\hat{\theta}^2\right] + \theta^2 - 2\theta\mathbb{E}\left[\hat{\theta}\right] \tag{45}$$

$$= \left(\mathbb{E}\left[\hat{\theta}\right]^2 + \theta^2 - 2\theta\mathbb{E}\left[\hat{\theta}\right]\right) + \left(\mathbb{E}\left[\hat{\theta}^2\right] - \mathbb{E}\left[\hat{\theta}\right]^2\right) \tag{46}$$

$$= \text{Bias}(\hat{\theta})^2 + \text{Var}\left[\hat{\theta}_m\right] \tag{47}$$

1.4.2 Maximum Likelihood Estimation (5.5)

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

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

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

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

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

$$D_{KL}(\hat{p}_{data} || p_{model}) = \mathbb{E}_{\boldsymbol{x} \sim \hat{p}_{data}} \left[\log \left(\hat{p}_{data}(\boldsymbol{x})\right) - \log \left(p_{model}(\boldsymbol{x})\right)\right] \tag{48}$$

$$= \left(\frac{1}{N}\left(\sum_{i=1}^{N} \log\left(\text{Counts}(\boldsymbol{x}_i)\right)\right) - \log N\right) - \mathbb{E}_{\boldsymbol{x} \sim \hat{p}_{data}} \left[\log \left(p_{model}(\boldsymbol{x})\right)\right] \tag{49}$$

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

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

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

where $\boldsymbol{x}^{(i)}$ are fed as *inputs* to the model; this is why we can formulate MLE as a conditional probability.

[7] The KL divergence is given by

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

1.4.3 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 the *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.

So what happens to $\boldsymbol{\theta}$ after we observe the data? We update it using Bayes' rule[8]:

$$p(\boldsymbol{\theta} \mid x^{(1)}, \ldots, x^{(m)}) = \frac{p(x^{(1)}, \ldots, x^{(m)} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta})}{p(x^{(1)}, \ldots, x^{(m)})} \tag{50}$$

Note that we still haven't mentioned how to actually make *predictions*. Since we no longer have just one value for $\boldsymbol{\theta}$, but rather we have a posterior *distribution* $p(\boldsymbol{\theta} \mid x^{(1)}, \ldots, x^{(m)})$, we must integrate over this to get the predicted likelihood of the next sample $x^{(m+1)}$:

$$p(x^{(m+1)} \mid x^{(1)}, \ldots, x^{(m)}) = \int p(x^{(m+1)} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta} \mid x^{(1)}, \ldots, x^{(m)}) \mathrm{d}\boldsymbol{\theta} \tag{51}$$

$$= \mathbb{E}_{\boldsymbol{\theta} \sim p(\boldsymbol{\theta} \mid x^{(1)}, \ldots, x^{(m)})} \left[p(x^{(m+1)} \mid \boldsymbol{\theta}) \right] \tag{52}$$

Linear Regression: MLE vs. Bayesian. Both want to model the conditional distribution $p(y \mid \boldsymbol{x})$ (the conditional likelihood). To derive the standard linear regression algorithm, we *define*

$$p(y \mid \boldsymbol{x}) = \mathcal{N}\left(y;\ \hat{y}(\boldsymbol{x}; \boldsymbol{w}),\ \sigma^2\right) \tag{53}$$

Assume σ^2 is some constant chosen by user.

$$\hat{y}(\boldsymbol{x}; \boldsymbol{w}) = \boldsymbol{w}^T \boldsymbol{x} \tag{54}$$

[8] In practice, we typically compute the denominator by simply normalizing the probability distribution, i.e. it is effectively the partition function.

- **Maximum Likelihood Approach**: We can use the definition above (and the i.i.d. assumption) to evaluate the conditional log-likelihood as

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

  where only the last term has any dependence on $\boldsymbol{w}$. Therefore, to obtain $\boldsymbol{w}_{ML}$ we take the derivative of the last term w.r.t. $\boldsymbol{w}$, set that to zero, and solve for $\boldsymbol{w}$. We see that finding the $\boldsymbol{w}$ that maximizes the conditional log-likelihood is equivalent to finding the $\boldsymbol{w}$ that minimizes the training MSE.

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

- **Bayesian Approach**: Our conditional likelihood is already given in equation 53. Next, we must define a prior distribution over $\boldsymbol{w}$. As is common, we choose a Gaussian prior to express our high degree of uncertainty about $\boldsymbol{\theta}$ (implying we'll choose a relatively large variance):

$$p(\boldsymbol{w}) := \mathcal{N}(\boldsymbol{w}; \boldsymbol{\mu}_0, \boldsymbol{\Lambda}_0) \tag{55}$$

  Typically assume $\boldsymbol{\Lambda}_0 = \text{diag}\,(\boldsymbol{\lambda}_0)$

  We can then compute [the unnormalized] $p(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y}) \propto p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{w})p(\boldsymbol{w})$ [and then normalize it].

Maximum A Posteriori (MAP) Estimation. Often we either prefer a point estimate for θ, or we find out that computing the posterior distribution is intractable and a point estimate offers a tractable estimation. The obvious way of obtaining this while still taking the Bayesian route is to just argmax the posterior and use that as your point estimate:

$$\boldsymbol{\theta}_{MAP} = \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{\theta} \mid \boldsymbol{x}) = \arg\max_{\boldsymbol{\theta}} \log p(\boldsymbol{x} \mid \boldsymbol{\theta}) + \log p(\boldsymbol{\theta}) \tag{56}$$

where the second form shows how this is basically maximum likelihood with incorporation of the prior. We don't want just any $\boldsymbol{\theta}$ that maximizes the likelihood of our data if there is virtually no chance of that value of $\boldsymbol{\theta}$ in the first place.

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:

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

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

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

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

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

Deep Networks: Modern Practices

Contents

Deep Feedforward Networks (Ch. 6)

Written by Brandon McKinzie

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

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

with ϕ defining a hidden layer.

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

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)

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

$$h_i = g(\boldsymbol{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i) \tag{57}$$

→ 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.

2.1.1 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],

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

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

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 (Ch. 7)

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."

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. 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 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 $\boldsymbol{x}$.

$$\boldsymbol{x}' \longleftarrow \boldsymbol{x} + \epsilon \cdot \text{sign}\left(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y})\right) \tag{58}$$

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.

Empirical Risk Minimization. The ultimate goal of any machine learning algorithm is to reduce the expected generalization error, also called the **risk**:

$$J^*(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim p_{data}}\left[L(f(\boldsymbol{x};\boldsymbol{\theta}), y)\right] \tag{59}$$

with emphasis that the risk is over the *true* underlying data distribution p_{data}. If we knew p_{data}, this would be an optimization problem. Since we don't, and only have a set of training samples, it is a machine learning problem. However, we can still just minimize the **empirical risk**, replacing p_{data} in the equation above with $\hat{p}_{data}$[10].

So, how is minimizing the empirical risk any different than familiar gradient descent approaches? Aren't they designed to do just that? Well, sort of, but it's technically not the same. When we say "minimize the empirical risk" in the context of optimization, we mean this very literally. Gradient descent methods emphatically do *not* just go and *set* the weights to values such that the empirical risk reaches its lowest possible value – that's not machine learning. Furthermore, many useful loss function such as 0-1 loss[11] do not have useful derivatives.

ERM ≠ GD

Surrogate Loss Functions and Early Stopping. In cases such as 0-1 loss, where minimization is intractable, one typically optimizes a **surrogate loss function** instead, such as the negative log-likelihood of the correct class. Also, an important difference between pure optimization and our training algorithms is that the latter usually don't halt at a local minimum. Instead, we usually must define some early stopping condition to terminate training before overfitting begins to occur.

> *We want to minimize the risk, but we don't have access to p_{data}, so ...*
> *We want to minimize the empirical risk, but it's prone to overfitting and our loss function's derivative may be zero/undefined, so ...*
> *We minimize a surrogate loss function iteratively over minibatches until early stopping is triggered.*

[10] This amount to a simple average over the loss function at each training point.
[11] The 0-1 loss function is defined as

$$L(\hat{y}, y) = I(\hat{y} \neq y) \tag{60}$$

Batch and Minibatch Algorithms. Computing $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ as an expectation over the entire training set is expensive, so we typically compute the expectation over a small subset of the examples. Recall that the standard deviation, or standard error $SE(\mu_m)$, of the mean taken over some subset of $m \leq n$ samples, $\mu_m = \frac{1}{m} \sum_{i \sim Rand(0,\ n,\ size=m)} x^{(i)}$, is given by $\sigma/\sqrt{m}$, where σ is the true [sample] standard deviation of the full n data samples. In other words, to improve such a gradient by a factor of 10 requires 100 times more samples-per-batch (and thus 100 times more computation). For this reason, most optimization algorithms actually *converge* much faster if they can rapidly compute approximate estimates of the gradient (re: smaller batches) rather than slowly computing the exact gradient.

The key points to consider when choosing your batch size:

1. Larger batches = more accurate estimates of the gradient, but with less than linear returns.
2. If examples in the batch are processed in parallel (as is typical), then memory roughly scales with batch size.
3. Small batches can offer a regularizing effect. Generalization error is often best for a batch size of 1. However, this requires a low learning rate to maintain stability and thus a longer overall training runtime.

Also, note that online SGD, where we never reuse data points, but simply update parameters as new data comes in, gives an unbiased estimator of the exact gradient of the generalization error (the risk). Once data samples are reused (e.g. when training with multiple epochs), the gradient estimates become biased. The interesting point here is that the availability of increasingly massive datasets is making single-epoch[12] training more common. In such cases, *overfitting is no longer an issue*, but rather underfitting and computational efficiency.

Ill-conditioning of the Hessian matrix $\boldsymbol{H}$ can cause SGD to get "stuck" in the sense that even very small steps increase the cost function. Recall that a second-order Taylor series expansion of the cost function predicts that an SGD step of $-\epsilon \boldsymbol{g}$ will add

$$\frac{1}{2}\epsilon^2 \boldsymbol{g}^T \boldsymbol{H} \boldsymbol{g} - \epsilon \boldsymbol{g}^T \boldsymbol{g} \tag{61}$$

to the cost. If $\boldsymbol{H}$ has a large condition number (re: if $\boldsymbol{H}$ is ill-conditioned), then the range of possible values, $[1/\lambda_{max}, 1/\lambda_{min}]$, for $\boldsymbol{g}^T \boldsymbol{H} \boldsymbol{g}$ can become very large. In particular, if $\boldsymbol{g}^T \boldsymbol{H} \boldsymbol{g}$ exceeds $\epsilon \boldsymbol{g}^T \boldsymbol{g}$, then the SGD step will *increase* the cost!

[12] Or even less, i.e. not using all of the training data.

Training algorithms. Below, I list some popular training algorithms and their update equations.

- **SGD**.

$$\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_{i}^{m} L\left(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}\right) \tag{62}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \boldsymbol{g} \tag{63}$$

- **Momentum**.

$$\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_{i}^{m} L\left(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}\right) \tag{64}$$

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g} \tag{65}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v} \tag{66}$$

- **Nesterov Momentum**. Gradient computations instead evaluated after current velocity is applied.

$$\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_{i}^{m} L\left(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta} + \alpha \boldsymbol{v}), \boldsymbol{y}^{(i)}\right) \tag{67}$$

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g} \tag{68}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v} \tag{69}$$

- **AdaGrad**. Different learning rate for each model parameter. Individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all historical squared values of the gradient. Empirically, can result in premature and excessive decrease in the effective learning rate.

$$\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_{i}^{m} L\left(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}\right) \tag{70}$$

$$\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g} \tag{71}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g} \tag{72}$$

  where the gradient accumulation variable $\boldsymbol{r}$ is initialized to the zero vector, and the fraction and square root in the last equation is applied element-wise.

- **RMSProp**. Modifies AdaGrad by changing the gradient accumulation into an exponentially weighted moving average.

$$\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_{i}^{m} L\left(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}\right) \tag{73}$$

$$\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho)\boldsymbol{g} \odot \boldsymbol{g} \tag{74}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g} \tag{75}$$

  It is also common to modify RMSProp to use Nesterov momentum.

- **Adam**. So-named to mean "adaptive moments." We now call $\boldsymbol{r}$ the 2nd moment (variance) variable, and introduce $\boldsymbol{s}$ as the 1st moment (mean) variable, where the moments are for the [true] gradient; the new variables act as estimates of the moments [since we estimate the gradient with a simple average over a minibatch]. Note that these moments are uncentered.

$$\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i}^{m} L\left(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}\right) \tag{76}$$

$$\boldsymbol{s} \leftarrow \frac{1}{1 - \rho_1^{t-1}} \left[\rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}\right] \tag{77}$$

$$\boldsymbol{r} \leftarrow \frac{1}{1 - \rho_2^{t-1}} \left[\rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}\right] \tag{78}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{s} \tag{79}$$

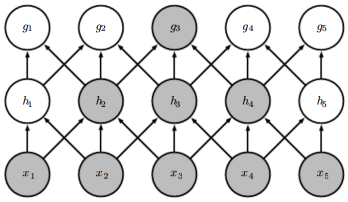
  where the factors proportional to the ρ values serve to correct for bias in the moment estimators.

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[13], 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.

[13] 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.

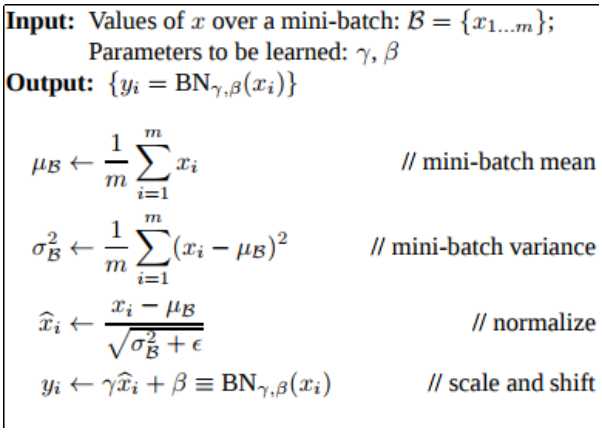
Pooling. Helps make the representation approximately **invariant** to small translations of the input. The use of pooling can be viewed as adding an infinitely strong prior[14] that the function the layer learns must be invariant to small translations.

Additional common tricks[15].

- **Local Response Normalization (LRN)**[16]. Purpose is to aid generalization ability. Let $a^i_{x,y}$ denote the activity of a neuron computed by applying kernel i at position (x, y) and then applying the ReLU nonlinearity. The response-normalized activity $b^i_{x,y}$ is given by the expression

$$b^i_{x,y} = \frac{a^i_{x,y}}{\left(k + \alpha \sum_j \left(a^j_{x,y}\right)^2\right)^\beta} \tag{80}$$

  where j runs from $[i - n/2]_+$ to $\min(N-1,\ i+n/2)$, and N is the total number of kernels in the given layer[17]. Authors used $k = 2$, $n = 5$, $\alpha = 10^{-4}$, $\beta = 0.75$.
- **Batch Normalization**[18]. BN Allows us to use much higher learning rates and be less careful about initialization. Algorithm defined in image below, where each element of the batch, $x_i \equiv x_i^{(k)}$ (where we drop the k for notational simplicity), represents the kth activation output from the previous layer [for the ith sample in the batch] and about to be fed as input to the current layer.



  Note that one can model each layer's activations as arising from some distribution. When we feed data to a network, we model the data as coming from some data-generating

[14] Where the distribution of this prior is over all possible functions learnable by the model.

[15] Collected on my own. In other words, not from the deep learning book, but rather a bunch of disconnected resources over time.

[16] From section 3.3 of Krizhevsky et al. (2012). AlexNet paper.

[17] In other words, the summation is over the adjacent kernel maps, with [total] window size n (manually chosen). The min/max just says $n/2$ to the left (right) unless that would be past the leftmost (rightmost) kernel map.

[18] From "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift" by Ioffe et al.

distribution. Similarly, we can model the activations that occur when feeding the data through as coming from some activation-layer-generating distribution. The problem with this model is that the process of updating our weights during training changes the distribution of activations for each layer, which can make the learning task more difficult. Batch normalization's goal is to reduce this *internal covariate shift*, and is motivated by the practice of normalizing our data to have zero mean and unit variance.

Intuition of some math.

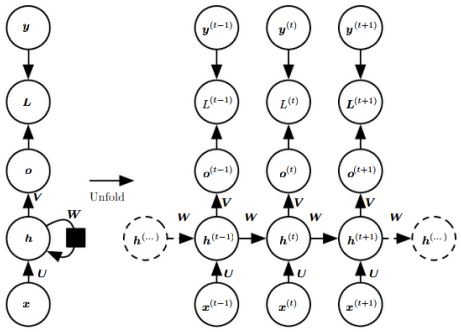
- **Q**: How to intuitively understand the commutativity of convolution?
  - **A**: You must first realize that, *independent of which formula we're thinking of*, as one index into either the image or kernel increases (decreases), the other decreases (increases); they increment in opposite directions. The difference between the two formulas (9.4 and 9.5 in textbook), is just that we start at different ends of the summations (when you actually substitute in the numbers). Note that this doesn't require any symmetry on the boundaries of the summation about zero; it truly is a property of the convolution.
- **Q**: What do the authors mean by "we have flipped the kernel"?
  - **A**: Not much, and it's poor wording. They didn't *do* anything, that is just part of the definition of the convolution. They literally just mean that the convolution has the property that as one index increases, the other decreases (see previous answer). The cross-correlation, however, has the property that as one index increases, the other increases, too.

2.5.1 Review: The Basics of RNNs

Notation/Architecture Used.

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

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



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

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

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

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

Note that this is an example of an RNN that maps input seqs to output seqs of the same length[19]. We can then compute, e.g., the log-likelihood loss $L = \sum_t L^{(t)}$ over all time steps as:

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

Convince yourself this is identical to cross-entropy.

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

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

Back-Propagation Through Time.

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

$$
\begin{aligned}
\left(\nabla_{\boldsymbol{o}^{(t)}} L\right)_i &= \frac{\partial L}{\partial \boldsymbol{o}_i^{(t)}} \\
&= \frac{\partial L}{\partial L^{(t)}} \cdot \frac{\partial L^{(t)}}{\partial \boldsymbol{o}_i^{(t)}} \\
&= (1) \cdot \frac{\partial L^{(t)}}{\partial \boldsymbol{o}_i^{(t)}} \\
&= \frac{\partial}{\partial \boldsymbol{o}_i^{(t)}} \left\{ -\log\left(\hat{\boldsymbol{y}}_{y^{(t)}}^{(t)}\right) \right\} \\
&= -\frac{\partial}{\partial \boldsymbol{o}_i^{(t)}} \left\{ \log\left(\frac{e^{\boldsymbol{o}_{y^{(t)}}^{(t)}}}{\sum_j e^{\boldsymbol{o}_j^{(t)}}} \right) \right\} \\
&= -\frac{\partial}{\partial \boldsymbol{o}_i^{(t)}} \left\{ \boldsymbol{o}_{y^{(t)}}^{(t)} - \log\left(\sum_j e^{\boldsymbol{o}_j^{(t)}} \right) \right\} \\
&= -\left\{ \mathbf{1}_{i,y^{(t)}} - \frac{\partial}{\partial \boldsymbol{o}_i^{(t)}} \log\left(\sum_j e^{\boldsymbol{o}_j^{(t)}} \right) \right\}
\end{aligned}
\tag{81}
$$

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

   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[20] $\nabla_a L = W^T \nabla_b L$. Using this result, we can then iterate and take gradients back in time from $t = \tau - 1$ to $t = 1$ as follows:

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

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

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

[20] More generally,

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

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

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

previously done with vectors after a quick abstraction: For any tensor $\mathbf{X}$ of arbitrary rank (e.g. if rank-4 then index like $\mathbf{X}_{ijk\ell}$), use single variable (e.g. i) to represent the complete tuple of indices[21].

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

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

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

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

$$\nabla_{\mathbf{V}} L = \sum_t^{\tau} \nabla_{\mathbf{V}} L^{(t)} \tag{82a}$$

$$= \sum_t^{\tau} \nabla_{\mathbf{V}} L^{(t)}(\boldsymbol{o}_1^{(t)}, \ldots, \boldsymbol{o}_{n_{out}}^{(t)}) \tag{82b}$$

$$= \sum_t^{\tau} \sum_i^{n_{out}} \left(\nabla_{\boldsymbol{o}^{(t)}} L\right)_i \nabla_{\mathbf{V}} \boldsymbol{o}_i^{(t)} \tag{82c}$$

$$= \sum_t^{\tau} \sum_i^{n_{out}} \left(\nabla_{\boldsymbol{o}^{(t)}} L\right)_i \nabla_{\mathbf{V}} \left\{ c_i + \sum_{j=1}^{n_{hid}} V_{ij} \boldsymbol{h}_j^{(t)} \right\} \tag{82d}$$

$$= \sum_t^{\tau} \sum_i^{n_{out}} \left(\nabla_{\boldsymbol{o}^{(t)}} L\right)_i \begin{bmatrix} 0 & 0 & \ldots & 0 \\ \vdots & \vdots & \ldots & \vdots \\ \boldsymbol{h}_1^{(t)} & \boldsymbol{h}_2^{(t)} & \ldots & \boldsymbol{h}_{n_{hid}}^{(t)} \\ \vdots & \vdots & \ldots & \vdots \\ 0 & 0 & \ldots & 0 \end{bmatrix} \tag{82e}$$

$$= \sum_t^{\tau} \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) (\boldsymbol{h}^{(t)})^T \tag{82f}$$

[21] 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

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

where if 82e confuses you, see the footnote[22].

- **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 $\boldsymbol{V}$[23]. The authors here define/choose, when evaluating $\nabla_{\mathbf{W}} h_i^{(t)}$ to only concern themselves with $\boldsymbol{W} := \boldsymbol{W}^{(t)}$, i.e. the direct connections to $\boldsymbol{h}$ at time t.

$$\nabla_{\mathbf{W}} L = \sum_t^\tau \nabla_{\mathbf{W}} L^{(t)} \tag{84a}$$

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

$$= \sum_t^\tau \sum_i^{n_{hid}} \left(\nabla_{\boldsymbol{h}^{(t)}} L\right)_i \left(\operatorname{diag}\left(1 - (\boldsymbol{h}^{(t)})^2\right) \begin{bmatrix} 0 & 0 & \dots & 0 \\ \vdots & \vdots & \dots & \vdots \\ \boldsymbol{h}_1^{(t-1)} & \boldsymbol{h}_2^{(t-1)} & \dots & \boldsymbol{h}_{n_{hid}}^{(t-1)} \\ \vdots & \vdots & \dots & \vdots \\ 0 & 0 & \dots & 0 \end{bmatrix} \right) \tag{84b}$$

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

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

$$\nabla_{\mathbf{U}} L = \sum_t^\tau \nabla_{\mathbf{U}} L^{(t)} \tag{85a}$$

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

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

---

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

$$\sum_i \nabla_{\mathbf{w}} \left[(\boldsymbol{W}x)_i\right] = \begin{bmatrix} \boldsymbol{x}^T \\ \boldsymbol{x}^T \\ \vdots \\ \boldsymbol{x}^T \end{bmatrix} \tag{83}$$

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

[23] 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) $\boldsymbol{W}$.

2.5.2 RNNs as Directed Graphical Models

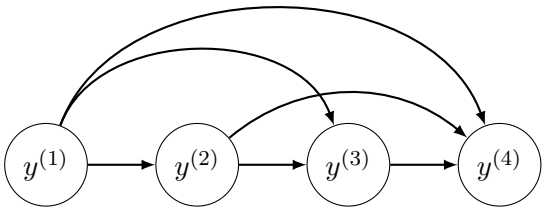
The advantage of RNNs is their efficient parameterization of the joint distribution over $\boldsymbol{y}^{(i)}$ via parameter sharing. This introduces a built-in assumption that we can model the effect of $y^{(i)}$ in the distant past on the current $y^{(t)}$ *via its effect on* $\boldsymbol{h}$. We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

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

$$P(\mathbb{Y}) \triangleq P(y^{(1)}, \ldots, y^{(\tau)}) = \prod_{t=1}^{\tau} P(y^{(t)} \mid y^{(t-1)}, \ldots, y^{(1)}) \tag{86}$$

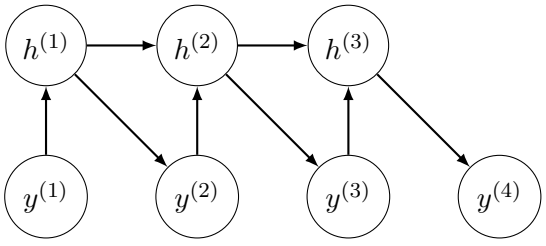
where I've drawn an example of the *complete graph* for $\tau = 4$ below.

The complete graph c represent the direct dependencies betweer any pairs of y values.



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

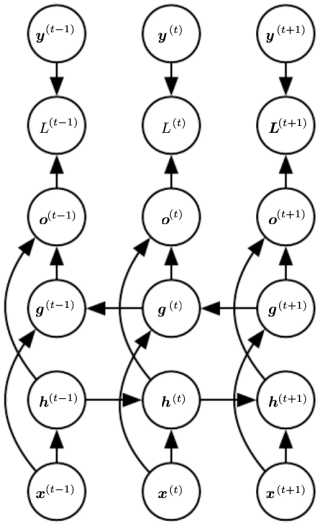
We could include latent variables h at each timestep that capture the dependencies, reminiscent of a classic RNN:



Since in the RNN case all factors $P(h^{(t)} \mid h^{(t-1)})$ are deterministic, we don't need any additional parameters to compute this probability[24], other than the single m^2 parameters needed to convert any $h^{(t)}$ to the next $h^{(t+1)}$ (which is shared across all transitions). Now, the number of parameters needed as a function of sequence length is constant, and as a function of k is just $\mathcal{O}(k)$.

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

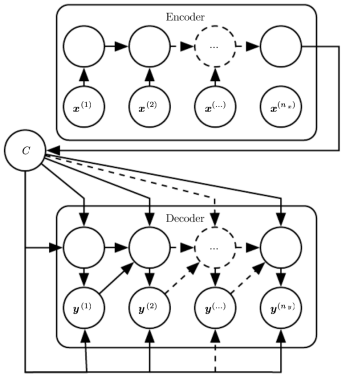
Bidirectional RNNs. In many applications, it is desirable to output a prediction of $\boldsymbol{y}^{(t)}$ that may depend on *the whole sequence*. For example, in speech recognition, the interpretation of words/sentences can also depend on what is *about* to be said. Below is a typical bidirectional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



Notice how the outp
units $\boldsymbol{o}^{(t)}$ have the r
property of dependir
both the past and fu
while being most
sensitive to input va
around time t.

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

Encoder-Decoder Seq2Seq Architectures (10.4) Here we discuss how an RNN can be trained to map an input sequence to output sequence which is not necessarily the same length. (Not really much of a discussion...figure below says everything.)



2.5.3 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.

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

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

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

> **Any component of $\boldsymbol{h}^{(0)}$ that isn't aligned with the largest eigenvector will eventually be discarded.**[25]

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

[25] 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.

be $\mathcal{O}(v^n)$[26]. *To obtain some desired variance v^* we may choose the individual weights with variance $v = \sqrt[n]{v^*}$.*

2.5.4 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[27]

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

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

The final hidden state can then be computed via

$$h_i^{(t)} = \tanh(s_i^{(t)}) \odot q_i^{(t)} \tag{89}$$

[26]Quick sketch of (my) proof:

$$\text{Var}\left[w^{(i)}\right] = v = \mathbb{E}\left[(w^{(i)})^2\right] - \cancel{\mathbb{E}\left[w^{(i)}\right]^2} \tag{87}$$

$$\text{Var}\left[\prod_t^n w^{(t)}\right] = \mathbb{E}\left[\left(\prod_t^n w^{(t)}\right)^2\right] = \prod_t^n \mathbb{E}\left[(w^{(t)})^2\right] = v^n \tag{88}$$

[27]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.

2.6.1 Natural Language Processing (12.4)

Begins on pg. 448

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

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

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

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

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

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

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

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

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

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

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

- **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[28], even for modest n.

2.6.2 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[29] 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*.

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

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

Deep Learning Research

Contents

Linear Factor Models (Ch. 13)

Written by Brandon McKinzie

Overview. Much research is in building a *probabilistic model*[30] 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

$$p_{model}(x) = \sum_h \Pr(h)\Pr(x \mid h) = \mathbb{E}_h\left[p_{model}(x \mid h)\right] \tag{90}$$

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

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

Note that $\boldsymbol{h}$ is a *vector* of arbitrary size, where we assume $p(\boldsymbol{h})$ is a **factorial distribution**: $p(\boldsymbol{h}) = \prod_i p(h_i)$. This roughly means we assume the elements of $\boldsymbol{h}$ are mutually independent[31]. 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:

$$\boldsymbol{x} = \boldsymbol{W}\boldsymbol{h} + \boldsymbol{b} + \text{noise} \tag{91}$$

Probabilistic PCA and Factor Analysis.

- **Factor analysis**:

$$\boldsymbol{h} \sim \mathcal{N}(\boldsymbol{h}; \boldsymbol{0}, \boldsymbol{I}) \tag{92}$$

$$\text{noise} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{\psi} \equiv \text{diag}(\boldsymbol{\sigma^2})) \tag{93}$$

$$\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{x};\ \boldsymbol{b}, \boldsymbol{W}\boldsymbol{W}^T + \boldsymbol{\psi}) \tag{94}$$

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

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

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

It is worth emphasizing the interpretation of $\boldsymbol{\psi}$ as the matrix of **conditional variances** σ_i^2. *Huh?* Let's take a step back. The fact that we were able to separate the distributions in the above relations for $\boldsymbol{h}$ and noise is from a built-in assumption that $\Pr(x_i|\boldsymbol{h}, x_{j\neq i}) = \Pr(x_i|\boldsymbol{h})$[32].

The Big Idea

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

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

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

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

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

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

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

$$\begin{aligned}
\boldsymbol{\mu}_x &= \mathbb{E}_{\boldsymbol{h}}\left[\boldsymbol{W}\boldsymbol{h} + \boldsymbol{b}\right] && (95)\\
&= \int p(\boldsymbol{h})(\boldsymbol{W}\boldsymbol{h} + \boldsymbol{b})\mathrm{d}\boldsymbol{h} && (96)\\
&= \boldsymbol{b} + \int \frac{1}{(2\pi)^{n/2}} e^{-\frac{1}{2}\sum_i^n h_i^2}\boldsymbol{W}\boldsymbol{h}\mathrm{d}\boldsymbol{h} && (97)\\
&= \boldsymbol{b} && (98)\\
\text{Cov}(\boldsymbol{x}) &= \mathbb{E}\left[(\boldsymbol{x} - \mathbb{E}\left[\boldsymbol{x}\right])(\boldsymbol{x} - \mathbb{E}\left[\boldsymbol{x}\right])^T\right] && (99)\\
&= \mathbb{E}\left[(\boldsymbol{W}\boldsymbol{h} + \text{noise})(\boldsymbol{h}^T\boldsymbol{W}^T + \text{noise}^T)\right] && (100)\\
&= \mathbb{E}\left[(\boldsymbol{W}\boldsymbol{h}\boldsymbol{h}^T\boldsymbol{W}^T)\right] + \boldsymbol{\psi} && (101)\\
&= \boldsymbol{W}\boldsymbol{W}^T + \boldsymbol{\psi} && (102)
\end{aligned}$$

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

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

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

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

Autoencoders (Ch. 14)

Written by Brandon McKinzie

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

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.

$$f(\cdot) \leftarrow I(\cdot) \tag{103}$$
$$\tilde{\boldsymbol{X}} = \boldsymbol{X} \tag{104}$$

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

$$f^{(k)} = \mathcal{L}(\tilde{\boldsymbol{X}}) \tag{105}$$
$$f(\cdot) \leftarrow f^{(k)}\left(f\left(\cdot\right)\right) \tag{106}$$
$$\tilde{\boldsymbol{X}} \leftarrow f^{(k)}(\tilde{\boldsymbol{X}}) \tag{107}$$

   In English: just apply the regular learning/training process for each layer/stage **sequentially and individually**[33].

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).

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

Motivation. In addition to classification, we can ask probabilistic models to perform other tasks such as density estimation ($\boldsymbol{x} \rightarrow p(\boldsymbol{x})$), denoising, missing value imputation, or sampling. What these [other] tasks have in common is they require a *complete understanding of the input*. Let's start with the most naive approach of modeling $p(\boldsymbol{x})$, where $\boldsymbol{x}$ contains n elements, each of which can take on k distinct values: we store a lookup table of all possible $\boldsymbol{x}$ and the corresponding probability value $p(\boldsymbol{x})$. This requires k^n parameters[34]. Instead, we use graphs to describe model structure (direct/indirect interactions) to drastically reduce the number of parameters.

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

$$p(\boldsymbol{x}) = \prod_i p(x_i \mid Pa_{\mathcal{G}}(x_i)) \tag{108}$$

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

$$\widetilde{p}(\boldsymbol{x}) = \prod_{\mathcal{C} \in \mathcal{G}} \phi(\mathcal{C}) \tag{109}$$

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

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

$$p(\boldsymbol{x}) = \frac{1}{Z}\widetilde{p}(\boldsymbol{x}) \tag{110}$$

$$Z = \int \widetilde{p}(\boldsymbol{x})\mathrm{d}\boldsymbol{x} \tag{111}$$

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

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

$$\widetilde{p}(\boldsymbol{x}) \triangleq \exp\left(-E(\boldsymbol{x})\right) \tag{112}$$

and $E(\boldsymbol{x})$ is known as the **energy function**[35]. Many algorithms need to compute not $p_{model}(\boldsymbol{x})$ but only $\log \widetilde{p}_{model}(\boldsymbol{x})$ (unnormalized log probabilities - logits!). For EBMs with latent variables $\boldsymbol{h}$, such algorithms are phrased in terms of the **free energy**:

$$\mathcal{F}(\boldsymbol{x} = x) = -\log \sum_h \exp\left(-E(\boldsymbol{x} = x,\ \boldsymbol{h} = h)\right) \tag{113}$$

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

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

- **Separation**. For *undirected* models. If variables a and b are connected by a path involving only unobserved variables (an **active** path), then a and b are *not* separated. Otherwise, they are separated. Any paths containing at least one observed variable are called **inactive**.
- **D-Separation**[36]. For *directed* models. Although there are rules that help determine whether a path between a and b is d-separated, it is simplest to just determine whether a is independent from b given any observed variables along the path.

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

[36] The D stands for dependence.

3.4.1 Sampling from Graphical Models

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

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

3.4.2 Inference and Approximate Inference

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

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

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\boldsymbol{b}^T\boldsymbol{v} - \boldsymbol{c}^T\boldsymbol{h} - \boldsymbol{v}^T\boldsymbol{W}\boldsymbol{h} \tag{114}$$

where $\boldsymbol{b}$, $\boldsymbol{c}$, and $\boldsymbol{W}$ are unconstrained, real-valued, learnable parameters. One could interpret the values of the bias parameters as the affinities for the associated variable being its given value, and the value $\boldsymbol{W}_{i,j}$ as the affinity of v_i being its value and h_j being its value at the same time[37].

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

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

that:

$$p(\boldsymbol{h} \mid \boldsymbol{v}) = \prod_i p(h_i \mid \boldsymbol{v}) \tag{115}$$

$$p(\boldsymbol{v} \mid \boldsymbol{h}) = \prod_i p(v_i \mid \boldsymbol{h}) \tag{116}$$

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

$$p(h_i = 1 \mid \boldsymbol{v}) = \sigma\left(c_i + \boldsymbol{v}^T \boldsymbol{W}_{:,i}\right) \tag{117}$$

allowing for efficient block Gibbs sampling.

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

$$s = \int p(\boldsymbol{x}) f(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} = \mathbb{E}_p \left[f(\mathbf{x}) \right] \quad \rightarrow \quad \hat{s}_n = \frac{1}{n} \sum_{i=1,\ \mathbf{x}^{(i)} \sim p}^{n} f(\boldsymbol{x}^{(i)}) \tag{118}$$

As we've seen before, the empirical average is an unbiased[38] 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:

$$p(\boldsymbol{x}) f(\boldsymbol{x}) = q(\boldsymbol{x}) \frac{p(\boldsymbol{x}) f(\boldsymbol{x})}{q(\boldsymbol{x})} \tag{122}$$

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

$$\hat{s}_q = \frac{1}{n} \sum_{i=1,\ \mathbf{x}^{(i)} \sim q}^{n} \frac{p(\boldsymbol{x}^{(i)}) f(\boldsymbol{x}^{(i)})}{q(\boldsymbol{x}^{(i)})} \tag{123}$$

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

$$\mathbb{E}_p \left[\hat{s}_n \right] = \frac{1}{n} \sum_{i=1}^{n} \mathbb{E}_p \left[f(\boldsymbol{x}^{(i)}) \right] \tag{119}$$

$$= \frac{1}{n} \sum_{i=1}^{n} s \tag{120}$$

$$= s \tag{121}$$

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

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

$$\mathrm{Var}\left[\hat{s}_q\right] = \mathrm{Var}\left[\frac{p(\boldsymbol{x})f(\boldsymbol{x})}{q(\boldsymbol{x})}\right]/n \tag{124}$$

with the optimal (minimum) value of q at:

$$q* = \frac{p(\boldsymbol{x}) \mid f(\boldsymbol{x}) \mid}{Z} \tag{125}$$

- **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:

$$\hat{s}_{BIS} = \frac{\sum_{i=1}^{n} \frac{\tilde{p}(\boldsymbol{x}^{(i)})}{\tilde{q}(\boldsymbol{x}^{(i)})} f(\boldsymbol{x}^{(i)})}{\sum_{i=1}^{n} \frac{\tilde{p}(\boldsymbol{x}^{(i)})}{\tilde{q}(\boldsymbol{x}^{(i)})}} \tag{126}$$

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 $n \to \infty$.

Noise Contrastive Estimation (NCE) (18.6). We now estimate

$$\log p_{model}(\boldsymbol{x}) = \log \tilde{p}_{model}(\boldsymbol{x}; \boldsymbol{\theta}) + c \tag{127}$$

and explicitly learn an approximation, c, for $-\log Z(\boldsymbol{\theta})$. Obviously MLE would just try jacking up c to maximize this, so we adopt a surrogate supervised training problem: binary classification that a given sample $\boldsymbol{x}$ belongs to the (true) data distribution p_{data} or to the noise distribution p_{noise}. We introduce binary variable y to indicate whether the sample is in the true data distribution (y=1) or the noise distribution (y=0). Our surrogate model is thus defined by

$$p_{joint}(y{=}1) = \frac{1}{2} \tag{128}$$

$$p_{joint}(\boldsymbol{x} \mid y{=}1) = p_{model}(\boldsymbol{x}) \tag{129}$$

$$p_{joint}(\boldsymbol{x} \mid y{=}0) = p_{noise}(\boldsymbol{x}) \tag{130}$$

We can now use MLE on the optimization problem,

$$\boldsymbol{\theta}, c = \arg\max_{\boldsymbol{\theta}, c} \mathbb{E}_{\boldsymbol{x}, y \sim p_{train}} \left[\log p_{joint}(y \mid \boldsymbol{x})\right] \tag{131}$$

$$p_{joint}(y{=}1 \mid \boldsymbol{x}) = \frac{p_{model}(\boldsymbol{x})}{p_{model}(\boldsymbol{x}) + p_{noise}(\boldsymbol{x})} \tag{132}$$

$$= \frac{1}{1 + p_{noise}(\boldsymbol{x}) / p_{model}(\boldsymbol{x})} \tag{133}$$

$$= \sigma\left(\log p_{model}\left(\boldsymbol{x}\right) - \log p_{noise}\left(\boldsymbol{x}\right)\right) \tag{134}$$

Approximate Inference (Ch. 19)

Written by Brandon McKinzie

Overview. Most graphical models with multiple layers of hidden variables have intractable posterior distributions. This is typically because the partition function scales exponentially with the number of units and/or due to marginalizing out latent variables. Many approximate inference approaches make use of the observation that exact inference can be described as an optimization problem.

Assume we have a probabilistic model consisting of observed variables $\boldsymbol{v}$ and latent variables $\boldsymbol{h}$. We want to compute $\log p(\boldsymbol{v};\boldsymbol{\theta})$, but it's too costly to marginalize out $\boldsymbol{h}$. Instead, we compute a lower bound $\mathcal{L}(\boldsymbol{v},\boldsymbol{\theta},q)$ – often called the **evidence lower bound** (ELBO) or negative **variational free energy** – on $\log p(\boldsymbol{v};\boldsymbol{\theta})$[39]:

$$\mathcal{L}(\boldsymbol{v},\boldsymbol{\theta},q) = \log p(\boldsymbol{v};\boldsymbol{\theta}) - D_{KL}\left(q(\boldsymbol{h}\mid\boldsymbol{v}) \;||\; p(\boldsymbol{h}\mid\boldsymbol{v};\boldsymbol{\theta})\right) \tag{135}$$

$$= -\mathbb{E}_{\boldsymbol{h}\sim q(\boldsymbol{h}|\boldsymbol{v})}\left[\log p(\boldsymbol{h},\boldsymbol{v})\right] + H(q(\boldsymbol{h}\mid\boldsymbol{v})) \tag{136}$$

where the second form is the more canonical definition[40]. Note that $\mathcal{L}(\boldsymbol{v},\boldsymbol{\theta},q)$ is a lower-bound on $\log p(\boldsymbol{v};\boldsymbol{\theta})$ by definition, since

$$\log p(\boldsymbol{v};\boldsymbol{\theta}) - \mathcal{L}(\boldsymbol{v},\boldsymbol{\theta},q) = D_{KL}\left(q(\boldsymbol{h}\mid\boldsymbol{v})||p(\boldsymbol{h}\mid\boldsymbol{v};\boldsymbol{\theta})\right) \geq 0$$

With equality (to zero) iff q is the same distribution as $p(\boldsymbol{h}\mid\boldsymbol{v})$. In other words, $\mathcal{L}$ can be viewed as a function parameterized by q that's maximized when q is $p(\boldsymbol{h}\mid\boldsymbol{v})$, and with maximal value $\log p(\boldsymbol{v})$. Therefore, we can cast the *inference* problem of computing the (log) probability of the observed data $\log p(\boldsymbol{v})$ into an *optimization* problem of maximizing $\mathcal{L}$. Exact inference can be done by searching over a family of functions that contains $p(\boldsymbol{h}\mid\boldsymbol{v})$.

[39] Recall that $D_{KL}(P||Q) = \mathbb{E}_{x\sim P(x)}\left[\log \frac{P(x)}{Q(x)}\right]$

[40] This can be derived easily from the first form. Hint:

$$\log \frac{q(\boldsymbol{h}\mid\boldsymbol{v})}{p(\boldsymbol{h}\mid\boldsymbol{v})} = \log \frac{q(\boldsymbol{h}\mid\boldsymbol{v})}{p(\boldsymbol{h},\boldsymbol{v};\boldsymbol{\theta})/p(\boldsymbol{v};\boldsymbol{\theta})}$$

Expectation Maximization (19.2). Technically not an approach to approximate inference, but rather an approach to learning with an approximate posterior. Popular for training models with latent variables. The EM algorithm consists of alternating between the followi[p]ng 2 steps until convergence:

1. **E-step**. For each training example $\boldsymbol{v}^{(i)}$ (in current batch or full set), set

$$q(\boldsymbol{h} \mid \boldsymbol{v}^{(i)}) = p(\boldsymbol{h} \mid \boldsymbol{v}^{(i)}; \boldsymbol{\theta}^{(0)}) \tag{137}$$

   where $\boldsymbol{\theta}^{(0)}$ denotes the current parameter values of the model at the beginning of the E-step. This can also be interpreted as maximizing $\mathcal{L}$ w.r.t. q.
2. **M-step**. Update the parameters $\boldsymbol{\theta}$ by completely or partially finding

$$\arg\max_{\boldsymbol{\theta}} \sum_i \mathcal{L}\left(\boldsymbol{v}^{(i)}, \boldsymbol{\theta}, q(\boldsymbol{h} \mid \boldsymbol{v}^{(i)}; \boldsymbol{\theta}^{(0)})\right) \tag{138}$$

Boltzmann Machines (20.1). An energy-based model over a d-dimensional binary random vector $\mathbf{x} \in \{0,1\}^d$. The energy function is simply $E(\boldsymbol{x}) = -\boldsymbol{x}^T\boldsymbol{U}\boldsymbol{x} - \boldsymbol{b}^T\boldsymbol{x}$, i.e. parameters between all pairs of x_i, x_j, and bias parameters for each x_i[41]. In settings where our data consists of samples of fully observed $\boldsymbol{x}$, this is clearly limited to very simple cases, since e.g. the probability of some x_i being on is given by logistic regression from the values of the other units.

Proof: prob of x_i being on is logistic regression on other units

It's important to be as specific as possible here, since the task stated as-is is ambiguous. We want to prove that the probability of some fully observed state $\boldsymbol{x}$ that has its ith element clamped to 1, which I'll denote as $p_{i=on}(\boldsymbol{x})$, is logistic regression over the other units.

To prove this, it's easier to use the conventional definition where $\boldsymbol{U}$ is symmetric with zero diagonal, and we write $E(\boldsymbol{x})$ as

$$E(\boldsymbol{x}) = -\sum_{i=1}^{d}\sum_{j=i+1}^{d} x_i U_{i,j} x_j - \sum_{i=1}^{d} b_i x_i \tag{139}$$

where the difference is that we explicitly only sum over the upper triangle of $\boldsymbol{U}$.

Intuitively, since $p(\{\boldsymbol{x}\}_{j\neq i}) = p_{i=on}(\boldsymbol{x}) + p_{i=off}(\boldsymbol{x})$, our final formula for $p_{i=on}$ should only contain terms involving the parameters that interact with x_i, and only for those cases where $x_i = 1$. This motivates exploring the formula for $\Delta E_i(\boldsymbol{x}) \triangleq E_{i=off} - E_{i=on}$ where I've dropped the explicit notation on $\boldsymbol{x}$ for simplicity/space. Before jumping in to deriving this, step back and realize that ΔE_i will only contain summation terms where either the row or column index of $\boldsymbol{U}$ is i, and only for terms with bias element b_i. Since our summation is over the upper triangle of $\boldsymbol{U}$, this means terms along the slices $U_{i,i+1:d}$ and $U_{1:i-1,i}$. Now there is no derivation needed and we can simply write

$$\Delta E_i = \sum_{k=i+1}^{d} U_{i,k} x_k + \sum_{k=1}^{i-1} x_k U_{k,i} + b_i \tag{140}$$

The goal is to use this to get a logistic-regression-like formula for $p_{i=on}$, so we should now think about the relationship between any given $p(\boldsymbol{x})$ and the associated $E(\boldsymbol{x})$. The critical observation is that $E(\boldsymbol{x}) = -\ln p(\boldsymbol{x}) - \ln Z$, which therefore means

$$\Delta E_i = \ln p_{i=on}(\boldsymbol{x}) - \ln p_{i=off}(\boldsymbol{x}) = -\ln\left(\frac{1 - p_{i=on}(\boldsymbol{x})}{p_{i=on}(\boldsymbol{x})}\right) \tag{141}$$

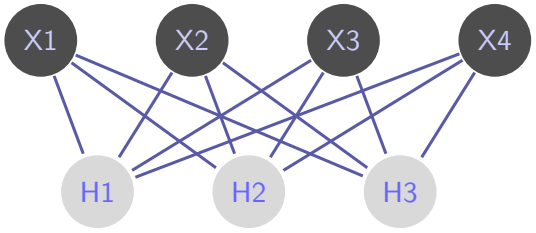
$$\exp(-\Delta E_i) = \frac{1 - p_{i=on}(\boldsymbol{x})}{p_{i=on}(\boldsymbol{x})} = \frac{1}{p_{i=on}(\boldsymbol{x})} - 1 \tag{142}$$

$$\therefore p_{i=on}(\boldsymbol{x}) = \frac{1}{1 + \exp(-\Delta E_i)} \tag{143}$$

Since ΔE_i is a linear function of all other units, we have proven that $p_{i=on}(\boldsymbol{x})$ for some state $\boldsymbol{x}$ reduces to logistic regression over the other units.

[41] Authors are being lazy because it's assumed the reader is familiar (which is fair, I guess). i.e. they aren't mentioning that this formula implies that $\boldsymbol{U}$ is either lower or upper triangular, and the diagonal is zero.

Restricted Boltzmann Machines (20.2). A BM with variables partitioned into two sets: hidden and observed. The graphical model is bipartite over the hidden and observed nodes, as I've drawn in the example below.



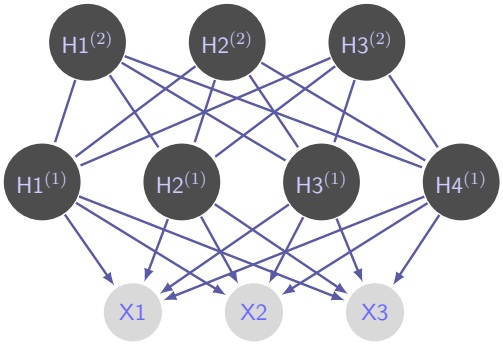
Although the joint distribution $p(\boldsymbol{x}, \boldsymbol{h})$ has a potentially intractable partition function, the conditional distributions can be computed efficiently by exploiting independencies:

$$p(\boldsymbol{h} \mid \boldsymbol{x}) = \prod_{j=1}^{n_h} \sigma\left([2\boldsymbol{h} - 1] \odot [\boldsymbol{c} + \boldsymbol{W}^T \boldsymbol{x}]\right)_j \tag{144}$$

$$p(\boldsymbol{x} \mid \boldsymbol{h}) = \prod_{i=1}^{n_x} \sigma\left([2\boldsymbol{x} - 1] \odot [\boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}]\right)_i \tag{145}$$

where $\boldsymbol{b}$ and $\boldsymbol{c}$ are the observed and hidden bias parameters, respectively.

Deep Belief Networks (20.3). Several layers of (usually binary) latent variables and a single observed layer. The "deepest" (away from the observed) layer connections are undirected, and all other layers are directed and pointing toward the data. I've drawn an example below.



We can sample from a DBN via first Gibbs sampling on the undirected layer, then ancestral sampling through the rest of the (directed) model to eventually obtain a sample from the visible units.

Deep Boltzmann Machines (20.4). Same as DBNs, but now all layers are undirected. Note that this is very close to the standard RBM, since we have a set of hidden and observed variables, except now we interpret certain subgroups of hidden units as being in a "layer", thus allowing for connections between hidden units in adjacent layers. What's interesting is that this still defines a bipartite graph, with odd-numbered layers on one side and even on the other[42].

Differentiable Generator Networks (20.10.2). Use a differentiable function $g(\boldsymbol{z}; \boldsymbol{\theta}^{(g)})$ to transform samples of latent variables $\mathbf{z}$ to either (a) samples $\mathbf{x}$, or (b) distributions over samples $\mathbf{x}$. For an example of case (a), the standard procedure for sampling from $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ is to first sample from $\mathcal{N}(\mathbf{0}, \boldsymbol{I})$ into a generator network consisting of a single affine layer:

$$\boldsymbol{x} \leftarrow g(\boldsymbol{z}) = \boldsymbol{\mu} + \boldsymbol{L}\boldsymbol{z}$$

where $\boldsymbol{L}$ is the *Cholesky decomposition*[43] of $\boldsymbol{\Sigma}$. In general, we think of the generator function g as providing a change of variables that transforms the distribution over $\mathbf{z}$ into the desired distribution $\mathbf{x}$. Of course, there *is* an exact formula for doing this,

$$p_x(\boldsymbol{x}) = \frac{p_z(g^{-1}(\boldsymbol{x}))}{\left|\det \frac{\partial g}{\partial \boldsymbol{z}}\right|} \tag{146}$$

but it's usually far easier to use indirect means of learning g, rather than trying to maximize/evaluation $p_x(\boldsymbol{x})$ directly.

For case (b), the common approach is to train the generator net to emit conditional probabilities

$$p(x_i \mid \boldsymbol{z}) = g(\boldsymbol{z})_i \qquad\qquad p(\boldsymbol{x}) = \mathbb{E}_{\boldsymbol{z}}\left[p(\boldsymbol{x} \mid \boldsymbol{z})\right] \tag{147}$$

which can also support generating discrete data (case a cannot). The challenge in training generator networks is that we often have a set of examples $\boldsymbol{x}$, but the value of $\boldsymbol{z}$ for each $\boldsymbol{x}$ is not fixed and known ahead of time. We'll now look at some ways of training generator nets given only training samples for $\boldsymbol{x}$. Note that such a setting is very unlike unsupervised learning, where we typically interpret $\boldsymbol{x}$ as inputs that we don't have labels for, while here we interpret $\boldsymbol{x}$ as *outputs* that we don't know the associated inputs for.

[42] Recall that this immediately implies that units in all odd layers are conditionally independent given the even layers (and vice-versa for even to odd).

[43] The [unique] Cholesky decomposition of a (real-symmetric) p.d. matrix $\boldsymbol{A}$ is a decomposition of the form $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^T$, where $\boldsymbol{L}$ is lower triangular.

Variational Autoencoders (20.10.3). VAEs are directed models that use learned approximate inference and can be trained purely with gradient-based methods. To generate a sample $\boldsymbol{x}$, the VAE first samples $\boldsymbol{z}$ from the *code distribution* $p_{model}(\boldsymbol{z})$. This sample is then fed through the a differentiable generator network $g(\boldsymbol{z})$. Finally, $\boldsymbol{x}$ is sampled from $p_{model}(\boldsymbol{x}; g(\boldsymbol{z})) = p_{model}(\boldsymbol{x} \mid \boldsymbol{z})$.

Papers and Tutorials

Contents

WaveNet

Written by Brandon McKinzie

Introduction.

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

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

$$p(x) = \prod_{t=1}^{T} p(x_t \mid x_1, \ldots, x_{t-1}) \tag{148}$$

which are modeled by a stack of convolutional layers (no pooling). 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[44].

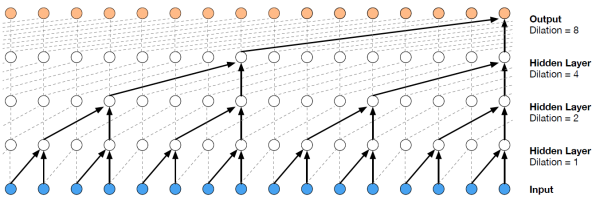


Figure 3: Visualization of a stack of *dilated* causal convolutional layers.

Dilated Convolution

> *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.*

[44] 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"

Softmax distributions. To deal with the fact that there are 2^{16} possible values, first apply a **μ-law companding transformation**[45] to data, and then quantize it to 256 possible values:

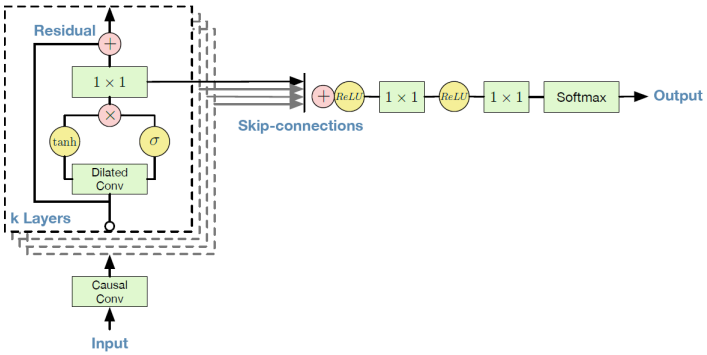
$$f(x_t) = \text{sign}(x_t)\frac{\ln(1+\mu|x_t|)}{\ln(1+\mu)} \tag{149}$$

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

Gated Activation and Residual/Skip Connections. Use the same gated activation unit as PixelCNN:

$$z = \tanh\left(W_{f,k} * x\right) \odot \sigma\left(W_{g,k} * x\right) \tag{150}$$

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.



The 1x1 blocks are 1x1 convolutions (i.e. position-wise dense layers).

[45] In telecommunication and signal processing **companding** (occasionally called compansion) is a method of mitigating the detrimental effects of a channel with limited dynamic range.

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

$$p(\boldsymbol{x} \mid \boldsymbol{h}) = \prod_{t=1}^{T} p(x_t \mid x_1, \ldots, x_{t-1}, h) \tag{151}$$

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

$$z = \tanh\left(W_{f,k} * \boldsymbol{x} + V_{f,k}^T \boldsymbol{h}\right) \odot \sigma\left(W_{g,k} * \boldsymbol{x} + V_{g,k}^T \boldsymbol{h}\right) \tag{152}$$

→ **Local conditioning**. Have a second time-series h_t. They first transform this h_t using a **transposed CNN (learned upsampling)** that maps it to a new time-series $\boldsymbol{y} = f(\boldsymbol{h})$ w/same resolution as $\boldsymbol{x}$.

$$z = \tanh\left(W_{f,k} * \boldsymbol{x} + V_{f,k} * \boldsymbol{y}\right) \odot \sigma\left(W_{g,k} * \boldsymbol{x} + V_{g,k}^T \boldsymbol{y}\right) \tag{153}$$

Experiments.

- **Multi-Speaker Speech Generation**. Dataset: multi-speaker corpus of 44 hours of data from 109 different speakers[46]. 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."

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

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

Neural Style

Written by Brandon McKinzie

Notation.

- **Content image**: $\boldsymbol{p}$
- **Filter responses**: the matrix $P^l \in \mathcal{R}^{N_l \times M_l}$ contains the activations of the filters in layer l, where P^l_{ij} would give the activation of the ith filter at position j in layer l. N_l is number of feature maps, each of size M_l (height $\times$ width of the feature map)[47].
- **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 $\boldsymbol{p}$.

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

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

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

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

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

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

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

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

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

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

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

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

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

Papers and Tutorials **February 8**

Neural Conversation Model

Table of Contents Local *Written by Brandon McKinzie*

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

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

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

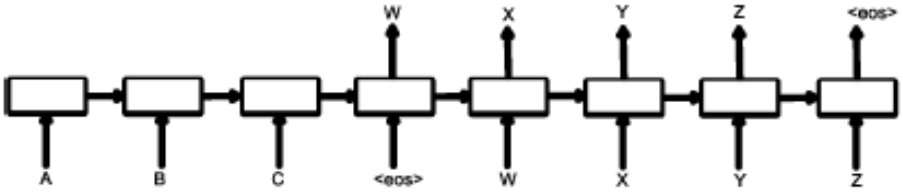
Related Work. The authors' approach is based on the following (linked and saved) papers on seq2seq:

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

Model. Succinctly described by the authors:

> The model reads the input sequence one token at a time, and predicts the output sequence, also one token at a time. During training, the true output sequence is given to the model, so learning can be done by backpropagation. The model is trained to maximize the cross entropy of the correct sequence given its context. During inference, in which the true output sequence is not observed, we simply feed the predicted output token as input to predict the next output. This is a "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][48].

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

IT Data & Experiment.

- **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.

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

NMT By Jointly Learning to Align & Translate

Written by Brandon McKinzie

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

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

Learning to Align[49] and translate.

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

$$p(y_i \mid y_1, \dots, y_{i-1}, \boldsymbol{x}) = g(y_{i-1}, s_i, c_i) \tag{154}$$

$$s_i = f(s_{i-1}, y_{i-1}, c_i) \tag{155}$$

  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:

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j \tag{156}$$

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})} \tag{157}$$

$$e_{ij} = a(s_{i-1}, h_j) \tag{158}$$

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

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

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

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

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

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

$$z_t = \sigma\left(W_z y_{t-1} + U_z[s_{t-1},\ c_t]\right) \qquad \textbf{[update gate]} \tag{159}$$

$$r_t = \sigma\left(W_r y_{t-1} + U_r[s_{t-1},\ c_t]\right) \qquad \textbf{[reset gate]} \tag{160}$$

$$\tag{161}$$

3. **Proposal state**:

$$\tilde{s}_t = \tanh\left(W y_{t-1} + U[r_t \circ s_{t-1},\ c_t]\right) \tag{162}$$

4. **Hidden state**:

$$s_t = (1 - z_t) \circ s_{t-1} + z_t \circ \tilde{s}_t \tag{163}$$

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:

$$a_j = a(s_{t-1}, h_j) = v_a^T \tanh\left(W_a s_{t-1} + U_a h_j\right) \tag{164}$$

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

$$a_j \leftarrow \frac{e^{a_j}}{\sum_{k=0}^{L_{enc}-1} e^{a_k}} \tag{165}$$

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

$$c = \sum_{j=1}^{L_{enc}} a_j h_j \tag{166}$$

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]:

$$\Pr(y \mid s, c) \propto e^{y^T W_o u} \tag{167}$$

$$u = [\max\{\tilde{u}_{2j-1}, \tilde{u}_{2j}\}]_{j=1,\ldots,\ell}^{T} \tag{168}$$

$$\tilde{u} = U_o[s_{t-1},\ c] + V_o y_{t-1} \tag{169}$$

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:

$$\tilde{\boldsymbol{h}}_t = \tanh\left(\boldsymbol{W}_c[\boldsymbol{c}_t;\ \boldsymbol{h}_t]\right) \tag{170}$$

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

$$p(y_t \mid y_{<t}, x) = \text{softmax}\left(\boldsymbol{W}_s \tilde{\boldsymbol{h}}_t\right) \tag{171}$$

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:

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^T \bar{\boldsymbol{h}}_s & \textit{dot} \\ \boldsymbol{h}_t^T \boldsymbol{W}_a \bar{\boldsymbol{h}}_s & \textit{general} \\ \boldsymbol{v}_a^T \tanh\left(\boldsymbol{W}_a[\boldsymbol{h}_t;\ \bar{\boldsymbol{h}}_s]\right) & \textit{concat} \end{cases} \tag{172}$$

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

$$\boldsymbol{a}_t(s) = \text{align}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) \tag{173}$$

$$= \frac{\exp(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s))}{\sum_{s'} \exp(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_{s'}))} \tag{174}$$

3. $\boldsymbol{c}_t$: The weighted average over all source (encoder) hidden states[50]:

$$\boldsymbol{c}_t = \sum_{i=1}^{L_{enc}} \boldsymbol{a}_t(i)\bar{\boldsymbol{h}}_i \tag{175}$$

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

$$\tilde{\boldsymbol{h}}_t = \tanh\left(\boldsymbol{W}_c[\boldsymbol{c}_t;\ \boldsymbol{h}_t]\right) \tag{176}$$

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

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

Using Large Vocabularies for NMT

 Written by Brandon McKinzie

Paper information:

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

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

$$x = (x_1, \dots, x_T) \qquad \textbf{[source sentence]} \tag{177}$$
$$h = (h_1, \dots, h_T) \qquad \textbf{[encoder state seq]} \tag{178}$$
$$h_t = f(x_t, h_{t-1}) \tag{179}$$

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:

$$y = (y_1, \dots, y'_T) \qquad [y_i \in \mathbb{Z}] \tag{180}$$
$$\Pr[y_t \mid y_{<t},\ x] \propto e^{q(y_{t-1},\ z_t,\ c_t)} \tag{181}$$
$$z_t = g(y_{t-1}, z_{t-1}, c_t) \qquad \textbf{[decoder hidden?]} \tag{182}$$
$$c_t = r(z_{t-1}, h_1, \dots, h_T) \qquad \textbf{[decoder inp?]} \tag{183}$$

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,

$$\theta^* = \arg\max_\theta \sum_{n=1}^{N} \sum_{t=1}^{T_n} \log\left(\Pr[y_t^n \mid y_{<t}^n,\ x^n]\right) \tag{184}$$

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.

$$c_t = \sum_{i=1}^{T} \alpha_i h_i \tag{185}$$

$$\alpha_t = \frac{e^{a(h_t, z_{t-1})}}{\sum_k e^{a(h_k, z_{t-1})}} \tag{186}$$

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 181 as[51] ,

$$\Pr[y_t \mid y_{<t},\ x] = \frac{1}{Z} e^{\mathbf{w}_t^T \phi(y_{t-1}, z_t, c_t) + b_t} \tag{187}$$

$$Z = \sum_{k:\ y_k \in V} e^{\mathbf{w}_k^T \phi(y_{t-1}, z_t, c_t) + b_k} \tag{188}$$

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

$$\nabla \log\left(\Pr[y_t \mid y_{<t},\ x]\right) = \nabla \mathcal{E}(y_t) - \sum_{k:\ y_k \in V} \Pr[y_k \mid y_{<t},\ x] \nabla \mathcal{E}(y_k) \tag{189}$$

$$\mathcal{E}(y_j) = \mathbf{w}_j^T \phi(y_{t-1}, z_t, c_t) + b_j \tag{190}$$

[51] 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.

[52] **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'.

$$\mathbb{E}_P[\nabla\mathcal{E}(y)] \approx \sum_{k:\ y_k \in V'} \frac{\omega_k}{\sum_{k':y_{k'}\in V'} \omega_{k'}} \nabla\mathcal{E}(y_k) \tag{191}$$

$$\omega_k = e^{\mathcal{E}(y_k) - \log Q(y_k)} \tag{192}$$

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

$$\omega_k = \frac{e^{\mathcal{E}(y_k)}}{Q(y_k)} \quad \text{thus} \quad p(y_k \mid y_{<t}, x) = \omega_k \frac{Q(y_k)}{Z} \tag{193}$$

$$\rightarrow \quad e^{\mathcal{E}(y_k)} = Z \cdot p(y_k \mid y_{<t}, x) = Q(y_k) \cdot \omega_k \tag{194}$$

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.

$$\mathbb{E}_P[\nabla\mathcal{E}(y)] = \sum_{k:\ y_k \in V} \frac{\omega_k \cdot Q(y_k)}{\sum_{k':y_{k'}\in V} \omega_{k'} \cdot Q(y_{k'})} \nabla\mathcal{E}(y_k) \tag{195}$$

$$\mathbb{E}_P[\nabla\mathcal{E}(y)] = \sum_{k:\ y_k \in V'} \frac{\omega_k}{\sum_{k':y_{k'}\in V'} \omega_{k'}} \nabla\mathcal{E}(y_k) \tag{196}$$

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

How to do in practice (very important).

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

Candidate Sampling – TensorFlow

Written by Brandon McKinzie

[Link to article]

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

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

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

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

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

$$\Pr\left(y_i^{true} = y \mid C_i,\ x_i\right) = \frac{\Pr\left(y_i^{true} = y \mid x_i\right) \cdot \Pr\left(C_i \mid y_i^{true} = y,\ x_i\right)}{\Pr\left(C_i \mid x_i\right)} \tag{197}$$

$$= \frac{\Pr\left(y \mid x_i\right)}{Q\left(y \mid x_i\right)} \cdot \frac{1}{K(x_i, C_i)} \tag{198}$$

where they've just defined

$$K(x_i, C_i) \triangleq \frac{\Pr\left(C_i \mid x_i\right)}{\prod_{y' \in C_i} Q(y' \mid x_i) \prod_{y' \in (L - C_i)} \left(1 - Q(y' \mid x_i)\right)} \tag{199}$$

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

$$F(x, y) = \log(\Pr(y \mid x)) + K(x) \tag{200}$$

which is the general form because

$$\text{Softmax}(\log(\Pr(y \mid x)) + K(x)) = \frac{e^{\log(\Pr(y|x))+K(x)}}{\sum_{y'} e^{\log(\Pr(y'|x)+K(x)}} \tag{201}$$

$$= \Pr(y \mid x) \tag{202}$$

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 200 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 198. 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

$$\text{Softmax Input: } F(x, y) - \log(Q(y \mid x)) \tag{203}$$

instead. That's it.

Attention Terminology

 Written by Brandon McKinzie

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

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

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

$$e_i = v^T \tanh(\text{FC}(s_{i-1}) + \text{FC}(h)) \tag{204}$$

  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[53]

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

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

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

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^T \bar{\boldsymbol{h}}_s & \textit{dot} \\ \boldsymbol{h}_t^T \boldsymbol{W}_a \bar{\boldsymbol{h}}_s & \textit{general} \\ \boldsymbol{v}_a^T \tanh\left(\boldsymbol{W}_a[\boldsymbol{h}_t;\ \bar{\boldsymbol{h}}_s]\right) & \textit{concat} \end{cases} \tag{205}$$

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

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

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

$$\boldsymbol{a}_t(s) = \text{align}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) \tag{206}$$

$$= \frac{\exp(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s))}{\sum_{s'} \exp(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_{s'}))} \tag{207}$$

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

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

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:

$$\tilde{\boldsymbol{h}}_t = \tanh\left(\boldsymbol{W}_c\left[\boldsymbol{c}_t,\ \boldsymbol{h}_t\right]\right) \tag{208}$$

$$p(y_t \mid y_{<t}, x) = \text{softmax}(\boldsymbol{W}_s \tilde{\boldsymbol{h}}_t) \tag{209}$$

TextRank

Written by Brandon McKinzie

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

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:

$$S(V_i) = (1-d) + d * \sum_{V_j \in \text{In}(V_i)} \frac{1}{|\ \text{Out}(V_i)\ |} S(V_j) \quad \text{where} \quad d \in \Re[0,1] \tag{210}$$

The facto
set to 0.8

and the damping factor d is interpreted as the probability of jumping from a given vertex[54] to another random vertex in the graph. In practice, the algorithm is implemented through the following steps:

(1) Initialize all vertices with arbitrary values.[55]
(2) Iterate over vertices, computing equation 4.9 until convergence [of the error rate] below a predefined threshold. The error-rate, defined as the difference between the "true score" and the score computed at iteration k, $S^k(V_i)$, is *approximated* as:

$$\text{Error}^k(V_i) \approx S^k(V_i) - S^{k-1}(V_i) \tag{211}$$

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

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

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

$$WS(V_i) = (1-d) + d * \sum_{j \in \text{In}(V_i)} \frac{w_{ji}}{\sum V_k \in \text{Out}(V_j) w_{jk}} WS(V_j) \tag{212}$$

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.

4.9.1 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[56].

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 $T = |V|/3$.[57] 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**[58]. 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.

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

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

[58] 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.

4.9.2 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:

$$\text{Similarity}(S_i, S_j) = \frac{\mid S_i \cap S_j \mid}{\log(\mid S_i \mid) + \log(\mid S_j \mid)} \tag{213}$$

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

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

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

Simple Baseline for Sentence Embeddings

Written by Brandon McKinzie

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

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

Algorithm.

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

$$\frac{1}{N}\sum_{i}^{N}\frac{a}{a+p(\boldsymbol{w}_i)}\boldsymbol{w}_i \tag{214}$$

The authors call t
weighted average
Smooth Inverse
Frequency (SIF

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

Algorithm 1 Sentence Embedding

Input: Word embeddings $\{v_w : w \in \mathcal{V}\}$, a set of sentences $\mathcal{S}$, parameter a and estimated probabilities $\{p(w) : w \in \mathcal{V}\}$ of the words.
Output: Sentence embeddings $\{v_s : s \in \mathcal{S}\}$
for all sentence s in $\mathcal{S}$ **do**
  $v_s \leftarrow \frac{1}{|s|}\sum_{w\in s}\frac{a}{a+p(w)}v_w$
end for
Compute the first principal component u of $\{v_s : s \in \mathcal{S}\}$
for all sentence s in $\mathcal{S}$ **do**
  $v_s \leftarrow v_s - uu^\top v_s$
end for

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

$$\Pr[w_t \mid c_t] \propto e^{c_t \cdot v_w} \tag{215}$$

- **The Random Walk**. If we assume that c_t doesn't change much over the words in a single sentence, we can assume it stays at some c_s. The authors claim that in their previous paper they showed that the MAP[59] estimate of c_s is – up to multiplication by a scalar – the average of the embeddings of the words in the sentence.
- **Improvements/Modifications to 215**.
  1. Additive term $\alpha p(w)$ where α is a scalar. Allows words to occur even if $c_t \cdot v_w$ is very small.
  2. Common discourse vector $c_0 \in \Re^d$. Correction term for the most frequent discourse that is often related to syntax.
- **Model**. Given the discourse vector c_s for a sentence s, the probability that w is in the sentence (at all (?)):

$$\Pr[w \mid c_s] = \alpha p(w) + (1-\alpha)\frac{e^{\tilde{c}_s \cdot v_w}}{Z_{\tilde{c}_s}} \tag{216}$$

$$\tilde{c}_s = \beta c_0 + (1-\beta)c_s \tag{217}$$

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

[59] Review of MAP:

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

Survey of Text Clustering Algorithms

 Written by Brandon McKinzie

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

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

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

Feature Selection.

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

  See ref 94 of the paper for more.

  $$s(t) = \Pr\left[t \in y \mid t \in x\right] \tag{218}$$

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

  $$s(t) = \frac{\text{Num pairs in which t occurs in both}}{\text{Num pairs in which t occurs in the first of the pair}} \tag{219}$$

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

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

  $$E(t) = -\sum_{i=1}^{n}\sum_{j=1}^{n}\left(S_{ij}\cdot\log(S_{ij}) + (1-S_{ij})\cdot\log(1-S_{ij})\right) \tag{220}$$

  $$S_{ij} = 2^{-d_{ij}/\bar{d}} \tag{221}$$

  where

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

LSI-based Methods. Latent Semantic Indexing is based on dimensionality reduction where the new (transformed) features are a linear combination of the originals. This helps magnify the semantic effects in the underlying data. LSI is quite similar to PCA[60], except that we use an approximation of the covariance matrix C which is appropriate for the sparse and high-dimensional nature of text data.

Let $\boldsymbol{A} \in \mathbb{R}^{n \times d}$ be term-document matrix, where $\boldsymbol{A}_{i,j}$ is the (normalized) frequency for term j in document i. Then $\boldsymbol{A}^T\boldsymbol{A} = n \cdot \Sigma$ is the (scaled) approximation to covariance matrix[61], assuming the data is mean-centered. Quick check/reminder:

$$(\boldsymbol{A}^T\boldsymbol{A})_{ij} = \boldsymbol{A}_{:,i}^T\boldsymbol{A}_{:,j} \triangleq \boldsymbol{a}_i^T\boldsymbol{a}_j \tag{222}$$

$$\approx n \cdot \mathbb{E}\left[\mathbf{a}_i\mathbf{a}_j\right] \tag{223}$$

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

As usual, we using the eigenvectors of $\boldsymbol{A}^T\boldsymbol{A}$ with the largest variance in order to represent the text[62]. In addition:

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

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

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

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

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

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

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

$$J = \frac{1}{2}||\boldsymbol{A} - \boldsymbol{U}\boldsymbol{V}^T||_F^2 \tag{224}$$

$$= \frac{1}{2}\left(tr(\boldsymbol{A}\boldsymbol{A}^T) - 2tr(\boldsymbol{A}\boldsymbol{V}\boldsymbol{U}^T) + tr(\boldsymbol{U}\boldsymbol{V}^T\boldsymbol{V}\boldsymbol{U}^T)\right) \tag{225}$$

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

$$\boldsymbol{a} \approx \boldsymbol{u} \cdot \boldsymbol{V}^T \tag{226}$$

$$= \sum_{i=1}^{k} \boldsymbol{u}_i \boldsymbol{V}_i^T \tag{227}$$

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

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

- Variables to optimize: All elements of both $\boldsymbol{U} = [u_{ij}]$ and $\boldsymbol{V} = [v_{ij}]$
- Constraint: non-negativity, i.e. $\forall i, j,\ u_{ij} \geq 0$ and $v_{ij} \geq 0$.
- Multipliers: Denote as matrices α and β, with same dimensions as $\boldsymbol{U}$ and $\boldsymbol{V}$, respectively.
- Lagrangian: I'll just show it here first, and then explain in this footnote[63]:

$$\mathcal{L} = J + tr(\alpha \cdot \boldsymbol{U}^T) + tr(\beta \cdot \boldsymbol{V}^T) \tag{228}$$

$$\text{where} \quad tr(\alpha \cdot \boldsymbol{U}^T) = \sum_{i=1}^{n} \alpha_i \cdot \boldsymbol{u}_i = \sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_{ij} u_{ij} \tag{229}$$

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

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

- Optimization: Set the partials of $\mathcal{L}$ w.r.t both U and V (separately) to zero[64]:

$$\frac{\partial \mathcal{L}}{\partial U} = -\boldsymbol{A} \cdot \boldsymbol{V} + \boldsymbol{U} \cdot \boldsymbol{V}^T \cdot \boldsymbol{V} + \boldsymbol{\alpha} = 0 \tag{230}$$

$$\frac{\partial \mathcal{L}}{\partial V} = -\boldsymbol{A}^T \cdot \boldsymbol{U} + \boldsymbol{V} \cdot \boldsymbol{U}^T \cdot \boldsymbol{U} + \boldsymbol{\beta} = 0 \tag{231}$$

  Since, ultimately, these just say [some matrix] $= 0$, we can multiply both sides (element-wise) by a constant ($x \times 0 = 0$). *Using*[65] the **Kuhn-Tucker conditions** $\alpha_{ij} \cdot u_{ij} = 0$ and $\beta_{ij} \cdot v_{ij} = 0$, we get:

$$(\boldsymbol{A} \cdot \boldsymbol{V})_{ij} \cdot u_{ij} - (\boldsymbol{U} \cdot \boldsymbol{V}^T \cdot \boldsymbol{V})_{ij} \cdot u_{ij} = 0 \tag{232}$$

$$(\boldsymbol{A}^T \cdot \boldsymbol{U})_{ij} \cdot v_{ij} - (\boldsymbol{V} \cdot \boldsymbol{U}^T \cdot \boldsymbol{U})_{ij} \cdot v_{ij} = 0 \tag{233}$$

- Update rules:

$$u_{ij} = \frac{(\boldsymbol{A} \cdot \boldsymbol{V})_{ij} \cdot u_{ij}}{(\boldsymbol{U} \cdot \boldsymbol{V}^T \cdot \boldsymbol{V})_{ij}} \tag{234}$$

$$v_{ij} = \frac{(\boldsymbol{A}^T \cdot \boldsymbol{U})_{ij} \cdot v_{ij}}{(\boldsymbol{V} \cdot \boldsymbol{U}^T \cdot \boldsymbol{U})_{ij}} \tag{235}$$

4.11.1 Distance-based Clustering Algorithms

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

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

- **Single Linkage Clustering**. Defines similarity between two groups (clusters) of documents as the largest similarity between any pair of documents from these two groups. First, (1) compute all similarity pairs [between documents; ignore cluster labels], then (2) sort in decreasing order, and (3) walk through the list in that order, merging clusters if the pair belong to different clusters. One drawback is *chaining*: the resulting clusters assume transitivity of similarity[66].

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

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

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

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

Distance-Based Partitioning Algorithms.

- **K-Medoid Clustering**. Use a set of points from training data as anchors (medoids) around which the clusters are built. Key idea is we are using an optimal set of representative documents *from the original corpus*. The set of k reps is successively improved via randomized inter-changes. In each iteration, we replace a randomly picked rep in the current set of medoids with a randomly picked rep from the collection, if it improves the clustering objective function. This approach is applied until convergence is achieved.
- **K-Means Clustering**. Successively (1) assigning points to the nearest cluster centroid and then (2) re-computing the centroid of each cluster. Repeat until convergence. Requires typically few iterations (about 5 for many large data sets). Disadvantage: sensitive to initial set of seeds (initial cluster centroids). One method for improving the initial set of seeds is to use some supervision - e.g. initialize with k pre-defined topic vectors (see ref. 4 in paper for more).

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

4.11.2 Probabilistic Document Clustering and Topic Models

Overview. Primary assumptions in any topic modeling approach:

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

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

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

$$\Pr[t_l \mid D_i] = \sum_{j=1}^{k} \Pr[t_l \mid T_j] \cdot \Pr[T_j \mid D_i] \tag{236}$$

which should be interpreted as a weighted average[67]. From here, we can generate a $n \times d$ matrix of probabilities.

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

$$J = -\log(\Pr[\boldsymbol{X}]) \tag{237}$$

$$= -\log\left(\prod_{i,l} \Pr[t_l \mid D_i]^{\boldsymbol{X}_{i,l}}\right) \tag{238}$$

$$= -\sum_{i,l} \boldsymbol{X}_{i,l} \cdot \log(\Pr[t_l \mid D_i]) \tag{239}$$

and we can plug-in eqn 236 to for evaluating $\Pr[t_l \mid D_i]$. We want to optimize the value of J, subject to the constraints:

$$(\forall T_j): \sum_{l} \Pr[t_l \mid T_j] = 1 \qquad (\forall D_i): \sum_{j} \Pr[T_j \mid D_i] = 1 \tag{240}$$

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

Latent Dirichlet Allocation (LDA). The term-topic probabilities and topic-document probabilities are modeled with a *Dirichlet distribution* as a prior[68]. Typically preferred over PLSI because PLSI more prone to overfitting.

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

[68] LDA is the Bayesian version of PLSI

4.11.3 Online Clustering with Text Streams

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

Overview. Maintaining text clusters in real time. One method is the **Online Spherical K-Means Algorithm** (OSKM)[69].

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

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

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

1. Initialize empty set of clusters $\mathcal{C} = \{\}$. As new data points arrive, unit clusters containing individual data points are created. Once a maximum number k of such clusters have been created, we can begin the process of online cluster maintenance.
2. For a new data point X, compute its similarity to each cluster C_j, denoted as $S(X, C_j)$.
   - If $S(X, C_{best}) > \text{thresh}_{outlier}$, or if there are no inactive clusters left[70], insert X to the cluster with maximum similarity.
   - Otherwise, a new cluster is created[71] containing the solitary data point X.

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

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

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

Misc.

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

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

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

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

Deep Sentence Embedding Using LSTMs

Written by Brandon McKinzie

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

Abstract. Sentence embeddings using LSTM cells, which automatically attenuate unimportant words and detect salient keywords. Main emphasis on applications for document retrieval (matching a query to a document[72]).

Introduction. Sentence embeddings are learned using a loss function defined on *sentence pairs*. For example, the well-known Paragraph Vector[73] is learned in an unsupervised manner as a distributed representation of sentences and documents, which are then used for sentiment analysis.

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

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

Related Work. (Identified by reference number)

- [2] Good for sentiment, but doesn't capture fine-grained sentence structure.
- [6] Unsupervised embedding method trained on the BookCorpus [7]. Not good for document retrieval task.
- [9] Semi-supervised Recursive Autoencoder (RAE) for sentiment prediction.
- [3] DSSM (uses bag-of-words) and [10] CLSM (uses bag of n-grams) models for IR and also sentence embeddings.
- [12] Dynamic CNN for sentence embeddings. Good for sentiment prediction and question type classification. In [13], a CNN is proposed for **sentence matching**[74]

[72] Note that this similar to topic extraction.
[73] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents."
[74] Might want to look into this.

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

1. Encode tth word [of the given sentence] in one-hot vector $\boldsymbol{x}(t)$.
2. Convert $\boldsymbol{x}(t)$ to a letter tri-gram vector $\boldsymbol{l}(t)$ using fixed hashing operator[75] $\boldsymbol{H}$:

$$\boldsymbol{l}(t) = \boldsymbol{H}\boldsymbol{x}(t) \tag{241}$$

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

$$\boldsymbol{h}(t) = \tanh\left(\boldsymbol{U}\boldsymbol{l}(t) + \boldsymbol{W}\boldsymbol{h}(t-1) + \boldsymbol{b}\right) \tag{242}$$

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

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

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)} + \sum_j P_{i,j}^f s_j^{(t-1)}\right) \tag{243}$$

$$s_i^{(t)} = f_i^{(t)} \odot s_i^{(t-1)} + g_i^{(t)} \odot \sigma\left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)}\right) \tag{244}$$

$$g_i^{(t)} = \sigma\left(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)} + \sum_j P_{i,j}^g s_j^{(t-1)}\right) \tag{245}$$

$$q_i^{(t)} = \sigma\left(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)} + \sum_j P_{i,j}^o s_j^{(t)}\right) \tag{246}$$

The final hidden state can then be computed via

$$h_i^{(t)} = \tanh(s_i^{(t)}) \odot q_i^{(t)} \tag{247}$$

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

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

$$L(\mathbf{\Lambda}) = \min_{\mathbf{\Lambda}} \left\{ -\log \prod_{r=1}^{N} \Pr\left[D_r^+ \mid Q_r\right] \right\} = \min_{\mathbf{\Lambda}} \sum_{r=1}^{N} l_r(\mathbf{\Lambda}) \tag{248}$$

$$l_r(\mathbf{\Lambda}) = \log\left(1 + \sum_{j=1}^{n} e^{-\gamma \cdot \Delta_{r,j}}\right) \tag{249}$$

where

- N is the number of (query, clicked-doc) pairs in the corpus, while n is the number of negative samples used during training.
- D_r^+ is the clicked document for rth query.
- $\Delta_{r,j} = R(Q_r, D_r^+) - R(Q_r, D_{r,j}^-)$ (R is just cosine similarity)[76].
- $\mathbf{\Lambda}$ is all the parameter matrices (and biases) in the LSTM.

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

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

Clustering Massive Text Streams

Written by Brandon McKinzie

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

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

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

$$t_0 \quad \text{s.t.} \quad f(t_0) = \frac{1}{2} f(0) \tag{250}$$

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

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

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

$$\mathcal{D}(t, \mathcal{C}) \triangleq (D\bar{F}2, D\bar{F}1, n, w(t), l) \tag{251}$$

  where
  - Entry k of the vector $D\bar{F}2$ is the (weighted) number of points in cluster $\mathcal{C}$ where the ith dimension had value x and the j dimension had value y. In other words, all pairwise combinations of values in the categorical vector[77]. $\sum_{i=1}^{d} \sum_{j \neq i}^{d} v_i v_j$ entries total[78].
  - Similarly, $D\bar{F}1$ consists of the (weighted) counts that some dimension i took on the value x. $\sum_{i=1}^{d} v_i$ entries total.

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

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

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

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

$$\mathcal{D}(t, \mathcal{C}) \triangleq (D\bar{F}2, D\bar{F}1, n, w(t), l) \tag{252}$$

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

Cluster Droplet Maintenance.

1. We first start of with k trivial clusters (the first k data points that arrived).
2. When a new point $\bar{X}$ arrives, the cosine similarity to each cluster's $D\bar{F}1$ is computed.
3. $\bar{X}$ is inserted into the cluster for which this is a maximum, so long as the associated $S(\bar{X}, D\bar{F}1) >$ thresh, a predefined threshold. If not above the threshold *and* some **inactive cluster** exists, a new cluster is created containing the solitary point $\bar{X}$, which replaces the inactive cluster. If not above threshold but no inactive clusters, then we just insert it into the max similarity cluster anyway.
4. If $\bar{X}$ was inserted (i.e. didn't replace an inactive cluster), then we need to:
   (a) Update the statistics to reflect the decay of the data points at the current moment in time[79]. This is done by multiplying entries in the droplet vectors by $2^{-\lambda \cdot (t-l)}$.
   (b) Add the statistics for each newly arriving data point to the cluster statistics.

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

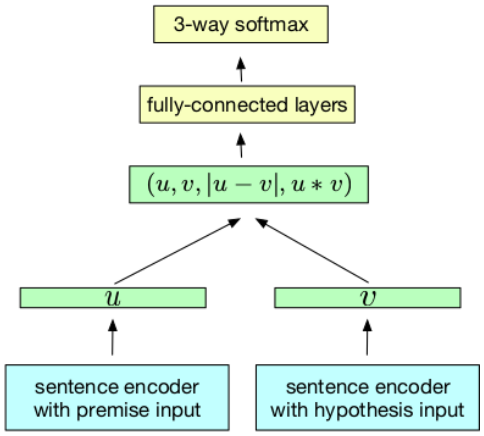
Supervised Universal Sentence Representations (InferSent)

Written by Brandon McKinzie

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

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

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



Note that the same sentence encoder is used for both u and v. To obtain a sentence vector from a BiLSTM encoder, they experiment with (1) the average h_t over all t (**mean pooling**), and (2) selecting the max value over each dimension of the hidden units [over all timesteps] (**max pooling**)[80].

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

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

 Written by Brandon McKinzie

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

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

Authors propose two new phrase/sentence representation learning objectives:

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

Distributed Sentence Representations. Existing models trained on text:

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

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

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

Novel Text-Based Methods.

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

     We then train the same LSTM-based encoder-decoder architecture as NMT, but with the denoising objective to predict (as target) the original source sentence S given a corrupted version $N(S \mid p_o, p_x)$ (as source).

Authors
$p_o = p_x$

- **FastSent**. Designed to be a more efficient/quicker to train version of SkipThought.

Latent Dirichlet Allocation

Written by Brandon McKinzie

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

Introduction. At minimum, one should be familiar with generative probabilistic models, mixture models, and the notion of latent variables before continuing. The "Dirichlet" in LDA of course refers to the **Dirichlet distribution**, which is a generalization of the beta distribution, B. It's PDF is defined as[81][82]:

$$\text{Dir}(\boldsymbol{x};\boldsymbol{\alpha}) = \frac{1}{B(\boldsymbol{\alpha})}\prod_{i=1}^{K}\boldsymbol{x}_i^{\boldsymbol{\alpha}_i-1} \quad \text{where} \quad B(\boldsymbol{\alpha}) = \frac{\prod_{i=1}^{K}\Gamma\left(\boldsymbol{\alpha}_i\right)}{\Gamma(\sum_{i=1}^{K}\boldsymbol{\alpha}_i)} \tag{253}$$

Main things to remember about LDA:

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

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

- **TF-IDF**. Design matrix $\boldsymbol{X} \in \mathbb{R}^{V\times M}$, where M is the number of docs, and $\boldsymbol{X}_{i,j}$ gives the TF-IDF value for ith word in vocabulary and corresp. to document j.
- **LSI**:[83] Performs SVD on the TF-IDF design matrix $\boldsymbol{X}$ to identify a linear subspace in the space of tf-idf features that captures most of the variance in the collection.
- **pLSI**: **TODO**

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

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

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

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

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

1. $N \sim \textbf{Poisson}(\lambda)$: Sample N, the number of words (length of $\boldsymbol{w}$), from Poisson(λ) = $e^{-\lambda}\frac{\lambda^n}{n!}$. The parameter λ *should* represent the average number of words per document.
2. $\theta \sim \textbf{Dir}(\alpha)$: Sample k-dimensional vector θ from the Dirichlet distribution (eq. 253), Dir(α). k is the number of topics (pre-defined by us). Recall that this means θ lies in the (k-1) simplex. The Dirichlet distribution thus tells us the probability density of θ over this simplex – it defines the probability of θ being at a given position on the simplex.
3. Do the following N times to generate the words for this document.
   (a) $z_n \sim \textbf{Multinomial}(\theta)$. Sample a topic z_n.
   (b) $w_n \sim \textbf{Pr}\left[w_n \mid z_n, \beta\right]$: Sample a word w_n from $\Pr\left[w_n \mid z_n, \beta\right]$, a "multinomial probability conditioned on topic z_n."[84] The parameter β gives the distribution of words given a topic:

$$\beta_{ij} = \Pr\left[w_j \mid z_i\right] \tag{254}$$

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

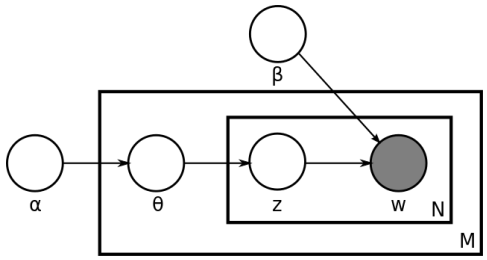
The defining equations for LDA are thus:

$$\Pr\left[\theta, \boldsymbol{z}, \boldsymbol{w} \mid \alpha, \beta\right] = \textbf{Pr}\left[\theta \mid \alpha\right] \prod_{n=1}^{N} \textbf{Pr}\left[z_n \mid \theta\right] \textbf{Pr}\left[w_n \mid z_n, \beta\right] \tag{255}$$

$$\Pr\left[\boldsymbol{w} \mid \alpha, \beta\right] = \int \textbf{Pr}\left[\theta' \mid \alpha\right] \left(\prod_{n=1}^{N} \sum_{z'_n} \textbf{Pr}\left[z'_n \mid \theta'\right] \textbf{Pr}\left[w_n \mid z'_n, \beta\right] \right) \mathrm{d}\theta' \tag{256}$$

$$\Pr\left[\mathcal{D} = \{\boldsymbol{w}^{(1)}, \ldots, \boldsymbol{w}^{(M)}\} \mid \alpha, \beta\right] = \prod_{d=1}^{M} \Pr\left[\boldsymbol{w}^{(d)} \mid \alpha, \beta\right] \tag{257}$$

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



- **Outermost Variables**: α and β. Both represent a (Dirichlet) prior distribution: α parameterizes the probability of a given *topic*, while β a given *word*.
- Document Plate. M is the number of documents, $\boldsymbol{\theta}_m$ gives the true distribution of topics for document m[85].

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

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

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

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

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

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

$$\Pr\left[\boldsymbol{\theta}, \boldsymbol{z} \mid \boldsymbol{w}, \boldsymbol{\alpha}, \boldsymbol{\beta}\right] = \frac{\Pr\left[\boldsymbol{\theta}, \boldsymbol{z}, \boldsymbol{w} \mid \boldsymbol{\alpha}, \boldsymbol{\beta}\right]}{\Pr\left[\boldsymbol{w} \mid \boldsymbol{\alpha}, \boldsymbol{\beta}\right]} \tag{258}$$

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

Conditional Random Fields

Written by Brandon McKinzie

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

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

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

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

Conditional Random Fields. Here we formalize the model and notation. Let $\mathbf{X}$ be a random variable over data sequences to be labeled (e.g. over all words/sentences), and let $\mathbf{Y}$ the random variable over corresponding label sequences[86]. Formal definition:

> *Let $G = (V, E)$ be a graph such that $Y = (Y_v)_{v \in V}$, so that Y is indexed by the vertices of G. Then (X, Y) is a CRF if, when conditioned on X, the random variables Y_v obey the Markov property with respect to the graph:*
>
> $$Pr\left[Y_v | X, Y_w, w \neq v\right] = Pr\left[Y_v | X, Y_w, w \sim v\right] \tag{259}$$
>
> *where $w \sim v$ means that w and v are neighbors in G.*

All this means is a CRF is a random field (discrete set of random-valued points in a space) where all points (i.e. globally) are conditioned on $\mathbf{X}$. If the graph $G = (V, E)$ of $\mathbf{Y}$ is a tree, its cliques[87] are the edges and vertices. Take note that $\mathbf{X}$ is not a member of the vertices

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

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

in G. G only contains vertices corresponding to elements of $\mathbf{Y}$. Accordingly, when the authors refer to cases where G is a "chain", remember that they just mean the $\mathbf{Y}$ vertex sequence.

By the fundamental theorem of random fields:

$$p_\theta(\mathbf{y} \mid \mathbf{x}) \propto \exp\left(\sum_{e \in E,k} \lambda_k f_k(e, \mathbf{y}|_e, \mathbf{x}) + \sum_{v \in V,k} \mu_k g_k(v, \mathbf{y}|_v, \mathbf{x})\right) \tag{260}$$

where $\mathbf{y}|_S$ is the set of components of $\mathbf{y}$ associated with the vertices in subgraph S. We assume the K feature [functions] f_k and g_k are given and fixed. Note that f_k are the feature functions over *transitions* y_{t-1} to y_t, and g_k are the feature functions over *states* y_t and x_t. Our estimation problem is thus to determine parameters $\theta = (\lambda_1, \lambda_2, \ldots; \mu_1, \mu_2, \ldots)$ from the labeled training data.

Linear-Chain CRF. Let $|\mathcal{Y}|$ denote the number of possible labels. At each position t in the observation sequence $\boldsymbol{x}$, we define the $|\mathcal{Y}| \times |\mathcal{Y}|$ matrix random variable $\boldsymbol{M}_t(\boldsymbol{x})$

$$\boldsymbol{M}_t(y', y, \mid \boldsymbol{x}) = \exp(\boldsymbol{\Lambda}_t(y', y \mid x)) \tag{261}$$

$$\boldsymbol{\Lambda}_t(y', y \mid x) = \sum_k \lambda_k f_k(y', y, \boldsymbol{x}) + \sum_k \mu_k g_k(y, \boldsymbol{x}) \tag{262}$$

where $y_{t-1} := y'$ and $y_t := y$. We can see that the individual elements correspond to specific values of e and v in the double-summations of $p_\theta(\boldsymbol{y} \mid \boldsymbol{x})$ above. Then the normalization (partition function) $Z_\theta(\boldsymbol{x})$ is the (y_0, y_{T+1}) entry (the fixed boundary states) of the product:

$$Z_\theta(\boldsymbol{x}) = \left[\prod_{t=1}^{T+1} \boldsymbol{M}_t(\boldsymbol{x})\right]_{y_0, y_{T+1}} \tag{263}$$

which includes all possible sequences $\boldsymbol{y}$ that start with the fixed y_0 and end with the fixed y_{T+1}. Now we can write the conditional probability as a function of just these matrices:

$$p_\theta(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{\prod_{t=1}^{T+1} \boldsymbol{M}_t(y_{t-1}, y_t \mid \boldsymbol{x})}{\left[\prod_{t=1}^{T+1} \boldsymbol{M}_t(\boldsymbol{x})\right]_{y_0, y_{T+1}}} \tag{264}$$

Parameter Estimation (for linear-chain CRFs). For each t in $[0, T+1]$, define the **forward vectors** $\alpha_t(\boldsymbol{x})$ with base case $\alpha_0(y \mid \boldsymbol{x}) = 1$ if $y = y_0$, else 0. Similarly, define the **backward vectors** $\beta_t(\boldsymbol{x})$ with base case $\beta_{T+1}(y \mid \boldsymbol{x}) = 1$ if $y = y_{T+1}$ else 0[88]. Their recurrence relations are

$$\alpha_t(\boldsymbol{x}) = \alpha_{t-1}(\boldsymbol{x})\boldsymbol{M}_t(\boldsymbol{x}) \tag{265}$$

$$\beta_t(\boldsymbol{x})^T = \boldsymbol{M}_{t+1}(\boldsymbol{x})\beta_{t+1}(\boldsymbol{x}) \tag{266}$$

[88]Remember that y_0 and y_{T+1} are their own fixed symbolic constants representing a fixed start/stop state.

Attention Is All You Need

Written by Brandon McKinzie

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

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

Model Architecture.

- **Encoder**. N=6 identical layers, each with 2 sublayers: (1) a **multi-head self-attention** mechanism and (2) a position-wise FC feed-forward network. They apply a residual connection and layer norm such that each sublayer, instead of outputting Sublayer(x), instead outputs LayerNorm(x + Sublayer(x)).
- **Decoder**. N=6 with 3 sublayers each. In addition to the two sublayers described for the encoder, the decoder has a third sublayer, which performs **multi-head** attention over the output of the encoder stack. Same residual connections and layer norm.

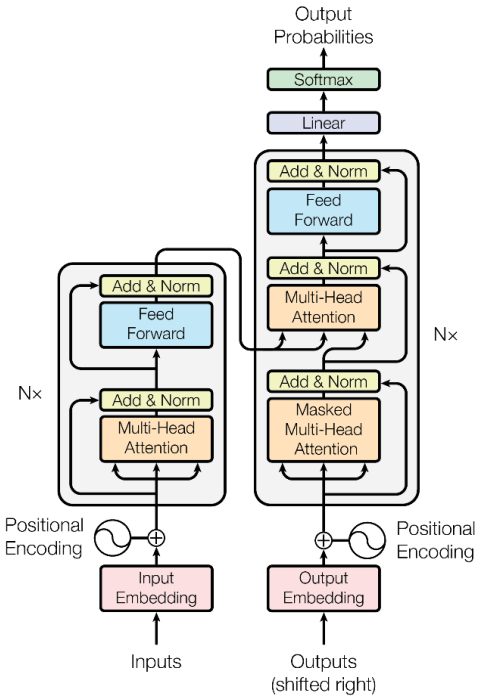


Figure shows encoder-decoder template layers. The actual model instantiates chain of 6 encoder layers and 6 decoders layers. The decoder's self-attention masks embeddings at future timesteps to zero.

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

$$\text{Attn}(\text{query}, \{(k_1, v_1), \ldots, \}) \Rightarrow \sum_i fn(\text{query}, k_i) v_i \tag{267}$$

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

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

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

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V \tag{268}$$

  Note that this is identical to the standard dot-product attention mechanism, except for the *scaling* factor (hence the name) of $1/\sqrt{d_k}$. The scaling factor is motivated by the fact that additive attention outperforms dot-product attention for large d_k and the authors stipulate this is due to the softmax having small gradients in this case, due to the large dot products[89].

  First, let's explicitly show which indices are being normalized over, since it can get confusing when presented with the highly vectorized version above. For a given input sequence of length T, and for the self-attention version where $K{=}Q{=}V \in \mathbb{R}^{T \times d_k}$, the output attention vector for timestep t is explicitly (ignoring the $\sqrt{d_k}$ for simplicity)

$$\text{Attention}(Q, K, V)_t = \left[\text{softmax}\left(QK^T\right) V\right]_t \tag{272}$$

$$= \sum_{t'}^{T} \frac{e^{Q_t \cdot K_{t'}}}{\sum_{t''}^{T} e^{Q_t \cdot K_{t''}}} V_{t'} \tag{273}$$

[89] Assume that $\boldsymbol{q}$ and $\boldsymbol{k}$ are vectors in $\mathbb{R}^d$ whose components are independent RVs with $\mathbb{E}[q_i] = \mathbb{E}[k_j] = 0$ $(\forall i, j)$, and $\text{Var}[q_i] = \text{Var}[k_j] = 1$ $(\forall i, j)$. Then

$$\mathbb{E}[\boldsymbol{q} \cdot \boldsymbol{k}] = \mathbb{E}\left[\sum_i^d q_i k_i\right] = \sum_i^d \mathbb{E}[q_i k_i] = \sum_i^d \mathbb{E}[q_i]\,\mathbb{E}[k_i] = 0 \tag{269}$$

$$\text{Var}[\boldsymbol{q} \cdot \boldsymbol{k}] = \text{Var}\left[\sum_i^d q_i k_i\right] = \sum_i^d \text{Var}[q_i k_i] = \sum_i^d \mathbb{E}\left[q_i^2 k_i^2\right] - \cancel{\mathbb{E}[q_i k_i]} \tag{270}$$

$$= \sum_i^d \mathbb{E}\left[q_i^2\right] \mathbb{E}\left[k_i^2\right] = \sum_i^d \text{Var}[q_i]\,\text{Var}[k_i] = d \tag{271}$$

See this S.O answer and/or these useful formulas for more details.

Next, the gradient of the dth softmax output w.r.t its inputs is

$$\frac{\partial \text{Softmax}_d(\boldsymbol{x})}{\partial x_j} = \text{Softmax}_d(\boldsymbol{x}) \left(\delta_{dj} - \text{Softmax}_d(\boldsymbol{x})\right) \tag{274}$$

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

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h) W^O \tag{275}$$

$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{276}$$

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

The Transformer uses multi-headed attention in 3 ways:

1. **Encoder-decoder attention**: the normal kind. Queries are previous decoder layer, and memory keys and values come from output of the [final layer of] encoder.
2. **Encoder self-attention**: all of the keys, values, and queries come from the previous *layer* in the encoder. Each position in the encoder can attend to all positions in the previous layer of the encoder.
3. **Decoder self-attention**: Similarly, self-attention layers in the decoder allow each position in the decoder to attend to all positions in the decoder up to and including that position (timestep). The masking is done on the inputs to the softmax, setting all inputs beyond the current timestep to $-\infty$.

Other Components.

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

$$\text{FFN}(x) = \max\left(0, xW_1 + b_1\right) W_2 + b_2 \tag{277}$$

- **Embeddings and Softmax**: use learned embeddings to convert input/output tokens to vectors of dimension d_{model}, *and* for the pre-softmax layer at the output of the decoder[90].
- **Positional Encoding**: how the authors deal with the lack of recurrence (to make use of the sequence order). They add a sinusoid function of the position (timestep) *pos* and vector index i to the input embeddings for the encoder and decoder[91]:

$$\text{PE}(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{278}$$

$$\text{PE}(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{279}$$

  The authors justify this choice:

[90] In other words, they use the same weight matrix for all three of (1) encoder input embedding, (2) decoder input embedding, and (3) (opposite direction) from decoder output to pre-softmax.

[91] Note that the positional encodings must necessarily be of dimension d_{model} to be summed with the input embeddings.

> *We chose this function because we hypothesized it would allow the model to easily learn to attend by relative positions, since for any fixed offset k, PE_{pos+k} can be represented as a linear function of PE_{pos}.*

Summary of Add-ons. Below is a list of all the little bells and whistles they add to the main components of the model that are easy to miss since they mention them throughout the paper in a rather unorganized fashion.

- Shared weights for encoder inputs, decoder inputs, and final softmax projection outputs.
- Multiply the encoder and decoder input embedding [shared] weights by $\sqrt{d_{model}}$. **TODO**: why? Also this must be highly correlated with their decision regarding weight initialization (mean/stddev/technique). Add whatever they use here if they mention it.
- Adam optimizer with β_1=0.9, β_2=0.98, ϵ=10^{-9}.
- Learning rate schedule $\text{LR}(s) = d_{model}^{-0.5} \cdot \min\left(s^{-0.5}, s \cdot w^{-1.5}\right)$ for global step s and warmup steps w=4000.
- Dropout on sublayer outputs pre-layernorm-and-residual. Specifically, they *actually* return LayerNorm(x + Dropout(Sublayer(x))). Use $P_{drop} = 0.1$.
- Dropout the summed embeddings+positional-encodings for both encoder and decoder stacks.
- Dropout on softmax outputs. So do Dropout(Softmax(QK))V.
- Label smoothing with $\epsilon_{ls} = 0.1$.

Hierarchical Attention Networks

Written by Brandon McKinzie

Yang et al., "Hierarchical Attention Networks for Document Classification."

Overview. Authors introduce the Hierarchical Attention Network (HAN) that is designed to capture insights regarding (1) the hierarchical structure of documents (words -> sentences -> documents), and (2) the context dependence between words and sentences. The latter is implemented by including two levels of attention mechanisms, one at the word level and one at the sentence level.

Hierarchical Attention Networks. Below is an illustration of the network. The first stage is familiar to sequence to sequence models - a bidirectional encoder for outputting sentence-level representations of a sequence of words. The HAN goes a step further by feeding this another bidirectional encoder for outputting document-level representations for sequences of sentences.

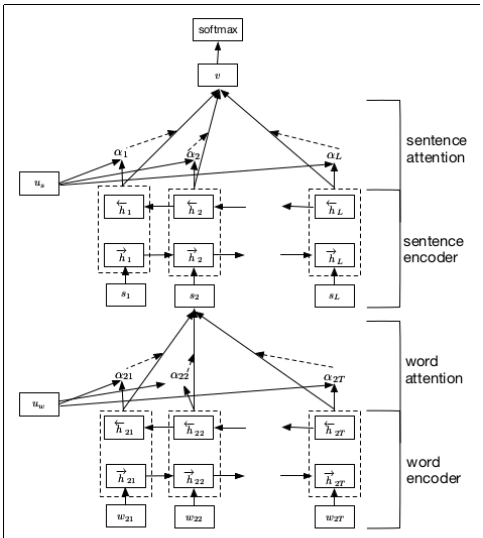


Figure 2: Hierarchical Attention Network.

The authors choose the GRU as their underlying RNN. For ease of reference, the defining equations of the GRU are shown below:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \tag{280}$$

$$z_t = \sigma\left(W_z x_t + U_z h_{t-1} + b_z\right) \tag{281}$$

$$\tilde{h}_t = \tanh\left(W_h x_t + r_t \odot (U_h h_{t-1}) + b_h\right) \tag{282}$$

$$r_t = \sigma\left(W_r x_t + U_r h_{t-1} + b_r\right) \tag{283}$$

Hierarchical Attention. Here I'll overview the main stages of information flow.

1. **Word Encoder**. Let the tth word in the ith sentence be denoted w_{it}. They embed the vectors with a word embedding matrix W_e, $x_{it} = W_e w_{it}$, and then feed x_{it} through a bidirectional GRU to ultimately obtain $h_{it} := [\overrightarrow{h}_{it}; \overleftarrow{h_{it}}]$.
2. **Word Attention**. Extracts words that are important to the meaning of the sentence and aggregates the representation of these informative words to form a sentence vector.

$$u_{it} = \tanh(W_w h_{it} + b_w) \tag{284}$$

$$\alpha_{it} = \frac{\exp(u_{it}^T u_w)}{\sum_t \exp(u_{it}^T u_w)} \tag{285}$$

$$s_i = \sum_t \alpha_{it} h_{it} \tag{286}$$

   Note the *context vector* u_w, which is shared for all words[92] and randomly initialized and jointly learned during the training process.
3. **Sentence Encoder**. Similar to the word encoder, but uses the sentence vectors s_i as the input for the ith sentence in the document. Note that the output of this encoder, h_i contains information from the neighboring sentences too (bidirectional) but focuses on sentence i.
4. **Sentence Attention**. For rewarding sentences that are clues to correctly classify a document. Similar to before, we now use a sentence level context vector u_s to measure the importance of the sentences.

$$u_i = \tanh(W_s h_i + b_s) \tag{287}$$

$$\alpha_i = \frac{\exp(u_i^T u_s)}{\sum_i \exp(u_i^T u_s)} \tag{288}$$

$$v = \sum_t \alpha_i h_i \tag{289}$$

   where v is the document vector that summarizes all the information of sentences in a document.

As usual, we convert v to a normalized probability vector by feeding through a softmax:

$$p = \text{softmax}(W_c v + b_c) \tag{290}$$

[92] To emphasize, there is only a single context vector u_w in the network, period. The subscript just tells us that it is the word-level context vector, to distinguish it from the sentence-level context vector in the later stage.

Configuration and Training. Quick overview of some parameters chosen by the authors:

- **Tokenization**: Stanford CoreNLP. Vocabulary consists of words occurring more than 5 times, all others are replaced with UNK token.
- **Word Embeddings**: train word2vec on the training and validation splits. Dimension of 200.
- **GRU**. Dimension of 50 (so 100 because bidirectional).
- **Context vectors**. Both u_w and u_s have dimension of 100.
- **Training**: batch size of 64, grouping documents of similar length into a batch. SGD with momentum of 0.9.

Joint Event Extraction via RNNs

Written by Brandon McKinzie

Nguyen, Cho, and Grishman, "Joint Event Extraction via Recurrent Neural Networks," (2016).

Event Extraction Task. Automatic Context Extraction (ACE) evaluation. Terminology:

- **Event**: something that happens or leads to some change of state.
- **Mention**: phrase or sentence in which an event occurs, including one trigger and an arbitrary number of arguments.
- **Trigger**: main word that most clearly expresses an event occurrence.
- **Argument**: an entity mention, temporal expression, or value that serves as a participant/attribute with a specific role in an event mention.

Example:

> In Baghdad, a **cameraman** **died**{*Die*} when an American tank **fired**{*Attack*} on the Palestine hotel.

TRIGGER
ARGUMEN

Each event subtype has its own set of roles to be filled by the event arguments. For example, the roles for the *Die* event subtype include *Place*, *Victim*, and *Time*.

Model.

- **Sentence Encoding**. Let w_i denote the ith token in a sentence. It is transformed into a real-valued vector x_i, defined as

$$x_i := [\text{GloVe}(w_i);\ \text{Embed}(\text{EntityType}(w_i));\ \text{DepVec}(w_i)] \tag{291}$$

  where "Embed" is an embedding we learn, and "DepVec" is the binary vector whose dimensions correspond to the possible relations between words in the dependency trees.
- **RNN**. Bidirectional LSTM on the inputs x_i.
- **Prediction**. Binary memory vector G_i^{trg} for triggers; binary memory matrices G_i^{arg} and $G_i^{arg/trg}$ for arguments (at each timestep i). At each time step i, do the following in order:
  1. Predict trigger t_i for w_i. First compute the feature representation vector R_i^{trig}, defined as:

$$R_i^{trig} := \left[h_i;\ L_i^{trg};\ G_{i-1}^{trg}\right] \tag{292}$$

     where h_i is the RNN output, L_i^{trg} is the local context vector for w_i, and G_{i-1}^{trg} is the memory vector from the previous step. $L_i^{trg} := [\text{GloVe}(w_{i-d}); \ldots; \text{GloVe}(w_{i+d})]$ for

some predefined window size d. This is then fed to a fully-connected layer with softmax activation, $\boldsymbol{F}^{trg}$, to compute the probability over possible trigger subtypes:

$$P_{i;t}^{trg} := F_t^{trg}(R_i^{trg}) \tag{293}$$

As usual, the predicted trigger type for w_i is computed as $t_i = \arg\max_t \left(P_{i;t}^{trg}\right)$. If w_i is not a trigger, t_i should predict "*Other*."

2. Argument role predictions, $a_{i1}, \ldots, a_{ik}$, for all of the [already known] entity mentions in the sentence, $e_1, \ldots, e_k$ with respect to w_i. a_{ij} denotes the argument role of e_j with respect to [the predicted trigger of] w_i. If NOT(w_i is trigger AND e_j is one of its arguments), then a_{ij} is set to *Other*. For example, if w_i was the word "died" from our example sentence, we'd hope that its predicted trigger would be $t_i = Die$, and that **the entity associated with "cameraman" would get a predicted argument role of *Victim*.**

```
def getArgumentRoles(triggerType=t, entities=e):
   k = len(e)
   if isOther(t):
      return [Other] * k
   else:
      for e_j in e:
```

$$R_{ij}^{arg} := \left[h_i;\ h_{i_j};\ L_{ij}^{arg};\ B_{ij};\ G_{i-1}^{arg}[j]; G_{i-1}^{arg/trg}[j]\right] \tag{294}$$

3. Update memory. TO BE CONTINUED...(moving onto another paper because this model is getting a *bit* too contrived for my tastes. Also not a fan of the reliance on a dependency parse.)

Event Extraction via Bidi-LSTM Tensor NNs

 Written by Brandon McKinzie

Y. Chen, S. Liu, S. He, K. Liu, and J. Zhao, "Event Extraction via Bidirectional Long Short-Term Memory Tensor Neural Networks."

Overview. The task/goal is the event extraction task as defined in *Automatic Content Extraction* (ACE). Specifically, given a text document, our goal is to do the following in order *for each sentence*:

1. Identify any event triggers in the sentence.
2. If triggers found, predict their subtype. For example, given the trigger "fired," we may classify it as having the *Attack* subtype.
3. If triggers found, identify their candidate argument(s). ACE defines an event argument as "an entity mention, temporal expression, or value that is involved in an event."
4. For each candidate argument, predict its role: "the relationship between an argument to the event in which it participates."

Context-aware Word Representation. Use pre-trained word embeddings for the input word tokens, the predicted trigger, and the candidate argument. Note: *we assume we already have predictions for the event trigger t and are doing a pass for one of (possibly many) candidate arguments a.*

1. Embed each word in the sentence with pre-trained embeddings. Denote the embedding for ith word as $e(w_i)$.
2. Feed each $e(w_i)$ through a bidirectional LSTM. Denote the ith output of the forward LSTM as $c_l(w_{i+1})$ and the output of the backward LSTM at the same time step as $c_r(w_{i-1})$. As usual, they take the general functional form:

$$c_l(w_i) = \overrightarrow{LSTM}\left(c_l(w_{i-1}), e(w_{i-1})\right) \tag{295}$$

$$c_r(w_i) = \overleftarrow{LSTM}\left(c_r(w_{i+1}), e(w_{i+1})\right) \tag{296}$$

$$\tag{297}$$

3. Concatenate $e(w_i)$, $c_l(w_i)$, $c_r(w_i)$ together along with the embedding of the candidate argument $e(a)$ and predicted trigger $e(t)$. Also include the relative distance of w_i to t or (??) a, denoted as pi for position information, and the embedding of the predicted event type pe of the trigger. Denote this massive concatenation result as x_i:

$$x_i := c_l(w_i) \oplus e(w_i) \oplus c_r(w_i) \oplus pi \oplus pe \oplus e(a) \oplus e(t) \tag{298}$$

Dynamic Multi-Pooling. This is easiest shown by example. Continue with our example sentence:

In California, **Peterson** was arrested for the **murder** of his wife and unborn son.

where the colors are given for *this specific case where murder is our predicted trigger and we are considering the candidate argument Peterson*[93]. Given our n outputs from the previous stage, $y^{(1)} \in \mathbb{R}^{n \times m}$, where n is the length of the sentence and m is the size of that huge concatenation given in equation 298. We split our sentence by trigger and candidate argument, then (confusingly) redefine our notation as

$$y_{1j}^{(1)} \leftarrow \begin{bmatrix} y_{1j}^{(1)} & y_{2j}^{(1)} \end{bmatrix} \tag{299}$$

$$y_{2j}^{(1)} \leftarrow \begin{bmatrix} y_{3j}^{(1)} & \cdots & y_{7j}^{(1)} \end{bmatrix} \tag{300}$$

$$y_{3j}^{(1)} \leftarrow \begin{bmatrix} y_{8j}^{(1)} & \cdots & y_{nj}^{(1)} \end{bmatrix} \tag{301}$$

where it's important to see that, for some $1 \leq j \leq m$, each new $y_{ij}^{(1)}$ is a *vector* of length equal to the number of words in segment i. Finally, the dynamic multi-pooling layer, $y^{(2)}$, can be expressed as

$$y_{i,j}^{(2)} := \max\left(y_{i,j}^{(1)}\right) \qquad 1 \leq i \leq 3,\ 1 \leq j \leq m \tag{302}$$

where the max is taken over each of the aforementioned vectors, leaving us with $3m$ values total. These are concatenated to form $y^{(2)} \in \mathbb{R}^{3m}$.

Output. To predict of each argument role [for the given argument candidate], $y^{(2)}$ is fed through a dense softmax layer,

$$O = W_2 y^{(2)} + b_2 \tag{303}$$

where $W_2 \in \mathbb{R}^{n_1 \times 3m}$ and n_1 is the number of possible argument roles (including "None"). The authors also use dropout on $y^{(2)}$.

[93] Yes, arrested could be another predicted trigger, but the network considers each possibility at separate times/locations in the architecture.

Reasoning with Neural Tensor Networks

Written by Brandon McKinzie

Socher et al., "Reasoning with Neural Tensor Networks for Knowledge Base Completion"

Overview. Reasoning over relationships between two entities. Goal: predict the likely truth of additional facts based on existing facts in the KB. This paper contributes (1) the new NTN and (2) a new way to represent entities in KBs. Each relation is associated with a distinct model. Inputs to a given relation's model are pairs of database entities, and the outputs score how likely the pair has the relationship.

Neural Tensor Networks for Relation Classification. Let $e_1, e_2 \in \mathbb{R}^d$ be the vector representations of the two entities, and let R denote the relation (and thus model) of interest. The NTN computes a score of how likely it is that e_1 and e_2 are related by R via:

$$g(e_1, R, e_2) = u_R^T \tanh\left(e_1^T W_R^{[1:k]} e_2 + V_R \begin{bmatrix} e_1 \\ e_2 \end{bmatrix} + b_R\right) \tag{304}$$

where the bilinear tensor product $e_1^T W_R^{[1:k]} e_2$ results in a vector $h \in \mathbb{R}^k$ with each entry computed by one slice $i = 1, \ldots, k$ of the tensor.

> *Intuitively, we can see each slice of the tensor as being responsible for one type of entity pair or instantiation of a relation... Another way to interpret each tensor slice is that it mediates the relationship between the two entity vectors differently.*

Training Objective and Derivatives. All models are trained with **contrastive max-margin objective functions** and minimize the following objective:

$$J(\mathbf{\Omega}) = \sum_{i=1}^{N} \sum_{c=1}^{C} \max\left(0, 1 - g\left(T^{(i)}\right) + g\left(T_c^{(i)}\right)\right) + \lambda ||\mathbf{\Omega}||_2^2 \tag{305}$$

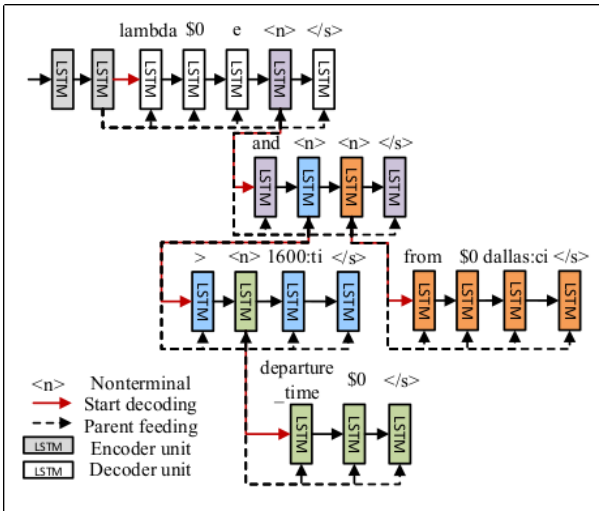
where c is for "corrupted" samples, $T_c^{(i)} := \left(e_1^{(i)}, R^{(i)}, e_c^{(i)}\right)$. Notice that this function is minimized when the difference, $g\left(T^{(i)}\right) - g\left(T_c^{(i)}\right)$, is maximized. The authors used minibatched **L-BFGS** for optimization.

Language to Logical Form with Neural Attention

Written by Brandon McKinzie

Dong and Lapata, "Language to Logical Form with Neural Attention," (2016)

Sequence-to-Tree Model. Variant of Seq2Seq that is more faithful to the compositional nature of meaning representations. It's schematic is shown below. The authors define a "nonterminal" $< n >$ token which indicates [the root of] a subtree.



where the author's have employed "parent-feeding": for a given subtree (logical form), at each timestep, the hidden vector of the parent nonterminal is concatenated with the inputs and fed into the LSTM (best understood via above illustration).

> *After encoding input q, the hierarchical tree decoder generates tokens at depth 1 of the subtree corresponding to parts of logical form a. If the predicted token is $< n >$, decode the sequence by conditioning on the nonterminal's hidden vector. This process terminates when no more nonterminals are emitted.*

Also note that the output posterior probability over the encoded input q is the product of subtree posteriors. For example, consider the decoding example in the figure below:

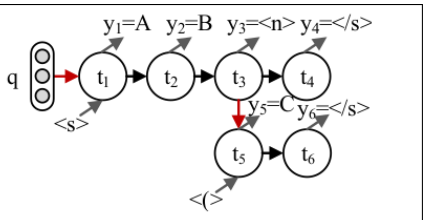


Figure 4: A SEQ2TREE decoding example for the logical form "*A B (C)*".

We would compute the output posterior as:

$$p(a \mid q) = p(y_1 y_2 y_3 y_4 \mid q) p(y_5 y_6 \mid y_{\leq 3}, q) \tag{306}$$

The model is trained by minimizing log-likelihood over the training data, using RMSProp for optimization. At inference time, greedy search or beam search is used to predict the most probable output sequence.

Seq2SQL: Generating Structured Queries from NL using RL

Written by Brandon McKinzie

Zhong, Xiong, and Socher, "Seq2SQL: Generating Structured Queries From Natural Language Using Reinforcement Learning"

Overview. Deep neural network for translating natural language questions to corresponding SQL queries. Outperforms state-of-the-art semantic parser.

Seq2Tree and Pointer Baseline. Baseline model is the Seq2Tree model from the previous note on Dong & Lapata's (2016) paper. Authors here argue their output space is unnecessarily large, and employ the idea of pointer networks with augmented inputs. The input sequence is the concatenation of (1) the column names, (2) the limited vocabulary of the SQL language such as `SELECT`, `COUNT`, etc., and (3) the question.

$$x := [< col >;\ x_1^c; x_2^c; \dots; x_N^c;\ \ < sql >;\ x^s;\ \ < question >;\ x^q] \tag{307}$$

where we also insert special ("sentinel") tokens to demarcate the boundaries of each section. The pointer network can then produce the SQL query by selecting exclusively from the input. Let g_s denote the sth decoder [hidden] state, and y_s denote the output (index/pointer to input query token).

$$\text{[ptr net]}\ y_s = \arg\max_t \left(\alpha_s^{ptr}\right) \quad \text{where} \quad \alpha_{s,t}^{ptr} = w^{ptr} \cdot \tanh\left(U^{ptr} g_s + V^{ptr} h_t\right) \tag{308}$$

Seq2SQL.

1. **Aggregation Classifier**. Our goal here is to predict which aggregation operation to use out of `COUNT`, `MIN`, `MAX`, `NULL`, etc. This is done by projecting the attention-weighted average of encoder states, κ^{agg}, to $\mathbb{R}^C$ where C denotes the number of unique aforementioned aggregation operations. The sequence of computations is summarized as follows:

$$\alpha_t^{inp} = w^{inp} \cdot h_t^{enc} \tag{309}$$
$$\beta^{inp} = \text{softmax}\left(\alpha^{inp}\right) \tag{310}$$
$$\kappa^{agg} = \sum_t^T \beta_t^{inp} h_t^{enc} \tag{311}$$
$$\alpha^{agg} = W^{agg} \tanh\left(V^{agg}\kappa^{agg} + b^{agg}\right) + c^{agg} \tag{312}$$
$$\beta^{agg} = \text{softmax}\left(\alpha^{agg}\right) \tag{313}$$

where β_i^{agg} gives the probability for the ith aggregation operation. We use cross entropy loss L^{agg} for determining the aggregation operation. Note that this part isn't really a sequence-to-sequence architecture. It's nothing more than an MLP applied to an attention-weighted average of the encoder states.

2. **Get Pointer to Column**. A pointer network is used for identifying which column in the input representation should be used in the query. Recall that $x_{j,t}^c$ denotes the tth word in column j. We use the last encoder state for a given column's LSTM[94] as its representation; T_j denotes the number of words in the jth column.

$$e_j^c = h_{j,T_j}^c \quad \text{where} \quad h_{j,t}^c = \text{LSTM}\left(\text{emb}(x_{j,t}^c), h_{j,t-1}^c\right) \tag{314}$$

To construct a representation for the question, compute another input representation κ^{sel} using the same architecture (but distinct weights) as for κ^{agg}. As usual, we compute the scores for each column j via:

$$\alpha_j^{sel} = W^{sel} \tanh\left(V^{sel}\kappa^{sel} + V^c e_j^c\right) \tag{315}$$

$$\beta^{sel} = \text{softmax}\left(\alpha^{sel}\right) \tag{316}$$

Similar to the aggregation, we train the `SELECT` network using cross entropy loss L^{sel}.

3. **WHERE Clause Pointer Decoder**. Recall from equation 308 that this is a model with recurrent connections from its *outputs leading back into its inputs*, and thus a common approach is to train it with **teacher forcing**[95]. However, since the boolean expressions within a `WHERE` clause can be swapped around while still yielding the same SQL query, reinforcement learning (instead of cross entropy) is used to *learn a policy to directly optimize the expected correctness of the execution result.* Note that this also implies that we will be sampling from the output distribution at decoding step s to obtain the next input for $s+1$ [instead of teacher forcing].

[94] Yes, we encode each column with an LSTM separately.

[95] Teacher forcing is just a name for how we train the decoder portion of a sequence-to-sequence model, wherein we feed the ground-truth output $y^{(t)}$ as input at time $t+1$ during training.

$$R\left(q(y), q_g\right) = \begin{cases} -2 & \text{if } q(y) \text{ is not a valid SQL query} \\ -1 & \text{if } q(y) \text{ is a valid SQL query and executes to an incorrect result} \\ +1 & \text{if } q(y) \text{ is a valid SQL query and executes to the correct result} \end{cases} \tag{317}$$

$$L^{whe} = -\mathbb{E}_y \left[R\left(q(y), q_g\right)\right] \tag{318}$$

$$\nabla L_\Theta^{whe} = -\nabla_\Theta \left(\mathbb{E}_{y \sim p_y}\left[R\left(q(y), q_g\right)\right]\right) \tag{319}$$

$$= -\mathbb{E}_{y \sim p_y}\left[R\left(q(y), q_g\right) \nabla_\Theta \sum_t \log p_y(y_t; \Theta)\right] \tag{320}$$

$$\approx -R\left(q(y), q_g\right) \nabla_\Theta \sum_t \log p_y(y_t; \Theta) \tag{321}$$

where

$\rightarrow$ $y = [y^1, y^2, \ldots, y^T]$ denotes the sequences of generated tokens in the `WHERE` clause.
$\rightarrow$ $q(y)$ denotes the query generated by the model.
$\rightarrow$ q_g denotes the ground truth query corresponding to the question.

and the gradient has been approximated in the last line using a single Monte-Carlo sample y.

Finally, the model is trained using gradient descent to minimize $L = L^{agg} + L^{sel} + L^{whe}$.

Speculations for Event Extraction. I want to explore using this paper's model for the task of event extraction. Below, I've replaced some words (shown in green) from a sentence in the paper in order to formalize this as event extraction.

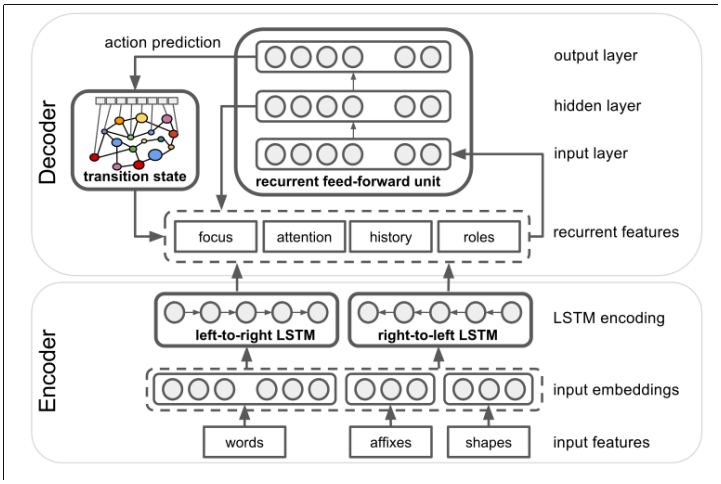
> *Seq2**Event** takes as input a **sentence** and the **possible event types** of an **ontology**. It generates the corresponding **event annotation**, which, during training, is **compared** against an **event template**. The result of the **comparison** is utilized to train the reinforcement learning algorithm*[96].

[96] Original: Seq2SQL takes as input a question and the columns of a table. It generates the corresponding SQL query, which, during training, is executed against a database. The result of the execution is utilized as the reward to train the reinforcement learning algorithm.

SLING: A Framework for Frame Semantic Parsing

Written by Brandon McKinzie

M. Ringgaard, R. Gupta, F. Pereira, "SLING: A framework for frame semantic parsing" (2017)



Model.

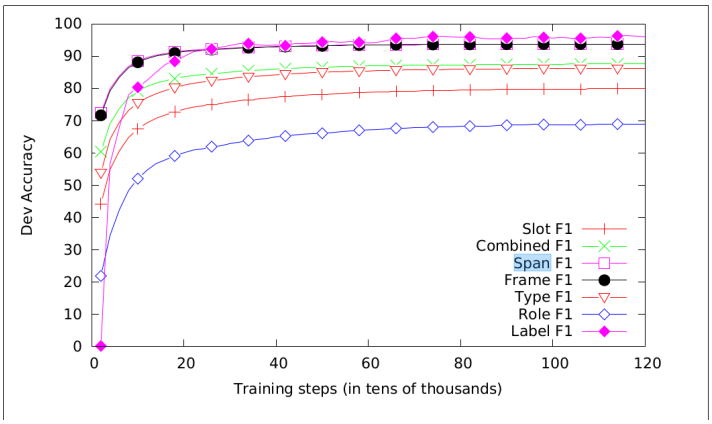
- **Inputs**. [words; affixes; shapes]
- **Encoder**.
  1. Embed.
  2. Bidirectional LSTM.
- **Inputs to TBRU**.
  - BLSTM [forward and backward] hidden state for the current token in the parser state.
  - **Focus**. Hidden layer activations corresponding to the transition steps that evoked/brought into *focus* the top-k frames in the attention buffer.
  - **Attention**. Recall that we maintain an **attention buffer**: an ordered list of frames, where the order represents closeness to center of attention. The attention portion of inputs for the TBRU looks at the top-k frames in the attention buffer, finds the phrases in the text (if any) that evoked them. The activations from the BLSTM for the last token of each of those phrases are included as TBRU inputs[97]
  - **History**. Hidden layer activations from the previous k steps.
  - **Roles**. Embeddings of (s_i, r_i, t_i), where the frame at position s_i in the attention buffer has a role (key) r_i with frame at position t_i as its value. Back-off features are added for the source roles (s_i, r_i), target role (r_i, t_i), and unlabeled roles (s_i, t_i).
- **Decoder** (TBRU). Outputs a softmax over possible transitions (actions).

[97] Okay, how is this attention at all? Seems misleading to call it attention.

Transition System. Below is the list of possible actions. Note that, since the system is trained to predict the correct *frame graph* result, it isn't directly told what order it should take a given set of actions[98].

- **SHIFT**. Move to next input token.
- **STOP**. Signal that we've reached end of parse.
- **EVOKE(type, num)**. New frame of **type** from next **num** tokens in the input; placed at front of attention buffer.
- **REFER(frame, num)**. New mention from next **num** tokens, evoking existing **frame** from attention buffer. Places at front.
- **CONNECT(source-frame, role, target-frame)**. Inserts **(role, target-frame)** slot into **source-frame**, move **source-frame** to front.
- **ASSIGN(source-frame, role, value)**. Same as CONNECT, but with primitive/constant **value**.
- **EMBED(target-frame, role, type)**. New frame of **type**, and inserts **(role, target-frame)** slot. New frame placed to front.
- **ELABORATE(source-frame, role, type)**. New frame of **type**. Inserts **(role, new-frame)** slot to **source-frame**. New frame placed at front.

Evaluation. Need some way of comparing an annotated document with its gold-standard annotation. This is done by constructing a virtual graph where the document is the start node. It is then connected to the spans (which are presumably nodes themselves), and the spans are connected to the frames they evoke. Frames that refer to other frames are given corresponding edges between them. Quality is computed by aligning the golden and predicted graphs and computing precision, recall, and F1. Specifically, these scores are computed separately for spans, frames, frame types, roles linking to other frames (referred to here as just "roles"), and roles that link to global constants (referred to here as just "labels"). Results are shown below.



[98]This is important to keep in mind, since more than one sequence of actions can result in a given predicted frame graph.

Poincaré Embeddings for Learning Hierarchical Representations

Written by Brandon McKinzie

M. Nickel and D. Kiela, "Poincaré Embeddings for Learning Hierarchical Representations" (2017)

Introduction. Dimensionality of embeddings can become prohibitively large when needed for complex data. Authors focus on mitigating this problem for large datasets whose objects can be organized according to a latent hierarchy[99]. They propose to compute embeddings in a particular model of hyperbolic space, the **Poincaré ball model**, claiming it is well-suited for gradient-based optimization (they make use of **Riemannian optimization**).

Prerequisite Math. Recall that a hyperbola is a set of points, such that for any point P of the set, the absolute difference of the distances $|PF_1|$, $|PF_2|$ to two fixed points F_1, F_2 (the foci), is constant, usually denoted by $2a$, $a > 0$. We can define a hyperbola by this set of points or by its canonical form, which are both given, respectively, as:

$$H = \{P| \; ||PF_2| - |PF_1|| = 2a\} \tag{322}$$

$$\frac{x^2}{a^2} - \frac{y^2}{b^2} = 1 \tag{323}$$

where $b^2 := c^2 - a^2$, $(\pm a, 0)$ are the two vertices, and $(\pm c, 0)$ are the two foci. Cannon et al. define n-dimensional hyperbolic space by the formula

$$H^n = \{x \in \mathbb{R}^{n+1} : \; x * x = -1\} \tag{324}$$

where $*$ denotes the non-euclidean inner product (subtracts last term; same as Minkowski sapce-time). Notice that this is the defining equation for a hyperboloid of two sheets, and Cannon et al. says "usually we deal only with one of the two sheets." Hyperbolic spaces are well-suited to model hierarchical data, since both circle length and disc area grow *exponentially* with r.

[99] This begs the question: how useful would a Poincaré embedding be for situations where this assumption isn't valid?

Poincaré Embeddings. Let $\mathcal{B}^d = \{\boldsymbol{x} \in \mathbb{R}^d \mid ||\boldsymbol{x}|| < 1\}$ be the open d-dimensional unit ball. The **Poincaré ball** model of hyperbolic space corresponds then to the Riemannian manifold[100] $(\mathcal{B}^d, g_{\boldsymbol{x}})$, where

$$g_{\boldsymbol{x}} = \left(\frac{2}{1 - ||\boldsymbol{x}||^2}\right)^2 g^E \tag{325}$$

is the **Riemannian metric tensor**, and g^E denotes the Euclidean metric tensor. The distance between two points $\boldsymbol{u}, \boldsymbol{b} \in \mathcal{B}^d$ is given as

$$d(\boldsymbol{u}, \boldsymbol{v}) = \operatorname{arccosh}\left(1 + 2\frac{||\boldsymbol{u} - \boldsymbol{v}||^2}{(1 - ||\boldsymbol{u}||^2)(1 - ||\boldsymbol{v}||^2)}\right) \tag{326}$$

The boundary of the ball corresponds to the sphere S^{d-1} and is denoted by $\partial\mathcal{B}$. Geodesics in $\mathcal{B}^d$ are then circles that are orthogonal to $\partial\mathcal{B}$. To compute Poincaré embeddings for a set of symbols $\mathcal{S} = \{x_i\}_{i=1}^n$, we want to find embeddings $\Theta = \{\boldsymbol{\theta}_i\}_{i=1}^n$, where $\boldsymbol{\theta}_i \in \mathcal{B}^d$. Given some loss function $\mathcal{L}$ that encourages semantically similar objects to be close as defined by the Poincaré distance, our goal is to solve the optimization problem

$$\Theta' \leftarrow \arg\min_{\Theta} \mathcal{L}(\Theta) \qquad s.t.\ \forall \boldsymbol{\theta}_i \in \Theta : ||\boldsymbol{\theta}_i|| < 1 \tag{327}$$

Optimization. Let $\mathcal{T}_\theta\mathcal{B}$ denote the **tangent space** of a point $\boldsymbol{\theta} \in \mathcal{B}^d$. Let $\nabla_R \in \mathcal{T}_{\boldsymbol{\theta}}\mathcal{B}$ denote the **Riemannian gradient** of $\mathcal{L}(\boldsymbol{\theta})$, and ∇_E the Euclidean gradient of $\mathcal{L}(\boldsymbol{\theta})$. Using **RSGD**, parameter updates take the form

$$\boldsymbol{\theta}_{t+1} = \mathfrak{R}_{\boldsymbol{\theta}_t}\left(-\eta_t \nabla_R \mathcal{L}(\boldsymbol{\theta}_t)\right) \tag{328}$$

where $\mathfrak{R}_{\boldsymbol{\theta}_t}$ denotes the **retraction** onto $\mathcal{B}$ at $\boldsymbol{\theta}$ and η_t denotes the learning rate at time t.

[100] All five analytic models of hyperbolic geometry in Cannon et al. are differentiable manifolds with a Riemannian metric. A **Riemannian metric** $\mathrm{d}s^2$ on Euclidean space $\mathbb{R}^n$ is a function that assigns at each point $p \in \mathbb{R}^n$ a positive definite symmetric inner product on the tangent space at p, this inner product varying differentiably with the point p. If $x_1, \ldots, x_n$ are the standard coordinates in $\mathbb{R}^n$, then $\mathrm{d}s^2$ has the form $\sum_{i,j} g_{ij}\mathrm{d}x_i\mathrm{d}x_j$, and the matrix (g_{ij}) depends differentiably on x and is positive definite and symmetric.

Enriching Word Vectors with Subword Information (FastText)

Written by Brandon McKinzie

P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching Word Vectors with Subword Information" (2017)

Overview. Based on the skipgram model, but where each word is represented as a bag of character n-grams. A vector representation is associated *each* character n-gram; words being represented as the *sum*[101] of these representations.

Skipgram with Negative Sampling. Since this is based on skipgram, recall the objective of skipgram which is to maximize:

$$\sum_{t=1}^{T} \sum_{c \in \mathcal{C}_t} \log \Pr\left[w_c \mid w_t\right] \tag{329}$$

for a sequence of words $w_1, \dots w_T$. One way of parameterizing $\Pr\left[w_c \mid w_t\right]$ is by computing a softmax over a scoring function $s : (w_t, w_c) \mapsto \mathbb{R}$,

$$\Pr\left[w_c \mid w_t\right] = \frac{e^{s(w_t, w_c)}}{\sum_{j=1}^{W} e^{s(w_t, j)}} \tag{330}$$

However, this implies that, given w_t, we only predict one context word w_c. Instead, we can frame the problem as a set of independent binary classification tasks, and independently predict the presence/absence of context words. Let $\ell : x \mapsto \log(1 + e^{-x})$ denote the standard logistic negative log-likelihood. Our objective is:

$$\sum_{t=1}^{T} \left[\sum_{c \in \mathcal{C}_t} \ell(s(w_t, w_c)) + \sum_{n \in \mathcal{N}_{t,c}} \ell(-s(w_t, n)) \right] \tag{331}$$

where $\mathcal{N}_{t,c}$ is a set of negative examples sampled from the vocabulary. A common scoring function involves associating a distinct input vector u_w and output vector v_w for each word w. Then the score is computed as $s(w_t, w_c) = \boldsymbol{u}_{w_t}^T \boldsymbol{v}_{w_c}$.

[101] It would be interesting to explore other aggregation operations than just summation.

FastText. Main contribution is a different scoring function s that utilizes subword information. Each word w is represented as a bag of character n-grams. Special symbols $<$ and $>$ delimit word boundaries, and the authors also insert the special sequence containing the full word (with the delimiters) in its bag of n-grams. The word *where* is thus represented by first building its bag of n-grams, for the choice of $n = 3$:

$$\text{where} \longrightarrow \{<wh, \quad whe, \quad her, \quad ere, \quad re>, \quad <where>\} \tag{332}$$

Such a set of n-grams for a word w is denoted $\mathcal{G}_w$. Each n-gram g for a word w has its own vector $\boldsymbol{z}_g$, and the final vector representation of w is the sum of these. The scoring function becomes

$$s(w, c) = \sum_{g \in \mathcal{G}_w} \boldsymbol{z}_g^T \boldsymbol{v}_c \tag{333}$$

DeepWalk: Online Learning of Social Representations

 Written by Brandon McKinzie

B. Perozzi, R. Al-Rfou, and S. Skiena, "DeepWalk: Online Learning of Social Representations," (2014).

Problem Definition. Classifying members of a social network into one or more categories.

> *Let $G = (V, E)$, where V are the members of the network, and E be its edges, $E \subseteq (V \times V)$. Given a partially labeled social network $G_L = (V, E, X, Y)$, with attributes $X \in \mathbb{R}^{|V| \times S}$ where S is the size of the feature space for each attribute vector, and $Y \in \mathbb{R}^{|V| \times |\mathcal{Y}|}$, $\mathcal{Y}$ is the set of labels.*

In other words, the elements of our training dataset, (X, Y), are the members of the social network, and we want to label each member, represented by a vector in $\mathbb{R}^S$, with one or more of the $|\mathcal{Y}|$ labels. We aim to learn features that capture the graph structure *independent* of the labels' distribution, and to do so in an unsupervised fashion.

Learning Social Representations. We want the representations to be adaptable, community-aware, low-dimensional, and continuous. The authors' method learns representations for vertices from a stream of short random walks, optimized with techniques from language modeling.

- **Random Walks**. Denote a random walk rooted at vertex v_i as $\mathcal{W}_{v_i}$, where the kth visited vertex is chosen at random from the neighbors of the $(k-1)^{th}$ visited vertex, and so on. Motivation for their use here is that they're "the foundation of a class of *output sensitive* algorithms which use them to compute local community structure information in time sublinear to the size of the input graph."
- **Language Modeling**. Authors present a generalization of language modeling, which traditionally aims to maximize $\Pr[w_n \mid w_0, \dots, w_{n-1}]$ over all words in a training corpus. The motivation of the generalization is to explore the graph through a stream of short random walks. The walks are thought of as short sentences/phrases in a special language, and we want to estimate the probability of observing vertex v_i given all previous vertices so far in the random walk. Since we want to learn a latent social representation of each vertex, and not simply a probability distribution over node co-occurrences, we condition on the *embeddings* of visited nodes in this latent space (rather than the nodes themselves directly)

$$\Pr[v_i \mid \Phi(v_1), \Phi(v_2), \dots, \Phi(v_{i-1})] \tag{334}$$

  where, in practice, the mapping function Φ is represented by a $|V| \times d$ matrix of free parameters (an embedding matrix). Since this becomes infeasible to compute as the walk length grows, the authors opt for an approach resembling CBOW: minimizing the NLL of vertices in the the context of a given vertex.

$$\min_{\Phi} \quad -\log \Pr[v_{i-w}, \dots, v_{i-1}, v_{i+1}, \dots, v_{i+w} \mid \Phi(v_i)] \tag{335}$$

Remember that, here, v_j is the jth vertex visited in some given random walk.

DeepWalk Algorithm. Below is a conceptual summary of the procedure, followed by a figure/illustration of the formal algorithm definition.

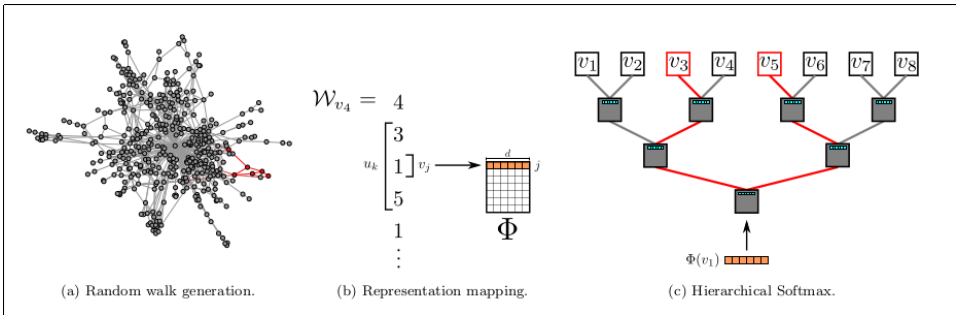
1. **Inputs**. Graph $G(V, E)$, window size w, embedding size d, walks per vertex γ, walk length t.
2. **Random Walk**. For each vertex v_i, compute $\mathcal{W}_{v_i} := RandomWalk(G, v_i, t)$.
3. **Updates**. Upon finishing a walk, $\mathcal{W}_{v_i}$, run skipgram on the sequence of walked vertices to update the embedding matrix Φ.
4. **Outputs**. The embedding matrix $\Phi \in \mathbb{R}^{|V| \times d}$.

Algorithm 1 DeepWalk(G, w, d, γ, t)

Input: graph $G(V, E)$
window size w
embedding size d
walks per vertex γ
walk length t
Output: matrix of vertex representations $\Phi \in \mathbb{R}^{|V| \times d}$

1: Initialization: Sample Φ from $\mathcal{U}^{|V| \times d}$
2: Build a binary Tree T from V
3: **for** $i = 0$ to γ **do**
4: $\quad \mathcal{O} = \text{Shuffle}(V)$
5: $\quad$ **for each** $v_i \in \mathcal{O}$ **do**
6: $\quad\quad \mathcal{W}_{v_i} = RandomWalk(G, v_i, \text{t})$
7: $\quad\quad$ SkipGram(Φ, $\mathcal{W}_{v_i}$, w)
8: $\quad$ **end for**
9: **end for**

where $SkipGram(\Phi, W_{v_i}, w)$ performs SGD updates on Φ to minimize $-\log \Pr\left[u_k \mid \Phi(v_j)\right]$ for each visited v_j, for each u_k in the "context" of v_j. Notice that a binary tree T is build from the set of vertices V (line 2) – this is done as preparation for computing each $\Pr\left[u_k \mid \Phi(v_j)\right]$ via a *hierarchical softmax*, to reduce computational burden of its partition function. Finally, a visual overview of the DeepWalk algorithm is shown below. The authors use this algorithm, com-



(a) Random walk generation. (b) Representation mapping. (c) Hierarchical Softmax.

bined with a one-vs-rest logistic regression implementation by LibLinear, for various multiclass multilabel classification tasks.

Review of Relational Machine Learning for Knowledge Graphs

 Written by Brandon McKinzie

Nickel, Murphy, Tresp, and Gabrilovich, "Review of Relational Machine Learning for Knowledge Graphs," (2015).

Introduction. Paper discusses latent feature models such as tensor factorization and multiway neural networks, and mining observable patterns in the graph. In **Statistical Relational Learning (SRL)**, the representation of an object can contain its relationships to other objects. The main goals of SRL include:

- Prediction of missing edges (relationships between entities).
- Prediction of properties of nodes.
- Clustering nodes based on their connectivity patterns.

We'll be reviewing how SRL techniques can be applied to large-scale **knowledge graphs** (KGs), i.e. graph structured knowledge bases (KBs) that store factual information in the form of relationships between entities.

Probabilistic Knowledge Graphs. Let $\mathcal{E} = \{e_1, \dots, e_{N_e}\}$ be the set of all entities and $\mathcal{R} = \{r_1, \dots, r_{N_r}\}$ be the set of all relation types in a KG. We model each *possible* triple $x_{ijk} = (e_i, r_k, e_j)$ as a binary random variable $y_{ijk} \in \{0, 1\}$ that indicates its existence. The full tensor $\boldsymbol{Y} \in \{0,1\}^{N_e \times N_e \times N_r}$ is called the *adjacency tensor*, where each possible realization of $\boldsymbol{Y}$ can be interpreted as a possible world.

Clearly, $\boldsymbol{Y}$ will be large and sparse in most applications. Ideally, a relational model for large-scale KGs should scale at most linearly with the data size, i.e., linearly in the number of entities N_e, linearly in the number of relations N_r, and linearly in the number of *observed* triples $|\mathcal{D}| = N_d$.

Types of SRL Modesls. The presence or absence of certain triples in relational data is correlated with (i.e. predictive of) the presence or absence of certain other triples. In other words, the random variables y_{ijk} are correlated with each other. There are three main ways to model these correlations:

1. **Latent feature models**: Assume all y_{ijk} are conditionally independent given latent features associated with the subject, object and relation type and additional parameters.
2. **Graph feature models**: Assume all y_{ijk} are conditionally independent given observed graph features and additional parameters.
3. **Markov Random Fields**: Assume all y_{ijk} have local interactions.

The first two model classes predict the existence of a triple x_{ijk} via a score function $f(x_{ijk}; \Theta)$

which represents the model's confidence that a triple exists given the parameters Θ. The conditional independence assumptions can be written as

$$\Pr\left[\boldsymbol{Y} \mid \mathcal{D}, \Theta\right] = \prod_{i=1}^{N_e} \prod_{j=1}^{N_e} \prod_{k=1}^{N_r} \operatorname{Ber}\left(y_{ijk} \mid \sigma\left(f(x_{ijk}; \Theta)\right)\right) \tag{336}$$

where Ber is the Bernoulli distribution[102]. Such models will be referred to as *probabilistic models*. We will also discuss *score-based models*, which optimize $f(\cdot)$ via maximizing the margin between existing and non-existing triples.

Latent Feature Models. We assume the variables y_{ijk} are conditionally independent given a set of global latent features and parameters. All LFMs explain triples (observable facts) via latent features of entities[103]. One task of all LFMs is to infer these [latent] features automatically from the data.

- **RESCAL**: a bilinear model. Models the score of a triple x_{ijk} as

$$f_{ijk}^{RESCAL} := \boldsymbol{e}_i^T \boldsymbol{W}_k \boldsymbol{e}_j = \sum_{a=1}^{H_e} \sum_{b=1}^{H_e} w_{abk} e_{ia} e_{jb} \tag{338}$$

  where the entity vectors $\boldsymbol{e}_i \in \mathbb{R}^{H_e}$ and H_e denotes the number of latent features in the model. The parameters of the model are $\Theta = \{\{\boldsymbol{e}_i\}_{i=1}^{N_e}, \{\boldsymbol{W}_k\}_{k=1}^{N_r}\}$. Note that entities have the same latent representation regardless of whether they occur as subjects or objects in a relationship (shared representation), *thus allowing the model to capture global dependencies in the data*. We can make a connection to **tensor factorization** methods by seeing that the equation above can be written compactly as

$$\boldsymbol{F}_k = \boldsymbol{E} \boldsymbol{W}_k \boldsymbol{E}^T \tag{339}$$

  where $\boldsymbol{F}_k \in \mathbb{R}^{N_e \times N_e}$ is the matrix holding all scores for the k-th relation, and the ith row of $\boldsymbol{E} \in \mathbb{R}^{N_e \times H_e}$ holds the latent representation of $\boldsymbol{e}_i$.

- **Multi-layer perceptrons**. We can rewrite RESCAL as

$$f_{ijk}^{RESCAL} := \boldsymbol{w}_k^T \boldsymbol{\phi}_{i,j}^{RESCAL} \tag{340}$$

$$\boldsymbol{\phi}_{i,j}^{RESCAL} := \boldsymbol{e}_j \otimes \boldsymbol{e}_i \tag{341}$$

  where $\boldsymbol{w}_k = \operatorname{vec}(\boldsymbol{W}_k)$ (vector of size H_e^2 obtained by stacking columns of $\boldsymbol{W}_k$). The

[102] Notation used:

$$\operatorname{Ber}(y \mid p) = \begin{cases} p & \text{if } y = 1 \\ 1 - p & \text{if } y = 0 \end{cases} \tag{337}$$

[103] It appears that "latent" is being used here synonymously with "not directly observed in the data".

authors extend this to what they call the E-MLP (E for entity) model:

$$f_{ijk}^{E-MLP} := \boldsymbol{w}_k^T \boldsymbol{g}(\boldsymbol{h}_{ijk}^a) \tag{342}$$

$$\boldsymbol{h}_{ijk}^a := \boldsymbol{A}_k^T \boldsymbol{\phi}_{ij}^{E-MLP} \tag{343}$$

$$\boldsymbol{\phi}_{ij}^{E-MLP} := [\boldsymbol{e}_i; \boldsymbol{e}_j] \tag{344}$$

Graph Feature Models. Here we assume that the existence of an edge can be predicted by extracting features from the observed edges in the graph. In contrast to LFMs, this kind of reasoning explains triples directly from the observed triples in the KG.

- **Similarity measures for uni-relational data**. Link prediction in graphs that consist only of a single relation (e.g. (Bob, isFriendOf, Sally) for a social network). Various **similarity indices** have been proposed to measure similarity of entities, of which there are three main classes:
  1. **Local** similarity indices: Common Neighbors, Adamic-Adar index, Preferential Attachment derive entity similarities from number of common neighbors.
  2. **Global** similarity indices: Katz index, Leicht-Holme-Newman index (ensembles of all paths bw entities). Hitting Time, Commute Time, PageRank (random walks).
  3. **Quasi-local** similarity indices: Local Katz, Local Random Walks.
- **Path Ranking Algorithm** (**PRA**): extends the idea of using random walks of bounded lengths for predicting links in multi-relational KGs. Let $\pi_L(i,j,k,t)$ denote a path of length L of the form $e_i \xrightarrow{r_1} e_2 \xrightarrow{r_2} e_3 \cdots \xrightarrow{r_L} e_j$, where t represents the sequence of edge types $t = (r_1, r_2, \ldots, r_L)$. We also require there to be a direct arc $e_i \xrightarrow{r_k} e_j$, representing the existence of a relationship of type k from e_i to e_j. Let $\Pi_L(i,j,k)$ represent the set of all such paths of length L, ranging over path types t.

  We can compute the probability of following a given path by assuming that at each step we follow an outgoing link uniformly at random. Let $\Pr[\pi_L(i,j,k,t)]$ be the probability of the path with type t. *The key idea in PRA is to use these path probabilities as features for predicting the probabilities of missing edges.* More precisely, the feature vector and score function (logistic regression) are as follows:

$$\boldsymbol{\phi}_{ijk}^{PRA} = [\Pr[\pi] : \pi \in \Pi_L(i,j,k)] \tag{345}$$

$$f_{ijk}^{PRA} := \boldsymbol{w}_k^T \boldsymbol{\phi}_{ijk}^{PRA} \tag{346}$$

TODO: Finish...

Fast Top-K Search in Knowledge Graphs

Written by Brandon McKinzie

S. Yang, F. Han, Y. Wu, X. Yan, "Fast Top-K Search in Knowledge Graphs."

Introduction. Task: Given a knowledge graph G, scoring function F, and a graph query Q, top-k subgraph search over G returns k answers with the highest matching scores. An example is searching for movie makers (directors) worked with "Brad" and have won awards, illustrated below:

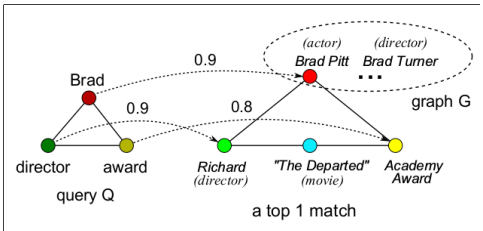


Fig. 1: Top-k subgraph querying

Clearly, it would be extremely inefficient to enumerate all possible matches and then rank them.

Preliminaries/Terminology.

- **Queries**. A query [graph] is defined as $Q = (V_Q, E_Q)$. Each query node in Q provides information/constraints about an entity, and an edge between two nodes specifies the relationship or the connectivity constraint posed on the two nodes. $Q*$ denotes a star-shaped query, which is basically a graph that looks like a star (central node with tree-like structure radially outward).
- **Subgraph Matching**. Given a graph query Q and a knowledge graph G, a match $\phi(Q)$ of Q in G is a subgraph of G, specified by a one-to-one matching function ϕ. It maps each node u (edge $e = (u, u')$) in Q to a node match $\phi(u)$ (edge match $\phi(e) = (\phi(u), \phi(u'))$) in $\phi(Q)$.

The matching score between query Q and its match $\phi(Q)$ is

$$F(\phi(Q)) = \sum_{v \in V_Q} F_V(v, \phi(v)) + \sum_{e \in E_Q} F_E(e, \phi(e)) \tag{347}$$

$$F_V(v, \phi(v)) = \sum_i \alpha_i f_i(v, \phi(v)) \tag{348}$$

$$F_E(e, \phi(e)) = \sum_j \beta_j f_j(e, \phi(e)) \tag{349}$$

Star-Based Top-K Matching.

1. **Query decomposition**: Given query Q, STAR decomposes Q to a set of star queries $\mathcal{Q}$. A star query contains a pivot node and a set of leaves as its neighbors in Q.
2. **Star querying engine**: Generate a set of top matches for each star query $\mathcal{Q}$.
3. **Top-k rank join**. The top matches for multiple star queries are collected and joined to produce top-k complete matches of Q.

Dynamic Recurrent Acyclic Graphical Neural Networks (DRAGNN)

Written by Brandon McKinzie

Kong et al., "DRAGNN: A Transition-based Framework for Dynamically Connected Neural Networks," (2017).

Transition Systems. Define a transition system $\mathcal{T} \triangleq \{\mathcal{S}, \mathcal{A}, t\}$, where

- $\mathcal{S} = \mathcal{S}(x)$ is a set of states, where that set depends on the input sequence x.
- A special start state $s^\dagger \in \mathcal{S}(x)$.
- A set of allowed decisions $\mathcal{A}(s, x)$ $\forall s \in \mathcal{S}(x)$.
- A transition function $t(s, d, x)$ returning a new state s' for any decision $d \in \mathcal{A}(s, x)$.

The authors then define a **complete structure** as a sequence of state/decision pairs $(s_1, d_1) \dots (s_n, d_n)$ such that $s_1 = s^\dagger$, $d_i \in \mathcal{A}(s_i)$ for $i = 1, \dots, n$ and $s_{i+1} = t(s_i, d_i)$, where $n = n(x)$ is the number of decisions for input x[104]. We'll use transition systems to map inputs x into a sequence of output symbols $d_1, \dots, d_n$.

Transition Based Recurrent Networks. When combining transition systems with recurrent networks, we will refer to them as **Transition Based Recurrent Units (TBRU)**, which consist of:

- Transition system $\mathcal{T}$.
- Input function $\boldsymbol{m}(s) : \mathcal{S} \mapsto \mathbb{R}^K$ that maps states to some fixed-size vector representation (e.g. an embedding lookup operation).
- Recurrence function $\boldsymbol{r}(s) : \mathcal{S} \mapsto \mathbb{P}\{1, \dots, i-1\}$ that maps states to a set of previous time steps, where $\mathbb{P}$ is the power set. Note that $|\boldsymbol{r}(s)|$ may vary with s. We use $\boldsymbol{r}$ to specify state-dependent recurrent links in the unrolled computation graph.
- The RNN cell $\boldsymbol{h}_s \leftarrow \mathbf{RNN}(\boldsymbol{m}(s), \{\boldsymbol{h}_i \mid i \in \boldsymbol{r}(s)\})$.

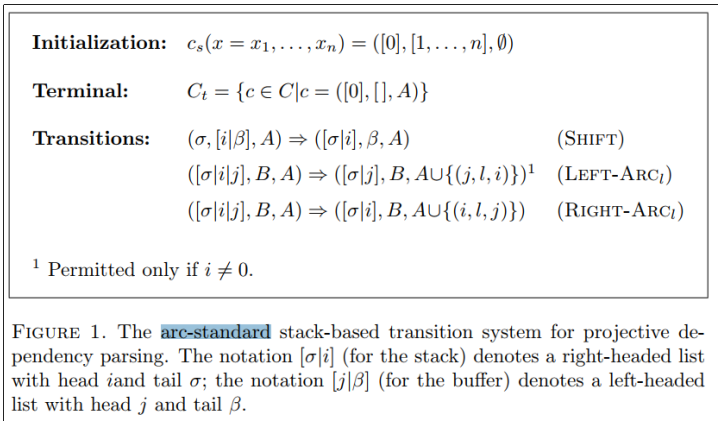
Example: Sequential tagging RNN. Let $\boldsymbol{x} = \{x_1, \dots, x_n\}$ be a sequence of input tokens. Let the ith output, d_i, be a tag from some predefined set of tags $\mathcal{A}$. Then our model can be defined as:

- Transition system: $\mathcal{T} = \{\ s_i{=}\mathcal{S}(x_i){=}\{1, \dots, d_{i-1}\},\ \ \mathcal{A},\ \ t(s_i, d_i, x_i){=}s_{i+1}{=}\ s_i{+}\{d_i\}\ \}$.
- Input function: $\boldsymbol{m}(s_i) = embed(x_i)$.
- Recurrence function: $\boldsymbol{r}(s_i) = \{i-1\}$ to connect the network to the previous state.
- RNN cell: $\boldsymbol{h}_i \leftarrow LSTM(\boldsymbol{m}(s_i) \mid \boldsymbol{r}(s_i) = \{i-1\})$.

[104] The authors state that we are only concerned with complete structures that have the same number of decisions $n(x)$ for the same input x.

Example: Parsey McParseface.

- Transition system: the **arc-standard** transition system, defined in image below[105].



Initialization:	$c_s(x = x_1, \ldots, x_n) = ([0], [1, \ldots, n], \emptyset)$	
Terminal:	$C_t = \{c \in C \mid c = ([0], [\,], A)\}$	
Transitions:	$(\sigma, [i\|\beta], A) \Rightarrow ([\sigma\|i], \beta, A)$	(SHIFT)
	$([\sigma\|i\|j], B, A) \Rightarrow ([\sigma\|j], B, A \cup \{(j, l, i)\})^1$	(LEFT-ARC$_l$)
	$([\sigma\|i\|j], B, A) \Rightarrow ([\sigma\|i], B, A \cup \{(i, l, j)\})$	(RIGHT-ARC$_l$)

[1] Permitted only if $i \neq 0$.

FIGURE 1. The arc-standard stack-based transition system for projective dependency parsing. The notation $[\sigma|i]$ (for the stack) denotes a right-headed list with head $i$and tail σ; the notation $[j|\beta]$ (for the buffer) denotes a left-headed list with head j and tail β.

  so the state contains all words and partially built trees (stack) as well as unseen words (buffer).
- Input function: $\boldsymbol{m}(s_i)$ is the concatenation of 52 feature embeddings extracted from tokens based on their positions in the stack and the buffer.
- Recurrence function: $\boldsymbol{r}(s_i)$ is empty, as this is a feed-forward network.
- RNN cell: a feed-forward MLP (so not an RNN...).

Inference with TBRUs. To predict the output sequence $\{d_1, \ldots, d_n\}$ given input sequence $\boldsymbol{x} = \{x_1, \ldots, x_n\}$, do:

1. Initialize $s_1 = s^\dagger$.
2. For $i = 1, \ldots, n$:
   (a) Compute $\boldsymbol{h}_i = \mathbf{RNN}(\boldsymbol{m}(s_i), \{\boldsymbol{h}_j \mid j \in \boldsymbol{r}(s_i)\})$.
   (b) Update transition state:

$$d_i \leftarrow \arg\max_{d \in \mathcal{A}(s_i)} \boldsymbol{w}_d^T \boldsymbol{h}_i \tag{350}$$

$$s_{i+1} \leftarrow t(s_i, d_i) \tag{351}$$

NOTE: This defines a locally normalized training procedure, whereas Andor et al. of Syntaxnet clearly conclude that their globally normalized model is the preferred choice.

[105] Image taken from "Transition-Based Parsing" by Joakim Nivre. Note that "right-headed" means "goes from left to right" or "headed *to* the right".

Combining multiple TBRUs. We connect multiple TBRUs with different transition systems via $\boldsymbol{r}(s)$.

1. We execute a list of T TBRU components sequentially, so that each TBRU advances a global step counter.
2. *Each* transition state, s^τ, from the τ'th component has access the *terminal* states from every prior transition system, and the recurrence function $\boldsymbol{r}(s^\tau)$ for any given component can pull hidden activations from every prior one as well.

Example: Multi-task bi-directional tagging. Say we want to do both POS and NER tagging (indices start at 1).

- Left-to-right: $\mathcal{T} =$ *shift-only*, $\boldsymbol{m}(s_i) = \boldsymbol{x}_i$, $\boldsymbol{r}(s_i) = \{i-1\}$.
- Right-to-left: $\mathcal{T} =$ *shift-only*, $\boldsymbol{m}(s_{n+i}) = \boldsymbol{x}_{(n-i)+1}$, $\boldsymbol{r}(s_{n+i}) = \{n+i-1\}$.
- POS Tagger: $\mathcal{T}_{POS} =$ *tagger*, $\boldsymbol{m}(s_{2n+i}) = \{\}$, $\boldsymbol{r}(s_{2n+i}) = \{i, (2n-i)+1\}$
- NER Tagger: $\mathcal{T}_{NER} =$ *tagger*, $\boldsymbol{m}(s_{3n+i}) = \{\}$, $\boldsymbol{r}(s_{3n+i}) = \{i, (2n-i)+1, 2n+i\}$

which illustrates the most important aspect of the TBRU:

> *A TBRU can serve as both an encoder for downstream tasks and a decoder for its own task simultaneously.*

For this example, the POS Tagger served as both an encoder for the NER task as well as a decoder for the POS task.

Training a DRAGNN. Assume training data consists of examples $\boldsymbol{x}$ along with gold decision sequences for a given TBRU in the DRAGNN. Given decisions $d_1, \ldots, d_N$ from prior components $1, \ldots, T-1$, the log-likelihood for training the T'th TBRU along its gold decision sequence $d^\star_{N+1}, \ldots, d^\star_{N+n}$ is then:

$$L(\boldsymbol{x},\ d^\star_{N+1:N+n};\ \theta) = \sum_i \log \Pr\left[d^\star_{N+i} \mid d_{1:N},\ d^\star_{N+1:N+i-1};\ \theta\right] \tag{352}$$

During training, the entire input sequence is unrolled and **backpropagation through structure** is used for gradient computation.

4.31.1 More Detail: Arc-Standard Transition System

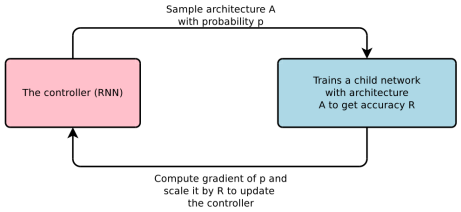
The arc-standard transition system is mentioned a lot, but with little detail. Here I'll synthesize what I find from external resources. The literature defines the states in a transition system slightly differently than the DRAGNN paper. Here we'll define them as a **configuration** $c = (\Sigma, B, A)$ triplet, where

- Σ is the **stack** of tokens in x that we've [partially] processed.
- B is the **buffer** of remaining tokens in x that we need to process.
- A is a set of **arcs** (w_i, w_j, ℓ) that link w_i to w_j, and label the arc/link as ℓ.

So, in the arc-standard transition system figure presented with Parsey McParseface earlier,

- SHIFT just means "move the head element of the buffer to the tail element of the buffer".
- Left-arc just means "make a link *from* the tail element of the stack *to* the element before it. Remove the pointed-to element from the stack."
- Right-arc just means "make a link *from* the element before the tail element in the stack *to* the tail element. Remove the pointed-to element from the stack."

B. Zoph and Q. Le, "Neural Architecture Search with Reinforcement Learning," (2017).



Controller RNN. Generates architectures with a predefined number of layers, which is increased manually as training progresses. At convergence, validation accuracy of the generated network is recorded. Then, the controller parameters θ_c are optimized to maximized the expected validation accuracy over a batch of generated architectures.

Reinforcement Learning to learn the controller parameters θ_c. Let $a_{1:T}$ denote a list of actions taken by the controller[106], which defines a generated architecture. We denote the resulting validation accuracy by R, which is the reward signal for our RL task. Concretely, we want our controller to maximize its expected reward, $J(\theta_c)$:

$$J(\theta_c) = \mathbb{E}_{P(a_{1:T};\, \theta_c)}\left[R\right] \tag{353}$$

Since the quantity $\nabla_{\theta_c} R$ is non-differentiable[107], we use a **policy gradient** method to iteratively update θ_c. All this means is that we instead compute gradients over the softmax outputs (the action probabilities), and use the value of R as a simple weight factor.

$$\nabla_{\theta_c} J(\theta_c) = R \sum_{t=1}^{T} \mathbb{E}_{P(a_{1:T};\, \theta_c)}\left[\nabla_{\theta_c} \log P(a_t \mid a_{1:t-1}; \theta_c)\right] \tag{354}$$

$$\approx \frac{1}{m} \sum_{k=1}^{m} R_k \sum_{t=1}^{T} \nabla_{\theta_c} \log P(a_t \mid a_{1:t-1}; \theta_c) \tag{355}$$

where the second equation is the empirical approximation (batch-average instead of expectation) over a batch of size m, an unbiased estimator for our gradient[108]. Also note that we *do*

[106] Note that T is not necessarily the number of layers, since a single generated layer can correspond to multiple actions (e.g. stride height, stride width, num filters, etc.).

[107] R is a function of the action sequence $a_{1:T}$ and the parameters θ_c, and implicitly depends on the samples used for the validation set. Clearly, we do not have access to an analytical form of R, and computing numerical gradients via small perturbations of $theta_c$ is computationally intractable.

[108] It is unbiased for the same reason that any average over samples x drawn from a distribution $P(x)$ is an unbiased estimator for $\mathbb{E}_P\left[x\right]$.

have access to the distribution $P(a_{1:T}; \theta_c)$ since it is *defined* to be the joint softmax probabilities of our controller, given its parameter values θ_c (i.e. this is not a p_{data} vs p_{model} situation). The approximation 354 is an unbiased estimator for 355, but has high variance. To reduce the variance of our estimator, the authors employ a baseline function b that does not depend on the current action:

$$\frac{1}{m} \sum_{k=1}^{m} \sum_{t=1}^{T} \nabla_{\theta_c} \log P(a_t \mid a_{1:t-1}; \theta_c)(R_k - b) \tag{356}$$

Joint Extraction of Events and Entities within a Document Context

Written by Brandon McKinzie

B. Yang and T. Mitchell, "Joint Extraction of Events and Entities within a Document Context," (2016).

Introduction. Two main reasons that state-of-the-art event extraction systems have difficulties:

1. They extract events and entities in separate stages.
2. They extract events independently from each individual sentence, ignoring the rest of the document.

This paper proposes an approach that simultaneously extracts events and entities within a document context. They do this by first decomposing the problem into 3 tractable subproblems:

1. Learning the dependencies between a single event [trigger] and all of its potential arguments.
2. Learning the co-occurrence relations between events across the document.
3. Learning for entity extraction.

and then combine these learned models into a joint optimization framework.

Learning Within-Event Structures. For now, assume we have some document x, a set of candidate event triggers $\mathcal{T}$, and a set of candidate entities $\mathcal{N}$. Denote the set of entity candidates that are potential arguments for trigger candidate i as $\mathcal{N}_i$. The joint distribution over the possible trigger types, roles, and entities for those roles, is given by

$$\Pr_{\boldsymbol{\theta}}\left[t_i, \boldsymbol{r}_i, \boldsymbol{a} \mid i, \mathcal{N}_i, x\right] \propto \tag{357}$$

$$\exp\left(\boldsymbol{\theta}_1^T f_1(t_i) + \sum_{j \in \mathcal{N}_i}\left[\boldsymbol{\theta}_2^T f_2(r_{ij}) + \boldsymbol{\theta}_3^T f_3(t_i, r_{ij}) + \boldsymbol{\theta}_4^T f_4(a_j) + \boldsymbol{\theta}_5^T f_5(r_{ij}, a_j)\right]\right) \tag{358}$$

where each f_i is a feature function, and I've colored the unary feature functions green. The unary features are tabulated in Table 1 of the original paper. They use simple indicator functions $1_{t,r}$ and $1_{r,a}$ for the pairwise features. They train using maximum-likelihood estimates with L2 regularization:

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) - \lambda||\boldsymbol{\theta}||_2^2 \tag{359}$$

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_i \log\left(\Pr_{\boldsymbol{\theta}}\left[t_i, \boldsymbol{r}_i, \boldsymbol{a} \mid i, \mathcal{N}_i, x\right]\right) \tag{360}$$

and use **L-BFGS** to optimize the training objective.

Learning Event-Event Relations. A pairwise model of event-event relations in a document. Training data consists of all pair of trigger candidates that co-occur in the same sentence or are connected by a co-referent subject/object if they're in different sentences. Given a trigger candidate pair (i, i'), we estimate the probabilities for their event types $(t_i, t_{i'})$ as

$$\Pr_\phi \left[t_i, t_{i'} \mid x, i, i'\right] \propto \exp\left(\phi^T g(t_i, t_{i'}, x, i, i')\right) \tag{361}$$

where g is a feature function that depends on the trigger candidate pair and their context. In addition to re-using the trigger features in Table 1 of the paper, they also introduce relational trigger features:

1. whether they're connected by a conjunction dependency relation
2. whether they share a subject or an object
3. whether they have the same head word lemma
4. whether they share a semantic frame based on FrameNet.

As before, they using L-BFGS to compute the maximum-likelihood estimates of the parameters ϕ.

Entity Extraction. Trained a standard linear-chain **CRF** using the BIO scheme. Their CRF features:

1. current words and POS tags
2. context words in a window of size 2
3. word type such as all-capitalized, is-capitalized, all-digits
4. Gazetteer-based entity type if the current word matches an entry in the gazetteers collected from Wikipedia.
5. pre-trained word2vec embeddings for each word

Joint Inference. Allows information flow among the 3 local models and finds globally-optimal assignments of all variables. Define the following objective:

$$\max_{\boldsymbol{t}, \boldsymbol{r}, \boldsymbol{a}} \sum_{i \in \mathcal{T}} E(t_i, \boldsymbol{r}_i, \boldsymbol{a}) + \sum_{i, i' \in \mathcal{T}} R(t_i, t_{i'}) + \sum_{j \in \mathcal{N}} D(a_j) \tag{362}$$

where

- The first term is the sum of confidence scores for individual event mentions from the within-event model.

$$E(t_i, \boldsymbol{r}_i, \boldsymbol{a}) = \log p_\theta(t_i) + \sum_{j \in \mathcal{N}_i} \left[\log p_\theta(t_i, r_{ij}) + \log p_\theta(r_{ij}, a_j)\right] \tag{363}$$

- The second term is the sum of confidence scores for event relations based on the pairwise event model.

$$R(t_i, t_{i'}) = \log p_\phi(t_i, t_{i'} \mid i, i', x) \tag{364}$$

- The third term is sum of confidence scores for entity mentions, where

$$D(a_j) = \log p_\psi(a_j \mid j, x) \tag{365}$$

  and $p_\psi(a_j \mid j, x)$ is the marginal probability derived from the linear-chain CRF.

The optimization is subject to agreement constraints that enforce the overlapping variables among the 3 components to agree on their values. The joint inference problem can be formulated as an **integer linear problem (ILP)**[109]. To solve it efficiently, they find solutions for the relaxation of the problem using a **dual decomposition algorithm** AD^3.

[109] From Wikipedia: An integer linear program in canonical form: maximize $c^T x$ subject to $Ax \leq b, x \geq 0, x \in \mathbb{Z}^n$

Globally Normalized Transition-Based Neural Networks

Written by Brandon McKinzie

D. Andor et al., "Globally Normalized Transition-Based Neural Networks," (2016).

Introduction. Authors demonstrate that simple FF neural networks can achieve comparable or better accuracies than LSTMs, as long as they are **globally normalized**. They don't use any recurrence, but perform **beam search** for maintaining multiple hypotheses and introduce global normalization with a **CRF** objective to overcome the label bias problem that locally normalized models suffer from.

Transition System. Given an input sequence x, define:

- Set of states $S(x)$.
- Start state $s^\dagger \in S(x)$.
- Set of decisions $\mathcal{A}(s, x)$ for all $s \in S(x)$.
- Transition function $t(s, d, x)$ returning new state s' for any decision $d \in A(s, x)$.

The scoring function $\rho(s, d; \theta)$, which gives the score for decision d in state s, will be defined:

$$\rho(s, d; \theta) = \phi(s; \theta^{(l)}) \cdot \theta^{(d)} \tag{366}$$

which is just the familiar logits computation for decision d. $\theta^{(l)}$ are the parameters of the network excluding the parameters at the final layer, $\theta^{(d)}$. $\phi(s; \theta^{(l)})$ gives the representation for state s computed by the neural network under parameters $\theta^{(l)}$.

Global vs. Local Normalization.

- **Local**. Conditional probabilities $\Pr[d_j \mid s_j; \theta]$ are normalized locally over the scores for each possible action d_j from the current state s_j.

$$\Pr_L[d_{1:n}] = \prod_{j=1}^{n} \Pr[d_j \mid s_j; \theta] = \frac{\exp\left(\sum_{j=1}^{n} \rho(s_j, d_j; \theta)\right)}{\prod_{j=1}^{n} Z_L(s_j; \theta)} \tag{367}$$

$$Z_L(s_j; \theta) = \sum_{d' \in \mathcal{A}(s_j)} \exp\left(\rho(s_j, d'; \theta)\right) \tag{368}$$

  Beam search can be used to attempt to find the action sequence with highest probability.
- **Global**. In contrast, a CRF defines:

$$\Pr_G[d_{1:n}] = \frac{\exp\left(\sum_{j=1}^{n} \rho(s_j, d_j; \theta)\right)}{Z_G(\theta)} \tag{369}$$

$$Z_G(\theta) = \sum_{d'_{1:n} \in \mathcal{D}_n} \exp\left(\sum_{j=1}^{n} \rho(s'_j, d'_j; \theta)\right) \tag{370}$$

where $\mathcal{D}_n$ is the set of all valid sequences of decisions of length n. The inference problem is now to find

$$\arg\max_{d_{1:n} \in \mathcal{D}_n} \Pr_G [d_{1:n}] = \arg\max_{d_{1:n} \in \mathcal{D}_n} \sum_{j=1}^{n} \rho(s_j, d_j; \theta) \tag{371}$$

and we can also use beam search to approximately find the argmax.

Training. SGD on the NLL of the data under the model. The NLL takes a different form depending on whether we choose a locally normalized model vs a globally normalized model.

- **Local**.

$$L_{local}(d^*_{1:n}; \theta) = -\ln \Pr_L [d^*_{1:n}; \theta] \tag{372}$$

$$= -\sum_{j=1}^{n} \rho(s^*_j, d^*_j; \theta) + \sum_{j=1}^{n} \ln Z_L(s^*_j; \theta) \tag{373}$$

- **Global**.

$$L_{global}(d^*_{1:n}; \theta) = -\ln \Pr_G [d^*_{1:n}; \theta] \tag{374}$$

$$= -\sum_{j=1}^{n} \rho(s^*_j, d^*_j; \theta) + \ln Z_G(\theta) \tag{375}$$

To make learning tractable for the globally normalized model, the authors use **beam search with early updates**, defined as follows. Keep track of the location of the gold path[110] in the beam as the prediction sequence is being constructed. If the gold path is not found in the beam after step j, run one step of SGD on the following objective:

$$L_{global-beam}(d^*_{1:j}, \theta) = -\sum_{t=1}^{j} \rho(d^*_{1:t-1}, d^*_t; \theta) - \ln \sum_{d'_{1:j} \in \mathcal{B}_j} \exp\left(\sum_{t=1}^{j} \rho(d'_{1:t-1}, d'_t; \theta)\right) \tag{376}$$

where $\mathcal{B}_j$ contains all paths in the beam at step j, *and* the gold path prefix $d^*1:j$. If the gold path remains in the beam throughout decoding, a gradient step is performed using $\mathcal{B}_T$, the beam at the end of decoding. When training the global model, the authors first pretrain[111] using the local objective function, and then perform additional training steps using the global objective function..

[110] The gold path is the predicted sequence that matches the true labeled sequence, up to the current timestep.

The Label Bias Problem. Locally normalized models often have a very weak ability to revise earlier decisions. Here we will prove that **globally normalized models are strictly more expressive than locally normalized models**[112]. Let $\mathcal{P}_L$ denote the set of all possible distributions $p_L(d_{1:n} \mid x_{1:n})$ under the local model as the scores ρ vary. Let $\mathcal{P}_G$ be the same, but for the global model.

Theorem 3.1 *$\mathcal{P}_L$ is a strict subset[113] of $\mathcal{P}_G$, that is $\mathcal{P}_L \subsetneq \mathcal{P}_G$.*

We are assuming that both P_L and P_G consist of log-linear distributions of scoring functions $\rho(d_{1:t-1}, d_t, x_{1:t})$

In other words, a globally normalized model can model any distribution that a locally normalized one can, but the converse is not true. I've worked through the proof below.

Proof: $P_L \subsetneq P_G$

Proof that $P_L \subseteq P_G$. For any locally normalized model with scores $\rho_L(d_{1:t-1}, d_t, x_{1:t})$, we can define a corresponding p_G over scores

$$\rho_G(d_{1:t-1}, d_t, x_{1:t}) = \ln p_L(d_t \mid d_{1:t-1}, x_{1:t}) \tag{377}$$

By definition, this means that $p_G(d_{1:t} \mid x_{1:t}) = p_L(d_{1:t} \mid x_{1:t})$.

Proof that $P_G \nsubseteq P_L$. A proof by example. Consider a dataset consisting entirely of one of the following tagged sequences:

$$\boldsymbol{x} = abc, \qquad \boldsymbol{d} = ABC \tag{378}$$
$$\boldsymbol{x} = abe, \qquad \boldsymbol{d} = ADE \tag{379}$$

Similar to a typical linear-chain CRF, let $\mathcal{T}$ denote the set of observed label transitions, and let $\mathcal{E}$ denote the set of observed (x_t, d_t) pairs. Let α be the single scalar parameter of this simple model, where

$$\rho(d_{1:t-1}, d_t, x_{1:t}) = \alpha \left(\mathbb{1}_{(d_{t-1}, d_t) \in \mathcal{T}} + \mathbb{1}_{(x_t, d_t) \in \mathcal{E}} \right) \tag{380}$$

for all t. This results in the following distributions p_G and p_L, evaluating on input sequence of length 3

$$p_G(d_{1:3} \mid x_{1:3}) = \frac{\exp\left(\alpha \sum_{t=1}^{3} (\mathbb{1}_{(d_{t-1}, d_t) \in \mathcal{T}} + \mathbb{1}_{(x_t, d_t) \in \mathcal{E}})\right)}{Z_G(x_{1:3})} \tag{381}$$
$$p_L(d_{1:3} \mid x_{1:3}) = p_L(d_1 \mid x_1) p_L(d_2 \mid d_1, x_{1:2}) p_L(d_3 \mid d_{1:2}, x_{1:3}) \tag{382}$$

where I've written p_L as a product over its local CPDs because it reveals the core observation that the proof is based on: *for any given subsequence $(d_{1:t-1}, x_{1:t})$, the local CPD is constrained to satisfy $\sum_{d_t} p_L(d_t \mid d_{1:t-1}, x_{1:t}) = 1$*. With this, the following comparison of p_G and p_L for large α completes the proof of $P_G \nsubseteq P_L$:

$$\lim_{\alpha \to \infty} p_G(ABC \mid abc) = \lim_{\alpha \to \infty} p_G(ADE \mid abe) = 1 \tag{383}$$
$$p_L(ABC \mid abc) + p_L(ADE \mid abe) \leq 1 \quad (\forall \alpha) \tag{384}$$

$\therefore P_L \subsetneq P_G$.

[112] This is for conditional models only.

[113] Note that $\subset$ and $\subsetneq$ mean the same thing. Matter of notational preference/being explicit/etc.

An Introduction to Conditional Random Fields

Written by Brandon McKinzie

Sutton et al., "An Introduction to Conditional Random Fields," (2012).

Graphical Modeling (2.1). Some notation. Denote factors as $\psi_a(\boldsymbol{y}_a)$ where $1 \leq a \leq A$ and A is the total number factors. $\boldsymbol{y}_a$ is an assignment to the subset $Y_a \subseteq Y$ of variables associated with ψ_a. The value returned by ψ_a is a non-negative scalar that can be thought of as a measure of how compatible the values $\boldsymbol{y}_a$ are with each other. Given a collection of subsets $\{Y_a\}_{a=1}^{A}$ of Y, an **undirected graphical model** is the set of all distributions that can be written as

$$p(y) = \frac{1}{Z} \prod_{a=1}^{A} \psi_a(\boldsymbol{y}_a) \tag{385}$$

$$Z = \sum_{\boldsymbol{y}} \prod_{a=1}^{A} \psi_a(\boldsymbol{y}_a) \tag{386}$$

for any choice of **factors** $\mathcal{F} = \{\psi_a\}$ that have $\psi_a(\boldsymbol{y}_a) \geq 0$ for all $\boldsymbol{y}_a$. The sum for the **partition function**, Z, is over all possible assignments $\boldsymbol{y}$ of the variables Y. We'll use the term ***random field*** to refer to a particular distribution among those defined by an undirected model[114].

We can represent the factorization with a **factor graph**: a bipartite graph $G = (V, F, E)$ in which one set of nodes $V = \{1, 2, \ldots, |Y|\}$ indexes the RVs in the model, and the set of nodes $F = \{1, 2, \ldots, A\}$ indexes the factors. A connection between a variable node Y_s for $s \in V$ to some factor node ψ_a for $a \in F$ means that Y_s is one of the arguments of ψ_a. It is common to draw the factor nodes as squares, and the variable nodes as circles.

Generative versus Discriminative Models (2.2). Naive Bayes is generative, while logistic regression (a.k.a maximum entropy) is discriminative. Recall that Naive Bayes and logistic are defined as, respectively,

$$p(y, \boldsymbol{x}) = p(y) \prod_{k=1}^{K} p(x_k \mid y) \tag{387}$$

$$p(y \mid \boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})} \exp\left(\sum_{k=1}^{K} \theta_k f_k(y, \boldsymbol{x}) \right) \tag{388}$$

where the f_k in the definition of logistic regression denote the feature functions. We could set them, for example, as $f_{y',j}(y, \boldsymbol{x}) = 1_{y'=y} x_j$.

[114] i.e. a particular set of factors.

An example generative model for sequence prediction is the **HMM**. Recall that an HMM defines

$$p(\boldsymbol{y}, \boldsymbol{x}) = \prod_{t=1}^{T} p(y_t \mid y_{t-1}) p(x_t \mid y_t) \tag{389}$$

where we are using the dummy notation of assuming an initial-initial state y_0 clamped to 0 and begins every state sequence, so we can write the initial state distribution as $p(y_1 \mid y_0)$.

We see that the generative models, like naive Bayes and the HMM, define a family of joint distributions that factorizes as $p(y, x) = p(y)p(x \mid y)$. Discriminative models, like logistic regression, define a family of conditional distributions $p(y \mid x)$. The main conceptual difference here is that a conditional distribution $p(y \mid x)$ doesn't include a model of $p(x)$. The principal advantage of discriminative modeling is that it's better suited to include rich, overlapping features. Discriminative models like CRFs make conditional independence assumptions both (1) among y and (2) about how the y can depend on x, but do *not* make conditional independence assumptions among x.

The difference between NB and LR is due *only* to the fact that NB is generative and LR is discriminative. Any LR classifier can be converted into a NB classifier with the same decision boundary, and vice versa. In other words, NB defines the same family as LR, if we interpret NB generatively as

$$p(y, \boldsymbol{x}) = \frac{\exp\left(\sum_k \theta_k f_k(y, \boldsymbol{x})\right)}{\sum_{\widetilde{y}, \widetilde{x}} \exp\left(\sum_k \theta_k f_k(\widetilde{y}, \widetilde{x})\right)} \tag{390}$$

and train it to maximize the conditional likelihood. Similarly, if the LR model is interpreted as above, and trained to maximize the joint likelihood, then we recover the same classifier as NB.

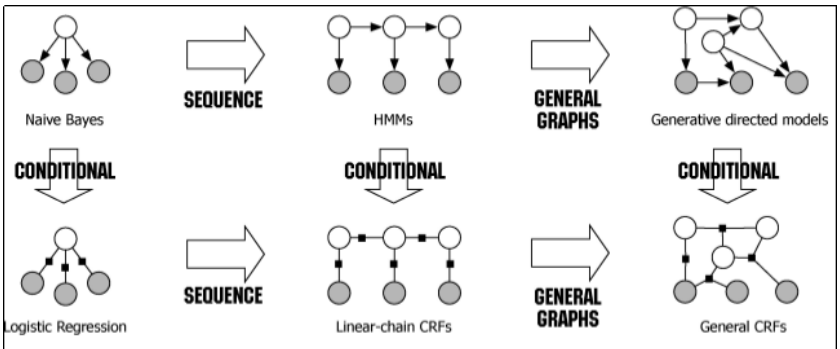


Fig. 2.4 Diagram of the relationship between naive Bayes, logistic regression, HMMs, linear-chain CRFs, generative models, and general CRFs.

Linear-Chain CRFs (2.3). Key point: the conditional distribution $p(\boldsymbol{y} \mid \boldsymbol{x})$ that follows from the joint distribution $p(\boldsymbol{y}, \boldsymbol{x})$ of an HMM is in fact a CRF with a particular choice of feature functions. First, we rewrite the HMM joint in a form that's more amenable to generalization:

$$p(\boldsymbol{y}, \boldsymbol{x}) = \frac{1}{Z} \prod_{t=1}^{T} \exp\left(\sum_{i,j \in S} \theta_{i,j} \mathbb{1}_{\{y_t = i, y_{t-1} = j\}} + \sum_{i \in S, o \in O} \mu_{o,i} \mathbb{1}_{\{y_t = i, x_t = o\}} \right) \tag{391}$$

$$= \frac{1}{Z} \prod_{t=1}^{T} \exp\left(\sum_{k=1}^{K} \theta_k f_k(y_t, y_{t-1}, x_t) \right) \tag{392}$$

and the latter provides the more compact notation[115]. We can use Bayes rule to then write $p(\boldsymbol{y} \mid \boldsymbol{x})$, which would give us a particular kind of linear-chain CRF that only includes features for the current word's identity. The general definition of linear-chain CRFs is given below:

> *Let Y, X be random vectors, $\theta = \{\theta_k\} \in \mathbb{R}^K$ be a parameter vector, and $\mathcal{F} = \{f_k(y, y', \boldsymbol{x}_t)\}_{k=1}^K$ be a set of real-valued feature functions. Then a* ***linear-chain conditional random field*** *is a distribution $p(\boldsymbol{y} \mid \boldsymbol{x})$ that takes the form:*
>
> $$p(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})} \prod_{t=1}^{T} \exp\left(\sum_{k} \theta_k f_k(y_t, y_{t-1}, \boldsymbol{x}_t) \right) \tag{393}$$
>
> $$Z(\boldsymbol{x}) = \sum_{\boldsymbol{y}} \prod_{t=1}^{T} \exp\left(\sum_{k} \theta_k f_k(y_t, y_{t-1}, \boldsymbol{x}_t) \right) \tag{394}$$

Notice that a linear chain CRF can be described as a factor graph over $\boldsymbol{x}$ and $\boldsymbol{y}$, where each local function (factor) ψ_t has the special log-linear form:

$$\psi_t(y_t, y_{t-1}, x_t) = \exp\left(\sum_{k} \theta_k f_k(y_t, y_{t-1}, x_t) \right) \tag{395}$$

General CRFs (2.4). Let G be a factor graph over X and Y. Then (X, Y) is a conditional random field if for any value $\boldsymbol{x}$ of X, the distribution $p(\boldsymbol{y} \mid \boldsymbol{x})$ factorizes according to G. If $F = \{\psi_a\}$ is the set of A factors in G, then the conditional distribution for a CRF is

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})} \prod_{a=1}^{A} \psi_a(\boldsymbol{y}_a, \boldsymbol{x}_a) \tag{396}$$

It is often useful to require that the factors be log-linear over a prespecified set of feature functions, which allows us to write the conditional distribution as

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})} \prod_{\psi_a \in F} \exp\left(\sum_{k=1}^{K(a)} \theta_{a,k} f_{a,k}(\boldsymbol{y}_a, \boldsymbol{x}_a) \right) \tag{397}$$

[115] Note how we collapsed the summations over i, j and i, o to simply k. This is purely notational. Each value k can be mapped to/from a unique i, j or i, o in the first version. Also note that, necessarily, each feature function f_k in the latter version maps to a specific indicator function $\mathbb{1}$ in the first.

In addition, most models rely extensively on **parameter tying**[116]. To denote this, we partition the factors of G into $\mathcal{C} = \{C_1, C_2, \ldots, C_P\}$, where each C_p is a **clique template**: a set of factors sharing a set of feature functions $\{f_{p,k}(\boldsymbol{x}_c, \boldsymbol{y}_c)\}_{k=1}^{K(p)}$ and a corresponding set of parameters $\boldsymbol{\theta}_p \in \mathbb{R}^{K(p)}$. A CRF that uses clique templates can be written as

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})} \prod_{C_p \in \mathcal{C}} \prod_{\psi_c \in C_p} \psi_c(\boldsymbol{x}_c, \boldsymbol{y}_c; \boldsymbol{\theta}_p) \tag{398}$$

$$= \frac{1}{Z(\boldsymbol{x})} \prod_{C_p \in \mathcal{C}} \prod_{c \in C_p} \exp\left\{ \sum_{k=1}^{K(p)} \theta_{p,k} f_{p,k}(\boldsymbol{x}_c, \boldsymbol{y}_c) \right\} \tag{399}$$

In a linear-chain CRF, typically one uses one clique template $C_0 = \{\psi_t\}_{t=1}^{T}$. Again, each factor in a given template *shares the same feature functions and parameters*, so the previous sentence means that we reuse the set of features and parameters for each timestep.

Feature engineering (2.5).

- **Label-observation features**. When our label variables are discrete, the features $f_{p,k}$ of a clique template C_p are ordinarily chosen to have a particular form:

$$f_{p,k}(\boldsymbol{y}_c, \boldsymbol{x}_c) = 1_{\{\boldsymbol{y}_c = \widetilde{\boldsymbol{y}}_c\}} q_{p,k}(\boldsymbol{x}_c) \tag{400}$$

  and we refer to the functions $q_{p,k}(\boldsymbol{x}_c)$ as *observation functions*.
- **Unsupported features**. Many observation-label pairs may never occur in our training data (e.g. having the word "with" being associated with label "CITY"). Such features are called unsupported features, and can be useful since often their weights will be driven negative, which can help prevent the model from making predictions in the future that are far from what was seen in the training data.
- **Edge-Observation and Node-Observation features**: the two most common types of label-observation features. Edge-observation features are for the transition factors, while node-observation features are the form introduced for label-observation features above.

$$\text{[edge-obs]} \quad f(y_t, y_{t-1}, \boldsymbol{x}_t) = q_m(\boldsymbol{x}_t) \mathbb{1}_{y_t = y, y_{t-1} = y'} \quad (\forall y, y' in \mathcal{Y}, \forall m) \tag{401}$$

$$\text{[node-obs]} \quad f(y_t, y_{t-1}, \boldsymbol{x}_t) = \mathbb{1}_{y_t = y, y_{t-1} = y'} \quad (\forall y, y' in \mathcal{Y}) \tag{402}$$

  and both use the same $f(y_t, \boldsymbol{x}_t) = q_m(\boldsymbol{x}_t) \mathbb{1}_{y_t = y}$ $(\forall y, \in \mathcal{Y}, \forall m)$. Recall that m is the index into our set of observation features[117].
- **Feature Induction**. The model begins with a number of base features, and the training procedure adds conjunctions of those features.

[116] Note how, for CRFs, the actual parameters θ are tightly coupled with the feature functions f.

[117] In CRFSuite, the observation features are all the attributes we define, and any features that use both label and observation are defined within CRFSuite itself.

There are two inference problems that arise:

1. Wanting to predict the labels of a new input $\boldsymbol{x}$ using the most likely labeling $\boldsymbol{y}^* = \arg\max_{\boldsymbol{y}} p(\boldsymbol{y} \mid \boldsymbol{x})$.
2. Computing marginal distributions (during parameter estimation, for example) such as node marginals $p(y_t \mid \boldsymbol{x})$ and edge marginals $p(y_t, y_{t-1} \mid \boldsymbol{x})$.

For linear-chain CRFs, the **forward-backward algorithm** is used for computing marginals, and the **Viterbi algorithm** for computing the most probable assignment. We'll first derive these for the case of HMMs, and then generalize to the linear-chain CRF case.

Forward-backward algorithm (HMMs). An efficient technique for computing marginals. We begin by writing out $p(\boldsymbol{x})$, and using the distributive law to convert the sum of products to a product of sums:

$$p(\boldsymbol{x}) = \sum_{\boldsymbol{y}} p(\boldsymbol{x}, \boldsymbol{y}) \tag{403}$$

$$= \sum_{\boldsymbol{y}} \prod_{t=1}^{T} \psi_t(y_t, y_{t-1}, x_t) \tag{404}$$

$$= \sum_{y_1} p(y_1)p(x_1 \mid y_1) \sum_{y_2} p(y_2 \mid y_1)p(x_2 \mid y_2) \sum_{y_3} \cdots \sum_{y_T} p(y_T \mid y_{T-1})p(x_T \mid y_T) \tag{405}$$

$$= \sum_{y_1} \psi_1(y_1, x_1) \sum_{y_2} \cdots \sum_{y_T} \psi_T(y_T, y_{T-1}, x_T) \tag{406}$$

$$= \sum_{T} \sum_{T-1} \psi_T(y_T, y_{t-1}, x_T) \sum_{y_{T-2}} \cdots \sum_{y_1} \psi_1(y_1, x_1) \tag{407}$$

We see that we can save an exponential amount of work by caching the inner sums as we go. Let M denote the number of possible states for the y variables. We define a set of T **forward variables** α_t, each of which is a vector of size M:

$$\alpha_t(j) \triangleq p(\boldsymbol{x}_{\langle 1 \ldots t \rangle}, y_t = j) \tag{408}$$

$$= \sum_{\boldsymbol{y}_{\langle 1 \ldots t-1 \rangle}} p(\boldsymbol{x}_{\langle 1 \ldots t \rangle}, \boldsymbol{y}_{\langle 1 \ldots t-1 \rangle}, y_t = j) \tag{409}$$

$$= \sum_{\boldsymbol{y}_{\langle 1 \ldots t-1 \rangle}} \psi_t(j, y_{t-1}, x_t) \prod_{t'=1}^{t'-1} \psi_{t'}(y_{t'}, y_{t-1}, x_{t'}) \tag{410}$$

$$= \sum_{i \in S} \psi_t(j, i, x_t) \sum_{\boldsymbol{y}_{\langle 1 \ldots t-2 \rangle}} \psi_{t-1}(y_{t-1}, y_{t-2}, x_{t-1}) \prod_{t'=1}^{t-2} \psi_{t'}(y_{t'}, y_{t-1}, x_{t'}) \tag{411}$$

$$= \sum_{i \in S} \psi_t(j, i, x_t) \alpha_{t-1}(i) \tag{412}$$

where S is the set of M possible states. By recognizing that $p(\boldsymbol{x}) = \sum_{j \in S} \sum_{\boldsymbol{y}_{\langle 1 \dots t-1 \rangle}} p(\boldsymbol{x}_{\langle 1 \dots t \rangle}, \boldsymbol{y}_{\langle 1 \dots t-1 \rangle}, j)$, we can rewrite $p(\boldsymbol{x})$ as

$$p(\boldsymbol{x}) = \sum_{j \in S} \alpha_T(j) \tag{413}$$

Notice how in the step from equation 408 to 409, we marginalized over all possible y subsequences that could've been aligned with $\boldsymbol{x}_{\langle 1 \dots t \rangle}$. We will repeat this pattern to derive the backward recursion for β_t, which is the same idea except now we go from T backward until some t (instead of going from 1 *forward* until some t).

$$\begin{aligned}
\beta_t(i) &\triangleq p(\boldsymbol{x}_{\langle t+1 \dots T \rangle} \mid y_t = i) && (414)\\
&= \sum_{\boldsymbol{y}_{\langle t+1 \dots T \rangle}} \psi_{t+1}(y_{t+1}, i, x_{t+1}) \prod_{t'=t+2}^{T} \psi_{t'}(y_{t'}, y_{t'-1}, x_{t'}) && (415)\\
&= \sum_{y_{t+1}} \psi_{t+1}(y_{t+1}, i, x_{t+1}) \beta_{t+1}(y_{t+1}) && (416)
\end{aligned}$$

Similar to how we obtained equation 413, we can rewrite $p(\boldsymbol{x})$ in terms of the β:

$$p(\boldsymbol{x}) = \beta_0(y_0) \triangleq \sum_{y_1} \psi_1(y_1, y_0, x_1) \beta_1(y_1) \tag{417}$$

We can then combine the definition for α and β to compute marginals of the form $p(y_{t-1}, y_t \mid \boldsymbol{x})$:

$$p(y_{t-1}, y_t \mid \boldsymbol{x}) = \frac{1}{p(\boldsymbol{x})} \alpha_{t-1}(y_{t-1}) \psi_t(y_t, y_{t-1}, x_t) \beta_t(y_t) \tag{418}$$

$$\text{where} \quad p(\boldsymbol{x}) = \sum_{j \in S} \alpha_T(j) = \beta_0(y_0) \tag{419}$$

In summary, the forward-backward algorithm consists of the following steps:

1. Compute α_t for all t using equation 413.
2. Compute β_t for all t using equation 417.
3. Return the marginal distributions computed from equation 418.

Viterbi algorithm (HMMs). For computing $\boldsymbol{y}^* = \arg\max_{\boldsymbol{y}} p(\boldsymbol{y} \mid \boldsymbol{x})$. The derivation is nearly the same as how we derived the forward-backward algorithm, except now we've replaced the summations in equation 407 with maximization. The analog of α for viterbi are defined as:

$$\delta_t(j) = \max_{\boldsymbol{y}_{\langle 1 \ldots t-1 \rangle}} \psi_t(j, y_{t-1}, x_t) \prod_{t'=1}^{t-1} \psi_{t'}(y_{t'}, y_{t'-1}, x_{t'}) \tag{420}$$

$$= \max_{i \in S} \psi_t(j, i, x_t) \delta_{t-1}(i) \tag{421}$$

and the maximizing assignment is computed by a backwards recursion,

$$y_T^* = \arg\max_{i \in S} \delta_T(i) \tag{422}$$

$$y_t^* = \arg\max_{i \in S} \psi_t(y_{t+1}^*, i, x_{t+1}) \delta_t(i) \text{ for } t < T \tag{423}$$

Computing the recursions for δ_t and y_t^* together is the *Viterbi algorithm.*

Forward-backward and Viterbi for linear-chain CRF. Generalizing to the linear-chain CRF, where now

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})} \prod_{t=1}^{T} \psi_t(y_t, y_{t-1}, x_t) \tag{424}$$

$$\text{where} \quad \psi_t(y_t, y_{t-1}, x_t) = \exp\left\{ \sum_k \theta_k f_k(y_t, y_{t-1}, x_t) \right\} \tag{425}$$

and the results for the forward-backward algorithm become

$$p(y_{t-1}, y_t \mid \boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})} \alpha_{t-1}(y_{t-1}) \psi_t(y_t, y_{t-1}, x_t) \beta_t(y_t) \tag{426}$$

$$p(y_t \mid \boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})} \alpha_t(y_t) \beta_t(y_t) \tag{427}$$

$$\text{where} \quad Z(\boldsymbol{x}) = \sum_{j \in S} \alpha_T(j) = \beta_0(y_0) \tag{428}$$

Note that the interpretation is also slightly different. The α, β, and δ variables should only be interpreted with the factorization formulas, and *not* as probabilities. Specifically, use

$$\alpha_t(j) = \sum_{\boldsymbol{y}_{\langle 1 \ldots t-1 \rangle}} \exp\left\{ \sum_k \theta_k f_k(j, y_{t-1}, x_t) \right\} \prod_{t'=1}^{t'-1} \exp\left\{ \sum_k \theta_k f_k(y_{t'}, y_{t'-1}, x_{t'}) \right\} \tag{429}$$

$$\beta_t(i) = \sum_{\boldsymbol{y}_{\langle t+1 \ldots T \rangle}} \exp\left\{ \sum_k \theta_k f_k(y_{t+1}, i, x_{t+1}) \right\} \prod_{t'=t+2}^{T} \exp\left\{ \sum_k \theta_k f_k(y_{t'}, y_{t'-1}, x_{t'}) \right\} \tag{430}$$

$$\delta_t(j) = \max_{\boldsymbol{y}_{\langle 1 \ldots t-1 \rangle}} \exp\left\{ \sum_k \theta_k f_k(j, y_{t-1}, x_t) \right\} \prod_{t'=1}^{t-1} \exp\left\{ \sum_k \theta_k f_k(y_{t'}, y_{t'-1}, x_{t'}) \right\} \tag{431}$$

Markov Chain Monte Carlo (MCMC). The two most popular classes of approximate inference algorithms are **Monte Carlo** algorithms and **variational** algorithms. In what follows, we drop the CRF-specific notation and refer to the more general joint distribution

$$p(\boldsymbol{y}) = Z^{-1} \prod_{a \in F} \psi_a(\boldsymbol{y}_a) \tag{432}$$

that factorizes according to some factor graph $G = (V, F)$. MCMC methods construct a Markov chain whose state space is the same as that of Y, and sample from this chain to approximate, e.g., the expectation of some function $f(\boldsymbol{y})$ over the distribution $p(\boldsymbol{y})$. MCMC algorithms aren't commonly applied in the context of CRFs, since parameter estimation by maximum likelihood requires calculating marginals many times.

4.35.2 Parameter Estimation (Sec. 5)

Maximum Likelihood for Linear-Chain CRFs. Since we're modeling the conditional distribution with CRFs, we use the **conditional log likelihood** $\ell(\boldsymbol{\theta})$ with l2-regularization:

$$\ell(\boldsymbol{\theta}) = \sum_{i=1}^{N} \log p(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) - \frac{1}{2\sigma^2} \sum_{k=1}^{K} \theta_k^2 \tag{433}$$

$$= \sum_{i=1}^{N} \sum_{t=1}^{T} \sum_{k=1}^{K} \theta_k f_k(y_t^{(i)}, y_{t-1}^{(i)}, \boldsymbol{x}_t^{(i)}) - \sum_{i=1}^{N} \log Z(\boldsymbol{x}^{(i)}) - \frac{1}{2\sigma^2} \sum_{k=1}^{K} \theta_k^2 \tag{434}$$

with regularization parameter $1/2\sigma^2$. The partial derivatives are

$$\frac{\partial \ell}{\partial \theta_k} = \sum_{i,t} f_k(y_t^{(i)}, y_{t-1}^{(i)}, \boldsymbol{x}_t^{(i)}) - \sum_{i,t,y,y'} f_k(y, y', \boldsymbol{x}_t^{(i)}) p(y, y' \mid \boldsymbol{x}^{(i)}) - \frac{\theta_k}{\sigma^2} \tag{435}$$

and the derivation for the partial derivative of $\log(Z(x))$ is

$$\frac{\partial \log Z(x)}{\partial \theta_k} = \frac{1}{Z(x)} \frac{\partial}{\partial \theta_k} \left[\sum_{\boldsymbol{y}\langle 1 \ldots T \rangle} \prod_{t=1}^{T} \exp\left\{ \sum_k \theta_k f_k(y_t, y_{t-1}, x_t) \right\} \right] \tag{436}$$

$$= \frac{1}{Z(x)} \sum_{\boldsymbol{y}\langle 1 \ldots T \rangle} \frac{\partial}{\partial \theta_k} \exp\left\{ \sum_t \sum_k \theta_k f_k(y_t, y_{t-1}, x_t) \right\} \tag{437}$$

$$= \frac{1}{Z(x)} \sum_{\boldsymbol{y}\langle 1 \ldots T \rangle} \sum_t f_k(y_t, y_{t-1}, x_t) \exp\left\{ \sum_t \sum_k \theta_k f_k(y_t, y_{t-1}, x_t) \right\} \tag{438}$$

$$= \sum_t \sum_{y_t} \sum_{y_{t-1}} f_k(y_t, y_{t-1}, x_t) \left[\sum_{\boldsymbol{y}\langle 1 \ldots t-2 \rangle} \sum_{\boldsymbol{y}\langle t+1 \ldots T \rangle} \frac{1}{Z(x)} \exp\left\{ \sum_t \sum_k \theta_k f_k(y_t, y_{t-1}, x_t) \right\} \right] \tag{439}$$

$$= \sum_t \sum_y \sum_{y'} f_k(y, y', x_t) p(y_t = y, y_{t-1} = y' \mid x) \tag{440}$$

We can rewrite this in the form of expectations. For now, let $\widetilde{p}(\boldsymbol{y}, \boldsymbol{x})$ denote the *empirical distribution*, and let $\hat{p}(\boldsymbol{y} \mid \boldsymbol{x}; \theta)\widetilde{p}(\boldsymbol{x})$ denote the *model distribution*.

$$\frac{\partial \ell}{\partial \theta_k} = \mathbb{E}_{\boldsymbol{x}, \boldsymbol{y} \sim \widetilde{p}(\boldsymbol{y}, \boldsymbol{x})} \left[\sum_t f_k(y_t, y_{t-1}, x_t) \right] - \mathbb{E}_{\boldsymbol{x}, \boldsymbol{y} \sim \hat{p}(\boldsymbol{y}, \boldsymbol{x})} \left[\sum_t f_k(y_t, y_{t-1}, x_t) \right] \tag{441}$$

Procedure: Training Linear-Chain CRFs

Here I summarize the main steps involved during a parameter update.

Inference. We need to compute the log probabilities for each instance in the dataset, under the current parameters. We will need them when evaluating $Z(\boldsymbol{x})$ and the marginals $p(y, y' \mid \boldsymbol{x})$ when computing gradients.

1. Initialize $\alpha_1(j) = \exp\{\sum_k \theta_k f_k(j, y_0, x_1)\}$ (y_0 is the fixed initial state) and $\beta_T(i) = 1$.
2. For all t, compute:

$$\alpha_t(j) = \sum_{i \in S} \exp\left\{ \sum_k \theta_k f_k(j, i, x_t) \right\} \alpha_{t-1}(i) \tag{442}$$

$$\beta_t(j) = \sum_{i \in S} \exp\left\{ \sum_k \theta_k f_k(i, j, x_{t+1}) \right\} \beta_{t+1}(i) \tag{443}$$

3. Compute the marginals:

$$p(y_t, y_{t-1}, \mid \boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})} \alpha_{t-1}(y_{t-1}) \psi_t(y_t, y_{t-1}, x_t) \beta_t(y_t) \tag{444}$$

$$p(y_t \mid \boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})} \alpha_t(y_t) \beta_t(y_t) \tag{445}$$

$$\text{where} \quad Z(\boldsymbol{x}) = \sum_{j \in S} \alpha_T(j) = \beta_0(y_0) \tag{446}$$

Gradients. For each parameter θ_k, compute:

$$\frac{\partial \ell}{\partial \theta_k} = \sum_{i,t} f_k(y_t^{(i)}, y_{t-1}^{(i)}, \boldsymbol{x}_t^{(i)}) - \sum_{i,t,y,y'} f_k(y, y', \boldsymbol{x}_t^{(i)}) p(y, y' \mid \boldsymbol{x}^{(i)}) - \frac{\theta_k}{\sigma^2} \tag{447}$$

CRF with latent variables. Now we have additional latent variables $\boldsymbol{z}$:

$$p(\boldsymbol{y}, \boldsymbol{z} \mid \boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})} \prod_t \psi_t(y_t, y_{t-1}, \boldsymbol{z}_t, \boldsymbol{x}_t) \tag{448}$$

Since we observe $\boldsymbol{y}$ during training, what if we instead treat this as a CRF with both $\boldsymbol{x}$ *and* $\boldsymbol{y}$ observed?

$$p(\boldsymbol{z} \mid \boldsymbol{y}, \boldsymbol{x}) = \frac{1}{Z(\boldsymbol{y}, \boldsymbol{x})} \prod_t \psi_t(y_t, y_{t-1}, \boldsymbol{z}_t, \boldsymbol{x}_t) \tag{449}$$

$$Z(\boldsymbol{y}, \boldsymbol{x}) = \sum_{\boldsymbol{z}} \prod_t \psi_t(y_t, y_{t-1}, \boldsymbol{z}_t, \boldsymbol{x}_t) \tag{450}$$

We can use the same inference algorithms as usual to compute $Z(\boldsymbol{y}, \boldsymbol{x})$, and the key result is that we can now write

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})} \sum_{\boldsymbol{z}} \prod_t \psi_t(y_t, y_{t-1}, \boldsymbol{z}_t, \boldsymbol{x}_t) = \frac{Z(\boldsymbol{y}, \boldsymbol{x})}{Z(\boldsymbol{x})} \tag{451}$$

Finally, we can write the gradient as[118]

$$\frac{\partial \ell}{\partial \theta_k} = \sum_{\boldsymbol{z}} p(\boldsymbol{z} \mid \boldsymbol{y}, \boldsymbol{x}) \frac{\partial}{\partial \theta_k} [\log p(\boldsymbol{y}, \boldsymbol{z} \mid \boldsymbol{x})] \tag{452}$$

$$= \sum_t \sum_{\boldsymbol{z}_t} \left[p(\boldsymbol{z}_t \mid \boldsymbol{y}, \boldsymbol{x}) f_k(y_t, y_{t-1}, \boldsymbol{z}_t, \boldsymbol{x}_t) - \sum_{y, y'} p(\boldsymbol{z}_t, y, y' \mid \boldsymbol{x}_t) f_k(y_t, y_{t-1}, \boldsymbol{z}_t, \boldsymbol{x}_t) \right] \tag{453}$$

where I've assumed there are no connections between any $\boldsymbol{z}_t$ and $\boldsymbol{z}_{t' \neq t}$.

Stochastic Gradient Methods. Now we compute gradients for a single example, and for a linear-chain CRF:

$$\frac{\partial \ell_i}{\partial \theta_k} = \sum_t f_k(y_t^{(i)}, y_{t-1}^{(i)}, \boldsymbol{x}_t^{(i)}) - \sum_{t, y, y'} f_k(y, y', \boldsymbol{x}_t^{(i)}) p(y, y' \mid \boldsymbol{x}^{(i)}) - \frac{\theta_k}{N\sigma^2} \tag{454}$$

which corresponds to parameter update (remember: we are using the LL, not the negative LL):

$$\theta^{(m)} = \theta^{(m-1)} + \alpha_m \nabla \ell_i(\theta^{(m-1)}) \tag{455}$$

where m denotes this is the mth update of the training process.

[118] This uses the trick

$$\frac{df}{d\theta} = f(\theta) \frac{d \log f}{d\theta}$$

MEMMs. Maximum-entropy Markov models. Essentially a Markov model in which the transition probabilities are given by logistic regression. Formally, a MEMM is defined by

$$p_{MEMM}(\boldsymbol{y} \mid \boldsymbol{x}) = \prod_{t=1}^{T} p(y_t \mid y_{t-1}, \boldsymbol{x}) \tag{456}$$

$$p(y_t \mid y_{t-1}, \boldsymbol{x}) = \frac{\exp\left\{\sum_{k=1}^{K} \theta_k f_k(y_t, y_{t-1}, \boldsymbol{x}_t)\right\}}{Z_t(y_{t-1}, \boldsymbol{x})} \tag{457}$$

$$Z_t(y_{t-1}, \boldsymbol{x}) = \sum_{y'} \exp\left\{\sum_{k=1}^{K} \theta_k f_k(y_t, y_{t-1}, \boldsymbol{x}_t)\right\} \tag{458}$$

which has some important differences compared to the linear-chain CRF. Notice how maximum-likelihood training of MEMMs does *not* require performance inference over full output sequences $\boldsymbol{y}$, because Z_t is a simple sum over the labels at a single position. MEMMs, however, suffer from *label bias*, while CRFs do not.

Bayesian CRFs. Instead of predicting the optimal labeling y^*_{ML} for input sequence x with maximum likelihood (ML), we can instead use a fully Bayesian (B) approach, both of which are shown below for comparison:

$$y^*_{ML} \leftarrow arg\, max_y p(y \mid x; \hat{\theta}) \tag{459}$$

$$y^*_{B} \leftarrow arg\, max_y \mathbb{E}_{\theta \sim p(\theta|x)}(p(y \mid x; \theta)) \tag{460}$$

Unfortunately, computing the exact integral (the expectation) is usually intractable, and we must resort to approximate methods like MCMC.

Co-sampling: Training Robust Networks for Extremely Noisy Supervision

Written by Brandon McKinzie

Han et al., "Co-sampling: Training Robust Networks for Extremely Noisy Supervision," (2018).

Introduction. The authors state that current methodologies [for training networks under noisy labels] involves estimating the **noise transition matrix** (which they don't define). Patrini et al. (2017) define the matrix as follows:

> *Denote by $T \in [0,1]^{c\times c}$ the noise transition matrix specifying the probability of one label being flipped to another, so that $\forall i,j\ T_{ij} \triangleq Pr\left[\widetilde{y} = e^j \mid y = e^i\right]$. The matrix is row-stochastic*[119] *and not necessarily symmetric across the classes.*

Algorithm. Authors propose a learning paradigm called **Co-sampling**. They maintain two networks f_{w_1} and f_{w_2} simultaneously. For each mini-batch data $\widehat{\mathcal{D}}$, each network selects $\mathcal{R}_T$ small-loss instances as a "clean" mini-batch $\widehat{\mathcal{D}}_1$ and $\widehat{\mathcal{D}}_2$, respectively. Each of the two networks then uses the clean mini-batch data to update the parameters w_2 (w_1) of its *peer* network.

- Why small-loss instances? Because deep networks tend to fit clean instances first, then noisy/harder instances progressively after.
- Why two networks? Because if we just trained a single network on clean instances, we would not be robust in extremely high-noise rates, since the training error would accumulate if the selected instances are not "fully clean."

The Co-sampling paradigm algorithm is shown below.

Input: w_1 and w_2; learning rate η; fixed $\mathfrak{D}$; epoch T_k and T_{max}; iteration N_{max};
for $T = 1, 2, \dots, T_{max}$ **do**
  Shuffle: training set $\widetilde{\mathcal{D}}$ //noisy dataset
  for $N = 1, \dots, N_{max}$ **do**
    Draw: mini-batch $\widehat{\mathcal{D}}$ from $\widetilde{\mathcal{D}}$
    Sample: $\widehat{\mathcal{D}}_1 = \arg\min_{\widehat{\mathcal{D}}} \ell(f_{w_1}, \widehat{\mathcal{D}}, \mathfrak{R}_T)$ //sample $\mathfrak{R}_T$ small-loss instances
    Sample: $\widehat{\mathcal{D}}_2 = \arg\min_{\widehat{\mathcal{D}}} \ell(f_{w_2}, \widehat{\mathcal{D}}, \mathfrak{R}_T)$ //sample $\mathfrak{R}_T$ small-loss instances
    Update: $w_1 = w_1 - \eta\nabla f_{w_1}(\widehat{\mathcal{D}}_2)$ //update w_1 by $\widehat{\mathcal{D}}_2$
    Update: $w_2 = w_2 - \eta\nabla f_{w_2}(\widehat{\mathcal{D}}_1)$ //update w_2 by $\widehat{\mathcal{D}}_1$
  end
  Update: $\mathfrak{R}_T = 1 - \min\{\frac{T}{T_k}\mathfrak{D}, \mathfrak{D}\}$
end
Output: f_{w_1} **and** f_{w_2}

[119] Each row sums to 1.

Hidden-Unit Conditional Random Fields

Written by Brandon McKinzie

Maaten et al., "Hidden-Unit Conditional Random Fields," (2011).

Introduction. Three key advantages of CRFs over HMMs:

1. CRFs don't assume that the observations are conditionally independent given the hidden (or target if linear-chain CRF) states.
2. CRFs don't suffer from the label bias problems of models that do local probability normalization[120].
3. For certain choices of factors, the negative conditional L.L. is convex.

The hidden-unit CRF (HUCRF), similar to discriminative restricted Boltzmann machines (RBMs), has binary stochastic hidden units that are conditionally independent given the data and the label sequence. By exploiting the conditional independence properties, we can efficiently compute:

1. The exact gradient of the C.L.L.
2. The most likely label sequence.
3. The marginal distributions over label sequences.

Hidden-Unit CRFs. At each time step t, the HUCRF employs H binary stochastic hidden units $\boldsymbol{z}_t$. It models the conditional distribution as

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{1}{\boldsymbol{Z}(\boldsymbol{x})} \sum_{\boldsymbol{z}} \exp\left(E(\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{y})\right) \tag{461}$$

$$\begin{aligned} E(\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{y}) = &\sum_{t=2}^{T} \left[\boldsymbol{y}_{t-1}^T \boldsymbol{A} \boldsymbol{y}_t\right] + \sum_{t=1}^{T} \left[\boldsymbol{x}_t^T \boldsymbol{W} \boldsymbol{z}_t + \boldsymbol{z}_t^T \boldsymbol{V} \boldsymbol{y}_t + \boldsymbol{b}^T \boldsymbol{z}_t + \boldsymbol{c}^T \boldsymbol{y}_t\right] \\ &+ \boldsymbol{y}_1^T \boldsymbol{\pi} + \boldsymbol{y}_T^T \boldsymbol{\tau} \end{aligned} \tag{462}$$

Since the hidden units are conditionally independent given the data and labels, the hidden units can be marginalized out one-by-one. This, along with the nice property that the hidden units have binary elements, allows us to write $p(\boldsymbol{y} \mid \boldsymbol{x})$ without writing any $\boldsymbol{z}_t$ explicitly, as

[120] See the introduction in my CRF notes for recap of label-bias.

shown below:

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{\exp\{\boldsymbol{y}_1^T \boldsymbol{\pi} + \boldsymbol{y}_T^T \boldsymbol{\tau}\}}{Z(\boldsymbol{x})} \prod_{t=1}^{T} \Bigg[\exp\{\boldsymbol{c}^T \boldsymbol{y}_t + \boldsymbol{y}_{t-1} \boldsymbol{A} \boldsymbol{y}_t\} \prod_{h=1}^{H} \sum_{z_h \in \{0,1\}} \exp\{z_h b_h + z_h \boldsymbol{w}_h^T \boldsymbol{x}_t + z_h \boldsymbol{v}_h^T \boldsymbol{y}_t\} \Bigg] \tag{463}$$

$$= \frac{\exp\{\boldsymbol{y}_1^T \boldsymbol{\pi} + \boldsymbol{y}_T^T \boldsymbol{\tau}\}}{Z(\boldsymbol{x})} \prod_{t=1}^{T} \Bigg[\exp\{\boldsymbol{c}^T \boldsymbol{y}_t + \boldsymbol{y}_{t-1} \boldsymbol{A} \boldsymbol{y}_t\} \prod_{h=1}^{H} \Big(1 + \exp\{b_h + \boldsymbol{w}_h^T \boldsymbol{x}_t + \boldsymbol{v}_h^T \boldsymbol{y}_t\}\Big) \Bigg] \tag{464}$$

For inference, we'll need an algorithm for computing the marginals $p(y_t \mid \boldsymbol{x})$ and $p(y_t, y_{t-1} \mid \boldsymbol{x})$. The equations are essentially the same as the forward-backward formulas for the linear-chain CRF, but with summations over $\boldsymbol{z}$:

$$p(y_t, y_{t-1} \mid \boldsymbol{x}) \propto \alpha_{t-1}(y_{t-1}) \Big[\sum_{\boldsymbol{z}_t} \Psi_t(\boldsymbol{x}_t, \boldsymbol{z}_t, y_t, y_{t-1}) \Big] \beta_t(y_t) \tag{465}$$

$$p(y_t \mid \boldsymbol{x}) \propto \alpha_t(y_t) \beta_t(y_t) \tag{466}$$

$$\tag{467}$$

$$\alpha_t(j) = \sum_{i \in \mathcal{Y}} \sum_{\boldsymbol{z}_t} \Psi_t(\boldsymbol{x}_t, \boldsymbol{z}_t, j, i) \alpha_{t-1}(i) \tag{468}$$

$$\beta_t(j) = \sum_{i \in \mathcal{Y}} \sum_{\boldsymbol{z}_{t+1}} \Psi_{t+1}(\boldsymbol{x}_{t+1}, \boldsymbol{z}_{t+1}, i, j) \beta_{t+1}(i) \tag{469}$$

Training. The conditional log likelihood for a single example $(\boldsymbol{x}, \boldsymbol{y})$ is (bias and initial-state terms omitted)

$$\mathcal{L} = \log p(\boldsymbol{y} \mid \boldsymbol{x}) \tag{470}$$

$$= \sum_{t=1}^{T} \log \left(\sum_{\boldsymbol{z}_t} \Psi_t \left(\boldsymbol{x}_t, \boldsymbol{z}_t, y_{t-1}, y_t \right) \right) - \log Z(\boldsymbol{x}) \tag{471}$$

$$\text{where} \qquad \Psi_t := \exp\{y_{t-1} \boldsymbol{A} y_t + \boldsymbol{x}_t^T \boldsymbol{W} \boldsymbol{z}_t + \boldsymbol{z}_t^T \boldsymbol{V} y_t\} \tag{472}$$

Let $\Upsilon = \{\boldsymbol{W}, \boldsymbol{V}, \boldsymbol{b}, \boldsymbol{c}, \}$ be the set of model parameters. The gradient w.r.t. the data-dependent[121] parameters $v \in \Upsilon$ is given by

$$\frac{\partial \mathcal{L}}{\partial v} = \sum_{t=1}^{T} \left[\sum_{k \in \mathcal{Y}} \left((\mathbb{1}_{y_t = k} - p(y_t = k \mid \boldsymbol{x})) \sum_{h=1}^{H} \sigma(o_{hk}(\boldsymbol{x}_t)) \frac{\partial o_{hk}(\boldsymbol{x}_t)}{\partial v} \right) \right] \tag{473}$$

$$\text{where} \qquad o_{hk}(\boldsymbol{x}_t) = b_h + c_k + V_{hk} + \boldsymbol{w}_h^T \boldsymbol{x}_t \tag{474}$$

Unfortunately, the negative CLL is **non-convex**, and so we are only guaranteed to converge to a *local* maximum of the CLL.

[121] The data-dependent parameters are each individual element of the elements of Υ. "Data" here means $(\boldsymbol{x}, \boldsymbol{y})$. Notice that Υ does not include $\boldsymbol{A}$, $\boldsymbol{\pi}$, or $\boldsymbol{\tau}$.

4.37.1 Detailed Derivations

Unfortunately, the paper leaves out a lot of details regarding derivations and implementations. I'm going to work through them here. First, a recap of the main equations, and with all biases/initial states included. Not leaving anything out[122]

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{\exp\left\{\boldsymbol{y}_1^T \boldsymbol{\pi} + \boldsymbol{y}_T^T \boldsymbol{\tau}\right\}}{Z(\boldsymbol{x})} \prod_{t=1}^{T} \left[\exp\{\boldsymbol{c}^T \boldsymbol{y}_t + \boldsymbol{y}_{t-1} \boldsymbol{A} \boldsymbol{y}_t\} \prod_{h=1}^{H} \left(1 + \exp\{b_h + \boldsymbol{w}_h^T \boldsymbol{x}_t + \boldsymbol{v}_h^T \boldsymbol{y}_t\}\right) \right] \tag{475}$$

$$NLL = -\sum_{i=1}^{N} \log p(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}) \tag{476}$$

$$= -\sum_{i=1}^{N} \left[\sum_{t=1}^{T} \log \left(\sum_{\boldsymbol{z}_t} \psi_t(\boldsymbol{x}_t, \boldsymbol{z}_t, \boldsymbol{y}_{t-1}, \boldsymbol{y}_t) \right) - \log Z(\boldsymbol{x}^{(i)}) \right] \tag{477}$$

The above formula for $p(\boldsymbol{y} \mid \boldsymbol{x})$ implies something that will be very useful:

$$\sum_{\boldsymbol{z}_t} \psi_t(\boldsymbol{x}_t, \boldsymbol{z}_t, \boldsymbol{y}_{t-1}, \boldsymbol{y}_t) = \exp\{\boldsymbol{c}^T \boldsymbol{y}_t + \boldsymbol{y}_{t-1} \boldsymbol{A} \boldsymbol{y}_t\} \prod_{h=1}^{H} \left(1 + \exp\{b_h + \boldsymbol{w}_h^T \boldsymbol{x}_t + \boldsymbol{v}_h^T \boldsymbol{y}_t\}\right) \tag{478}$$

Using the generalization of the product rule for derivatives over N products, we can derive that

$$\frac{\partial}{\partial v} \prod_h (1 + \exp\{o(h)\}) = \left[\prod_h (1 + \exp\{o(h)\}) \right] \left[\sum_h \sigma\left(o\left(h\right)\right) \frac{\partial o(h)}{\partial v} \right] \tag{479}$$

Which means the derivatives of $\sum_z \psi$ for the data-dependent params v_{dat} and transition params v_{tr}, are:

$$\frac{\partial \sum_{\boldsymbol{z}_t} \psi_t}{\partial v_{dat}} = \left[\sum_h \sigma\left(o\left(h, y_t\right)\right) \frac{\partial o(h, y_t)}{\partial v_{dat}} \right] \sum_{\boldsymbol{z}_t} \psi_t \tag{480}$$

$$\frac{\partial \sum_{\boldsymbol{z}_t} \psi_t}{\partial v_{tr}} = \left[\frac{\partial}{\partial v_{tr}} \boldsymbol{c}^T \boldsymbol{y}_t + \boldsymbol{y}_{t-1} \boldsymbol{A} \boldsymbol{y}_t \right] \sum_{\boldsymbol{z}_t} \psi_t \tag{481}$$

which also conveniently means that

$$\frac{\partial}{\partial v_{dat}} \log \left(\sum_{\boldsymbol{z}_t} \psi_t \right) = \sum_h \sigma\left(o\left(h, y_t\right)\right) \frac{\partial o(h, y_t)}{\partial v_{dat}} \tag{482}$$

$$\frac{\partial}{\partial v_{tr}} \log \left(\sum_{\boldsymbol{z}_t} \psi_t \right) = \frac{\partial}{\partial v_{tr}} \boldsymbol{c}^T \boldsymbol{y}_t + \boldsymbol{y}_{t-1} \boldsymbol{A} \boldsymbol{y}_t \tag{483}$$

I'll now proceed to derive the gradients of negative (conditional) log-likelihood for the main parameters. We can save some time by getting the base formula for any of the gradients with

[122] The equation for $p(\boldsymbol{y} \mid \boldsymbol{x})$ from the paper, and thus here, is technically incorrect. The term $\exp\{\boldsymbol{c}^T \boldsymbol{y}_t + \boldsymbol{y}_{t-1} \boldsymbol{A} \boldsymbol{y}_t\}$ should not be included in the product over t for $t = 1$.

respect to a specific single parameter v:

$$\frac{\partial NLL}{\partial v} = -\sum_{i=1}^{N}\left[\sum_{i=1}^{T}\frac{\partial}{\partial v}\log\left(\sum_{\boldsymbol{z}_t}\psi_t(\boldsymbol{x}_t^{(i)},\boldsymbol{z}_t,\boldsymbol{y}_{t-1}^{(i)},\boldsymbol{y}_t^{(i)})\right) - \frac{\partial}{\partial v}\log Z(\boldsymbol{x}^{(i)})\right] \tag{484}$$

$$\frac{\partial \log Z(\boldsymbol{x}^{(i)})}{\partial v} = \frac{1}{Z(\boldsymbol{x}^{(i)})}\frac{\partial}{\partial v}\sum_{\boldsymbol{y}_{\langle 1\ldots T\rangle}}\widetilde{p}(\boldsymbol{y}_1,\ldots,\boldsymbol{y}_T \mid \boldsymbol{x}^{(i)}) \tag{485}$$

$$\frac{\partial}{\partial v}\widetilde{p}(\boldsymbol{y}_1,\ldots,\boldsymbol{y}_T \mid \boldsymbol{x}^{(i)}) = \frac{\partial}{\partial v}\prod_t\sum_{\boldsymbol{z}_t}\psi_t \tag{486}$$

$$= \left[\prod_t\sum_{\boldsymbol{z}_t}\psi_t\right]\left[\sum_t\frac{\frac{\partial}{\partial v}\sum_{\boldsymbol{z}_t}\psi_t}{\sum_{\boldsymbol{z}_t}\psi_t}\right] \tag{487}$$

where I've done some regrouping on the last line to be more gradient-friendly.

Data-dependent parameters

All params v that are not transition params.

$$\frac{\partial NLL}{\partial v_{dat}} = -\sum_{i=1}^{N}\left[\sum_{t=1}^{T}\frac{\partial}{\partial v_{dat}}\log\left(\sum_{\boldsymbol{z}_t}\psi_t\right) - \frac{\partial}{\partial v}\log Z(\boldsymbol{x}^{(i)})\right] \tag{488}$$

$$= -\sum_{i=1}^{N}\left[\sum_t\sum_h\sigma\left(o\left(h,y_t\right)\right)\frac{\partial o(h,y_t)}{\partial v_{dat}} - \frac{1}{Z(\boldsymbol{x}^{(i)})}\sum_{\boldsymbol{y}_{\langle 1\ldots T\rangle}}\left[\prod_t\sum_{\boldsymbol{z}_t}\psi_t\right]\left[\sum_t\frac{\frac{\partial}{\partial v}\sum_{\boldsymbol{z}_t}\psi_t}{\sum_{\boldsymbol{z}_t}\psi_t}\right]\right] \tag{489}$$

$$= -\sum_{i=1}^{N}\left[\sum_t\sum_h\sigma\left(o\left(h,y_t\right)\right)\frac{\partial o(h,y_t)}{\partial v_{dat}} - \frac{1}{Z(\boldsymbol{x}^{(i)})}\sum_{\boldsymbol{y}_{\langle 1\ldots T\rangle}}\left[\prod_t\sum_{\boldsymbol{z}_t}\psi_t\right]\left[\sum_t\sum_h\sigma\left(o\left(h,y_t\right)\right)\frac{\partial o(h,y_t)}{\partial v_{dat}}\right]\right] \tag{490}$$

$$= -\sum_{i=1}^{N}\left[\sum_t\sum_h\sigma\left(o\left(h,y_t\right)\right)\frac{\partial o(h,y_t)}{\partial v_{dat}} - \sum_t\sum_y\sum_{y'}\left[\sum_h\sigma\left(o\left(h,y_t\right)\right)\frac{\partial o(h,y_t)}{\partial v_{dat}}\right]\xi_{t,y,y'}\right] \tag{491}$$

$$= -\sum_{i=1}^{N}\left[\sum_t\sum_h\sigma\left(o\left(h,y_t\right)\right)\frac{\partial o(h,y_t)}{\partial v_{dat}} - \sum_t\sum_y\left[\sum_h\sigma\left(o\left(h,y_t\right)\right)\frac{\partial o(h,y_t)}{\partial v_{dat}}\right]\gamma_{t,y}\right] \tag{492}$$

$$= -\sum_{i=1}^{N}\left[\sum_t\left(\sum_h\sigma\left(o\left(h,y_t\right)\right)\frac{\partial o(h,y_t)}{\partial v_{dat}} - \sum_y\left[\sum_h\sigma\left(o\left(h,y_t\right)\right)\frac{\partial o(h,y_t)}{\partial v_{dat}}\right]\gamma_{t,y}\right)\right] \tag{493}$$

$$= -\sum_{i=1}^{N}\left[\sum_t\sum_y\left(\left(\mathbb{1}_{y_t=y}-\gamma_{t,y}\right)\sum_h\sigma\left(o(h,y_t)\right)\frac{\partial o(h,y_t)}{\partial v_{dat}}\right)\right] \tag{494}$$

NOTE: Although I haven't thoroughly checked the last few steps, they are required to be true in order to match the paper's results.

Transition parameters

$$\frac{\partial NLL}{\partial v_{tr}} = -\sum_{i=1}^{N}\left[\sum_{t=1}^{T}\frac{\partial}{\partial v_{tr}}\log\left(\sum_{\boldsymbol{z}_t}\psi_t\right) - \frac{\partial}{\partial v_{tr}}\log Z(\boldsymbol{x}^{(i)})\right] \tag{495}$$

$$= -\sum_{i}^{N}\left[\sum_{t}\frac{\partial}{\partial v_{tr}}\left[\boldsymbol{c}^T\boldsymbol{y}_t + \boldsymbol{y}_{t-1}\boldsymbol{A}\boldsymbol{y}_t\right] - \frac{\partial}{\partial v_{tr}}\log Z(\boldsymbol{x}^{(i)})\right] \tag{496}$$

$$= -\sum_{i=1}^{N}\left[\sum_{t}\frac{\partial}{\partial v_{tr}}\left[\boldsymbol{c}^T\boldsymbol{y}_t + \boldsymbol{y}_{t-1}\boldsymbol{A}\boldsymbol{y}_t\right] - \frac{1}{Z(\boldsymbol{x}^{(i)})}\sum_{\boldsymbol{y}_{\langle 1\ldots T\rangle}}\left[\prod_t\sum_{\boldsymbol{z}_t}\psi_t\right]\left[\sum_t\frac{\partial}{\partial v_{tr}}\left[\boldsymbol{c}^T\boldsymbol{y}_t + \boldsymbol{y}_{t-1}\boldsymbol{A}\boldsymbol{y}_t\right]\right]\right] \tag{497}$$

$$= -\sum_{i=1}^{N}\left[\sum_t\sum_y\left(\left(\mathbb{1}_{y_t=y} - \gamma_{t,y}\right)\frac{\partial}{\partial v_{tr}}\left[\boldsymbol{c}^T\boldsymbol{y}_t + \boldsymbol{y}_{t-1}\boldsymbol{A}\boldsymbol{y}_t\right]\right)\right] \tag{498}$$

Boundary parameters

$$\frac{\partial NLL}{\partial \pi_\ell} = -\sum_{i=1}^{N}\left[\mathbb{1}_{y_1=\ell} - \gamma_{1,\ell}\right] \tag{499}$$

$$\frac{\partial NLL}{\partial \tau_\ell} = -\sum_{i=1}^{N}\left[\mathbb{1}_{y_T=\ell} - \gamma_{T,\ell}\right] \tag{500}$$

The results of each of the boxes above are summarized below, for the case of $N = 1$ to save space.

$$\frac{\partial NLL}{\partial v_{dat}} = -\sum_t\sum_y\left(\left(\mathbb{1}_{y_t=y} - \gamma_{t,y}\right)\sum_h\sigma\left(o(h,y)\right)\frac{\partial o(h,y)}{\partial v_{dat}}\right) \tag{501}$$

$$\frac{\partial NLL}{\partial v_{tr}} = -\sum_t\sum_y\left(\left(\mathbb{1}_{y_t=y} - \gamma_{t,y}\right)\frac{\partial}{\partial v_{tr}}\left[\boldsymbol{c}^T\boldsymbol{y}_t + \boldsymbol{y}_{t-1}\boldsymbol{A}\boldsymbol{y}_t\right]\right) \tag{502}$$

$$\frac{\partial NLL}{\partial \pi_\ell} = -\sum_{i=1}^{N}\left[\mathbb{1}_{y_1=\ell} - \gamma_{1,\ell}\right] \tag{503}$$

$$\frac{\partial NLL}{\partial \tau_\ell} = -\sum_{i=1}^{N}\left[\mathbb{1}_{y_T=\ell} - \gamma_{T,\ell}\right] \tag{504}$$

Now I'll further go through and show how the equations simplify for each type of data-

dependent parameter.

$$\frac{\partial NLL}{\partial W_{c,h}} = -\sum_t \sum_y \left((\mathbb{1}_{y_t=y} - \gamma_{t,y}) \sum_{h'} \sigma\left(o(h', y)\right) \frac{\partial}{\partial W_{c,h}} \left(b_{h'} + c_y + V_{h',y} + \boldsymbol{w}_{h'}^T \boldsymbol{x}_t \right) \right) \tag{505}$$

$$= -\sum_t \sum_y \left((\mathbb{1}_{y_t=y} - \gamma_{t,y}) \sum_{h'} \sigma\left(o(h, y)\right) \mathbb{1}_{h=h'} \mathbb{1}_{c \in \boldsymbol{x}_t} \right) \tag{506}$$

$$= -\sum_t \sum_y (\mathbb{1}_{y_t=y} - \gamma_{t,y}) \sigma\left(o(h, y)\right) \mathbb{1}_{c \in \boldsymbol{x}_t} \tag{507}$$

$$\frac{\partial NLL}{\partial V_{h,y}} = -\sum_t (\mathbb{1}_{y_t=y} - \gamma_{t,y}) \sigma\left(o(h, y)\right) \tag{508}$$

$$\frac{\partial NLL}{\partial b_h} = -\sum_t \sum_y (\mathbb{1}_{y_t=y} - \gamma_{t,y}) \sigma\left(o(h, y)\right) \tag{509}$$

$$\frac{\partial NLL}{\partial c_y} = -\sum_t (\mathbb{1}_{y_t=y} - \gamma_{t,y}) \sum_h \sigma\left(o(h, y)\right) \tag{510}$$

$$\tag{511}$$

Alternative Approach. The above was a bit more cumbersome than needed. I'll now derive it in an easier way.

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{\exp\left\{\boldsymbol{y}_1^T \boldsymbol{\pi} + \boldsymbol{y}_T^T \boldsymbol{\tau}\right\}}{Z(\boldsymbol{x})} \prod_{t=1}^{T} \left[\exp\{\boldsymbol{c}^T \boldsymbol{y}_t + \boldsymbol{y}_{t-1} \boldsymbol{A} \boldsymbol{y}_t\} \prod_{h=1}^{H} \left(1 + \exp\{b_h + \boldsymbol{w}_h^T \boldsymbol{x}_t + \boldsymbol{v}_h^T \boldsymbol{y}_t\}\right) \right] \tag{512}$$

$$= \frac{\exp\{I + T\}}{Z(\boldsymbol{x})} \prod_{t=1}^{T} \prod_{h=1}^{H} \left(1 + \exp\{b_h + \boldsymbol{w}_h^T \boldsymbol{x}_t + \boldsymbol{v}_h^T \boldsymbol{y}_t\}\right) \tag{513}$$

$$NLL = -\sum_{i=1}^{N} \log p(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}) \tag{514}$$

$$= -\sum_{i=1}^{N} \left[I + T + \sum_{t=1}^{T} \sum_{h=1}^{H} \left[\log\left(1 + \exp\{b_h + \boldsymbol{w}_h^T \boldsymbol{x}_t + \boldsymbol{v}_h^T \boldsymbol{y}_t\}\right) \right] - \log Z(\boldsymbol{x}^{(i)}) \right] \tag{515}$$

Now, focusing on the regular log-likelihood for a single example, we have

$$\frac{\partial \mathcal{L}_i}{\partial v} = \frac{\partial}{\partial v} \log p(\boldsymbol{y}^{(i)}, \boldsymbol{x}^{(i)}) \tag{516}$$

$$= \frac{\partial}{\partial v} \left[I + T + \sum_{t,h} \log\left(1 + \exp\left\{b_h + \boldsymbol{w}_h^T \boldsymbol{x}_t + \boldsymbol{v}_h^T \boldsymbol{y}_t\right\}\right) - \log Z(\boldsymbol{x}^{(i)}) \right] \tag{517}$$

$$\frac{\partial \log Z(\boldsymbol{x}^{(i)})}{\partial v} = \frac{1}{Z(\boldsymbol{x}^{(i)})} \sum_{\boldsymbol{y}_{\langle 1 \ldots T \rangle}} \frac{\partial}{\partial v} \widetilde{p}(\boldsymbol{y} \mid \boldsymbol{x}^{(i)}) \tag{518}$$

$$= \frac{1}{Z(\boldsymbol{x}^{(i)})} \sum_{\boldsymbol{y}_{\langle 1 \ldots T \rangle}} \widetilde{p}(\boldsymbol{y} \mid \boldsymbol{x}^{(i)}) \frac{\partial}{\partial v} \log \widetilde{p}(\boldsymbol{y} \mid \boldsymbol{x}^{(i)}) \tag{519}$$

as our base formula for partial derivatives.

Transition parameters

$$\frac{\partial \mathcal{L}_i}{\partial A_{i,j}} = \frac{\partial}{\partial A_{i,j}} \left[I + T + \sum_{t,h} \log \left(1 + \exp \left\{ b_h + \boldsymbol{w}_h^T \boldsymbol{x}_t + \boldsymbol{v}_h^T \boldsymbol{y}_t \right\} \right) - \log Z(\boldsymbol{x}^{(i)}) \right] \tag{520}$$

$$= \frac{\partial}{\partial A_{i,j}} [I + T] - \sum_{\boldsymbol{y}_{\langle 1 \dots T \rangle}} p(\boldsymbol{y} \mid \boldsymbol{x}^{(i)}) \frac{\partial}{\partial A_{i,j}} \left[y_1^T \pi + y_T^T \tau + \sum_t y_{t-1} A y_t \right] \tag{521}$$

$$= \sum_t \mathbb{1}_{y_{t-1}^{(i)}=i} \mathbb{1}_{y_t^{(i)}=j} - \sum_{\boldsymbol{y}_{\langle 1 \dots T \rangle}} \frac{1}{Z(\boldsymbol{x}^{(i)})} \widetilde{p}(\boldsymbol{y} \mid \boldsymbol{x}^{(i)}) \sum_{t=1}^{T} \mathbb{1}_{y_{t-1}=i} \mathbb{1}_{y_t=j} \tag{522}$$

$$= \sum_t \mathbb{1}_{y_{t-1}^{(i)}=i} \mathbb{1}_{y_t^{(i)}=j} - \sum_t \sum_{y_t} \sum_{y_{t-1}} \mathbb{1}_{y_{t-1}=i} \mathbb{1}_{y_t=j} \sum_{\boldsymbol{y}_{\langle 1 \dots t-2 \rangle}} \sum_{\boldsymbol{y}_{\langle t+1 \dots T \rangle}} \frac{1}{Z(\boldsymbol{x}^{(i)})} \widetilde{p}(\boldsymbol{y} \mid \boldsymbol{x}^{(i)}) \tag{523}$$

Pre-training of Hidden-Unit CRFs

Written by Brandon McKinzie

Kim et al., "Pre-training of Hidden-Unit CRFs," (2018).

Model Definition. The Hidden-Unit CRF (HUCRF) accepts the usual observation sequence $\boldsymbol{x} = x_1, \ldots, x_n$, and associated label sequence $\boldsymbol{y} = y_1, \ldots y_n$ for training. The HUCRF also has a hidden layer of binary-valued $\boldsymbol{z} = z_1 \ldots z_n$. It defines the joint probability

$$p_{\theta,\gamma}(\boldsymbol{y}, \boldsymbol{z} \mid \boldsymbol{x}) = \frac{\exp\left(\boldsymbol{\theta}^T \Phi(\boldsymbol{x}, \boldsymbol{z}) + \boldsymbol{\gamma}^T \Psi(\boldsymbol{z}, \boldsymbol{y})\right)}{\sum_{\boldsymbol{z}', \boldsymbol{y}' \in \mathcal{Y}(\boldsymbol{x}, \boldsymbol{z}')} \exp\left(\boldsymbol{\theta}^T \Phi(\boldsymbol{x}, \boldsymbol{z}') + \boldsymbol{\gamma}^T \Psi(\boldsymbol{z}', \boldsymbol{y}')\right)} \tag{524}$$

where

- $\mathcal{Y}(\boldsymbol{x}, \boldsymbol{z})$ is the set of all possible label sequences for $\boldsymbol{x}$ and $\boldsymbol{z}$.
- $\Phi(\boldsymbol{x}, \boldsymbol{z}) = \sum_{j=1}^{n} \phi(x, j, z_j)$
- $\Psi(\boldsymbol{z}, \boldsymbol{y}) = \sum_{j=1}^{n} \psi(z_j, y_{j-1}, y_j)$.

Also note that we model $(z_i \perp z_{j \neq i} \mid \boldsymbol{x}, \boldsymbol{y})$.

Pre-training HUCRFs. Since the objective for HUCRFs is non-convex, we should choose a better initialization method than random initialization. This is where pre-training comes in, a simple 2-step approach:

1. Cluster observed tokens from M unlabeled sequences and treat the clusters as labels to train an intermediate HUCRF. Let $C(u^{(i)})$ be the sequence of cluster assignments/labels for the unlabeled sequence $u^{(i)}$. We compute:

$$(\theta_1, \gamma_1) \approx \arg\max_{\theta,\gamma} \sum_{i=1}^{M} \log p_{\theta,\gamma}(C(u^{(i)}) \mid u^{(i)}) \tag{525}$$

2. Train a final model on the labeled data $\{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$, using θ_1 as an initialization point:

$$(\theta_2, \gamma_2) \approx \arg\max_{\theta,\gamma} \sum_{i=1}^{N} \log p_{\theta,\gamma}(y^{(i)} \mid x^{(i)}) \tag{526}$$

Note that pre-training only defines the initialization for θ, the parameters between $\boldsymbol{x}$ and $\boldsymbol{z}$. We still train γ, the parameters from $\boldsymbol{z}$ to $\boldsymbol{y}$, from scratch.

Canonical Correlation Analysis (CCA). A general technique that we will need to understand as a prerequisite for the multi-sense clustering approach (defined in the next section). Given n samples of the form $(x^{(i)}, y^{(i)})$, where each $x^{(i)} \in \{0,1\}^d$ and $y^{(i)} \in \{0,1\}^{d'}$, CCA returns *projection matrices* $A \in \mathbb{R}^{d \times k}$ and $B \in \mathbb{R}^{d' \times k}$ that we can use to project the samples to k dimensions:

$$x \longrightarrow A^T x \tag{527}$$

$$y \longrightarrow B^T y \tag{528}$$

The CCA algorithm is outlined below.

Algorithm: CCA

1. Calculate $\boldsymbol{D} \in \mathbb{R}^{d \times d'}$, $\boldsymbol{u} \in \mathbb{R}^d$, and $\boldsymbol{v} \in \mathbb{R}^{d'}$ as follows:

$$D_{i,j} = \sum_{l=1}^{n} \mathbb{1}_{x_i^{(l)}=1} \mathbb{1}_{y_j^{(l)}=1} \tag{529}$$

$$u_i = \sum_{l=1}^{n} \mathbb{1}_{x_i^{(l)}=1} \tag{530}$$

$$v_i = \sum_{l=1}^{n} \mathbb{1}_{y_i^{(l)}=1} \tag{531}$$

2. Define $\hat{\Omega} = \text{diag}(\boldsymbol{u})^{-1/2}\ \boldsymbol{D}\ \text{diag}(\boldsymbol{v})^{-1/2}$.
3. Calculate rank-k SVD $\hat{\Omega}$. Let $\boldsymbol{U} \in \mathbb{R}^{d \times k}$ and $\boldsymbol{V} \in \mathbb{R}^{d' \times k}$ contain the left and right, respectively, singular vectors for the largest k singular values.
4. Return $A = \text{diag}(\boldsymbol{u})^{-1/2}\boldsymbol{U}$ and $B = \text{diag}(\boldsymbol{v})^{-1/2}\boldsymbol{V}$.

Multi-sense clustering. For each word type, use CCA to create a set of context embeddings corresponding to all occurrences of that word type. Then, cluster these embeddings with k-means. Set the number of word senses k to the number of label types occurring in the labeled data.

TODO: finish this note

Structured Attention Networks

Written by Brandon McKinzie

Kim et al., "Structured Attention Networks," (2017).

Background: Attention Networks. Goal: produce a context c based on input sequence x and query q. We assume we have an attention distribution[123] $z \sim p(z \mid x, q)$. Interpret z as a categorical latent variable over T categories, where $T = len(x)$ is the length of the input sequence. We can then compute the context, $c = \mathbb{E}_{z \sim p(z|x,q)}\left[f(x, z)\right]$, where $f(x, z)$ is an *annotation function*[124].

> *We interpret the attention mechanism as taking the expectation of an annotation function $f(x, z)$ with respect to a latent variable $z \sim p$, where p is parameterized to be a function of x and q.*

For comparisons later on with the traditional attention mechanism, here it is:

$$c = \sum_{t}^{T} p(z = t \mid x, q)\boldsymbol{x}_t \tag{532}$$

$$p(z = t \mid x, q) = \text{softmax}(\theta_t) \tag{533}$$

where usually x is the sequence of hidden state of the encoder RNN, q is the hidden state of the decoder RNN at the most recent time step, z gives the source position to be attended to, and $\theta_t = \text{score}(x_t, q)$.

Structured Attention. In a **structured attention model**, z is now a *vector* of discrete random variables $z_1, \ldots, z_m$ and the attention distribution $p(z \mid x, q)$ is now modeled as a *conditional random field*, specifying the structure of the z variables. We also assume now that the annotation function f factors into clique annotation functions $f(x, z) = \sum_C f_C(x, z_C)$, where the summation is over the C factors, ψ_C, of the CRF. Our context vector takes the form:

$$c = \mathbb{E}_{z \sim p(z|x,q)}\left[f(x, z)\right] = \sum_{C} \mathbb{E}_{z \sim p(z_C|x,q)}\left[f_C(x, z_C)\right] \tag{534}$$

$$p(z \mid x, q) = \frac{1}{Z(x, q)} \prod_{C} \psi_C(z_C) \tag{535}$$

[123] Also called the "alignments". It is the output of the softmax layer of attention scores in the majority of cases.

[124] In all applications I've seen, $f(x, z) = x_z$.

Example 1: Subsequence Selection. Let $m = T$, and let each $z_i \in \{0, 1\}$ be a binary R.V. Let $f(x, z) = \sum_t^T f_t(x, z_t) = \sum_t^T \boldsymbol{x}_t \mathbb{1}_{z_t=1}$[125]. This yields the context vector,

$$c = \mathbb{E}_{z_1,\ldots,z_T}[f(x, z)] = \sum_t^T p(z_t = 1 \mid x, q)\boldsymbol{x}_t \tag{536}$$

Although this looks similar to equation 532, we haven't yet revealed the functional form for $p(z \mid x, q)$. Two possible choices:

$$\textbf{Linear-Chain CRF:} \quad p(z_1, \ldots, z_T \mid x, q) = \frac{1}{Z(x, q)} \prod_t^T \psi_t(z_t, z_{t-1}) \tag{537}$$

$$\textbf{Bernoulli:} \quad p(z_1, \ldots, z_T \mid x, q) = \prod_t^T p(z_t = 1 \mid x, q) = \prod_t^T \sigma(\psi_t(z_t)) \tag{538}$$

These show why equation 536 is fundamentally different than equation 532:

- It allows for multiple inputs (or no inputs) to be selected for a given query.
- We can incorporate structural dependencies across the z_t's.

Also note that all methods can use potentials from the same neural network or RNN that takes x and q as input. By this we mean, for example, that we can take the same parameters we'd use when computing the scores in our attention layer, and reinterpret them as e.g. CRF parameters. Then, we can compute the marginals $p(z_t \mid x)$ using the forward-backward algorithm[126].

> *Crucially this generalization from vector softmax to forward-backward is just a* ***series of differentiable steps****, and we can compute gradients of its output (marginals) with respect to its input (potentials), allowing the structured attention model to be trained end-to-end as part of a deep model.*

[125] Ok, so equivalently, $z^T x$, i.e. the indicator function can just be replace by z_t here

[126] This is different than the simple softmax we usually use in an attention layer, which does not model any interdependencies between the z_t. The marginals we end up with when using the CRF originate from a *joint* distribution over the entire sequence $z_1, \ldots z_T$. This seems potentially incredibly powerful. Need to analyze in depth.

Do and Artieres, "Neural Conditional Random Fields," (2010).

Neural CRFs. Essentially, we feed the input sequence $\boldsymbol{x}$ through a feed-forward network whose output layer has a linear activation function. The output layer is then connected with the target variable sequence $\mathbf{Y}$. In other words, instead of feeding instances $\boldsymbol{x}$ of the observation variables $\mathbf{X}$, we feed the hidden layer activations of the NN. This results in the conditional probability

$$p(\boldsymbol{y} \mid \boldsymbol{x}) \propto \prod_{c \in C} e^{-E_c(\boldsymbol{x}, \boldsymbol{y}_c, \boldsymbol{w})} = \prod_{c \in C} e^{\langle \boldsymbol{w}_c^{\boldsymbol{y}_c}, \boldsymbol{\Phi}_c(\boldsymbol{x}, \boldsymbol{w}_{NN}) \rangle} \tag{539}$$

We can set Φ
shared-weigh

where

- $\boldsymbol{w}_{NN}$ are the weights for the NN.
- $\boldsymbol{w}_c^{\boldsymbol{y}_c}$ are the weights (for clique c) for the CRF.
- $\boldsymbol{\Phi}_c(\boldsymbol{x}, \boldsymbol{w}_{NN})$ is the output of the NN. It symbolizes the high-level feature representation of the input $\boldsymbol{x}$ at clique c computed by the NN.

The authors refer to the linear output layer (containing the CRF weights) as the *energy outputs*. For the sake of writing this in more familiar notation for the linear-chain CRF case, here is the above equation translated for the case where each clique corresponds to a timestep t of the input sequence and is either a label-label clique or a state-label clique.

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{1}{Z(x)} \prod_t^T \exp\left\{-E_t(\boldsymbol{x}, \boldsymbol{y}_t, \boldsymbol{y}_{t-1}, \boldsymbol{w})\right\} \tag{540}$$

$$= \frac{1}{Z(x)} \prod_t^T \exp\left\{-E_{loc}(\boldsymbol{x}, t, y_t, \boldsymbol{w}) - E_{trans}(\boldsymbol{x}, t, y_{t-1}, y_t, \boldsymbol{w})\right\} \tag{541}$$

$$\tag{542}$$

where the authors are using a blanket $\boldsymbol{w}$ to denote all model parameters[127].

[127] Also note that the authors allow for utilizing the input sequence $\boldsymbol{x}$ in the transition energy function, E_{trans}, although usually we implement E_{trans} using only y_{t-1} and y_t.

Initialization & Fine-Tuning. The hidden layers of the NN are initialized layer-by-layer in an unsupervised manner using RBMs. It's important to note that the hidden layers of the NN consist of binary units. Then, using the pre-trained hidden layers, the CRF layer is initialized by training it in the usual way, and keeping the pretrained NN weights fixed.

Next, fine-tuning is used to learn all parameters globally.

$$\frac{\partial L(\boldsymbol{w})}{\partial \boldsymbol{w}} = \frac{1}{n}\sum_{i=1}^{n}\frac{\partial L_i(\boldsymbol{w})}{\partial \boldsymbol{w}} \tag{543}$$

$$\frac{\partial L_i(\boldsymbol{w})}{\partial \boldsymbol{w}} = \frac{\partial L_i(\boldsymbol{w})}{\partial \boldsymbol{E}(\boldsymbol{x}^{(i)})}\frac{\partial \boldsymbol{E}(\boldsymbol{x}^{(i)})}{\partial \boldsymbol{w}} \tag{544}$$

$$\Big[\boldsymbol{E}(\boldsymbol{x}^{(i)})\Big]_t = E_{loc}(\boldsymbol{x}, t, y_t, \boldsymbol{w}) + E_{trans}(\boldsymbol{x}, t, y_{t-1}, y_t, \boldsymbol{w}) \tag{545}$$

$$\tag{546}$$

where $\frac{\partial \boldsymbol{E}_i}{\partial \boldsymbol{w}}$ is the Jacobian matrix of the NN outputs for input sequence $\boldsymbol{x}^{(i)}$ w.r.t. weights $\boldsymbol{w}$. By setting $\frac{\partial L_i(\boldsymbol{w})}{\partial \boldsymbol{E}_i}$ as backprop errors of the NN output units, we can backpropagate and get $\frac{\partial L_i(\boldsymbol{w})}{\partial \boldsymbol{w}}$ using the chain rule over the hidden layers.

Bidirectional LSTM-CRF Models for Sequence Tagging

Written by Brandon McKinzie

Huang et al., "Bidirectional LSTM-CRF Models for Sequence Tagging," (2015).

BI-LSTM-CRF Networks. Consider the matrix of scores $f_\theta(\boldsymbol{x})$ for input sentence $\boldsymbol{x}$. The element $[f_\theta]_{\ell,t}$ gives the score for label ℓ with the t-th word. This is output by the LSTM network parameterized by θ. We let $[A]_{i,j}$ denote the transition score from label i to label j within the CRF. The total set of parameters is denoted $\widetilde{\theta} = \theta \cup \boldsymbol{A}$. The total score for input sentence $\boldsymbol{x}$ and predicted label sequence y is then

$$s(\boldsymbol{x}, \boldsymbol{y}, \widetilde{\theta}) = \sum_t^T \left(A_{y_{t-1}, y_t} + [f_\theta]_{y_t, t} \right) \tag{547}$$

Features. The authors incorporate 3 distinct types of input features:

- **Spelling features**. Various standard lexical/syntactical features. One-hot encoded as usual.
- **Context features**. Unigram, bigram, and sometimes tri-gram features. One-hot encoded.
- **Word embeddings**. Distinct from the word features. Use a pretrained embedding for each word.

Although it's not entirely clear, it appears they concatenate all of the aforementioned features together as input to the BI-LSTM. This necessarily means they are learning an embedding for the one-hot encoded spelling and word features. They also add direct connections from the input to the CRF for the spelling and word features.

EDIT: they may actually replace the one-hot encoded word features with the word embeddings. Unclear.

Pawar et al., "Relation Extraction: A Survey," (December 2017).

Feature-based Methods. For each entity pair (e_1, e_2), generate a set of features and train a classifier to predict the relation[128]. Some useful features are shown in the figure below.

Feature Types	Example
Words: Words of both the mentions and all the words in between	`M11_leaders`, `M21_Venice`; `B1_of`, `B2_Italy`, `B3_'s`, `B4_left-wing`, `B5_government`, `B6_were`, `B7_in`
Entity Types: Entity types of both the mentions	`E1_PERSON`, `E2_GPE`
Mention Level: Mention types (NAME, NOMINAL or PRONOUN) of both the mentions	`M1_NOMINAL`, `M2_NAME`
Overlap: #words separating the two mentions, #other mentions in between, flags indicating whether the two mentions are in the same NP, VP or PP	`7_Words_Apart`, `2_Mentions_In_Between` (`Italy & government`), `Not_Same_NP`, `Not_Same_VP`, `Not_Same_PP`
Dependency: Words, POS and chunk labels of words on which the mentions are dependent in the dependency tree, #links traversed in dependency tree to go from one mentions to another	`M1W_were`, `M1P_VBD`, `M1C_VP`, `M2W_in`, `M2P_IN`, `M2C_PP`, `DepLinks_3`
Parse Tree: Path of non-terminals connecting the two mentions in the parse tree, and the path annotated with head words	`PERSON-NP-S-VP-PP-GPE`, `PERSON-NP:leaders-S -VP:were-PP:in-GPE`

Authors found that SVMs outperform MaxEnt (logistic reg) classifiers for this task.

Kernel methods. Instead of explicit feature engineering, we can design kernel functions for computing similarities between representations of two relation instances[129] (a relation instance is a triplet of the form (e_1, e_2)), and SVM for the classification.

[128] If there are N unique relations for our data, it is common to train the classifier to predict $2N$ total relations, to handle both possible orderings of relation arguments.

[129] Recall that kernel methods are for the general optimization problem

$$\min_w \sum_{i=1}^{n} \text{loss}\left(\sum_{j=1}^{n} \alpha_j \phi(x_j)^T \phi(x_i), y_i\right) + \lambda \sum_{j=1}^{n}\sum_{k=1}^{n} \alpha_j \alpha_k \phi(x_j)^T \phi(x_k) \tag{548}$$

which changes the direct focus from feature engineering to "similarity" engineering.

One approach is the **sequence kernel**. We represent each relation instance as a sequence of feature vectors:

$$(e_1, e_2) \to (\boldsymbol{f}_1, \ldots, \boldsymbol{f}_N)$$

where N might be e.g. the number of words between the two entities, and the dimension of each f is the same, and could correspond to e.g. POS tag, NER tag, etc. More formally, define the *generalized subsequence kernel*, $K_n(s, t, \lambda)$, that computes some number of weighted subsequences u such that

- There exist index sequences $ii := (i_1, \ldots i_n)$ and $jj := (j_1, \ldots j_n)$ of length n such that

$$u_i \in \boldsymbol{s}_i \qquad \forall i \in ii \tag{549}$$

$$u_j \in \boldsymbol{t}_j \qquad \forall j \in jj \tag{550}$$

$$\tag{551}$$

- The weight of u is $\lambda^{l(ii)+l(jj)}$, where $l(x) = \max(x) - \min(x)$ and $0 < \lambda < 1$. Sparser (more spaced out) subsequences get lower weight.

The authors then provide the recursion formulas for K, and describe some extensions of sequence kernels for relation extraction.

Syntactic Tree Kernels. Structural properties of a sentence are encoded by its constituent parse tree. The tree defines the syntax of the sentence in terms of constituents such as noun phrases (NP), verb phrases (VP), prepositional phrases (PP), POS tags (NN, VB, IN, etc.) as non-terminals and actual words as leaves. The syntax is usually governed by Context Free Grammar (CFG). Constructing a constituent parse tree for a given sentence is called *parsing*. The **Convolution Parse Tree Kernel** K_T can be used for computing similarity between two syntactic trees.

Dependency Tree Kernels. For grammatical relations between words in a sentence. Words are the nodes and dependency relations are the edges (in the tree), typically from dependent to parent. In the **relation instance representation**, we use the smallest subtree containing the entity pair of a given sentence. Each node is augmented with additional features like POS, chunk, entity level (name, nominal, pronoun), hypernyms, relation argument, etc. Formally, an *augmented dependency tree* is defined as a tree T where

- Each node t_i has features $\phi(t_i) = \{v_1, \ldots, v_d\}$.
- Let $t_i[\boldsymbol{c}]$ denote all children of t_i, and let $Pa(t_i)$ denote its parent.
- For comparison of two nodes we use:
  - **Matching function** $m(t_i, t_j)$: equal to 1 if some important features are shared between t_i and t_j, else 0.
  - **Similarity function** $s(t_i, t_j)$: returns a positive real similarity score, and defined as

$$s(t_i, t_j) = \sum_{v_q \in \phi(t_i)} \sum_{v_r \in \phi(t_j)} \text{Compat}(v_q, v_r) \tag{552}$$

    over some compatibility function between two feature values.

Finally, we can define the overall dependency tree kernel $K(T_1, T_2)$ for similarity between trees T_1 and T_2 as follows. Let r_i denote the root node of tree T_i.

$$K(T_1, T_2) = \begin{cases} 0 & \text{if } m\left(r_1, r_2\right) = 0 \\ s(r_1, r_2) + K_c(r_1[\boldsymbol{c}], r_2[\boldsymbol{c}]) & \text{otherwise} \end{cases} \tag{553}$$

$$K_c(t_i[\boldsymbol{c}], t_j[\boldsymbol{c}]) = \sum_{\boldsymbol{a},\boldsymbol{b},l(\boldsymbol{a})=l(\boldsymbol{b})} \lambda^{d(\boldsymbol{a})+d(\boldsymbol{b})} K(t_i[\boldsymbol{a}], t_j[\boldsymbol{b}]) \tag{554}$$

The interpretation is that, whenever a pair of matching nodes is found, *all* possible matching subsequences[130] of their children are found. Two subsequences $\boldsymbol{a}$ and $\boldsymbol{b}$ are said to "match" if $m(a_i, b_1) = 1(\forall i < n)$. Similar to the sequence kernel seen earlier, λ is a decay factor that penalizes sparser subsequences.

[130]Note that a summation over subsequences of a sequence $\boldsymbol{a}$, denoted here as $\sum_{\boldsymbol{a}}$, expands to $\{a_1, \ldots a_n, a_1a_2, a_1a_3, \ldots a_1a_n, a_1a_2a_3, \ldots, a_2a_5a_6, \ldots\}$ and so on and so forth.

Neural Relation Extraction with Selective Attention over Instances

Written by Brandon McKinzie

Lin et al., "Neural Relation Extraction with Selective Attention over Instances," (2016).

Introduction. A common distant supervision approach for RE is aligning a KB with text. For any (e_1, r, e_2) in the KB, it assumes that all text mentions of (e_1, e_2) express the relation r. Of course, this assumption will often not be true. This motivates the notion of **multi-instance learning**, wherein we predict whether a set of instances $\{x_1, \dots x_n\}$ (each of which contain mention(s) of (e_1, e_2)) imply the existence of (e_1, r, e_2) being true.

Input Representation. Each instance sentence x is tokenized into a sequence of words. Each word w_i is transformed into a concatenation $\boldsymbol{w}_i \in \mathbb{R}^d$ $(d = d_w + 2d_{pos})$,

$$\boldsymbol{w}_i := [\text{word2Vec}(w_i); \text{dist}(w_i, e_1); \text{dist}(w_i, e_2)] \tag{555}$$

where dist(a, b) returns the [embedded] relative distance (num tokens) between a and b in the given sentence (positive integer)[131].

Convolutional Network. We use a CNN to encode a sentence of embeddings $\{\boldsymbol{w}_1, \dots, \boldsymbol{w}_T\}$ into a single sentence vector representation $\boldsymbol{x}$. Denote the kernel/filter/window size as ℓ and the number of words in the given sentence as T. Let $\boldsymbol{q}_i \in \mathbb{R}^{\ell \cdot d}$ denote the vector for the ith window,

$$\boldsymbol{q}_i = \boldsymbol{w}_{i-\ell+1:i} \qquad (1 \leq i \leq T + \ell - 1) \tag{556}$$

and let $\boldsymbol{Q} \in \mathbb{R}^{(T+\ell-1)\times \ell \cdot d}$ be defined such that row $\boldsymbol{Q}_i = \boldsymbol{q}_i^T$. It follows that, for convolution matrix $\boldsymbol{W} \in \mathbb{R}^{K \times (\ell \cdot d)}$, the output of the kth filter, and subsequent max-pooling, is

$$\boldsymbol{p}_k = \left[\boldsymbol{W}\boldsymbol{Q}^T\right]_k + \boldsymbol{b} \tag{557}$$

$$[\boldsymbol{x}]_k = [\max(\boldsymbol{p}_{k1}); \max(\boldsymbol{p}_{k2}); \max(\boldsymbol{p}_{k3})] \tag{558}$$

where we've divided $\boldsymbol{p}_k$ into three segments, corresponding to before entity 1, middle, and after entity 2 of the given sentence. The sentence vector $\boldsymbol{x} \in \mathbb{R}^{3K}$ is the concatenation of all of these, after feeding through a non-linear activation function like a ReLU.

[131] More specifically, it is an embedded representation of the relative distance. To actually implement it, you'd first shift the relative distances such that they begin at 0, and learn 2 * window_size + 1 embedding vectors for each of the possible position offsets. Anything outside the window is embedded into the zero vector.

Selective Attention over Instances. An attention mechanism is employed over all n sentence instances x_i for some candidate entity pair (e_1, e_2). The output is a **set vector** $\boldsymbol{s}$, a real-valued vector representation of the set of instances, where

$$\boldsymbol{s} = \sum_i \alpha_i \boldsymbol{x}_i \tag{559}$$

$$\alpha_i = \text{softmax}(\boldsymbol{x}_i \boldsymbol{A} \boldsymbol{r}) \tag{560}$$

is an attention-weighted sum over the instance embeddings. Note that they constrain $\boldsymbol{A}$ to be diagonal. Finally, the predictive distribution is defined as

$$p(r \mid \mathcal{S}, (e_1, e_2)\theta) = \text{softmax}\left(\boldsymbol{M}\boldsymbol{s} + \boldsymbol{d}\right)_r \tag{561}$$

where $\mathcal{S}$ is the set of n sentences for the given entity pair (e_1, e_2).

On Herding and the Perceptron Cycling Theorem

Written by Brandon McKinzie

Gelfand et al., "On Herding and the Perceptron Cycling Theorem," (2010).

Introduction. Begin with the familiar learning rule of Rosenblatt's perceptron, after some *incorrect* prediction $\hat{y}_i = \text{sgn}(\boldsymbol{w}^T \boldsymbol{x}_i)$,

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \boldsymbol{x}_i(y_i - \hat{y}_i) \tag{562}$$

which has the effect that a subsequent prediction on $\boldsymbol{x}_i$ will (before taking the sign) be $||\boldsymbol{x}_i||_2^2$ closer to the correct side of the hyperplane. The **perceptron cycling theorem** (PCT) states that if the data is *not* linearly separable, the weights will still remain bounded and won't diverge to infinity. **This paper shows that the PCT implies that certain moments are conserved on average**. Formally, their result says that, for some N number of iterations over samples selected from training data (with replacement)[132],

$$\left|\left| \frac{1}{N}\sum_i^N \boldsymbol{x}_i y_i - \frac{1}{N}\sum_i^N \boldsymbol{x}_i \hat{y}_i \right|\right| \sim \mathcal{O}\left(\frac{1}{N}\right) \tag{563}$$

where it's important to remember that $\hat{y}_i$ here is the prediction for $\boldsymbol{x}_i$ when it was encountered at that training iteration. This result shows that perceptron learning generate predictions that correlate with the input attributes the same way as the true labels do, and [the correlations] converge to the sample mean with a rate of 1/N. This also hints at why averaged perceptron algorithms (using the average of weights across training) makes sense, as opposed to just selected the best weights. This paper also shows that *supervised perceptron algorithms and unsupervised herding algorithms can all be derived from the PCT.*

Below are some theorems that will be used throughout the paper. Let $\{\boldsymbol{w}_t\}$ be a sequence of vectors $\boldsymbol{w}_t \in \mathbb{R}^D$, each generated according to iterative updates $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \boldsymbol{v}_t$, where $\boldsymbol{v}_t$ is an element of a *finite* set $\boldsymbol{V}$, and the norm of $\boldsymbol{v}_t$ is bounded: $\max ||\boldsymbol{v}_t|| = R < \infty$.

> ***PCT**: $\forall t \geq 0$: If $\boldsymbol{w}_t^T \boldsymbol{v}_t \leq 0$, $\exists M > 0$ s.t. $||\boldsymbol{w}_t - \boldsymbol{w}_0|| < M$.*
> ***Convergence Thm**: If PCT holds, then $||\frac{1}{T}\sum_{t=1}^T \boldsymbol{v}_t|| \sim \mathcal{O}\frac{1}{T}$.*

[132] Unclear whether this is only for samples that corresponded to an update, or just all samples during training.

Herding. Consider a fully observed Markov Random Field (MRF) over m variables, each of which can take on an integer value in the range $[1, K]$. In herding, our energy function and weight updates for observation $\boldsymbol{x}$ (over all m variables in $\mathcal{X}$),

$$E(\boldsymbol{x}) = -\boldsymbol{w}^T \phi(\boldsymbol{x}) \tag{564}$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \bar{\phi} - \phi(\boldsymbol{x}_t^*) \tag{565}$$

$$\text{where} \quad \bar{\phi} = \mathbb{E}_{\boldsymbol{x}^{(i)} \sim p_{data}} \left[\phi(\boldsymbol{x}^{(i)}) \right] \tag{566}$$

$$\text{and} \quad \boldsymbol{x}_t^* = \arg\max_{\boldsymbol{x}} \boldsymbol{w}_t^T \phi(\boldsymbol{x}) \tag{567}$$

What if we consider more complicated features that depend on the weights $\boldsymbol{w}$? This situation may arise in e.g. models with hidden units $\boldsymbol{z}$, where our feature function would take the form $\phi(\boldsymbol{x}, \boldsymbol{z})$, and we always select $\boldsymbol{z}$ via

$$\boldsymbol{z}(\boldsymbol{x}, \boldsymbol{w}) = \arg\max_{\boldsymbol{z}'} \boldsymbol{w}^T \phi(\boldsymbol{x}, \boldsymbol{z}') \tag{568}$$

and therefore our feature function ϕ depends on weights $\boldsymbol{w}$ through $\boldsymbol{z}$. In this case, our herding update terms from above take the form

$$\bar{\phi} = \mathbb{E}_{\boldsymbol{x}^{(i)} \sim p_{data}} \left[\phi(\boldsymbol{x}^{(i)}, \boldsymbol{z}(\boldsymbol{x}^{(i)}, \boldsymbol{w})) \right] \tag{569}$$

$$\boldsymbol{x}_t^*, \boldsymbol{z}_t^* = \arg\max_{\boldsymbol{x}, \boldsymbol{z}} \boldsymbol{w}_t^T \phi(\boldsymbol{x}, \boldsymbol{z}) \tag{570}$$

Conditional Herding. Main contribution of this paper. It's basically identical to regular herding, but now we decompose $\boldsymbol{x}$ into inputs and outputs $(\boldsymbol{x}, \boldsymbol{y})$ for interpreting in a discriminative setting. In the paper, they express $\boldsymbol{w}^T \phi(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ identically as a discriminative RBM. The parameter update for mini-batch $\mathcal{D}_t$ is given by

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \frac{\boldsymbol{\eta}}{|\mathcal{D}_t|} \sum_{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}) \in \mathcal{D}_t} \Big(\phi(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}, \boldsymbol{z}) - \phi(\boldsymbol{x}^{(i)}, \boldsymbol{y}^*, \boldsymbol{z}^*) \Big) \tag{571}$$

Non-Convex Optimization for Machine Learning

Written by Brandon McKinzie

P. Jain and P. Kar, "Non-convex Optimization for Machine Learning," (2017).

Convex Analysis (2.1). First, let's summarize some definitions.

Convex Combination

*A **convex combination** of a set of n vectors $\boldsymbol{x}_i \in \mathbb{R}^p$, $i = 1 \dots n$ is a vector $\boldsymbol{x}_\theta := \sum_{i=1}^n \theta_i \boldsymbol{x}_i$, where $\theta_i \geq 0$ and $\sum_{i=1}^n \theta_i = 1$.*
My interp: A weighted average where the weights can be interpreted as probability mass associated with each vector.

Convex Set. Sets that contain all [points in] line segments that join any 2 points in the set.

*A set $\mathcal{C}$ is called a **convex set** if $\forall \boldsymbol{x}, \boldsymbol{y} \in \mathcal{C}$ and $\lambda \in [0, 1]$, we have that $(1-\lambda)\boldsymbol{x} + \lambda \boldsymbol{y} \in \mathcal{C}$ as well.*

Proving conv. comb. of 3 vectors $\in \mathcal{C}$ too.

After reading the definition of a convex set above, it seemed intuitive that any convex combination of points $\in \mathcal{C}$ should also be in it as well (i.e. generalizing the pairwise definition). Let $x, y, z \in \mathcal{C}$. How can we prove that $\theta_1 x + \theta_2 y + \theta_3 z$ (where θ_i satisfy the constraints of a convex comb.) is also in $\mathcal{C}$? Here is how I ended up doing it:

- If we can prove that $\theta_1 x + \theta_2 y = (1 - \theta_3) v$ for some $v \in \mathcal{C}$, then our work is done. This is pretty easy to show via simple arithmetic.
- Case 1: assume $\theta_3 < 1$, so that we can divide both sides by $1 - \theta_3$:

$$v = \frac{\theta_1}{1-\theta_3} x + \frac{\theta_2}{1-\theta_3}$$

  Clearly, the two coefficients here sum 1 and satisfy the constraints of a convex combination, and therefore we know that $v \in \mathcal{C}$, and this case is done.
- Case 2: assume $\theta_3 = 1$. Well, that means $\theta_1 = \theta_2 = 0$. Trivially, $z \in \mathcal{C}$ and this case is done.

Convex Function

*A continuously differentiable function $f : \mathbb{R}^p \mapsto \mathbb{R}$ is a **convex function** if $\forall \boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^p$, we have that*

$$f(\boldsymbol{y}) \geq f(\boldsymbol{x}) + \langle \nabla f(\boldsymbol{x}), \boldsymbol{y} - \boldsymbol{x} \rangle \tag{572}$$

While thinking about how to gain intuition for the above, I came across chapter 3 of "Convex Optimization" which describes this in much more detail. It's crucial to recognize that the RHS of the inequality is the 1st-order Taylor expansion of the function f about $\boldsymbol{x}$, evaluated at $\boldsymbol{y}$. In other words, *the first-order Taylor approximation is a **global underestimator** of any convex function f*[133].

[133] Consider what this implies about all the 1st-order gradient-based optimizers we use.

Strongly Convex/Smooth Function. Informally, strong convexity ensures a convex function doesn't grow too *slow*, while strong smoothness ensures a ~~convex~~[134] function doesn't grow too *fast*. Formally,

A continuously differentiable function is considered α***-strongly convex*** *(SC) and* β***-strongly smooth*** *(SS) if* $\forall \boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^p$ *we have*

$$\frac{\alpha}{2}||\boldsymbol{x}-\boldsymbol{y}||_2^2 \leq f(\boldsymbol{y}) - f(\boldsymbol{x}) - \langle \nabla f(\boldsymbol{x}), \boldsymbol{y}-\boldsymbol{x} \rangle \leq \frac{\beta}{2}||\boldsymbol{x}-\boldsymbol{y}||_2^2 \tag{573}$$

Considering the aforementioned 1st-order Taylor approximation interpretation, we see that α determines just how much larger $f(\boldsymbol{y})$ must be compared to its linear approximation. Conversely, β determines the upper bound for how large this discrepancy is allowed to be[135].

Exercise 2.1: SS does not imply convexity

Construct a ***non-convex*** *function* $f : \mathbb{R}^p \mapsto \mathbb{R}$ *that is* 1*-SS.*

We need to find a function whose linear approximation is always more than $\frac{1}{2}$ times the magnitude of the difference in inputs **squared**, compared to the true value. Intuitively, I'd expect any *concave* function to satisfy this, since its linear approximation is a global *overestimator* of the true value. So, for example, $f(\boldsymbol{y}) = -||\boldsymbol{y}||_2^2$ would satisfy $1-SS$ while being non-convex.

Lipschitz Function

A function f *is* ***B-Lipschitz*** *if* $\forall \boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^p$,

$$|f(\boldsymbol{x}) - f(\boldsymbol{y})| \leq B \cdot ||\boldsymbol{x}-\boldsymbol{y}||_2 \tag{574}$$

Jensen's Inequality. Generalizes behavior of convex functions on convex combinations[136].

If X *is a R.V. taking values in the domain of a convex function* f*, then*

$$\mathbb{E}\left[f(X)\right] \geq f(\mathbb{E}\left[X\right]) \tag{575}$$

[134] Strong smoothness alone does not imply convexity.

[135] Notice that SC and SS are *quadratic* lower and upper bounds, respectively. This means that the allowed deltas grow as a function of the distance between $\boldsymbol{x}$ and $\boldsymbol{y}$, whereas things like Lipschitzness grow linearly.

[136] It should be obvious that expectations are convex combinations.

Convex Projections (2.2). Given any closed set $\mathcal{C} \in \mathbb{R}^p$, the projection operator $\Pi_{\mathcal{C}}(\cdot)$ is defined as

$$\Pi_{\mathcal{C}}(\boldsymbol{z}) := \arg\min_{\boldsymbol{x} \in \mathcal{C}} ||\boldsymbol{x} - \boldsymbol{z}||_2 \tag{576}$$

If $\mathcal{C}$ is a convex set, then the above reduces to a convex optimization problem. Projections onto convex sets have three particularly interesting properties. For each of them, the setup is: "For any convex set $\mathcal{C} \subset \mathbb{R}^p$, and any $\boldsymbol{z} \in \mathbb{R}^p$, let $\hat{\boldsymbol{z}} := \Pi_{\mathcal{C}}(\boldsymbol{z})$. Then $\forall \boldsymbol{x} \in \mathcal{C}$, ..."

- **Property-O**[137]: $||\hat{\boldsymbol{z}} - \boldsymbol{z}||_2 \leq ||\boldsymbol{x} - \boldsymbol{z}||_2$. Informally: "the projection results in the point $\hat{\boldsymbol{z}}$ in $\mathcal{C}$ that is closest to the original $\boldsymbol{z}$". This basically just restates the optimization problem.
- **Property-I**. $\langle \boldsymbol{x} - \hat{\boldsymbol{z}}, \boldsymbol{z} - \hat{\boldsymbol{z}} \rangle \leq 0$. Informally: "from the perspective of $\hat{\boldsymbol{z}}$, all points $\boldsymbol{x} \in \mathcal{C}$ are in the (informally) opposite direction of $\boldsymbol{z}$."
- **Property-II**. $||\hat{\boldsymbol{z}} - \boldsymbol{x}||_2 \leq ||\boldsymbol{z} - \boldsymbol{x}||_2$. Informally: "the projection brings the point closer to all points in $\mathcal{C}$ compared to its original location."

Proving Property-I

A proof by contradiction.

1. Assume that $\exists \boldsymbol{x} \in \mathcal{C}$ s.t. $\langle \boldsymbol{x} - \hat{\boldsymbol{z}}, \boldsymbol{z} - \hat{\boldsymbol{z}} \rangle > 0$.
2. We know that $\hat{\boldsymbol{z}}$ is also in $\mathcal{C}$, and since $\mathcal{C}$ is convex, then for any $\lambda \in [0, 1]$,

$$\boldsymbol{x}_\lambda := \lambda x + (1 - \lambda)\hat{\boldsymbol{z}} \tag{577}$$

   must also be in $\mathcal{C}$.
3. If we can show that some value of λ guarantees that $||\boldsymbol{z} - \boldsymbol{x}_\lambda||_2 < ||\boldsymbol{z} - \hat{\boldsymbol{z}}||_2$, this would directly contradict property-O, implying $\hat{\boldsymbol{z}}$ is not the closest member of $\mathcal{C}$ to $\boldsymbol{z}$. I'm not sure how to actually derive the range of λ values that satisfy this, though.

(Convex) Projected Gradient Descent (2.3). Our optimization problem is

$$\min_{\boldsymbol{x} \in \mathbb{R}^p} f(\boldsymbol{x}) \quad \text{s.t.} \quad \boldsymbol{x} \in \mathcal{C} \tag{578}$$

where $\mathcal{C} \subset \mathbb{R}^p$ is a convex constraint set, and $f : \mathbb{R}^p \mapsto \mathbb{R}$ is a convex objective function. Projected gradient descent iteratively updates the value of $\boldsymbol{x}$ that minimizes f as usual, but additionally projects the current iterate (value of best $\boldsymbol{x}$) onto $\mathcal{C}$ at the end of each iteration. That's the only difference.

[137] In this case only, $\mathcal{C}$ need not be convex

Non-Convex Projections (3.1). Here we look at a few special cases where projecting onto a non-convex set can still be carried out efficiently.

- **Projecting into sparse vectors**. The set of s-sparse vectors (vectors with at most s nonzero elements) is denoted as

$$\mathcal{B}_0(s) \triangleq \{\boldsymbol{x} \in \mathbb{R}^p \mid ||\boldsymbol{x}||_0 \leq s\} \tag{579}$$

  It turns out that $\hat{\boldsymbol{z}} := \Pi_{\mathcal{B}_0(s)}(\boldsymbol{z})$ is obtained by setting all except the top-s elements of $\boldsymbol{z}$ to zero.
- **Projecting into low-rank matrices**. The set of $m \times n$ matrices with rank at most r is denoted as

$$\mathcal{B}_{rank}(r) \triangleq \{A \in \mathbb{R}^{m \times n} \mid \text{rank}(A) \leq r\} \tag{580}$$

  and we want to project some matrix A onto this set,

$$\Pi_{\mathcal{B}_{rank}(r)}(A) := \underset{X \in \mathcal{B}_{rank}(r)}{\arg\min} ||A - X||_F \tag{581}$$

  This can be done efficiently via SVD on A and retaining the top r singular values and vectors.

Restricted Strong Convexity and Smoothness (3.2).

Restricted Convexity

A continuously differentiable function $f : \mathbb{R}^p \mapsto \mathbb{R}$ is said to satisfy restricted convexity over a (possibly non-convex) region $\mathcal{C} \subseteq \mathbb{R}^p$ if $\forall \boldsymbol{x}, \boldsymbol{y} \in \mathcal{C}$, we have that

$$f(\boldsymbol{y}) \geq f(\boldsymbol{x}) + \langle \nabla f(\boldsymbol{x}), \boldsymbol{y} - \boldsymbol{x} \rangle \tag{582}$$

and a similar rephrasing for restricted strong convexity (RSC) and restricted strong smoothness (RSS).

Improving Language Understanding by Generative Pre-Training

Written by Brandon McKinzie

Radford et al., "Improving Language Understanding by Generative Pre-Training," (2018).

Unsupervised Pre-Training (3.1). Given unsupervised corpus of tokens $\mathcal{U} = \{u_1, \dots, u_n\}$, train with a standard LM objective:

$$L_1(\mathcal{U}) = \sum_i^n \log P(u_i \mid u_{i-k}, \dots, u_{i-1}; \Theta) \tag{583}$$

The authors use a **Transformer decoder**, i.e. literally just the decoder part of the Transformer in "Attention is all you need."

Supervised Fine-Tuning (3.2). Now we have a labeled corpus $\mathcal{C}$, where each instance consists of a sequence of input tokens $x^1, \dots, x^m$, along with a label y. They just feed the inputs through the transformer until they obtain the final transformer block's activation h_l^m, and linearly project it to output space:

$$P(y \mid x^1, \dots, x^m) = \text{softmax}(h_l^m W_y) \tag{584}$$

$$L_2(\mathcal{C}) = \sum_{(x,y)} \log P(y \mid x^1, \dots, x^m) \tag{585}$$

They also found that including a language modeling auxiliary objective helped learning,

$$L_3(\mathcal{C}) = L_2(\mathcal{C}) + \lambda L_1(\mathcal{C}) \tag{586}$$

. . . that's it. Extremely simple, yet somehow effective.

Deep Contextualized Word Representations

Written by Brandon McKinzie

Peters et al., "Deep Contextualized Word Representations," (2018).

Bidirectional Language Models (3.1). Given a sequence of N tokens, a forward LM computes the probability of the sequence via

$$p(t_1, \dots, t_N) = \prod_{k=1}^{N} p(t_k \mid t_1, \dots, t_{k-1}) \tag{587}$$

A common approach is learning context-independent token representations $\boldsymbol{x}_k$ and passing these through L layers of forward LSTMs. The top layer LSTM output at step k, $\overrightarrow{\boldsymbol{h}}_{k,L}$, is used to predict t_{k+1} with a softmax layer. The authors' biLM combines a forward and backward LM to jointly maximize

$$\begin{aligned} \sum_{k=1}^{N} \Big[& \log p(t_k \mid t_1, \dots, t_{k-1}; \Theta_x, \overrightarrow{\Theta}_{LSTM}, \Theta_s) \\ & + \log p(t_k \mid t_{k+1}, \dots, t_N; \Theta_x, \overleftarrow{\Theta}_{LSTM}, \Theta_s) \Big] \end{aligned} \tag{588}$$

and it's important to note the shared parameters Θ_x (token representation) and Θ_s (output softmax).

ELMo (3.2). A task-specific linear combination of the intermediate representations.

$$\boldsymbol{ELMo}_k^{task} = \gamma^{task} \sum_{j=0}^{L} s_j^{task} \boldsymbol{h}_{k,j}^{LM} \tag{589}$$

where $\boldsymbol{s}^{task}$ are softmax-normalized weights (so the combination is convex). The authors also mention that, in some cases, it helped to apply **layer normalization** to each biLM layer before weighting.

Using biLMs for Supervised NLP (3.3). Given a pretrained biLM and a supervised architecture, we can learn the ELMo representations (jointly with the given supervised task) as follows.

1. Freeze the weights of the [pretrained] biLM.
2. Concatenate the token representations (e.g. GloVe) with the ELMo representation.
3. Pass the concatenated representation into the supervised architecture.

The authors found it beneficial to some dropout to ELMo, and in some cases add L2-regularization on the ELMo weights.

Pretrained biLM Architecture (3.4). In addition to the biLM we introduced earlier, the authors make the following changes/specifications for their pretrained biLMs:

- **residual connections** between LSTM layers[138].
- Halved all embedding and hidden dimensions from the CNN-BIG-LSTM model in *Exploring the Limits of Language Modeling*.
- The $\boldsymbol{x}_k$ token representations use 2048 character n-gram convolutional filters followed by two **highway layers**.

[138] So the output of some layer, instead of being LSTM(x), becomes (x + LSTM(x))

Josefina et al., "Exploring the Limits of Language Modeling," (2016).

NCE and Importance Sampling (3.1). In this section, assume any $p(w)$ is shorthand for $p(w \mid \{w_{prev}\})$.

- **Noise Contrastive Estimation** (NCE). Train a classifier to discriminate between true data (from distribution p_d) or samples coming from some arbitrary noise distribution p_n. If these distributions were known, we could compute

$$p(Y{=}true \mid w) = \frac{p(w \mid \text{true})p(\text{true})}{p(w)} \tag{590}$$

$$= \frac{p_d(w)p(\text{true})}{p(w,\text{true}) + p(w,\text{false})} \tag{591}$$

$$= \frac{p_d(w)p(\text{true})}{p_d(w)p(\text{true}) + p_n(w)p(\text{false})} \tag{592}$$

$$= \frac{p_d(w)}{p_d(w) + kp_n(w)} \tag{593}$$

  where k is the number of negative samples per positive word. The idea is to train a logistic classifier $p(Y{=}true \mid w) = \sigma(\log p_{model} - \log kp_n(w))$, then $\text{softmax}(\log p_{model})$ is a good approx of $p_d(\boldsymbol{w})$.
- **Importance Sampling**. Estimates the partition function. Consider that now we have a set of $k+1$ words $W = \{w_1, \ldots, w_{k+1}\}$, where w_1 is the word coming from the true data, and the rest are from the noise distribution. We train a multinomial logistic regression over $k+1$ classes,

$$p(Y{=}i \mid W) = \frac{p_d(w_i)}{p_n(w_i)} \frac{1}{\sum_{i'=1}^{k+1} p_d(w_{i'})/p_n(w_{i'})} \tag{594}$$

$$\propto_Y \frac{p_d(w_i)}{p_n(w_i)} \tag{595}$$

  and we end up seeing that IS is the same as NCE, except in the multiclass setting and with cross entropy loss instead of logistic loss.

CNN Softmax (3.2). Typically the logit for word w is given by $z_w = h^T e_w$, where h is often the output state of an LSTM, and e_w is a vector of parameters that could be interpreted as the word embedding for w. Instead of this, the authors propose what they call *CNN Softmax*, where we compute $e_w = CNN(chars_w)$. Although this makes the function mapping from w to e_w much smoother (due to the tied weights), it ends up having a hard time distinguishing between similarly spelled words that may have entirely different meanings. The authors use a correction factor, learned for each word, such that

$$z_w = h^T CNN(chars_w) + h^T M corr_w \tag{596}$$

where M projects low-dimensional $corr_w$ back up to the dimensionality of the LSTM state h.

Char LSTM Predictions (3.3). To reduce the computational burden of the partition function, the authors feed the word-level LSTM state h through a character-level LSTM that predicts the target word one character at a time.

Connectionist Temporal Classification

 Written by Brandon McKinzie

Graves et al., "Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks," (2006).

Temporal Classification.

- **Input space**: Let $\mathcal{X} = (\mathbb{R}^m)^*$ be the set of all sequences of m dimensional real-valued vectors.
- **Output space**: Let $\mathcal{Y} = L^*$ be the set of all sequences of a finite vocabulary of L labels.
- **Data distribution**: Denote by $\mathcal{D}_{\mathcal{X}\times\mathcal{Y}}$ the probability distribution over samples $(\boldsymbol{x}, \boldsymbol{y})$. Let S denote a set of training examples drawn from this distribution.

From Network Outputs to Labellings (3.1). Let $L' = L \cup \{\epsilon\}$ denote the set of unique labels combined with the blank token ϵ. We refer to the alignment sequences of length T (same length as $\boldsymbol{x}$), i.e. elements of the set $(L')^T$, as **paths** and denote them π, where

$$p(\pi \mid \boldsymbol{x}) = \prod_{t=1}^{T} y^t_{\pi_t} \qquad (\forall \pi \in (L')^T) \tag{597}$$

and y^t_k denoting the probability of observing label k at time t. Now that we have paths π, we need to convert them to label sequences by (1) merging repeated contiguous labels, and then (2) removing blank tokens. We denote this procedure as a many-to-one map $\mathcal{B} : L'^T \mapsto L^{\leq T}$. In other words, $\mathcal{B}(\pi) \to \boldsymbol{\ell}$. We can then write the conditional posterior over possible output sequences $\boldsymbol{\ell}$:

$$p(\boldsymbol{\ell} \mid \boldsymbol{x}) = \sum_{\pi \in \mathcal{B}^{-1}(\boldsymbol{\ell})} p(\pi \mid \boldsymbol{x}) \tag{598}$$

Constructing the Classifier (3.2). There is no tractable algorithm for exact decoding, i.e. computing

$$h(\boldsymbol{x}) := \arg\max_{\boldsymbol{\ell} \in L^{\leq T}} p(\boldsymbol{\ell} \mid \boldsymbol{x}) \tag{599}$$

However, the following two approximate methods work well in practice:

1. **Best Path Decoding**. $h(\boldsymbol{x}) \approx \mathcal{B}(\pi^*) \quad \text{where} \quad \pi^* = \arg\max_{\pi \in L'^T} p(\pi \mid \boldsymbol{x})$.
2. **Prefix Search Decoding**.

The authors end up using neither of the above, but rather a heuristic approach:

We divide the output sequence into sections that are very likely to begin and end with a blank. We do this by choosing boundary points where the probability of observing a blank label is above a certain threshold. We then calculate the most probable labelling for each section individually and concatenate these to get the final classification.

The CTC Forward-Backward Algorithm (4.1). Define the total probability of the first s output labels, $\boldsymbol{\ell}_{\langle 1\ldots s\rangle}$, given the first t inputs as

$$\alpha_t(s) \triangleq p(\boldsymbol{\ell}_{\langle 1\ldots s\rangle} \mid \boldsymbol{x}_{\langle 1\ldots t\rangle}) \tag{600}$$

$$= \sum_{\substack{\pi\in L'^T \\ \mathcal{B}(\boldsymbol{\pi}_{\langle 1\ldots t\rangle})=\boldsymbol{\ell}_{\langle 1\ldots s\rangle}}} \prod_{t'=1}^{t} y^{t'}_{\pi_{t'}} \tag{601}$$

Note that the summation here could contain duplicate $\boldsymbol{\pi}_{\langle 1\ldots t\rangle}$ that differ only in their elements beyond t.

We insert a blank token at the beginning and end of $\boldsymbol{\ell}$ and between each pair of labels, and call this augmented sequence $\boldsymbol{\ell}'$. We have the following rules for initializing α at the first intput step t=1, followed by the recursion rule:

$$\alpha_1(s) = \begin{cases} y^1_\epsilon & s{=}1 \\ y^1_{\boldsymbol{\ell}_1} & s{=}2 \\ 0 & s>2 \end{cases} \tag{602}$$

$$\alpha_t(s) = \begin{cases} \bar{\alpha}_t(s) y^t_{\boldsymbol{\ell}'_s} & \boldsymbol{\ell}'_s{=}b \text{ or } \boldsymbol{\ell}'_{s-2} \\ (\bar{\alpha}_t(s) + \alpha_{t-1}(s-2)) y^t_{\boldsymbol{\ell}'_s} & \text{otherwise} \end{cases} \tag{603}$$

$$\bar{\alpha}_t(s) \triangleq \alpha_{t-1}(s) + \alpha_{t-1}(s-1) \tag{604}$$

Interpretation:

- *Initialization*: Given the first input x_1, the only valid paths[139] that could potentially result in the final labeling $\boldsymbol{\ell}$ are $\{\epsilon\}$ and $\{\ell_1\}$. These respectively correspond to the augmented labelings $\{\epsilon\}$ and $\{\epsilon, \ell_1\}$. This is what the authors are referring to when they say "We allow all prefixes to start with either a blank or the first symbol in $\boldsymbol{\ell}$."
- *Case 1.* $\bar{\alpha}_t(s)$ corresponds to a right arrow ([s, t-1] $\rightarrow$ [s, t]) and a right-down-one arrow ([s-1, t-1] $\rightarrow$ [s, t]) (see explanations below).
- *Case 2.* $\bar{\alpha}_t(s) + \alpha_{t-1}(s-2))$ corresponds to the 2 arrows from case 1 and additional a right-down-two arrow ([s-2, t-1] $\rightarrow$ [s, t]).

Reading the lattice arrow diagrams. The usage of the augmented label sequence $\boldsymbol{\ell}'$ can make reading these very confusing. Here's how to read them:

- *Rows*: row s corresponds to ℓ'_s.

[139] which remember are always the same length as x

- *Columns*: column t corresponds to $\boldsymbol{x}_t$.
- *Arrows/Traversal*: this was the confusing part for me. The arrows correspond to a transition in a given *path*. Each arrow direction has a different interpretation:
  - right: the path either had a repeated character (which would've been merged into a single output label) or a repeated blank token (if the row is a blank token) which would've been removed entirely in the final output label sequence.
  - right-down-one: transition from either label-to-blank or blank-to-label.
  - right-down-two: direct transition between two unique characters.

Ok, I finally get it now. In my opinion, introducing $\boldsymbol{\ell}'$ as the "augmented label sequence" is confusing/misleading/unnecessary/stupid/etc. It is literally just introduced so we can meaningfully talk about transitions between elements of a given path π. The traversal being done in the α formulas is referencing the valid *paths*, NOT the final labels (at least directly). Especially confusing is when the distill article says crap like "can't jump over z_{s-1}" which is just total nonsense – no one is jumping over anything! Literally all they mean is "transition between unique characters in a path". Lost so many hours confused over the wording in the distill article, which only served to confuse me further (just read the paper).

The final probability of the label sequence $\boldsymbol{\ell}$ given input sequence $\boldsymbol{x}$ is thus:

$$p(\boldsymbol{\ell} \mid \boldsymbol{x}) = \alpha_T(|\boldsymbol{\ell}'|) + \alpha_T(|\boldsymbol{\ell}'| - 1) \tag{605}$$

which corresponds to "total probability of all paths ending in a blank token plus total probability of all paths ending in the final label of $\boldsymbol{\ell}$."

4.49.1 Sequence Modeling With CTC

Notes on this distill article (note that I do NOT recommending reading this article – the wording is horrendously confusing compared to what's actually going on).

Introduction. CTC is an approach for mapping input sequences $X = \{x_1, \dots, x_T\}$ to label sequences $Y = \{y_1, \dots, y_U\}$, where the lengths may vary ($T \neq U$).

Alignment. An **alignment** between input sequence X and label sequence Y is a function $f : X \mapsto Y$. Take for example the label sequence $Y = \{h, e, l, l, o\}$ and some input sequence (e.g. raw audio) $X = \{x_1, \dots, x_{12}\}$. CTC places the following constraints:

1. It must be the same length as the input sequence X.
2. It has the same vocabulary as Y, plus an additional token ϵ to denote blanks.
3. At position i, it either (a) repeats the aligned token at $i-1$, (b) assigns the empty token ϵ, or (c) assigns the next letter of the label sequence.

For our example, we could have an aligned sequence $A = \{h, h, e, \epsilon, \epsilon, l, l, l, \epsilon, l, l, o\}$. Then we apply the following two steps (can interpret as functions) to map from A to Y:

1. Merge any repeated [contiguous] characters.

2. Remove any ϵ tokens.

Number of Valid Alignments

Given X of length T and Y of length $U \leq T$ (and no repeating letters), how many valid alignments exist?

The differences between alignments fall under two categories:

1. Indices where we transition from one label to the next.
2. Indices where we insert the blank token, ϵ.

Stated even simpler, the alignments differ first and foremost by *which elements of X are "extra" tokens*, where I'm using "extra" to mean either blank or repeat token. Given a set of T tokens, there are $\binom{T}{T-U}$ different ways to assign $T-U$ of them as "extra." The tricky part is that we can't just randomly decide to repeat or insert a blank, since a sequence of one or more blanks is *always* followed by a transition to next letter, by definition. And remember, we have defined Y to have no repeated [contiguous] labels.

Apparently, the answer is $\binom{T+U}{T-U}$ total valid alignments.

Loss Function. When you hear someone say "the CTC loss," they usually mean "MLE using a CTC posterior." In other words, there is no "CTC loss" function, but rather there is the standard maximum likelihood objective, but we use a particular form for the posterior $p(Y \mid X)$ over possible output labels Y given raw input sequence X:

$$p(Y \mid X) = \sum_{\mathcal{A} \in \mathcal{A}_{X,Y}} \prod_{t=1}^{T} p_t(a_t \mid X) \tag{606}$$

where $\mathcal{A}$ is one of the valid alignments from $\mathcal{A}_{X,Y}$. The value of a_t obeys the set of three constraints listed above.

How can we compute the loss efficiently? Let $\boldsymbol{z} \triangleq [\epsilon, y_1, \epsilon, y_2, \ldots, \epsilon, y_U, \epsilon]$, and let α denote the **score of the merged alignments** at a given node [in the **CTC lattice**]. We compute the forward probabilities $\alpha_t(s)$, defined as the probability of arriving at [prefix of] augmented label sequence $\boldsymbol{z}_{\langle 1 \ldots s \rangle}$ given unmerged alignments up to input step t:

$$\alpha_t(s) \triangleq p(\boldsymbol{z}_{\langle 1 \ldots s \rangle} \mid \boldsymbol{x}_{\langle 1 \ldots t \rangle})$$

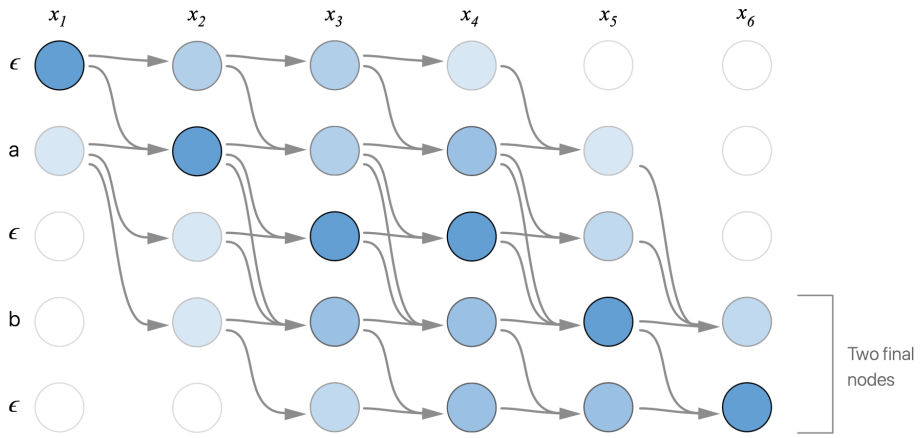
The key insight is that we can compute α_t as long as we know α_{t-1}. There are two cases to consider.

1. (1.1) z_s **is the blank token** ϵ**.** At the previous RNN output (time $t-1$), we could've emitted either a blank token ϵ or the previous token in the augmented label sequence, z_{s-1}. In other words,

$$\alpha_t(s) = p(z_s{=}\epsilon \mid x_t) \cdot (\alpha_{t-1}(s) + \alpha_{t-1}(s-1)) \tag{607}$$

2. (1.2) z_s **is the same label as at step** $s-2$. This occurs when Y has repeated labels next to each other.

$$\alpha_t(s) = p(z_s{=}z_{s-2} \mid x_t) \cdot (\alpha_{t-1}(s) + \alpha_{t-1}(s-1)) \tag{608}$$



In this situation, $\alpha_{t-1}(s)$ corresponds to us just emitting the same token as we did at $t-1$ or emitting a blank token ϵ, and $\alpha_{t-1}(s-1)$ corresponds to a transition to/from ϵ and a label.

3. (2) z_{s-1} **is the blank token** ϵ **between unique characters.** In addition to the two α_{t-1} terms from before, we now also must consider the possibility that our RNN emitted z_{s-2} at the previous time $(t-1)$ and then emitted z_s immediately after at time t.

$$\alpha_t(s) = p(z_s \mid x_t) \cdot (\alpha_{t-1}(s-2) + \alpha_{t-1}(s-1) + \alpha_{t-1}(s)) \tag{609}$$

BERT

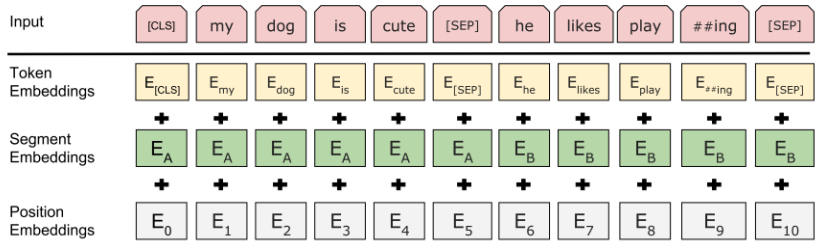
Written by Brandon McKinzie

Devlin et al., "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," Google AI Language (Oct 2018).

TL;DR. Bidirectional Encoder Representations from Transformers. Pretrained by jointly conditioning on left and right context, and can be fine-tuned with one additional non-task-specific output layer. Authors claim the following contributions:

- *Demonstrate importance of bidirectional pre-training for language representations.* Ok, congrats.
- *Show that pre-trained representations eliminate needs of task-specific architectures.* We already knew this. Seriously, how is this news?
- *Advances SOTA for eleven NLP tasks.*

BERT. Instead of using the unidirectional transformer *decoder*, they use the bidirectional *encoder* architecture for their pre-trained model[140].



The input representation is shown in the figure above. The input is a sentence pair, as commonly seen in tasks like QA/NLI.

[140] Is this seriously paper-worthy?? I'm taking notes so I can easily refer back on popular approaches, but I don't see what's so special here.

Pre-training Tasks (3.3).

1. **Masked LM**: Mask 15% of all tokens, and try to predict only those masked tokens[141] Furthermore, at training time, the mask tokens are either fed through as (a) the special `[MASK]` token 80% of the time, (b) a random word 10% of the time, and (c) the original word unchanged 10% of the time. Now *this* is just hackery.
2. **Next Sentence Prediction**: Given two input sentences A and B, train a binary classifier to predict whether sentence B came directly after sentence A.

They do the pretraining jointly, using a loss function that's the sum of the mean masked LM likelihood and mean next sentence prediction likelihood.

[141] This is the only "novel" idea I've seen in the paper. Seems hacky-ish but also reasonable.

Wasserstein is all you need

Written by Brandon McKinzie

Singh et al., "Wasserstein is all you need," EPFL Switzerland (August 2018).

TL;DR. Unsupervised representations of objects/entities via distributional + point estimate. Made possible by **optimal transport**.

Optimal Transport (3). First, notation. Let...

- Ω denote a space of possible outcomes.
- μ denote an **empirical** probability measure, defined as some convex combination $\mu(\boldsymbol{x}) = \sum_{i=1}^{n} a_i \delta(x_i)$, where $x_i \in \Omega$.
- ν denote a similar measure, also a convex combination, $\nu(\boldsymbol{y}) = \sum_{j}^{m} b_j \delta(y_j)$.
- M_{ij} denote the ground cost of moving from point x_i to y_j.

Intuition break: recognize that μ and ν are just a formal description of probability densities via normalized "counts" a_i and/or b_j. Those weights are basically probability mass The **Optimal Transport** distance between μ and ν is the following **linear program**:

$$\mathrm{OT}(\mu, \nu; \boldsymbol{M}) = \min_{\boldsymbol{T} \in \mathbb{R}_{+}^{n \times m}} \sum_{ij} T_{ij} M_{ij} \quad \text{s.t.} \quad (\forall i) \sum_{j} T_{ij} = a_i, \quad (\forall j) \sum_{i} T_{ij} = b_j \tag{610}$$

where $\boldsymbol{T}$ is called the **transportation matrix**. Informally, the constraints are simply enforcing bijection to/from μ and ν, in that "all the mass sent from element i must be exactly a_i, and all mass sent to element j must be exactly b_j". A particular case called the **p-Wasserstein distance**, where $\Omega = \mathbb{R}^d$ and M_{ij} is a distance metric over $\mathbb{R}^d$, is defined as

$$W_p(\mu, \nu) \triangleq \mathrm{OT}(\mu, \nu; D_{\Omega}^{p})^{1/p} \tag{611}$$

where D is just a distance metric, e.g. for $p = 2$ it could be euclidean distance.

Distributional Estimate (4.1). Let $\mathcal{C} \triangleq \{c\}_i$ be the set of possible contexts, where each context c_i can be a word/phrase/sentence/etc. For a given word w and our set of observed contexts for that word, we essentially want to embed its context histogram into a space Ω (where typically $\Omega = \mathbb{R}^d$). Let $\boldsymbol{V}$ denote a matrix of context embeddings, such that $V_{i,:} = \boldsymbol{c}_i \in \mathbb{R}^d$, the embedding for context c_i in what the authors call the *ground space*. Combining the histogram H^w containing observed context counts for word w with $\boldsymbol{V}$, the **distributional estimate** of the word w is defined as

$$P_{\boldsymbol{V}}^{w} \triangleq \sum_{c \in \mathcal{C}} (H^w)_c \delta(\boldsymbol{v}_c) \tag{612}$$

Also, the *point estimate* is just $\boldsymbol{v}_w$, i.e. the embedding of the word w when viewed as a context.

Distance (4.2). Given some distance metric D_Ω in ground space $\Omega = \mathbb{R}^d$, the distance between words w_i and w_j is the solution to the following OT problem[142]:

$$\mathrm{OT}(P_{\boldsymbol{V}}^{w_i}, P_{\boldsymbol{V}}^{w_j}; D_\Omega^p) := W_p^\lambda(P_{\boldsymbol{V}}^{w_i}, P_{\boldsymbol{V}}^{w_j})^p \tag{613}$$

Concrete Framework (5). The authors make use of the **shifted positive pointwise mutual information** (SPPMI), $\boldsymbol{S}_s^\alpha$, for computing the word histograms:

$$(H^w)_c := \frac{\boldsymbol{S}_s^\alpha(w,c)}{\sum_{c' \in \mathcal{C}} \boldsymbol{S}_s^\alpha(w,c')} \tag{614}$$

$$\boldsymbol{S}_s^\alpha(w,c) \triangleq \max\left[\log\left(\frac{\mathrm{Count}(w,c)\sum_{c'}\mathrm{Count}(c')^\alpha}{\mathrm{Count}(w)\mathrm{Count}(c)^\alpha}\right) - \log(s),\ 0\right] \tag{615}$$

[142] I'm not sure whether the rightmost exponent of p is a typo in the paper, but that is how it is written.

Noise Contrastive Estimation

Written by Brandon McKinzie

M. Gutman and A. Hyvärinen, "Noise contrastive estimation: A new estimation principle for unnormalized statistical models," University of Helsinki (2010).

TL;DR: A few ways of thinking about NCE:

- Instead of directly modeling a normalized word distribution $p_d(w)$, we can just model the unnormalized $\widetilde{p}_d$ distribution (and an additional parameter $c = -\ln Z$) by training our model to distinguish between true samples from p_d and noise samples from some distribution p_n that we choose.
- Instead of modeling $p(w \mid c)$, we model $p(D \mid w, c)$, where D is binary RV indicating whether w, c are from the true data distribution p_d or the noise distribution p_n.

Introduction. Setup & notation:

- We observe $\boldsymbol{x} \sim p_d(\cdot)$ but $p_d(\cdot)$ itself is unknown.
- We model p_d by some model $p_m(\cdot; \alpha)$ parameterized by α[143].

So, can we get away with modeling the *unnormalized* density $\widetilde{p}_m$ instead of requiring the normalization constraint to be baked in to our optimization problem? Similar to approaches like **contrastive divergence** and **score matching**, **noise-contrastive estimation** (NCE) aims to address this question.

Noise-contrastive estimation (2.1). Let c be an estimator for $-lnZ(\alpha)$, and let $\theta = \{\alpha, c\}$ denote all of our parameters. Given observed data $X = \{\boldsymbol{x}_1, \dots, \boldsymbol{x}_T\}$, and noise $Y = \{\boldsymbol{y}_1, \dots, \boldsymbol{y}_T\}$, we seek parameters $\hat{\theta}_T$ that maximize $J_T(\theta)$[144]:

$$J_T(\theta) = \frac{1}{2T} \sum_t \ln\left[h(\boldsymbol{x}_t)\right] + \ln\left[1 - h(\boldsymbol{y}_t)\right] \tag{616}$$

$$h(\boldsymbol{u}) = \sigma\left(G\left(\boldsymbol{u}\right)\right) \tag{617}$$

$$G(\boldsymbol{u}) = \ln p_m(\boldsymbol{u}) - \ln p_n(\boldsymbol{u}) \tag{618}$$

$$\ln p_m(\cdot; \theta) := \ln \widetilde{p}_m(\cdot; \alpha) + c \tag{619}$$

where for compactness reasons I've removed the explicit dependence of all functions above (except p_n) on θ. Notice how this fixes the issue of the model just setting c arbitrarily high to obtain a high likelihood[145].

[143] The implicit assumption here is that $\exists \alpha^*$ such that $p_d(\cdot) = p_m(\cdot; \alpha^*)$.
[144] This all assumes of course that p_n is fully defined.
[145] The primary reason why MLE is traditionally unable to parameterize the partition function.

Connection to supervised learning (2.2). The NCE objective can be interpreted as binary logistic regression that discriminates whether a point belongs to the data (p_d) or to the noise (p_n).

$$P(C{=}1 \mid \boldsymbol{u} \in X \cup Y) = \frac{P(C{=}1)p(\boldsymbol{u} \mid C{=}1)}{p(\boldsymbol{u})} \tag{620}$$

We model with a uniform prior: $P(C{=}1) = P(C{=}0) = 1/2$

$$= \frac{p_m(\boldsymbol{u})}{p_m(\boldsymbol{u}) + p_n(\boldsymbol{u})} \tag{621}$$

$$\equiv h(\boldsymbol{u}; \theta) \tag{622}$$

where we're now using the union of X and Y, $U := \{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_{2T}\}$. The log-likelihood of the data under the parameters θ is

$$\ell(\theta) = \sum_t^{2T} \left[C_t \ln P(C_t{=}1 \mid \boldsymbol{u}_t; \theta) + (1 - C_t) \ln P(C_t{=}0 \mid \boldsymbol{u}_t; \theta)\right] \tag{623}$$

$$= \sum_t^{2T} \left[C_t \ln h(\boldsymbol{u}_t; \theta) + (1 - C_t) \ln \left[1 - h(\boldsymbol{u}_t; \theta)\right]\right] \tag{624}$$

$$= \sum_t^{T} \left[\ln h(\boldsymbol{x}_t; \theta) + \ln \left[1 - h(\boldsymbol{y}_t; \theta)\right]\right] \tag{625}$$

which is (up to a constant factor) the same as our NCE objective.

Properties of the estimator (2.3). As $T \to \infty$, $J_T(\theta) \to J(\theta)$, where

$$J(\theta) \triangleq \lim_{T\to\infty} J_T(\theta) = \frac{1}{2}\mathbb{E}\left[\ln h(\boldsymbol{x}; \theta) + \ln \left[1 - h(\boldsymbol{y}; \theta)\right]\right] \tag{626}$$

$$\widetilde{J}(f) \triangleq \frac{1}{2}\mathbb{E}\left[\ln\left[\sigma\Big(f(\boldsymbol{x}) - \ln p_n(\boldsymbol{x})\Big)\right] + \ln\left[1 - \sigma\Big(f(\boldsymbol{y}) - \ln p_n(\boldsymbol{y})\Big)\right]\right] \tag{627}$$

Theorem 1

$\widetilde{J}$ attains exactly one maximum, located at $f(\cdot) = \ln p_d(\cdot)$, provided $p_d(\cdot){\neq}0 \implies p_n(\cdot){\neq}0$.

4.52.1 Self-Normalized NCE

Notes from A. Mnih and Y. Teh, "A fast and simple algorithm for training neural probabilistic language models" (2012).

Maximum likelihood learning. Let $P_\theta^h(w)$ denote the probability of observing word w given context h. For neural LMs, we assume this is the softmax of a scoring function $s_\theta(w, h)$ (logits). In what follows, I'll drop the explicit θ and h subscript/superscript notation for brevity.

$$\frac{\partial \log P(w)}{\partial \theta} = \frac{\partial}{\partial \theta} s(w,h) - \frac{\partial}{\partial \theta} \log \left[\sum_{w'} e^{s(w',h)} \right] \tag{628}$$

$$= \frac{\partial}{\partial \theta} s(w,h) - \sum_{w'} P(w') \frac{\partial}{\partial \theta} s(w',h) \tag{629}$$

$$= \frac{\partial}{\partial \theta} s(w,h) - \mathbb{E}_{w \sim P_\theta^h} \left[\frac{\partial}{\partial \theta} s(w,h) \right] \tag{630}$$

where the expectation (in red) is expensive due to requiring $s(w, h)$ evaluated for all words in the vocabulary. One approach is **importance sampling** where we sample a subset of k words from the vocab and compute the probabilities from that approximation:

$$\frac{\partial \log P(w)}{\partial \theta} = \frac{\partial}{\partial \theta} s(w,h) - \sum_{w'} P(w') \frac{\partial}{\partial \theta} s(w',h) \tag{631}$$

$$\approx \frac{\partial}{\partial \theta} s(w,h) - \frac{1}{V} \sum_{j=1}^{k} v(x_j) \frac{\partial}{\partial \theta} s(w',h) \tag{632}$$

$$\text{where} \quad v(x) = \frac{e^{s_\theta(x,h)}}{Q^h(w{=}x)} \tag{633}$$

and we refer to v as the **importance weights**. In NLP, we typically set Q to the **Zipfian distribution**[146]

[146] TensorFlow seems to define this as

$$P_{Zipf}(w) = \frac{\log(w+2) - \log(w+1)}{\log(V+1)} \tag{634}$$

where V is the vocabulary size. I can't seem to find this definition anywhere else though. A more common form seems to be

$$P(w) = \frac{\frac{1}{w}}{\sum_{w'} \frac{1}{w'^s}} \tag{635}$$

I plotted both on WolframAlpha (link here) and they do indeed look basically the same, especially for any reasonably large V.

NCE. In NCE, we introduce [unigram] noise distribution $P_n(w)$ and impose a prior that noise samples are k times more frequent than data samples from $P_d^h(w)$, resulting in the joint distribution,

$$P^h(D, w) = P(D{=}1)P_d^h(w) + P(D{=}0)P_n(w) \tag{636}$$

$$= \frac{1}{k+1}P_d^h(w) + \frac{k}{k+1}P_n(w) \tag{637}$$

Our goal is to learn the posterior distribution $P^h(D{=}1 \mid w)$ (so we replace P_d with P_θ):

$$P^h(D{=}1 \mid w, \theta) = \frac{P_\theta^h(w)}{P_\theta^h(w) + kP_n(w)} \tag{638}$$

In NCE, we re-parameterize P_θ by treating $-\log Z$ as a parameter itself, c^h[147].

$$P_\theta^h(w) := P_{\theta^0}^h \exp(c^h) \tag{639}$$

where $P_{\theta^0}^h$ denotes the unnormalized distribution. It turns out that, in practice, we can impose that $\exp(c^h){=}1$ and use the unnormalized $P_{\theta^0}^h$ in place of the true probabilities in all that follows. *Critically, note that this means we rewrite equation 638 using the unnormalized probabilities in place of* P_θ^h. The NCE objective is to find[148] as follows, where I've shown each step of the derivation:

$$\theta^* = \arg\max_\theta J^h(\theta) \tag{640}$$

$$J^h(\theta) = \mathbb{E}_{(D,w)\sim P^h}\left[\log P(D \mid w, \theta)\right] \tag{641}$$

$$= \sum_{D=0}^{1}\sum_w P^h(D, w) \log P^h(D \mid w, \theta) \tag{642}$$

$$= \sum_w P^h(0, w) \log P^h(0 \mid w) + \sum_w P^h(1, w) \log P^h(1 \mid w) \tag{643}$$

$$= \frac{1}{k+1}\sum_w \left[kP_n(w) \log P^h(0 \mid w) + P_d^h(w) \log P^h(1 \mid w)\right] \tag{644}$$

$$= \frac{1}{k+1}\left[k\mathbb{E}_{P_n}\left[\log P^h(0 \mid w)\right] + \mathbb{E}_{P_d^h}\left[\log P^h(1 \mid w)\right]\right] \tag{645}$$

$$\propto k\mathbb{E}_{P_n}\left[\log P^h(0 \mid w)\right] + \mathbb{E}_{P_d^h}\left[\log P^h(1 \mid w)\right] \tag{646}$$

The gradient of the NCE objective is thus

$$\frac{\partial}{\partial\theta}J^h(\theta) = \sum_w \frac{kP_n(w)}{P_\theta^h(w) + kP_n(w)}\left(P_d^h(w) - P_\theta^h(w)\right)\frac{\partial}{\partial\theta}\log P_\theta^h(w) \tag{647}$$

TODO: incorporate more info from Chris Dyer's excellent notes.

[147] Reminder that the h is a reminder that Z is a function of the context h.
[148] I'll drop off θ dependence wherever obvious for the sake of compactness.

Neural Ordinary Differential Equations

Written by Brandon McKinzie

R. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud, "Neural Ordinary Differential Equations," University of Toronto (Oct 2018).

Introduction (1). Let T denote the number of layers of our network. In the limit of $T \to \infty$ and small $\delta \boldsymbol{h}(t)$ between each "layer", we can parameterize the dynamics via an ODE:

$$\frac{d\boldsymbol{h}(t)}{dt} = f\left(\boldsymbol{h}\left(t\right), t, \theta\right) \tag{648}$$

Benefits of defining models using ODE solvers:

- **Memory**. Constant as a function of depth, since we don't need to store intermediate values from the forward pass.
- **Adaptive computation**. Modern ODE solvers adapt evaluation strategy on the fly.
- **Parameter efficiency**. Nearby "layers" have shared parameters.

Review: ODE

Remember the basic idea with ODEs like the one shown above. Our goal is to solve for $\boldsymbol{h}(t)$.

$$d\boldsymbol{h}(t) = f\left(\boldsymbol{h}\left(t\right), t, \theta\right) \mathrm{d}t \tag{649}$$

$$\int d\boldsymbol{h}(t) = \int f\left(\boldsymbol{h}\left(t\right), t, \theta\right) \mathrm{d}t \tag{650}$$

$$\boldsymbol{h}(t) + c_1 = \int f\left(\boldsymbol{h}\left(t\right), t, \theta\right) \mathrm{d}t \tag{651}$$

$$\tag{652}$$

and so the solution of an ODE is often represented as an integral.

Reverse-mode automatic differentiation of ODE solutions (2). Our goal is to optimize

$$L(\boldsymbol{z}(t_1)) = L\left(\int_{t0}^{t_1} f(\boldsymbol{z}(t), t, \theta)\mathrm{d}t\right) \tag{653}$$

Given our starting definition (eq 648), we can say

$$\boldsymbol{z}(t+\epsilon) = z(t) + \int_t^{t+\epsilon} f\left(\boldsymbol{z}\left(t\right), t, \theta\right) \mathrm{d}t := T_\epsilon(\boldsymbol{z}(t), t) \tag{654}$$

which we can use to define the **adjoint** $a(t)$:

$$a(t) \triangleq -\frac{\partial L}{\partial \boldsymbol{z}(t)} = -\frac{\partial L}{\partial \boldsymbol{z}(t+\epsilon)}\frac{d\boldsymbol{z}(t+\epsilon)}{d\boldsymbol{z}(t)} \tag{655}$$

$$= a(t+\epsilon)\frac{\partial T_\epsilon(\boldsymbol{z}(t), t)}{\partial \boldsymbol{z}(t)} \tag{656}$$

$$\frac{da(t)}{dt} = -a(t)^T \frac{\partial f(\boldsymbol{z}(t), t, \theta)}{\partial \boldsymbol{z}} \tag{657}$$

where $\frac{da(t)}{dt}$ can be derived using the limit definition of a derivative. We'll now outline the algorithm for computing gradients. We use a black box ODE solver as a subroutine that solves a first-order ODE initial value problem. As such, it accepts an initial state, its first derivative, the start time, the stop time, and parameters θ as arguments.

Reverse-mode derivative

Given start time t_0, stop time t_1, final state $\boldsymbol{z}(t_1)$, parameters θ, and gradient $\frac{\partial L}{\partial \boldsymbol{z}(t_1)}$ compute all gradients of L.

1. Compute t_1 gradient:

$$\frac{\partial L}{\partial t_1} = \frac{\partial L}{\partial \boldsymbol{z}(t_1)}^T \frac{\partial \boldsymbol{z}(t_1)}{\partial t_1} = \frac{\partial L}{\partial \boldsymbol{z}(t_1)}^T f(\boldsymbol{z}(t_1), t_1, \theta) \tag{658}$$

2. Initialize the augmented state:

$$s_0 := \left[\boldsymbol{z}(t_1),\ \boldsymbol{a}(t_1),\ \mathbf{0},\ -\frac{\partial L}{\partial t_1}\right] \tag{659}$$

3. Define augmented sate dynamics:

$$\frac{ds}{dt} \triangleq \left[f(\boldsymbol{z}(t), t, \theta),\ -\boldsymbol{a}(t)^T\frac{\partial f}{\partial \boldsymbol{z}},\ -\boldsymbol{a}(t)^T\frac{\partial f}{\partial \theta},\ -\boldsymbol{a}(t)^T\frac{\partial f}{\partial t}\right] \tag{660}$$

4. Solve **reverse-time**[a] ODE:

$$\left[\boldsymbol{z}(t_0),\ \frac{\partial L}{\partial \boldsymbol{z}(t_0)},\ \frac{\partial L}{\partial \theta},\ \frac{\partial L}{\partial t_0}\right] = \text{ODESolve}\left(s_0, \frac{ds}{dt}, t_1, t_0, \theta\right) \tag{661}$$

5. Return $\frac{\partial L}{\partial \boldsymbol{z}(t_0)}$, $\frac{\partial L}{\partial \theta}$, $\frac{\partial L}{\partial t_0}$, $\frac{\partial L}{\partial t_1}$

[a] Notice how our "initial state" actually corresponds to t_1, and we pass in t_1 and t_0 in the opposite order we typically do.

On the Dimensionality of Word Embedding

Written by Brandon McKinzie

Z. Yin and Y. Shen, "On the Dimensionality of Word Embedding," Stanford University (Dec 2018).

Unitary Invariance of Word Embeddings (2.1). Authors interpret result any unitary transformation (e.g. a rotation) on embedding matrix as equivalent to the original.

Word Embeddings from Explicit Matrix Factorization (2.2). Let M be the $V \times V$ co-occurrence counts matrix. One way of getting embeddings is doing a truncated SVD on $M = UDV^T$. If we want k-dimensional embedding vectors, we can do

$$\boldsymbol{E} = \boldsymbol{U}_{1:k}\boldsymbol{D}^{\alpha}_{1:k,1:k} \tag{662}$$

for some $\alpha \in [0, 1]$.

PIP Loss (3). Given embedding matrix $E \in \mathbb{R}^{V \times d}$, define its **Pairwise Inner Product** (PIP) matrix to be

$$\text{PIP}(\boldsymbol{E}) \triangleq \boldsymbol{E}\boldsymbol{E}^T \tag{663}$$

Notice that $\text{PIP}(\boldsymbol{E})_{i,j} = \langle \boldsymbol{w}_i, \boldsymbol{w}_j \rangle$. Define the **PIP loss** for comparing two embeddings $\boldsymbol{E}_1$ and $\boldsymbol{E}_2$ for a common vocab of V words:

$$||\text{PIP}(\boldsymbol{E}_1) - \text{PIP}(\boldsymbol{E}_2)||_F = \sqrt{\sum_{i,j}\left(\langle \boldsymbol{w}_i^{(1)}, \boldsymbol{w}_j^{(1)} \rangle - \langle \boldsymbol{w}_i^{(2)}, \boldsymbol{w}_j^{(2)} \rangle\right)^2} \tag{664}$$

Generative Adversarial Nets

Written by Brandon McKinzie

Goodfellow et al., "Generative Adversarial Nets," (June 2014)

TL;DR. The abstract is actually quite good:

> *. . . we simultaneously train two models: a generative model **G** that captures the data distribution, and a discriminative model **D** that estimates the probability that a sample came from the training data rather than **G**. The training procedure for **G** is to maximize the probability of **D** making a mistake. This framework corresponds to a minimax two-player game.*

Adversarial Nets (3). As usual, first go over notation:

- Generator produces data samples[149], $\boldsymbol{x} := G(\boldsymbol{z}; \theta_g)$, where $\boldsymbol{z} \sim p_n$ (noise distribution prior).
- Discriminator, $D(\boldsymbol{x}; \theta_d)$, outputs probability that $\boldsymbol{x}$ came from (true) p_{data} instead of G.

Our two-player minimax optimization problem can be written as:

$$\min_G \max_D V(D, G) = \mathbb{E}_{\boldsymbol{x} \sim p_{data}} [\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p_n} [\log (1 - D(G(\boldsymbol{z})))] \tag{665}$$

Theoretical Results (4). Below is the training algorithm.

SGD with GANs

Repeat the following for each training iteration.

1. Train D. For k steps, repeat:
   (a) Sample m noise samples $\{\boldsymbol{z}_1, \dots, \boldsymbol{z}_m\}$ from noise prior p_n.
   (b) Sample m data samples $\{\boldsymbol{x}_1, \dots, \boldsymbol{x}_m\}$ from data distribution p_{data}.
   (c) Update discriminator by *ascending* $\nabla_{\theta_d} V(D, G)$.
2. Train G: Sample another m noise samples $\{\boldsymbol{z}_1, \dots, \boldsymbol{z}_m\}$ and *descend* on $\nabla_{\theta_g} V(D, G)$.

[149] Note that G outputs samples $\boldsymbol{x}$, not probabilities. By doing this, it *implicitly* defines a probability distribution $p_g(\boldsymbol{x})$. This is what the authors say.

Global Optimality of $p_g \equiv p_{data}$ (4.1).

Proposition 1

For fixed G, the optimal D is

$$D_G^*(\boldsymbol{x}) = \frac{p_{data}(\boldsymbol{x})}{p_{data}(\boldsymbol{x}) + p_g(\boldsymbol{x})} \tag{666}$$

Derivation of $D_G^*(\boldsymbol{x})$.

Aside: Law of the unconscious statistician (LotUS). The distribution $p_g(\boldsymbol{x})$ should be read as "the probability that the output of G yields the value $\boldsymbol{x}$." Take a step back and recognize that G is simply a function of a random variable $\boldsymbol{z}$. As such, we can apply familiar rules like

$$\mathbb{E}\left[G(\boldsymbol{z})\right] = \mathbb{E}_{\boldsymbol{z}\sim p_n}\left[G(\boldsymbol{z})\right] \tag{667}$$

$$= \int_{\boldsymbol{z}} p_n(\boldsymbol{z}) G(\boldsymbol{z}) \mathrm{d}\boldsymbol{z} \tag{668}$$

However, recall that functions of random variables can themselves be interpreted as random variables. In other words, we can also use the interpretation that G evaluates to some output $\boldsymbol{x}$ with probability $p_g(\boldsymbol{x})$.

$$\mathbb{E}\left[G\right] = \mathbb{E}_{\boldsymbol{x}\sim p_g}\left[\boldsymbol{x}\right] \tag{669}$$

$$= \int_{\boldsymbol{x}} p_g(\boldsymbol{x}) \boldsymbol{x} \mathrm{d}\boldsymbol{x} \tag{670}$$

As this blog post details, this equivalence is NOT due to a change of variables, but rather by the **Law of the unconscious statistician**.

The Proof: We can directly use LotUS to rewrite $V(G, D)$:

$$V(G,D) = \mathbb{E}_{\boldsymbol{x}\sim p_{data}}\left[\log D(\boldsymbol{x})\right] + \mathbb{E}_{\boldsymbol{z}\sim p_n}\left[\log\left(1 - D(G(\boldsymbol{z}))\right)\right] \tag{671}$$

$$= \mathbb{E}_{\boldsymbol{x}\sim p_{data}}\left[\log D(\boldsymbol{x})\right] + \mathbb{E}_{\boldsymbol{x}\sim p_g}\left[\log\left(1 - D(\boldsymbol{x})\right)\right] \tag{672}$$

$$= \int_{\boldsymbol{x}} \left[p_{data}(\boldsymbol{x}) \log\left(D(\boldsymbol{x})\right) + p_g(\boldsymbol{x}) \log\left(1 - D(\boldsymbol{x})\right)\right] \mathrm{d}\boldsymbol{x} \tag{673}$$

LotUS allowed us to express $V(G, D)$ as a continuous function over $\boldsymbol{x}$. More importantly, it means we can evaluate $\frac{\partial V}{\partial D}$ and take the derivative inside the integral[a]. Setting the derivative to zero and solving for D yields $D{*}_G$, the form that maximizes V.

[a] Also remember that $D(\cdot) \in [0, 1]$ since it is a probability distribution.

The authors use this proposition to define the virtual training criterion $C(G) \triangleq V(G, D_G^*)$:

$$C(G) = \mathbb{E}_{\boldsymbol{x} \sim p_{data}} \left[\log \frac{p_{data}(\boldsymbol{x})}{p_{data}(\boldsymbol{x}) + p_g(\boldsymbol{x})} \right] + \mathbb{E}_{\boldsymbol{x} \sim p_g} \left[\log \frac{p_g(\boldsymbol{x})}{p_{data}(\boldsymbol{x}) + p_g(\boldsymbol{x})} \right] \tag{674}$$

Theorem 1.

The global minimum of $C(G)$ is achieved IFF $p_g = p_{data}$. At that point $C(G) = -\log 4$.

Proof: Theorem 1

The authors subtract $V(D_G^*, G; p_g{=}p_{data})$ from both sides of 674, do some substitions, and find that

$$C(G) = 2 \cdot JSD(p_{data} || p_g) - \log 4 \tag{675}$$

where JSD is the **Jensen-Shannon divergence**[a]. Since $0 \leq JSD(p||q)$ always, with equivalence only if $p \equiv q$, this proves Theorem 1 above.

[a] Recall that the JSD represents the divergence of each distribution from the mean of the two

A Framework for Intelligence and Cortical Function

 Written by Brandon McKinzie

J. Hawkins et al., "A Framework for Intelligence and Cortical Function Based on Grid Cells in the NeoCortex," Numata Inc. (Oct 2018).

Introduction. Authors propose new framework based on location processing that provides supporting evidence to the theory that **all regions of the neocortex are fundamentally the same**. We've known that **grid cells** exist in the hippocampal complex of mammals, but only recently have seen evidence that they may be present in the neocortex.

How Grid Cells Represent Location. Grid cells in the **entorhinal cortex**[150] represent space and location. The main concepts, in order such that they build on one another, are as follows:

- A **single grid cell** is a neuron that fires [when the agent is] at [one of many] multiple locations in a physical environment[151].
- A **grid cell module** is a set of grid cells that activate with the *same lattice spacing and orientation* but at shifted locations within an environment.
- **Multiple grid cell modules** that differ in tile spacing and/or orientation can provide *unique location* information[152].

Crucially, the number of unique locations that can be represented by a set of grid cell modules **scales exponentially** with the number of modules. Every learned environment is associated with a set of unique locations (firing patterns of the grid cells).

Grid Cells in the Neocortex. The authors propose that we learn the structures of objects (like pencils, cups, etc) via grid cells in the *neocortex*. Specifically, they propose:

1. Every cortical column has neurons that perform a function similar to grid cells.
2. Cortical columns learn models of *objects* similarly to how grid/place cells learn models of *environments*.

[150] The entorhinal cortex is located in the medial temporal lobe and functions as a hub in a widespread network for memory, navigation and the perception of time.

[151] For example, there may be a grid cell in my brain that fires when I'm at certain locations inside my room. Those locations tend to form a lattice of sorts.

[152] A single module alone cannot, because it repeats periodically. In other words, it can only provide relative location information.

Large-Scale Study of Curiosity Driven Learning

Written by Brandon McKinzie

Burda et al., "Large-Scale Study of Curiosity Driven Learning," OpenAI and UC Berkeley (Aug 2018).

An agent sees observation x_t, takes action a_t, and transitions to the next state with observation x_{t+1}. Goal: incentivize agent with reward r_t relating to how informative the transition was. The main components in what follows are:

- Observation embedding $\phi(x)$.
- Forward dynamics network for prediction $P(\phi(x_{t+1} \mid x_t, a_t)$.
- Exploration reward (*surprisal*):

$$r_t = -\log p\left(\phi\left(x_{t+1}\right) \mid x_t, a_t\right) \tag{676}$$

The authors choose to model the next state embedding with a Gaussian,

$$\phi(x_{t+1}) \mid x_t, a_t \sim \mathcal{N}(f(x_t, a_t), \epsilon) \tag{677}$$

$$r_t = ||f(x_t, a_t) - \phi(x_{t+1})||_2^2 \tag{678}$$

where f is the learned dynamics model.

Feature spaces (2.1). Some possible ways to define ϕ:

- **Pixels**: $\phi(x) = x$.
- **Random Features**: Literally feeding $\phi(x) = ConvNet(x)$ where ConvNet is *randomly initialized* and fixed.
- **VAE**: Use the mapping to the mean [of the approximated distribution] as the embedding network ϕ.

Interpretation. It seems that this works because after awhile, it is boring and predictable to take actions that result in losing a game. The most surprising actions seem to be those that advance us forward, to new and uncharted territory. However, these experiments are all on games that have a very "linear" uni-directional-like sequence of success. I wonder how successful this would be in a game like rocket league, where there is no tight coupling of direction with success and novelties (e.g. moving forward in mario bros).

Universal Language Model Fine-Tuning for Text Classification

Written by Brandon McKinzie

J. Howard and S. Ruder, "Universal Language Model Fine-Tuning for Text Classification," (May 2018).

TL;DR. ULMFiT is a transfer learning method. They introduce techniques for fine-tuning language models.

Universal Language Model Fine-tuning (3). Define the general **inductive transfer learning** setting for NLP:

> *Given a source task $\mathcal{T}_s$ and target task $\mathcal{T}_T \neq \mathcal{T}_S$, improve performance on $\mathcal{T}_T$.*

ULMFiT is defined by the following three steps:

1. General-domain LM pretraining.
2. Target task LM fine-tuning.
3. Target task classifier fine-tuning.

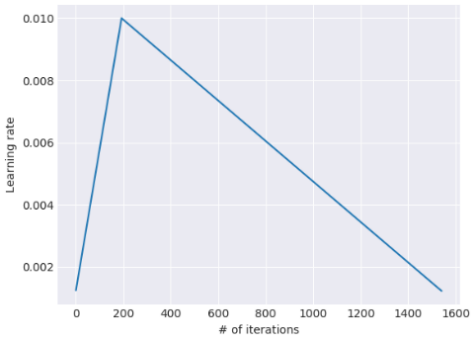
Target Task LM Fine-tuning (3.2). For step 2, the authors propose what they call **discriminative fine-tuning** and **slanted triangular learning rates**.

- **Discriminative fine-tuning**. Tune each layer with different learning rates:

$$\theta_t^\ell = \theta_{t-1}^\ell - \eta^\ell \cdot \nabla_{\theta^\ell} J(\theta) \tag{679}$$

  The authors suggest setting $\eta^{\ell-1} = \eta^\ell / 2.6$.

- **Slanted triangular learning rates**. A type of learning rate schedule that looks like the picture below.



First, we define the following hyperparameters:

- T: *total* number of training iterations[153]
- $cfrac$: fraction of T (in num iterations) where we're *increasing* the learning rate.
- cut: $\lfloor T \cdot cfrac \rfloor$. Iteration where we switch from increasing the LR to decreasing it.
- $ratio$: η_{max}/η_{min}. We of course must also define η_{max}.

We can now compute the learning rate for a given iteration t:

$$\eta_t = \eta_{max} \cdot \frac{1 + p \cdot (ratio - 1)}{ratio} \tag{680}$$

$$p = \begin{cases} \frac{t}{cut} & \text{if } t < cut \\ 1 - \frac{t - cut}{cut \cdot (1/cfrac - 1)} & \text{otherwise} \end{cases} \tag{681}$$

Target Task Classifier Fine-tuning (3.3). Augment the LM with two fully-connected layers. The first with ReLU activation and the second with softmax. Each uses batch normalization and dropout. The first is fed the output hidden state of the LM concatenated with the max- and mean-pooled hidden states over all timesteps[154]:

$$\boldsymbol{h}_c = [\boldsymbol{h}_t, \text{maxpool}(\boldsymbol{H}), \text{meanpool}(\boldsymbol{H})] \tag{682}$$

In addition to DF-T and STLR from above, they also employ the following techniques:

- **Gradual Unfreezing**: first unfreeze the last layer and fine-tune it alone with all other layers frozen *for one epoch*. Then, unfreeze the next layer and fine-tune the last-two layers only *for the next epoch*. Continue until the entire network is being trained, at which time we just train until convergence.
- **BPTT for Text Classification**. Divide documents into fixed-length "batches"[155] of size b. They initialize the ith section with the final state of the model run on section $i - 1$.

[153] Steps-per-epoch times number of epochs.
[154] Or just as much as we can fit into GPU memory.
[155] Not a fan of how they overloaded this term here.

The Marginal Value of Adaptive Gradient Methods in Machine Learning

Written by Brandon McKinzie

Wilson et al., "The Marginal Value of Adaptive Gradient Methods in Machine Learning," (May 2018).

TL;DR. For simple overparameterized problems, adaptive methods (a) often find drastically different solutions than SGD, and (b) tend to give undue influence to spurious features that have no effect on out-of-sample generalization. They also found that tuning the initial learning and decay scheme for Adam yields significant improvements over its default settings in all cases.

Background (2). The gradient updates for general stochastic gradient, stochastic momentum, and adaptive gradient methods, respectively, can be formalized as follows[156]

$$\text{[regular]} \quad \Delta w_{k+1} = -\alpha_k \widetilde{\nabla} f(w_k) \tag{683}$$

$$\text{[momentum]} \quad \Delta w_{k+1} = -\alpha_k \widetilde{\nabla} f\left(w_k + \gamma_k \Delta w_k\right) + \beta_k \Delta w_k \tag{684}$$

$$\text{[adaptive]} \quad \Delta w_{k+1} = -\alpha_k \boldsymbol{H}_k^{-1} \widetilde{\nabla} f\left(w_k + \gamma_k \Delta w_k\right) + \beta_k \boldsymbol{H}_k^{-1} \boldsymbol{H}_{k-1} \Delta w_k \tag{685}$$

where H_k is a p.d. matrix involving the entire sequence of iterates $(w_1, \dots w_k)$. For example, regular momentum would be γ_k=0, and Nesterov momentum would be γ_k=β_k. In practice, we basically always define H_k as the diagonal matrix:

$$H_k := \text{diag}\left[\left(\sum_{i=1}^{k} \eta_i \boldsymbol{g}_i \odot \boldsymbol{g}_i\right)^{1/2}\right] \tag{686}$$

The Potential Perils of Adaptivity (3). Consider the binary least-squares classification problem, where we aim to minimize

$$R_s[w] := \frac{1}{2} ||Xw - y||_2^2 \tag{687}$$

where $X \in \mathbb{R}^{n \times d}$ and $y \in \{-1, 1\}^n$.

Lemma 3.1

If there exists a scalar c s.t. $X\,sign(X^T y) = cy$, then (assuming $w_0 := 0$) AdaGrad, Adam, and RMSProp all converge to the unique solution $w \propto sign(X^T y)$.

[156] I'm defining $\Delta w_{k+1} \triangleq w_{k+1} - w_k$.

Y. Gal and Z. Ghahramani, "A Theoretically Grounded Application of Dropout in Recurrent Neural Networks," *University of Cambridge* (Oct 2016).

Background (3). In Bayesian regression, we want to infer the parameters $\boldsymbol{\omega}$ of some function $\boldsymbol{y} = f^{\boldsymbol{\omega}}(\boldsymbol{x})$. We define a prior, $p(\boldsymbol{\omega})$, and a likelihood,

$$p(y{=}d \mid \boldsymbol{x}, \boldsymbol{\omega}) = \text{Cat}_d\left(\text{softmax}\left(f^{\boldsymbol{\omega}}(\boldsymbol{x})\right)\right) \tag{688}$$

for a classification setting. Given a dataset $\boldsymbol{X}, \boldsymbol{Y}$, and some new point $\boldsymbol{x}^*$, we can predict its output via

$$p(\boldsymbol{y}^* \mid \boldsymbol{x}^*, \boldsymbol{X}, \boldsymbol{Y}) = \int p(\boldsymbol{y}^* \mid \boldsymbol{x}^*, \boldsymbol{\omega}) p(\boldsymbol{\omega} \mid \boldsymbol{X}, \boldsymbol{Y}) \mathrm{d}\boldsymbol{\omega} \tag{689}$$

In a **Bayesian neural network**, we place the prior over the NNs weights (typically Gaussians). The posterior $p(\boldsymbol{\omega} \mid \boldsymbol{X}, \boldsymbol{Y})$ is usually intractable, so we resort to **variational inference** to approximate it. We define our approximating distribution $\boldsymbol{q}(\boldsymbol{\omega})$ and aim to minimize the KLD:

$$\begin{aligned} \text{KL}\left(q(\boldsymbol{\omega}) || p(\boldsymbol{\omega} \mid \boldsymbol{X}, \boldsymbol{Y})\right) &\propto -\int q(\boldsymbol{\omega}) \log p(\boldsymbol{Y} \mid \boldsymbol{X}, \boldsymbol{\omega}) \mathrm{d}\boldsymbol{\omega} + \text{KL}(q(\boldsymbol{\omega}) || p(\boldsymbol{\omega})) && (690) \\ &= \sum_{i=1}^{N} \int q(\boldsymbol{\omega}) \log p(\boldsymbol{y}_i \mid f^{\boldsymbol{\omega}}(\boldsymbol{x}_i)) \mathrm{d}\boldsymbol{\omega} + \text{KL}(q(\boldsymbol{\omega}) || p(\boldsymbol{\omega})) && (691) \end{aligned}$$

Variational Inference in RNNs (4). The authors use MC integration to approximate the integral. The use only a single sample $\hat{\boldsymbol{\omega}} \sim q(\boldsymbol{\omega})$ for each of the N summations, resulting in an unbiased estimator. Plugging this in, we obtain our objective:

$$\mathcal{L} \approx -\sum_{i=1}^{N} \log p\left(\boldsymbol{y}_i \mid f_{\boldsymbol{y}}^{\hat{\boldsymbol{\omega}}_i}\left(f_{\boldsymbol{h}}^{\hat{\boldsymbol{\omega}}_i}\left(\boldsymbol{x}_{i,T}, \boldsymbol{h}_{T-1}\right)\right)\right) + \text{KL}(q(\boldsymbol{\omega}) || p(\boldsymbol{\omega})) \tag{692}$$

The crucial observations here are:

- For each sequence $\boldsymbol{x}_i$, we sample a new realization $\hat{\boldsymbol{\omega}}_i$.
- For each of the T symbols in $\boldsymbol{x}_i$, we use that *same* realization.

We define our approximating distribution to factorize over the weight matrices and their rows in $\boldsymbol{\omega}$. For each weight matrix row $\boldsymbol{w}_k$, we have

$$q(\boldsymbol{w}_k) \triangleq p\mathcal{N}(\boldsymbol{w}_k; \boldsymbol{0}, \sigma^2 I) + (1-p)\mathcal{N}(\boldsymbol{w}_k; \boldsymbol{m}_k, \sigma^2 I) \tag{693}$$

with $\boldsymbol{m}_k$ **variational parameter** (row vector).

Improving Neural Language Models with a Continuous Cache

Written by Brandon McKinzie

E. Grave, A. Joulin, and N. Usunier, "Improving Neural Language Models with a Continuous Cache," *Facebook AI Research* (Dec 2016).

The cache stores pairs (h_t, x_{t+1}) of the final hidden-state representation at time t, along with the word which was *generated*[157] based on this representation.

$$p_{vocab}(w \mid \boldsymbol{x}_{\langle 1\ldots t\rangle}) \propto \exp\left(h_t^T o_w\right) \tag{694}$$

$$p_{cache}(w \mid \boldsymbol{h}_{\langle 1\ldots t\rangle}, \boldsymbol{x}_{\langle 1\ldots t\rangle}) \propto \sum_{i=1}^{t-1} \mathbb{1}_{x_{i+1}=w} \exp\left(\theta h_t^T h_i\right) \tag{695}$$

$$= \sum_{\substack{(x,h)\in cache \\ s.t.\ x=w}} \exp\left(\theta h_t^T h\right) \tag{696}$$

$$p(w \mid \boldsymbol{h}_{\langle 1\ldots t\rangle}, \boldsymbol{x}_{\langle 1\ldots t\rangle}) = (1-\lambda) p_{vocab}(w \mid h_t) + \lambda p_{cache}(w \mid \boldsymbol{h}_{\langle 1\ldots t\rangle}, \boldsymbol{x}_{\langle 1\ldots t\rangle}) \tag{697}$$

where θ is a scalar parameter that controls the flatness of the cache distribution.

[157] They say this, but everything else in the paper strongly suggests they mean the next gold-standard input instead.

A. Bhowmick et al., "Protection Against Reconstruction and Its Applications in Private Federated Learning," *Apple, Inc.* (Dec 2018).

Introduction (1). In many scenarios, it is possible to reconstruct model inputs x given just $\nabla_\theta \ell(\theta; x, y)$. **Differential privacy** is one approach for obscuring the gradients such that guarantees can be made regarding risk of compromising user data x. **Locally private** algorithms, however, are preferred to DP when the user wants to keep their data private even from the data collector. The authors want to find a way to perform SGD while providing both local privacy to individual data X_i and stronger guarantees on the global privacy of the output $\hat{\theta}_n$ of their procedure.

Formally, say we have two users' data x and x' (both in $\mathcal{X}$) and some randomized mechanism $M : \mathcal{X} \mapsto \mathcal{Z}$. We say that M is **ε-local differentially private** if $\forall x, x' \in \mathcal{X}$ and sets $S \subset \mathcal{Z}$:

$$\frac{\Pr\left[M(x) \in S\right]}{\Pr\left[M(x') \in S\right]} \leq e^{\varepsilon} \tag{698}$$

Clearly, the RHS will need to be pretty big for this to be achievable. The authors claim that allowing $\varepsilon >> 1$ "may [still] provide meaningful privacy protections."

Privacy Protections (2). The focus here is on the **curious onlooker**: an individual (e.g. Apple PFL engineer) who can observe all updates to a model and communication from individual devices. Let X denote some user data. Let ΔW denote the weights difference after some model update using the data X. Let Z be the result of the randomized mapping $\Delta W \mapsto Z$. Our setting can be described with the Markov chain $X \to \Delta W \to Z$. The onlooker observes Z and wants to estimate some function $f(X)$ on the private data.

Separated private mechanisms (2.2). The authors propose, instead of a simple mapping $\Delta W \to Z$, to split it up into two parts: $Z_1 = M_1(U)$ and $Z_2 = M_2(R)$, where

$$U = \frac{\Delta W}{||\Delta W||_2} \tag{699}$$

$$R = ||\Delta W||_2 \tag{700}$$

Separated Differential Privacy

> *A pair of mechanisms M_1, M_2 mapping from $\mathcal{U} \times \mathcal{R}$ to $\mathcal{Z}_1 \times \mathcal{Z}_2$ is $(\varepsilon_1, \varepsilon_2)$-**separated differentially private** if M_1 is ε_1-locally differentially private and M_2 is ε_2-locally differentially private.*

Privatizing Unit ℓ_2 Vectors with High Accuracy (4.1). Given some vector $u \in \mathbb{S}^{d-1}$[158], we want to generate an ε-differentially private vector Z such that

$$\mathbb{E}\left[Z \mid u\right] = u \qquad \forall u \in \mathbb{S}^{d-1} \tag{701}$$

Privatized Unit Vector: `PrivUnit`$_2$

Sample random vector V:

$$V \sim \begin{cases} U\left(\{v \in \mathbb{S}^{d-1} \mid \langle v, u\rangle \geq \gamma\}\right) & \text{with probability } p \\ U\left(\{v \in \mathbb{S}^{d-1} \mid \langle v, u\rangle < \gamma\}\right) & \text{otherwise} \end{cases} \tag{702}$$

where $\gamma \in [0, 1]$ and $\mathrm{p} \geq \frac{1}{2}$ together control *accuracy* and *privacy*.

Let $\alpha = \frac{d-1}{2}$, $\tau = \frac{1+\gamma}{2}$, and

$$m = \frac{(1-\gamma^2)\alpha}{2^{d-2}(d-1)}\left[\frac{p}{B(\alpha,\alpha) - B(\tau;\alpha,\alpha)} - \frac{1-p}{B(\tau;\alpha,\alpha)}\right] \tag{703}$$

where $B(x, \alpha, \beta)$ is the incomplete beta function (see paper pg 17 for details).

Return $Z = \frac{1}{m} \cdot V$

Privatizing the Magnitude (4.3). We also need to privatize the weight delta norms. We want to return values $r \in [0, r_{max}]$ for some $r_{max} < \infty$.

[158] Here, this denotes an n-sphere:

$$\mathbb{S}^n \triangleq \{x \in \mathbb{R}^{n+1} : ||x|| = r\}$$

Context Dependent RNN Language Model

Written by Brandon McKinzie

T. Mikolov and G. Zweig, "Context Dependent Recurrent Neural Network Language Model," *BRNO and Microsoft* (2012).

Model Structure (2). Given one-hot input vector $\boldsymbol{x}_t$, output a probability distribution $\boldsymbol{y}_t$ for the next word. Incorporate a *feature vector* $\boldsymbol{f}_t$ that will contain topic information.

$$\boldsymbol{y}_t = \text{Softmax}\left(\boldsymbol{V}\boldsymbol{h}_t + \boldsymbol{G}\boldsymbol{f}_t\right) \tag{704}$$

$$\boldsymbol{h}_t = \sigma\left(\boldsymbol{U}\boldsymbol{x}_t + \boldsymbol{W}\boldsymbol{h}_{t-1} + \boldsymbol{F}\boldsymbol{f}_t\right) \tag{705}$$

LDA for Context Modeling (3). "Documents" fed to LDA here will be individual sentences. The generative process assumed by LDA is compactly defined by the following sequence of operations[159]:

$$N \sim \text{Poisson}(\xi) \tag{706}$$

$$\Theta \sim \text{Dir}(\alpha) \tag{707}$$

$$z_n \sim \text{Multinomal}(\Theta) \tag{708}$$

$$w_n \sim \Pr\left[w_n \mid z_n, \beta\right] \tag{709}$$

N: numb
$\Theta_i \equiv p(t$
z_n: topic

where $\Pr\left[w_n{=}a \mid z_n{=}b\right] = \beta_{b,a}$, so we are really just sampling from row z_n of β, where $\beta \in [0,1]^{Z\times V}$ (where Z is number of topics). The result of LDA is a learned value for α, and the topic distributions β.

$$\boldsymbol{f}_t = \frac{1}{Z}\prod_{i=0}^{K} \boldsymbol{t}_{x_{t-i}} \tag{710}$$

$$\boldsymbol{f}_t = \frac{1}{Z}\boldsymbol{f}_{t-1}^{\gamma}\boldsymbol{t}_{x_t}^{(}1-\gamma) \tag{711}$$

[159] α is a vector with number of elements equal to number of topics.

Strategies for Training Large Vocabulary Neural Language Models

Written by Brandon McKinzie

Chen et al., "Strategies for Training Large Vocabulary Neural Language Models," *FAIR* (Dec 2018). arXiv:1512.04906

Setup/Notation. Note that in everything below, the authors are using a rather primitive feed-forward network as their language model. To predict w_t it just concatenates the embeddings of the previous n words and feeds it through a k-layer FF network. Then, layer $k+1$ is the dense projection and softmax:

$$h^{k+1} = W^{k+1}h^k + b^{k+1} \in \mathbb{R}^V \tag{712}$$

$$y = \frac{1}{Z}\exp\left\{h^{k+1}\right\} \tag{713}$$

Using cross-entropy loss, the derivative of $\log p(w_t{=}i)$ wrt the jth element of the logits is:

$$\frac{\partial \log y_i}{\partial h_j^{k+1}} = \frac{\partial}{\partial h_j^{k+1}}\left[h_i^{k+1} - \log Z\right] \tag{714}$$

$$= \delta_{ij} - y_j \tag{715}$$

When computing gradients of the cross-entropy loss, y_i here is the ground truth. Therefore, to increase the probability of the correct token, we need to increase the logits element for that index, and decrease the elements for the others. Note how this implies we must compute the final activations for *all words in the vocabulary.*

Hierarchical Softmax (2.2). Group words into one of two clusters $\{c_1, c_2\}$, based on unigram frequency[160]. Then model $p(w_t \mid x) = p(c_t \mid x)p(w_t \mid c_t)$ where c_t is the class that word w_t was assigned to.

[160] For example, you could just put the top 50% in c_1 and the rest in c_2.

NCE (2.5). Define $P_{noise}(w)$ by the unigram frequency distribution. For each real token w_t in the training set, sample K noise tokens $\{n_k\}_{k=1}^{K}$. NCE aims to minimize

$$L_{NCE}(\{\boldsymbol{w}_1, \ldots, \boldsymbol{w}_N\}) = \sum_{i=1}^{N} \sum_{t=1}^{len(\boldsymbol{w}^{(i)})} \left[\log h(w_t^{(i)}) + \sum_{k=1}^{K} \log(1 - h(n_k^{(i)})) \right] \tag{716}$$

$$h(w_t) = \frac{P_{model}(w)}{P_{model}(w) + kP_{noise}(w)} \tag{717}$$

$$\approx \frac{\widetilde{P}_{model}(w)}{\widetilde{P}_{model}(w) + kP_{noise}(w)} \tag{718}$$

where the final approximation is what makes NCE less computationally expensive in practice than standard softmax. This would seem to imply that NCE should approach standard softmax (in terms of correctness) as k increases.

Takeaways.

- Hierarchical softmax is the fastest.
- NCE performs well on large-volume large-vocab datasets.
- Similar NCE values can result in very different validation perplexities.
- Sampled softmax shows good results if the number of negative samples is at 30% of the vocab size or larger.
- Sampled softmax has a lower ppl reduction per step than others.

Product quantization for nearest neighbor search

 Written by Brandon McKinzie

Jégou, "Product quantization for nearest neighbor search," (2011)

Vector Quantization. Denote the *index set* $\mathcal{I} = [0..k-1]$ and the set of reproduction values (a.k.a. **centroids**) c_i as $\mathcal{C} = \{\boldsymbol{c}_i \in \mathbb{R}^D : i \in \mathcal{I}\}$. We refer to $\mathcal{C}$ as the **codebook** of size k. A **quantizer** is a function $q : \boldsymbol{x} \mapsto \boldsymbol{q}(\boldsymbol{x})$, where $x \in \mathbb{R}^D$ and $\boldsymbol{q}(\boldsymbol{x}) \in \mathcal{C}$. We typically evaluate the quality of a quantizer with mean squared error of the reconstruction:

$$MSE(q) = \mathbb{E}_{\boldsymbol{x}\sim p(\boldsymbol{x})}\left[||\boldsymbol{x} - \boldsymbol{q}(\boldsymbol{x})||_2^2\right] \tag{719}$$

In order for an optimizer to be optimal, it must satisfy the **LLoyd optimality conditions**:

$$\textbf{(1)} \quad \boldsymbol{q}(\boldsymbol{x}) = \underset{c_i \in mathcalC}{\arg\min} ||\boldsymbol{x} - \boldsymbol{c}_i||_2 \tag{720}$$

$$\textbf{(2)} \quad \boldsymbol{c}_i = \mathbb{E}_{\boldsymbol{x}}\left[\boldsymbol{x} \mid i\right] \triangleq \int_{\mathcal{V}_i} \boldsymbol{x} p(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} \tag{721}$$

a.k.a. literally just K-means.

Product Quantization. Input vector $\boldsymbol{x} \in \mathcal{R}^D$ is split into m distinct subvectors $\boldsymbol{u}_j \in \mathbb{R}^{D/m}$, where $j \in [1..m]$. Note that D must be an integer multiple of m (i.e. $D = am$ for some $a \in \mathbb{Z}$).

$$\boldsymbol{x} = \underbrace{\boldsymbol{x}_{\langle 1 \dots D^* \rangle}}_{\boldsymbol{u}_1(\boldsymbol{x})}, \dots, \underbrace{\boldsymbol{x}_{\langle D - D^* + 1 \dots D \rangle}}_{\boldsymbol{u}_m(\boldsymbol{x})} \quad \rightarrow \quad \boldsymbol{q}_1(\boldsymbol{u}_1(\boldsymbol{x})), \dots, \boldsymbol{q}_m(\boldsymbol{u}_m(\boldsymbol{x})) \tag{722}$$

Note that each subquantizer q_j has its own index set $\mathcal{I}_j$ and codebook $\mathcal{C}_j$. Therefore, the final reproduction value of a product quantizer is identified by an element of the product set $\mathcal{I} = \mathcal{I}_1 \times \cdots \times \mathcal{I}_m$. The associated final codebook is $\mathcal{C} = \mathcal{C}_1 \times \cdots \times \mathcal{C}_m$.

Large Memory Layers with Product Keys

 Written by Brandon McKinzie

Lample et al., "Large Memory Layers with Product Keys," *FAIR* (July 2019). arXiv:1907.05242v1

Memory Design (3.1). The high-level structure (sequence of ops) is as follows:

1. **Query network** computes some query vector $\boldsymbol{q}$.
2. Compare $\boldsymbol{q}$ with each **product key** via some scoring function.
3. Select the k highest scoring product keys.
4. Compute output $\boldsymbol{m}(\boldsymbol{x})$ as weighted sum over the values associated with each of the top k keys from the previous step.

The **query network** is usually just a dense layer[161]. Since they want it to output query vectors with good coverage over the key space, the put a batch normalization layer before the query network[162].

The *standard* way of doing key assignment/weighting is as follows:

$$\text{[KNN]} \quad \mathcal{I} \triangleq \text{TopK}(\boldsymbol{q}(\boldsymbol{x})^T \boldsymbol{k}_i) \qquad 1 \leq i \leq \mathcal{K} \tag{723}$$

$$\text{[Normalize]} \quad \boldsymbol{w} = \text{Softmax}\left(\mathcal{I}\right) \tag{724}$$

$$\text{[Aggregate]} \quad \boldsymbol{m}(\boldsymbol{x}) = \sum_{i \in \mathcal{I}} w_i \boldsymbol{v}_i \tag{725}$$

where equation 723 is inefficient for large memory (key-value) stores. The authors propose instead a structured set of keys that they call **product keys**. Spoiler alert: it's just product quantization with m=2 subvectors. Instead of using the flat key set $\mathcal{K} \triangleq \{\boldsymbol{k}_1, \ldots, \boldsymbol{k}_{|\mathcal{K}|}\}$ with each $\boldsymbol{k}_i \in \mathbb{R}^{d_q}$ from earlier, we redefine it as

$$\mathcal{K} \triangleq \{(\boldsymbol{c}, \boldsymbol{c}') \mid c \in \mathcal{C},\ c' \in \mathcal{C}'\} \tag{726}$$

where both $\mathcal{C}$ and $\mathcal{C}'$ are sets of *sub-keys* $\boldsymbol{k}_i \in \mathbb{R}^{d_q/2}$.

Then...

1. Just run each of subvectors q_1 and q_2 through the standard TopK. You'll have k sub-keys for both, defined by their index into their respective codebook.
2. Let $\mathcal{K} := \{(\boldsymbol{c}_i, \boldsymbol{c}'_j) \mid i \in \mathcal{I}_{\mathcal{C}},\ \mathcal{I}_{\mathcal{C}'}\}$. This new reduced-size key set $\mathcal{K}$ has only $k \times k$ entries.

[161] They choose $d_q = 512$ as the output dimensionality of their query network.

[162] Recall that batch norm just normalizes the batch inputs to have 0 mean and unit standard deviation, followed by a scaling and bias factor.

3. Run the standard algorithm using the new reduced key set $\mathcal{K}$.

TODO: finish this note

Show, Ask, Attend, and Answer

Written by Brandon McKinzie

V. Kazemi and A. Elqursh, "Show, Ask, Attend, and Answer: A Strong Baseline For Visual Question Answering" *Google Research* (April 2017). arXiv:1704.03162v2

TL;DR: Good for a high-level overview of the VQA task, but extremely vague with so many details omitted it renders the paper fairly useless.

Method (3). Given a training set of image-question-answer triplets (I, q, a), learn to estimate the most likely answer $\hat{a}$ out of the set of most frequent answers[163] in the training data:

$$\hat{a} = \arg\max_a \Pr\left[a \mid I, q\right] \tag{727}$$

The method utilizes the following architectural components:

- **Image Embedding** (3.1). Extracts features ϕ=CNN(I).
- **Question Embedding** (3.2). Encode question q as the final state of LSTM: $\boldsymbol{s}$=LSTM(Embed(q)).
- **Stacked Attention** (3.3)[164] Seems like they feed Concat$[\boldsymbol{s}, \phi]$ through two layers of convolution to produce an output F_c for $c \in [1..C]$ (meaning they do C such convolutions separately an in parallel, like multi-head attention). This represents the scores for the attention function. The attention output, as usual, is computed as

$$\boldsymbol{x}_c = \sum_\ell \alpha_{c,\ell}\phi_\ell \tag{728}$$

$$\alpha_{c,\ell} \propto \exp F_c(\boldsymbol{s}, \phi_\ell) \tag{729}$$

  where ℓ appears to be over all [flattened] spatial indices of ϕ.
- **Classifier** (3.4). Concat the **image glimpses** $\boldsymbol{x}_c$ with the LSTM output $\boldsymbol{s}$ and feed through a couple FC layers to eventually obtain softmax probabilities over each possible answer a_i, $i \in [1..M]$.

[163] Same approach as how we define vocabulary in language modeling tasks.

[164] Authors do a laughably poor job at describing this part in any detail, so I'm taking the liberty of filling in the blanks. Blows my mind that papers this sloppy can even be published.

Dataset. Although, again, the authors are horribly vague/sloppy here, it *seems* like the data they use actually provides K "correct" answers for each image-question pair. The model loss is therefore an average NLL over the K true classes.

Did the Model Understand the Question?

 Written by Brandon McKinzie

Mudrakarta et al., "Did the Model Understand the Question?" *Univ. Chicago & Google Brain* (May 2018). arXiv:1805.05492v1

TL;DR. Basically all QA-related networks are dumb and don't learn what we think they learn.

- Networks tend to make predictions based a tiny subset of the input words. Due to this, altering the non-important words in ways that may drastically change the meaning of the question can have virtually no impact on the network's prediction.
- Networks assign high-importance to words like "there", "what", "how", etc. These are actually low-information words that the network should not heavily rely on.
- Networks rely on the image far more than the question.

Integrated Gradients (IG) (3). Purpose: "isolate question words that a DL system uses to produce an answer."

$$F(\boldsymbol{x}{=}\{x_1, \dots x_n\}) \in [0, 1] \tag{730}$$

$$A_F(\boldsymbol{x}, \boldsymbol{x}') = \{a_1, \dots a_n\} \in \mathbb{R}^n \tag{731}$$

where $\boldsymbol{x}'$ is some baseline input we use to compute the relative attribution of input x. The authors set $\boldsymbol{x}'$ as the "empty question" (sequence of padding values)[165].

> *Given an input x and baseline x', the **integrated gradient** along the ith dimension is as follows.*
>
> $$IG_i(x, x') \triangleq (x_i - x_i') \times \int_{\alpha=0}^{1} \frac{\partial F(x' + \alpha \times (x - x'))}{\partial x_i} \mathrm{d}\alpha \tag{732}$$

Interpretation: seems like IG gives us a better idea of the *total* "attribution" of each input dimension x_i relative to baseline x_i' along the line connecting x_i and x_i', instead of just the immediate derivative around x_i. Although, the fact that infinitesimal contributions could cancel each other out seems problematic (positive and negative gradients along the interpolation).

[165] The use the same context though (e.g. the associated image for VQA). Only the question is changed.

XLNet

 Written by Brandon McKinzie

Yang et al., "XLNet: Generalized Autoregressive Pretraining for Language Understanding" *CMU & Google Brain* (May 2018).

TL;DR: Instead of minimizing the NLL using $p(w_1, \dots, w_T)$, minimize over NLL's using every possible order of the given word sequence.

Background. Recall that BERT does denoising auto-encoding. Given text sequence $\boldsymbol{x} = \{x_1, \dots x_T\}$, BERT constructs a corrupted version $\hat{x}$ by randomly masking out some tokens. Let $\bar{\boldsymbol{x}}$ denote the tokens that were masked. The BERT training objective is then

$$\text{[BERT]} \quad \max_{\theta} \quad \log p_\theta(\bar{\boldsymbol{x}} \mid \hat{\boldsymbol{x}}) \approx \sum_{\bar{x} \in \bar{\boldsymbol{x}}} \log p_\theta(\bar{x} \mid \hat{\boldsymbol{x}}) \tag{733}$$

$$p(\bar{x} \mid \hat{\boldsymbol{x}}) = \text{Softmax}\left(H_\theta(\hat{\boldsymbol{x}})_t^T e(\bar{x})\right) \tag{734}$$

Objective & Architecture. Their proposed **permutation language modeling objective** is:

$$\max_{\theta} \quad \mathbb{E}_{\boldsymbol{z} \sim \mathcal{Z}_T} \left[\sum_{t=1}^{T} \log p_\theta(x_{z_t} \mid \boldsymbol{x}_{\langle z_1 \dots z_{t-1} \rangle}) \right] \tag{735}$$

where $\mathcal{Z}_T$ is the set of all possible permutations of the length-T index sequence $[1..T]$. To implement this, the authors had to re-parameterize the next-token distribution to be **target position aware**:

$$p_\theta(X_{z_t}{=}x \mid \boldsymbol{x}_{\langle z_1 \dots z_{t-1} \rangle}) = \text{Softmax}\left(g_\theta\left(\boldsymbol{x}_{\langle z_1 \dots z_{t-1} \rangle}, z_t\right)^T e\left(x\right) \right) \tag{736}$$

They accomplish this via **two-stream self-attention**, a technique that utilizes two sets of hidden representations (instead of one):

- **Content representation**: $h_{z_t} \triangleq h_\theta(\boldsymbol{x}_{\langle z_1 \dots z_t \rangle})$.
- **Query representation**: $g_{z_t} \triangleq g_\theta(\boldsymbol{x}_{\langle z_1 \dots z_{t-1} \rangle}, z_t)$.

The query stream is initialized with some vector $g_i^{(0)}=w$, and the content stream is initialized with word embedding $h_i^{(0)}=e(x_i)$. For the subsequent attention layers $1 \leq m \leq M$, they are computed respectively as follows:

$$g_{z_t}^{(m)} \leftarrow \text{Attention}(Q=g_{z_t}^{(m-1)}, K=V=\boldsymbol{h}_{z<t}^{(m-1)}) \tag{737}$$

$$h_{z_t}^{(m)} \leftarrow \text{Attention}(Q=h_{z_t}^{(m-1)}, K=V=\boldsymbol{h}_{z\leq t}^{(m-1)}) \tag{738}$$

In practice, in order to speed up optimization, the authors do **partial prediction**: only train to predict over $\boldsymbol{x}_{z_{>c}}$ targets rather than all of them.

Incorporating Ideas from Transformer-XL. Often times, sequences are too long to feed all at once. The authors adopt relative positional encoding and segment-level recurrence from Transformer-XL. To compute the attention update with memory on a given segment, we use the content representations from the *previous* segment, $\widetilde{\boldsymbol{h}}$, along with the current segment, $\boldsymbol{h}_{z_{\leq t}}$ as follows:

$$h_{z_t}^{(m)} \leftarrow \text{Attention}\left(Q=h_{z_t}^{(m-1)}, K=V=\left[\widetilde{\boldsymbol{h}}^{(m-1)}; \boldsymbol{h}_{z\leq t}^{(m-1)}\right]\right) \tag{739}$$

Transformer-XL

Written by Brandon McKinzie

Dai et al., "Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context" *CMU & Google Brain* (Jan 2019).

Segment-Level Recurrence with State Reuse. Denote two consecutive segments of length L as $\boldsymbol{s}_\tau = [x_{\tau,1}, \dots, x_{\tau,L}]$ and $\boldsymbol{s}_{\tau+1} = [x_{\tau+1,1}, \dots, x_{\tau+1,L}]$. Denote output of layer n given input segment $\boldsymbol{s}_\tau$ as $\boldsymbol{h}_\tau^n \in \mathbb{R}^{L\times d}$, where d is the hidden dimension. To obtain the output of layer n given the next segment, $\boldsymbol{s}_{\tau+1}$, do:

$$\begin{aligned}
\boldsymbol{h}_{\tau+1}^n &= \text{TransformerLayer}(\boldsymbol{q}_{\tau+1}^n, \boldsymbol{k}_{\tau+1}^n, \boldsymbol{v}_{\tau+1}^n) && (740)\\
&= \text{TransformerLayer}(\boldsymbol{h}_{\tau+1}^{n-1}\boldsymbol{W}_q^T,\ \widetilde{\boldsymbol{h}}_{\tau+1}^{n-1}\boldsymbol{W}_k^T,\ \widetilde{\boldsymbol{h}}_{\tau+1}^{n-1}\boldsymbol{W}_v^T) && (741)\\
\widetilde{\boldsymbol{h}}_{\tau+1}^{n-1} &= \left[\text{SG}\left(\boldsymbol{h}_\tau^{n-1}\right); \boldsymbol{h}_{\tau+1}^{n-1}\right] && (742)
\end{aligned}$$

where the concat in 742 is along the length (time) dimension. In other words, $\boldsymbol{Q}$ remains the same, but $\boldsymbol{K}$ and $\boldsymbol{V}$ get the previous segment prepended. Ultimately this only changes the inner dot products in the attention mechanism to attend over both segments. The L output attention vectors are therefore each weighted sums over the previous $2L$ timesteps instead of just L.

Relative Positional Encodings. Instead of absolute positional encodings (as regular transformers do), only encode the *relative* positional information in the hidden states. Ignoring the scale factor of $1/\sqrt{d_k}$, we can write the score for query vector $q_i = W_q(e_{x_i} + u_i)$ and key vector $k_j = W_k(e_{x_j} + u_j)$, for input embeddings e and positional encodings u as follows. Below it we show the authors proposed re-parameterized relative encoding version.

$$\begin{aligned}
A_{i,j}^{abs} &= e_{x_i}^T W_q^T W_k e_{x_j} + e_{x_i}^T W_q^T W_k u_j + u_i^T W_q^T W_k e_{x_j} + u_i^T W_q^T W_k u_j && (743)\\
A_{i,j}^{res} &= \underbrace{e_{x_i}^T W_q^T W_{k,E} e_{x_j}}_{\text{cont-based addr}} + \underbrace{e_{x_i}^T W_q^T W_{k,R} r_{i-j}}_{\text{cont-dep pos bias}} + \underbrace{\boldsymbol{u}^T W_{k,E} e_{x_j}}_{\text{global cont bias}} + \underbrace{\boldsymbol{v}^T W_{k,R} r_{i-j}}_{\text{global pos bias}} && (744)
\end{aligned}$$

where content is abbreviated as "cont" and positional is abbrev as "pos". I've shown all differences introduced by the second version in **red** font. It appears that r_{i-j} is literally just u_{i-j} but I guess using new letters is cool. Note that they separate W_k into **content-based** $W_{k,E}$ and **location-based** $W_{k,R}$.

Efficient Softmax Approximation for GPUs

Written by Brandon McKinzie

Grave et al., "Efficient Softmax Approximation for GPUs" *FAIR* (June 2017).

Adaptive Softmax: Two-Level

Partition the vocabulary $\mathcal{V}$ into two clusters $\mathcal{V}_h$ and $\mathcal{V}_t$, where

- $\mathcal{V}_h$ denotes the *head*, consisting of the most frequent words.
- $\mathcal{V}_t$ denotes the *tail*, associated with a **large number** of rare words.
- $|\mathcal{V}_h| << |\mathcal{V}_t|$ and $P(\mathcal{V}_h) >> P(\mathcal{V}_t)$.

To compute the probability of some word w given context h, do:

$$\Pr[w \mid h] = \begin{cases} P_{\mathcal{V}_h}(w \mid h) & \text{if } w \in \mathcal{V}_h \\ P_{\mathcal{V}_t}(w \mid h) P_{\mathcal{V}_h}(\text{tail} \mid h) & \text{otherwise} \end{cases} \tag{745}$$

where both $P_{\mathcal{V}_h}$ and $P_{\mathcal{V}_t}$ are modeled with a softmax over the words in their respective clusters ($P_{\mathcal{V}_h}$ also includes the special "tail" token).

More generally, we can extend the above algorithm to N clusters (instead of 2). We can also adapt the *capacity* of each cluster (varying their embedding size). The authors recommend, for each successive tail cluster, reducing the output size by a factor of 4. Of course, this then has to be followed by projecting back up to the number of words associated with the given cluster.

TODO: detail out how cross entropy loss is computed under this setup.

Adaptive Input Representations for Neural Language Modeling

Written by Brandon McKinzie

A. Baevski and M. Auli, "Adaptive Input Representations for Neural Language Modeling" *FAIR* (Feb 2019).

TL;DR: Literally just adaptive softmax but for the input embeddings. Official implementation can be found here.

Adaptive Input Representations (3). Same as Grave et al., they partition the vocabulary $\mathcal{V}$ into

$$\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2 \cup \ldots \cup \mathcal{V}_n \tag{746}$$

where $\mathcal{V}_1$ is the head and the rest are the tails (ordered by decreasing frequency). They reduce the capacity of each cluster by a factor of k=4 (also same as Grave et al.). Finally, they add linear projections for each cluster's embeddings in order to ensure they all result in d-dimensional output embeddings (even $\mathcal{V}_1$).

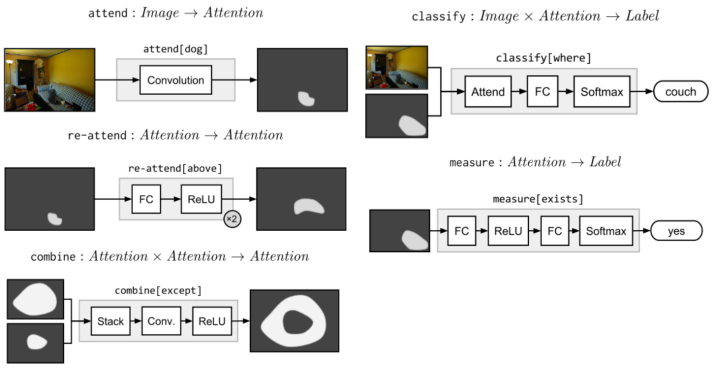
Neural Module Networks

Written by Brandon McKinzie

Andreas et al., "Deep Compositional Question Answering with Neural Module Networks" *UC Berkeley* (Nov 2015).

NMNs for Visual QA (4). Model and task overview:

- **Data**: 3-tuples (w, x, y) containing the question, image, and answer, respectively.
- **Model**: fully specified by a collection of **modules** $\{m\}$. Each module m has parameters θ_m and a **network layout predictor** $P(w)$ that maps from strings to networks. The high-level procedure is, for each (w, x, y), do:
  1. Instantiate a network based on $P(w)$.
  2. Pass the image x (and possibly w again) as inputs to the network.
  3. Obtain network outputs encoding $p(y \mid w, x; \theta)$.
- **Modules**:



From strings to networks (4.2).

1. Parse question w with the Stanford Parser to obtain universal dependency representation.
2. Filter dependencies to those connected to the wh-word in the question. Some examples:
   - *what is standing in the field* $\mapsto$ `what(stand)`
   - *what color is the truck* $\mapsto$ `color(truck)`
   - *is there a circle next to a square* $\mapsto$ `is(circle, next-to(square))`
3. Assign identities of modules (already have full network structure).
   - Leaves become `attend` modules.
   - Internal nodes become `re-attend` or `combine` modules.
   - Root nodes become `measure` (y/n questions) or `classify` (everything else) modules.

Answering natural language questions (4.3). They combine the results from the module network with an LSTM, which is fed the question as input and outputs a predictive distribution over the set of answers[166]. The final prediction is a geometric average of the LSTM output probabilities and the root module output probabilities.

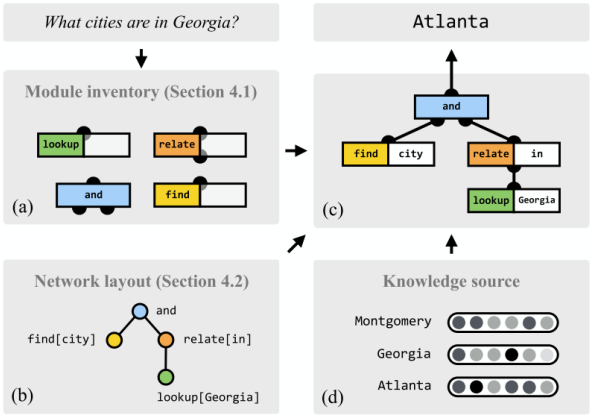
[166]This is the same distribution that the root module is trying to predict

Learning to Compose Neural Networks for QA

Written by Brandon McKinzie

Andreas et al., "Learning to Compose Neural Networks for Question Answering" *UC Berkeley* (June 2016).

TL;DR. Improve initial NMN work (previous note) by (1) learning network predictor ($P(w)$ in previous paper) instead of manually specifying it, and (2) extending visual primitives from previous work to reason over structured world representations.



Model (4). Training data consists of (world, question, answer) triplets (w, x, y). The model is built around two distributions:

- **layout model** $p(z \mid x; \theta_\ell)$ which predicts a layout z for sentence x.
- **execution model** $p_z(y \mid w; \theta_e)$ which applies the network specified by z to the world representation w.

Evaluating Modules (4.1). The execution model is defined as

$$p_z(y \mid w) = (\llbracket z \rrbracket_w)_y \tag{747}$$

where $\llbracket z \rrbracket_w$ denotes the output of network with layout z on input world w. The defining equations for all modules is as follows ($\sigma \equiv$ ReLU, $sm \equiv$ softmax):

$$\llbracket \texttt{lookup[i]} \rrbracket = e_{f(i)} \tag{748}$$

$$\llbracket \texttt{find[i]} \rrbracket = \mathrm{sm}(a \odot \sigma(Bv^i \oplus CW \oplus d)) \tag{749}$$

$$\llbracket \texttt{relate[i](h)} \rrbracket = \mathrm{sm}(a \odot \sigma(Bv^i \oplus CW \oplus D\bar{w}(h) \oplus e)) \tag{750}$$

$$\llbracket \texttt{and}(h^1, h^2, \ldots) \rrbracket = h^1 \odot h^2 \odot \cdots \tag{751}$$

$$\llbracket \texttt{describe[i](h)} \rrbracket = sm(A\sigma(B\bar{w}(h)) + v^i) \tag{752}$$

$$\llbracket \texttt{exists(h)} \rrbracket = sm\left(\left(\max_k h_k\right) a + b\right) \tag{753}$$

To train, maximize

$$\sum_{(w,y,z)} \log p_z(y \mid w; \theta_e) \tag{754}$$

Assembling Networks (4.2). **TODO**: finish note

End-to-End Module Networks for VQA

Written by Brandon McKinzie

R. Hu, J. Andreas, et al., "Learning to Reason: End-to-End Module Networks for Visual Question Answering" *UC Berkeley, FAIR, BU* (Sep 2017).

End-to-End Module Networks (3). High-level sequence of operations, given some input question and image:

1. Layout policy predicts a coarse functional expression that describes the structure of the computation.
2. Some subset of function applications within the expression receive parameter vectors predicted from the text.
3. Network is assembled with the modules according to layout expression to output an answer.

Attentional Neural Modules (3.1). A neural module m is a parameterized function $y = f_m(a_1, a_2, \ldots; x_{vis}, x_{txt}, \theta_m)$, where the a_i are image attention maps and the output y is either an image attention map or a probability distribution over answers. The full set of modules used by the authors, along with their inputs/outputs, is tabulated below.

Module name	Att-inputs	Features	Output	Implementation details
`find`	(none)	x_{vis}, x_{txt}	att	$a_{out} = \text{conv}_2(\text{conv}_1(x_{vis}) \odot W x_{txt})$
`relocate`	a	x_{vis}, x_{txt}	att	$a_{out} = \text{conv}_2(\text{conv}_1(x_{vis}) \odot W_1 \text{sum}(a \odot x_{vis}) \odot W_2 x_{txt})$
`and`	a_1, a_2	(none)	att	$a_{out} = \text{minimum}(a_1, a_2)$
`or`	a_1, a_2	(none)	att	$a_{out} = \text{maximum}(a_1, a_2)$
`filter`	a	x_{vis}, x_{txt}	att	$a_{out} = \texttt{and}(a, \texttt{find}[x_{vis}, x_{txt}]())$, *i.e.* reusing `find` and `and`
`[exist, count]`	a	(none)	ans	$y = W^T \text{vec}(a)$
`describe`	a	x_{vis}, x_{txt}	ans	$y = W_1^T (W_2 \text{sum}(a \odot x_{vis}) \odot W_3 x_{txt})$
`[eq_count, more, less]`	a_1, a_2	(none)	ans	$y = W_1^T \text{vec}(a_1) + W_2^T \text{vec}(a_2)$
`compare`	a_1, a_2	x_{vis}, x_{txt}	ans	$y = W_1^T (W_2 \text{sum}(a_1 \odot x_{vis}) \odot W_3 \text{sum}(a_2 \odot x_{vis}) \odot W_4 x_{txt})$

Note that, whereas the original NMN paper (see previous note) instantiated module types based on words (e.g. describe[shape] vs describe[where]) and gave different instantiations different parameters, this paper has a single module for each module type (no"instances" anymore). To distinguish between cases where e.g. `describe` should describe a shape vs describing a location, the module incorporates a text feature $x_{txt}^{(m)}$ computed separately/identically for each module m:

$$x_{txt}^{(m)} = \sum_{i=1}^{T} \alpha_i^{(m)} w_i \tag{755}$$

Layout Policy with Seq2Seq RNN (3.2). **TODO** finish note

Fast Multi-language LSTM-based Online Handwriting Recognition

Written by Brandon McKinzie

Carbune et al., "Fast Multi-language LSTM-based Online Handwriting Recognition" *Google AI Perception* (Feb 2019).

Introduction (1). Task: given input strokes, i.e. sequences of points (x, y, t), output it in the form of text.

Model Architecture (2). The high-level sequences of operations is:

1. Input time series $(v_1, \dots, v_T)$ encoding user input. The authors experiment with two representations:
   (a) *Raw touch points*: sequence of 5-dimensional points $(x_i, y_i, t_i, p_i, n_i)$, where t_i is seconds since first touch point in current observation, p_i is binary-valued equal to 0 if pen-up, else 1 if pen-down, and n_i is binary on start-of-new-stroke (1 if True).
   (b) *Bézier curves*: TL;DR is that they model x, y, t each as a cubic polynomial over a new variable $s \in [0, 1]$. Ultimately this means solving for some coefficients Ω of a linear system of equations: $V^T Z = V^T V \Omega$.
2. Several BiLSTM layers for contextual character embedding.
3. Softmax layer providing character probabilities at each time step.
4. CTC decoding with beam search. They also incorporate **feature functions** into the output logits to help with decoding. They use the following 3 feature functions:
   (a) Character language models. A 7-gram LM over Unicode codepoints using Stupid back-off.
   (b) Word language models. 3-grams pruned to between 1.25M and 1.5M entries.
   (c) Character classes. Scoring heuristic which boosts the score of characters from the LM's alphabet.

Training (3). Training happens in two stages, each on a different dataset:

1. Training neural network model with CTC loss on large dataset.
2. Decoder tuning using **Bayesian optimization** through **Gaussian Processes** in Vizier[167].

[167] Vizier is a program made by Google for black-box tuning

Multi-Language Online Handwriting Recognition

 Written by Brandon McKinzie

Keysers et al., "Multi-Language Online Handwriting Recognition" *Google* (June 2017).

System Architecture (3). Segment-and-decode approach consisting of the following steps:

- Preprocessing (4).
  1. Resampling.
  2. Slope correction.
- **Segmentation**[168] and search lattice creation (5).
  1. Segmentation goal: obtain high *recall* of all actual character boundaries. Accomplished via a heuristic which creates a set of potential cut points and then a neural net which assigns a score to each.
  2. Segmentation lattice: a graph (V, E) of ink segments. Each segment is identified by a unique integer index.
     - Nodes (in V) define the path of ink segments up to that point (e.g. $\{1, 0, 2\}$) (i.e. a character hypothesis)
     - Edges (in E) from a given node v indicate the ink segments which are grouped in a character hypothesis. For example, if v=$\{i, j\}$ has some edge k, then that edge will have node $\{i, j, k\}$ on the other end, and $\{i, j, k\}$ is a valid character hypothesis.

     It appears that each node (assign from the empty start node) is passed to the next stage as a character hypothesis to be scored/classified.
- Generation & scoring of **character hypotheses**[169] (5.3). Goal: cetermine the characters most likely to have been written.
  1. Feature extraction: they make a fixed-length dense feature vector containing *point-wise* and *character-global* features.
  2. Classification: single hidden layer NN with tanh activation followed by softmax.
  3. Create a labeled latice which will later be decoded to find the final recognition result.
- Best path search in the resulting lattice using additional knowledge sources (6).

[168] **Segmentation/cut point**: a point at which another character my start. **Segment**: the (partial) strokes between 2 consecutive segmentation points.

[169] **Character hypothesis**: a set of one or more segments (not necessarily consecutive).

Language Models (6.1). They utilize two types of language models:

- Stupid-backoff entropy-pruned 9-gram character LM. This is their "main" LM. Depending on the language, they use about 10M to 100M n-grams.
- Word-based probabilistic finite automaton. Creating using 100K most frequent words of a language.

Search (6.2). Goal: obtain a recognition result by finding the best path from the source node (no ink recognized) to the target node (all ink recognized). Algorithm: ink-aligned beam search that starts at the start node and proceeds through the lattice in topological order.

Modular Generative Adversarial Networks

Written by Brandon McKinzie

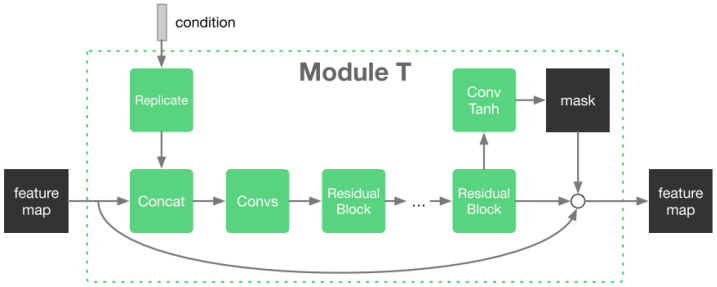
Zhao et al., "Modular Generative Adversarial Networks" *UBC, Tencent AI Lab* (April 2018).

TL;DR. Task(s): multi-domain image **generation** and image-to-image **translation**.

Network Construction (3.2). Let x and y denote the input and target image, respectively, wherever applicable. Let $\boldsymbol{A} = \{A_1, A_2, \cdots, A_n\}$ denote an **attribute set**. Four types of modules are used:

1. Initial module is task-dependent (below). Output is feature map in $\mathbb{R}^{C\times H\times W}$.
   - **[translation] encoder** $\boldsymbol{E}$: $x \mapsto \boldsymbol{E}(x)$
   - **[generation] generator** $\boldsymbol{G}$: $(z, a_0) \mapsto \boldsymbol{G}(z, a_0)$ where z is random noise and a_0 is a condition vector representing auxiliary information.
2. **transformer(s)** $\boldsymbol{T}_i$: $E(x) \mapsto \boldsymbol{T}_i(\boldsymbol{E}(x), a_i)$. Modifies repr of attrib a_i in the FM.
3. **reconstructor** $\boldsymbol{R}$: $(\boldsymbol{T}_i, \boldsymbol{T}_j, \ldots) \mapsto y$. Reconstructs image from an intermediate FM.
4. **discriminator** $\boldsymbol{D}_i$: $\boldsymbol{R} \mapsto \{0, 1\} \times \text{Val}(a_i)$. Predicts probability that R came from p_{true}, and the [transformed] value of a_i.

The authors emphasize that the transformer module is their core module. It's architecture is illustrated below.



Loss Function (3.4).

$$\mathcal{L}_D(\boldsymbol{D}) = -\sum_{i=1}^{n} \mathcal{L}_{adv_i} + \lambda_{cls} \sum_{i=1}^{n} \mathcal{L}^r_{cls_i} \tag{756}$$

$$\mathcal{L}_G(\boldsymbol{E}, \boldsymbol{T}, \boldsymbol{R}) = \sum_{i=1}^{n} \mathcal{L}_{adv_i} + \lambda_{cls} \sum_{i=1}^{n} \mathcal{L}^f_{cls_i} + \lambda_{cyc} \left(\mathcal{L}^{\boldsymbol{ER}}_{cyc} + \sum_{i=1}^{n} \mathcal{L}^{\boldsymbol{T}_i}_{cyc} \right) \tag{757}$$

$$\begin{aligned} \mathcal{L}_{adv_i}(\boldsymbol{E}, \boldsymbol{T}_i, \boldsymbol{R}, \boldsymbol{D}_i) = {} & \mathbb{E}_{y \sim p_{data}(y)} \left[\log \boldsymbol{D}_i(y) \right] + \\ & \mathbb{E}_{x \sim p_{data}(x)} \left[\log \left(1 - \boldsymbol{D}_i \left(\boldsymbol{R} \left(\boldsymbol{T}_i \left(\boldsymbol{E}(x) \right) \right) \right) \right) \right] \end{aligned} \tag{758}$$

$$\mathcal{L}^r_{cls_i} = -\mathbb{E}_{x, c_i} \left[\log \boldsymbol{D}_{cls_i}(c_i \mid x) \right] \tag{759}$$

$$\mathcal{L}^f_{cls_i} = -\mathbb{E}_{x, c_i} \left[\log \boldsymbol{D}_{cls_i}(c_i \mid \boldsymbol{R}(\boldsymbol{E}(\boldsymbol{T}_i(x)))) \right] \tag{760}$$

$$\mathcal{L}^{\boldsymbol{ER}}_{cyc} = \mathbb{E}_x \left[||\boldsymbol{R}(\boldsymbol{E}(x)) - x||_1 \right] \tag{761}$$

$$\mathcal{L}^{\boldsymbol{T}_i}_{cyc} = \mathbb{E}_x \left[||\boldsymbol{T}_i(\boldsymbol{E}(x)) - \boldsymbol{E}(\boldsymbol{R}(\boldsymbol{T}_i(\boldsymbol{E}(x))))||_1 \right] \tag{762}$$

where n is the total number of controllable attributes.

Transfer Learning from Speaker Verification to TTS

Written by Brandon McKinzie

Jia et al., "Transfer Learning from Speaker Verification to Multispeaker Text-To-Speech Synthesis" *Google* (Jan 2019).

TL;DR: TTS that's able to generate speech in the voice of different speakers, including those unseen during training.

Multispeaker Speech Synthesis Model (2). System is composed of three independently trained NNs:

1. **Speaker Encoder**. Computes a fixed-dimensional vector from a speech signal.
2. **Synthesizer**. Predicts a **mel spectrogram** from a sequence of **grapheme** or **phoneme** inputs, conditioned on the speaker vector. Extension of **Tacotron 2** to support multiple speakers.
3. **Vocoder**. Autoregressive WaveNet, which converts the spectrogram into time domain waveforms.

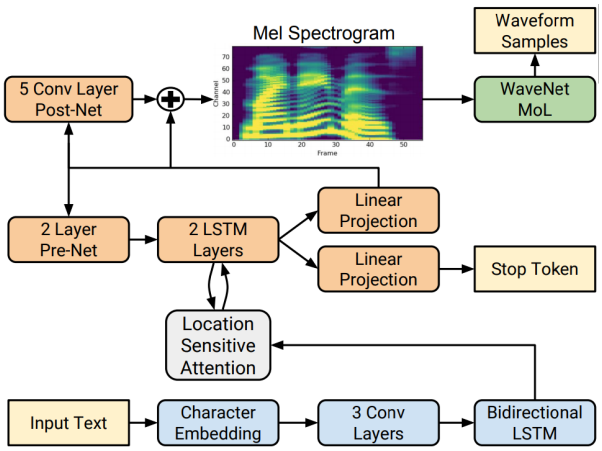
Tacotron 2

Written by Brandon McKinzie

Shen et al., "NATURAL TTS SYNTHESIS BY CONDITIONING WAVENET ON MEL SPECTROGRAM PREDICTIONS" *Google*, *UCB* (Feb 2018).

TL;DR: Seq2seq network for mapping characters to mel-scale spectrograms[170], followed by a modified WaveNet vocoder.

Spectrogram Prediction Network (2.2). Illustrated below.



- Encoder: character language model architecture. They use convolution layers in the middle under the hypothesis that these should learn ngram-like representations.
- Attention Decoder: **location-sensitive** encoder-decoder attention[171]. At each timestep, instead of incorporating the previous decoder prediction, they first feed it through a **pre-net** (2 FF layers). Furthermore, they also have a **post-net** which predicts a residual to add to the prediction "to improve the overall reconstruction".

  Finally, they also project the decoder/attention outputs to a scalar (sigmoid activation) for predicting probability that output sequence has completed.

[170] A **mel-frequency spectrogram** is obtained by applying a nonlinear transform to the frequency axis of the short-time Fourier transform (STFT).

[171] Basically additive attention with cumulative weights.

WaveNet Vocoder (2.3). Inverts the mel spectrogram feature representation into time-domain waveform samples. Specifically, I assume this means they take each time slice of the predicted spectrogram (a vector over frequencies?) and do **local conditioning** (as described in WaveNet paper) with $\boldsymbol{h}_t$ the spectrogram at time slice t.

Digression: Fourier Transforms

We measure sound by pressure/amplitudes as a function of time. What we actually measure is the *sum total* of all the individual sinusoidal waves each with their own frequency/amplitude. To reiterate: in the *time domain* our x-axis is time (duh) and our y-axis is amplitude. The **Fourier transform** maps our time-domain representation into a *frequency domain*. Now, the x-axis is frequency (duh) and the y-axis represents how much of our original signal consisted of waves with a given frequency. For example, if our original signal was literally a 3 Hz wave superimposed over some 5 Hz wave, our frequency-domain plot would show two spikes at x=3 and x=5, and be zero everywhere else.

Formally, let $g(t)$ denote the amplitude of some sound wave at time t.

$$g(t)e^{-2\pi ift} \tag{763}$$

where

- The negative sign in the exponent is a convention that we should think about *clockwise* rotations.
- f denotes the "winding" frequency: the number of full cycles represented in our wound up graph.
- Multiplying by $g(t)$ means that the distance-to-origin (magnitude) at time t will always equal $g(t)$.

Similarly, the **inverse Fourier transform** maps from frequency domain to time domain.
Remember that the trick for identifying the component frequencies is by computing the **center of mass** of this formula:

$$\frac{1}{t_2 - t_1}\int_{t_1}^{t_2} g(t)e^{-2\pi ift}\mathrm{d}t \tag{764}$$

where, to do this for one full circle, we'd set t_1=0 and t_2=$1/f$. If our signal $g(t)$ does contain the frequency f, this integral will be relatively large in comparison with other frequencies. The final formula for the Fourier transform is just removing the scale factor (i.e. the FT is just the CoM scaled by the time interval of our signal):

$$\hat{g}(\omega) \triangleq \Re\left\{\int_{t_1}^{t_2} g(t)e^{-2\pi i\omega t}\mathrm{d}t\right\} \tag{765}$$

More intuition regarding removal of the scale factor: if there is a component wave in $g(t)$ that only exists for a small portion of time δt, it would have a smaller value of $\hat{g}(\omega)$ than a component of some different frequency (but same amplitude) that persisted throughout the entirety of our time interval.

Digression: Spectrograms

Now that we know about Fourier transforms (above example), we can define what a spectrogram is. Say that, instead of a single function $g(t)$, I split time into various windows $\{t_0, t_1, t_2, t_T\}$ and store a separate function, $g_\tau(t)$ with $1 \leq \tau \leq T$, for each window. Then, I apply the FT on each function, resulting in a set of $\hat{g}_\tau(\omega)$. If I plot a 2D heatmap with my x-axis denoting the window τ, the y-axis denoting frequency ω, and values (color) denoting $\hat{g}_\tau(\omega)$, I have a **spectrogram** (technically I need to transform my y-axis to $\log\omega$ and my color/value axis to Decibels).

Glow

Written by Brandon McKinzie

Kingma et al., "Glow: Generative Flow with Invertible 1x1 Convolutions" *OpenAI*, (July 2018).

Flow-based Generative Models (2).

$$\boldsymbol{z} \sim p_\theta(\boldsymbol{z}) \tag{766}$$

$$\boldsymbol{x} = \boldsymbol{g}_\theta(\boldsymbol{z}) \tag{767}$$

where $\boldsymbol{g}$ is invertible. *Inference* is done by $\boldsymbol{z} = \boldsymbol{f}_\theta(\boldsymbol{x}) = \boldsymbol{g}_\theta^{-1}(\boldsymbol{x})$.

$$\log p_\theta(\boldsymbol{x}) = \log p_\theta(\boldsymbol{z}) + \log|\det \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}| \tag{768}$$

$$= \log p_\theta(\boldsymbol{z}) + \sum_{i=1}^{K} \log|\det \frac{\partial \boldsymbol{h}_i}{\partial \boldsymbol{h}_{i-1}}| \tag{769}$$

The log-determininant gives the change in log-density when going $\boldsymbol{h}_{i-1} \to \boldsymbol{h}_i$. We can see that maximum likelihood will encourage $\boldsymbol{f}_\theta(\boldsymbol{x})$ to increase volume.

WaveGlow

Written by Brandon McKinzie

Prenger et al., "WaveGlow: A Flow-Based Generative Network for Speech Synthesis"' *NVIDIA*, (Oct 2018).

TL;DR: flow-based **vocoder**[172]

WaveGlow (2). Model the distribution of audio samples *conditioned on* a mel-spectrogram. Note that the *forward pass* is defined as going from $\boldsymbol{x}$ to *vecz*. Train by minimizing the negative log-likelihood:

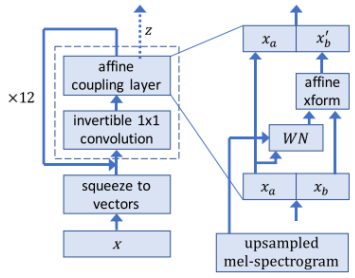
$$\boldsymbol{z} \sim \mathcal{N}(\boldsymbol{z}; \boldsymbol{0}, \boldsymbol{I}) \tag{770}$$

$$\boldsymbol{x} = \boldsymbol{f}_0 \circ \boldsymbol{f}_1 \circ \cdots \circ \boldsymbol{f}_k(\boldsymbol{z}) \tag{771}$$

$$\log p_\theta(\boldsymbol{x}) = \log p_\theta(\boldsymbol{z}) + \sum_{i=1}^{k} \log |\det \left(\boldsymbol{J}\left(\boldsymbol{f}_i^{-1}(\boldsymbol{x}) \right) \right)| \tag{772}$$

$$\boldsymbol{z} = \boldsymbol{f}_k^{-1} \circ \boldsymbol{f}_{k-1}^{-1} \circ \cdots \circ \boldsymbol{f}_0^{-1}(\boldsymbol{x}) \tag{773}$$

and $\boldsymbol{x}$ denotes the output audio. The mel-spectrogram conditioning happens in the affine coupling layer(s). Training penalizes the norm of $\boldsymbol{z}$ and encourages each layer f_i^{-1} to increase the log-density volume of the previous layer.



$$\boldsymbol{x}_a, \boldsymbol{x}_b = split(\boldsymbol{x}) \tag{774}$$

$$\log \boldsymbol{s}, \boldsymbol{t} = WN(\boldsymbol{x}_a, mel - spectrogram) \tag{775}$$

$$\boldsymbol{x}'_b = \boldsymbol{s} \odot \boldsymbol{x}_b + \boldsymbol{t} \tag{776}$$

$$\boldsymbol{f}^{-1}_{coupling}(\boldsymbol{x}) = concat(\boldsymbol{x}_a, \boldsymbol{x}_b) \tag{777}$$

[172] Vocoder: network that transforms time-aligned features (e.g. spectrograms) into audio samples.

Solving Rubik's Cube with a Robot Hand

Written by Brandon McKinzie

Akkaya et al., "Solving Rubik's Cube with a Robot Hand"' *OpenAI*, (Oct 2019).

TL;DR: **Automatic Domain Randomization** (ADR) + robot platform = solving rubik's cube from simulation alone. The two main structural components are (1) the hand and (2) a few cameras that observe the pose/state of the hand/cube.

ADR (5). They denote an "environment" as e_λ, parameterized by $\lambda \in \mathbb{R}^d$. In other words, there are d parameters/knobs they can fiddle with in simulation to change various characteristics of the given environment[173].. Here, the authors choose $d' = 2d$. Let $\phi^L, \phi^H \in \mathbb{R}^d$ be some partition of ϕ.

$$P_\phi(\lambda) = \prod_{i=1}^{d} P_\phi(\lambda_i) = \prod_{i=1}^{d} U(\phi_i^L, \phi_i^H) \tag{778}$$

$$\mathcal{H}(P_\phi) = -\frac{1}{d}\int P_\phi(\lambda)\ln P_\phi(\lambda)\mathrm{d}\lambda = \frac{1}{d}\sum_{i=1}^{d}\ln\left(\phi_i^H - \phi_i^L\right) \tag{779}$$

where they define a normalized entropy $\mathcal{H}$ in nats/dimension. The pairs (ϕ_i^L, ϕ_i^H) are referred to as **boundary values**.

[173] I guess it comes down to semantics whether we interpret a $\Delta\lambda$ as a new environment or just the same environment with different characteristics.

Automatic Domain Randomization

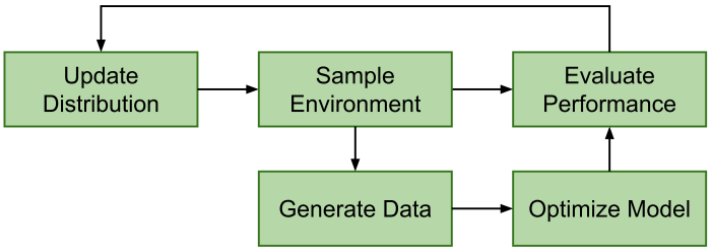
TL;DR: algorithm for updating parameter ranges ϕ_i in distribution P_ϕ.

Init with **performance buffers** $\{D_i^L, D_i^H\}_{i=1}^d$ (presumably empty at start), **thresholds** m, t_L, t_H ($t_L < t_H$). Repeat the following until "training is complete":

1. Sample $\lambda \sim P_\phi$.
2. Randomly sample index $1 \leq i \leq d$, and number $x \sim U(0, 1)$.
3. If $x < 0.5$, set D_i=D_i^L, λ_i=ϕ_i^L, else D_i=D_i^H, λ_i=ϕ_i^H.
4. Collect model performance p on environment parameterized by λ.
5. $D_i \leftarrow D_i \cup \{p\}$.
6. If Length$(D_i) \geq m$:
   (a) $\bar{p} \leftarrow$ Average(D_i)
   (b) CLEAR(D_i)
   (c) if $\bar{p} \geq t_H$ increase ϕ_i by Δ. If $\bar{p} \leq t_L$, decrease ϕ_i by Δ. (not sure if this means increase range scale, or shift range left/right (seems like the former))

In English: we sample environment params λ at each iteration, but then set a random dimension of λ to a boundary value. We measure model performance and append to performance buffer index associated with the boundary value. After we have enough performance measurements for a given boundary value index i, we increase the associated pair (ϕ_i^L, ϕ_i^H).

To generate training data, they use the ADR algorithm above in conjunction with sampling from the [changing each time ADR is run] distribution P_ϕ and running the model (running the model results in new training data[174]).



TODO: finish note

[174] **TODO**: how? how does running a model generate data? what is the data anyway?

Fine-Tuning Language Models from Human Preferences

Written by Brandon McKinzie

Ziegler et al., "Fine-Tuning Language Models from Human Preferences" *OpenAI*, (Sep 2019).

Methods (2). Setup: vocab Σ, language model ρ. Want to model probability of response sequence y given input sequence x. They initialize a **policy** $\pi = \rho$, then fine-tune π with RL. Instead of *defining* a reward function $r : X \times Y \mapsto \mathbb{R}$, the *learn* one using a dataset S of human annotations. Each element of S is a tuple $(x, y_0, y_1, y_2, y_3, b)$ – each y_i is a multiple-choice option presented to the human labeler, and the human selects option $0 \leq b < 3$. They train the reward model r with loss

$$\text{loss}(r) = \mathbb{E}_{x,\{y_i\},b\sim S}\left[\log \frac{e^{r(x,y_b)}}{\sum_i e^{r(x,y_i)}}\right] \tag{780}$$

They initialize r as a random linear function of the final embedding output of ρ. They fine-tune π to optimize r, performing RL on the modified reward:

$$R(x,y) = r(x,y) - \beta \log \frac{\pi(y \mid x)}{\rho(y \mid x)} \tag{781}$$

how is this any different from just continuing training the LM on the human annotations?

Deep Double Descent

Written by Brandon McKinzie

Nakkiran et al., "Deep Double Descent: Where Bigger Models and More Data Hurt" *OpenAI*, (Dec 2019).

TL;DR: For a variety of DL tasks, increasing model size (or training epochs) first leads to *worse* performance, and then gets better. Define a new complexity measure called **effective model complexity**.

> *Informally, our intuition is that for model-sizes at the interpolation threshold, there is effectively only one model that fits the train data and this interpolating model is very sensitive to noise in the train set and/or model mis-specification. That is, since the model is just barely able to fit the train data, forcing it to fit even slightly-noisy or mis-specified labels will destroy its global structure, and result in high test error.*

Introduction (1). Bias-variance tradeoff suggests that increasing model complexity results in lower bias and higher variance. After a certain threshold [of increasing model complexity], conventional wisdom says that the model will "overfit" with the variance term dominating the test error. Therefore, increasing the complexity beyond this threshould *should* merely result in a larger variance term and thus worse MSE (i.e. *larger models are worse* [after a certain point]). Conventioanal wisdom also tells us that *more data is always better* [for improving the test MSE].

Results (2). Define a **training procedure** $\mathcal{T}$ to be any procedure that takes as input a training set $S = \{(x_i, y_i)\}_{i=1}^n$ and outputs a classifier $\mathcal{T}(S) : x \mapsto y$.

Effective Model Complexity

> *The **EMC** of $\mathcal{T}$, wrt distribution $\mathcal{D}$ and $\epsilon > 0$, is defined as:*
>
> $$\text{EMC}_{\mathcal{D},\epsilon}(\mathcal{T}) := \max\{n \mid \mathbb{E}_{S\sim\mathcal{D}^n}\left[\text{Error}_S(\mathcal{T}(S))\right] \leq \epsilon\} \tag{782}$$
>
> *where $\text{Error}_S(M)$ is the mean error of model M on train samples S.*

In other words, the EMC of $\mathcal{T}$ is the maximum number of training samples for which $\mathcal{T}$ gets [on average] zero *training error*. The authors then hypothesize the 3 following regimes (assuming n training examples):

- **Under-parameterized**: $\text{EMC}_{\mathcal{D},\epsilon}(\mathcal{T}) << n$. Any $\delta\mathcal{T}$ resulting in larger $\text{EMC}_{\mathcal{D},\epsilon}(\mathcal{T})$ will *decrease* test error.
- **Over-parameterized**: $\text{EMC}_{\mathcal{D},\epsilon}(\mathcal{T}) >> n$. (same as above).
- **Critically parameterized**: $\text{EMC}_{\mathcal{D},\epsilon}(\mathcal{T}) \approx n$. Any $\delta\mathcal{T}$ resulting in larger $\text{EMC}_{\mathcal{D},\epsilon}(\mathcal{T})$ may *decrease or increase* test error.