Contents

Math and Machine Learning Basics

Contents

Linear Algebra (Quick Review) (Ch. 2)

Written by Brandon McKinzie

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Functions of RVs.

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

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

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

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

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

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

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

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

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

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

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

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

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

Gradient-based optimization. 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

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

The Hessian is the Jacobian of the gradient.

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

The second derivative in a specific direction $\hat{\boldsymbol{d}}$ is given by $\hat{\boldsymbol{d}}^T \boldsymbol{H} \hat{\boldsymbol{d}}$[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.

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

[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 $\mathbb{E}_{\boldsymbol{x} \sim \hat{p}_{data}} \left[\log \left(\hat{p}_{data}(\boldsymbol{x})\right)\right]$:

$$\begin{aligned} D_{KL}(\hat{p}_{data} || p_{model}) &= \mathbb{E}_{\boldsymbol{x} \sim \hat{p}_{data}} \left[\log \left(\hat{p}_{data}(\boldsymbol{x})\right) - \log \left(p_{model}(\boldsymbol{x})\right)\right] &(48)\\ &= \left(\frac{1}{N}\left(\sum_{i=1}^{N} \log\left(\text{Counts}(\boldsymbol{x}_i)\right)\right) - \log N\right) - \mathbb{E}_{\boldsymbol{x} \sim \hat{p}_{data}} \left[\log \left(p_{model}(\boldsymbol{x})\right)\right] &(49)\end{aligned}$$

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

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

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

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.

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

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

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

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* $p(\boldsymbol{\theta} \mid x^{(1)}, \dots, x^{(m)})$, we must integrate over this to get the predicted likelihood of the next sample $x^{(m+1)}$:

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

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

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*

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

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

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

[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.

1.4.4 Supervised Learning Algorithms (5.7)

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

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

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

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

Equation 5.81 is the definition of logistic regression

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

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

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

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

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

Deep Networks: Modern Practices

Contents

Deep Feedforward Networks (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...

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

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

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

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

→ 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],

$\boldsymbol{x} \in \mathbb{R}^m$
$\boldsymbol{y} \in \mathbb{R}^n$
$z : \mathbb{R}^n \to \mathbb{R}$
$g : \mathbb{R}^m \to \mathbb{R}^n$

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

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

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 $\boldsymbol{x}$.

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

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

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**:

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

risk definitions

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 $\neq$ 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."[13] 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.

$$\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_{i}^{m} L\left(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}\right) \tag{77}$$

$$\boldsymbol{s} \leftarrow \frac{1}{1-\rho_1^{t-1}} \left[\rho_1 \boldsymbol{s} + (1-\rho_1)\boldsymbol{g}\right] \tag{78}$$

$$\boldsymbol{r} \leftarrow \frac{1}{1-\rho_2^{t-1}} \left[\rho_2 \boldsymbol{r} + (1-\rho_2)\boldsymbol{g} \odot \boldsymbol{g}\right] \tag{79}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{s} \tag{80}$$

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

[13] Review of moments: The nth moment of a real-valued continuous function $f(x)$ of a real variable about a value c is

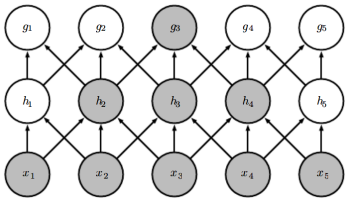
$$\mu_n = \int_{-\infty}^{\infty} (x-c)^n f(x) \mathrm{d}x \tag{76}$$

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[14], 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.

[14] 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.

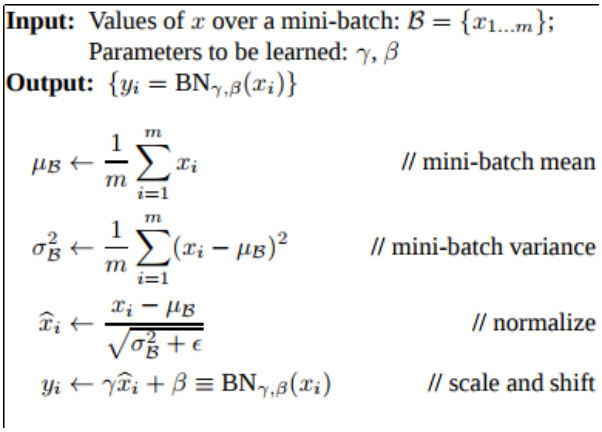
Pooling. Helps make the representation approximately **invariant** to small translations of the input. The use of pooling can be viewed as adding an infinitely strong prior[15] that the function the layer learns must be invariant to small translations.

Additional common tricks[16].

- **Local Response Normalization (LRN)**[17]. 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

$$b^i_{x,y} = \frac{a^i_{x,y}}{\left(k + \alpha \sum_j \left(a^j_{x,y}\right)^2\right)^\beta} \tag{81}$$

  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[18]. Authors used $k = 2$, $n = 5$, $\alpha = 10^{-4}$, $\beta = 0.75$.
- **Batch Normalization**[19]. 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

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

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

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

[18] 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.

[19] 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.

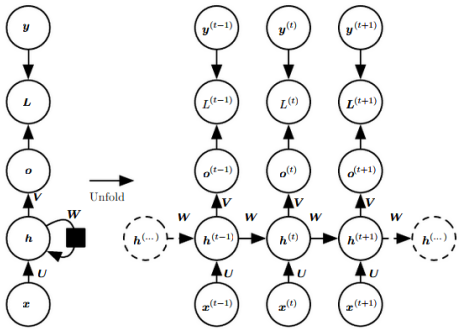
- **Q**: How to intuitively understand the commutativity of convolution?
  - **A**: You must first realize that, *independent of which formula we're thinking of*, as one index into either the image or kernel increases (decreases), the other decreases (increases); they increment in opposite directions. 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[20]. 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.

[20] 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.

$$
\begin{aligned}
\left(\nabla_{\boldsymbol{o}^{(t)}} L\right)_i &= \frac{\partial L}{\partial \boldsymbol{o}_i^{(t)}} \\
&= \frac{\partial L}{\partial L^{(t)}} \cdot \frac{\partial L^{(t)}}{\partial \boldsymbol{o}_i^{(t)}} \\
&= (1) \cdot \frac{\partial L^{(t)}}{\partial \boldsymbol{o}_i^{(t)}} \\
&= \frac{\partial}{\partial \boldsymbol{o}_i^{(t)}} \left\{ -\log\left(\hat{\boldsymbol{y}}_{y^{(t)}}^{(t)}\right) \right\} \\
&= -\frac{\partial}{\partial \boldsymbol{o}_i^{(t)}} \left\{ \log\left(\frac{e^{\boldsymbol{o}_{y^{(t)}}^{(t)}}}{\sum_j e^{\boldsymbol{o}_j^{(t)}}} \right) \right\} \\
&= -\frac{\partial}{\partial \boldsymbol{o}_i^{(t)}} \left\{ \boldsymbol{o}_{y^{(t)}}^{(t)} - \log\left(\sum_j e^{\boldsymbol{o}_j^{(t)}} \right) \right\} \\
&= -\left\{ \mathbf{1}_{i,y^{(t)}} - \frac{\partial}{\partial \boldsymbol{o}_i^{(t)}} \log\left(\sum_j e^{\boldsymbol{o}_j^{(t)}} \right) \right\}
\end{aligned}
\tag{82}
$$

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

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

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

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

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

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

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

Before proceeding, **notice the following useful pattern**: If two nodes a and b, each containing n_a and n_b neurons, are fully connected by parameter matrix $W_{n_b \times n_a}$ and directed like $a \to b \to L$, then[21] $\nabla_a L = W^T \nabla_b L$. Using this result, we can then iterate and take gradients back in time from $t = \tau - 1$ to $t = 1$ as follows:

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

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

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

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

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

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

[21] 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[22].

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

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

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

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

$$\nabla_{\mathbf{V}} L = \sum_t^{\tau} \nabla_{\mathbf{V}} L^{(t)} \tag{83a}$$

$$= \sum_t^{\tau} \nabla_{\mathbf{V}} L^{(t)}(\boldsymbol{o}_1^{(t)}, \ldots, \boldsymbol{o}_{n_{out}}^{(t)}) \tag{83b}$$

$$= \sum_t^{\tau} \sum_i^{n_{out}} \left(\nabla_{\boldsymbol{o}^{(t)}} L\right)_i \nabla_{\mathbf{V}} \boldsymbol{o}_i^{(t)} \tag{83c}$$

$$= \sum_t^{\tau} \sum_i^{n_{out}} \left(\nabla_{\boldsymbol{o}^{(t)}} L\right)_i \nabla_{\mathbf{V}} \left\{ c_i + \sum_{j=1}^{n_{hid}} V_{ij} \boldsymbol{h}_j^{(t)} \right\} \tag{83d}$$

$$= \sum_t^{\tau} \sum_i^{n_{out}} \left(\nabla_{\boldsymbol{o}^{(t)}} L\right)_i \begin{bmatrix} 0 & 0 & \ldots & 0 \\ \vdots & \vdots & \ldots & \vdots \\ \boldsymbol{h}_1^{(t)} & \boldsymbol{h}_2^{(t)} & \ldots & \boldsymbol{h}_{n_{hid}}^{(t)} \\ \vdots & \vdots & \ldots & \vdots \\ 0 & 0 & \ldots & 0 \end{bmatrix} \tag{83e}$$

$$= \sum_t^{\tau} \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) (\boldsymbol{h}^{(t)})^T \tag{83f}$$

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

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

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

where if 83e confuses you, see the footnote[23].

- **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 $\boldsymbol{V}$[24]. 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.

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

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

$$= \sum_t^\tau \sum_i^{n_{hid}} \left(\nabla_{\boldsymbol{h}^{(t)}} L\right)_i \left(\operatorname{diag}\left(1 - (\boldsymbol{h}^{(t)})^2\right) \begin{bmatrix} 0 & 0 & \dots & 0 \\ \vdots & \vdots & \dots & \vdots \\ \boldsymbol{h}_1^{(t-1)} & \boldsymbol{h}_2^{(t-1)} & \dots & \boldsymbol{h}_{n_{hid}}^{(t-1)} \\ \vdots & \vdots & \dots & \vdots \\ 0 & 0 & \dots & 0 \end{bmatrix} \right) \tag{85b}$$

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

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

$$\nabla_{\mathbf{U}} L = \sum_t^\tau \nabla_{\mathbf{U}} L^{(t)} \tag{86a}$$

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

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

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

$$\sum_i \nabla_{\mathbf{W}} \left[(\boldsymbol{W}x)_i\right] = \begin{bmatrix} \boldsymbol{x}^T \\ \boldsymbol{x}^T \\ \vdots \\ \boldsymbol{x}^T \end{bmatrix} \tag{84}$$

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

[24] 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

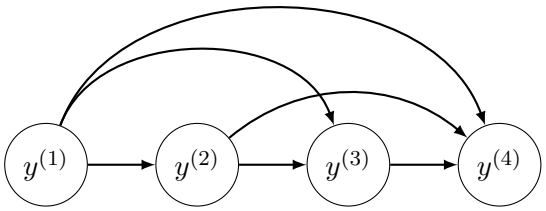
The advantage of RNNs is their efficient parameterization of the joint distribution over $\boldsymbol{y}^{(i)}$ via parameter sharing. This introduces a built-in assumption that we can model the effect of $y^{(i)}$ in the distant past on the current $y^{(t)}$ *via its effect on* $\boldsymbol{h}$. We 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:

$$P(\mathbb{Y}) \triangleq P(y^{(1)}, \ldots, y^{(\tau)}) = \prod_{t=1}^{\tau} P(y^{(t)} \mid y^{(t-1)}, \ldots, y^{(1)}) \tag{87}$$

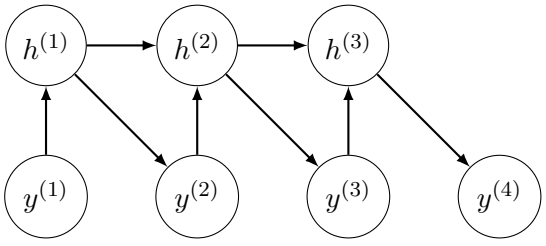
where I've drawn an example of the *complete graph* for $\tau = 4$ below.

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



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

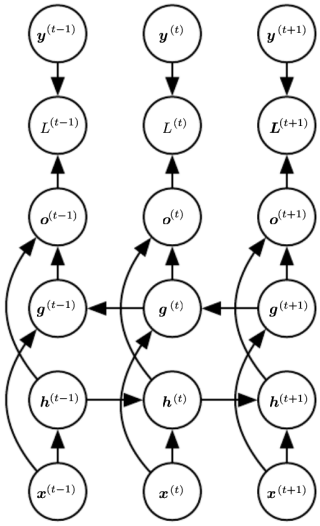
We could include latent variables h at each timestep that capture the dependencies, reminiscent of a classic RNN:



Since in the RNN case all factors $P(h^{(t)} \mid h^{(t-1)})$ are deterministic, we don't need any additional parameters to compute this probability[25], other than the single m^2 parameters needed to convert any $h^{(t)}$ to the next $h^{(t+1)}$ (which is shared across all transitions). Now, the number of parameters needed as a function of sequence length is constant, and as a function of k is just $\mathcal{O}(k)$.

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

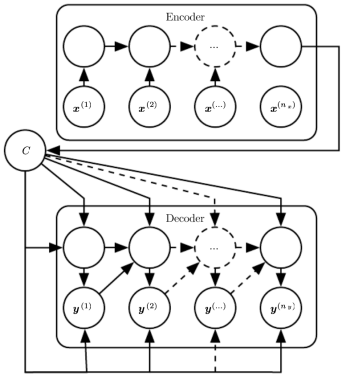
Bidirectional RNNs. In many applications, it is desirable to output a prediction of $\boldsymbol{y}^{(t)}$ that may depend on *the whole sequence*. For example, in speech recognition, the interpretation of words/sentences can also depend on what is *about* to be said. Below is a typical bidirectional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



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

[25] 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.

$$\boldsymbol{h}^{(t)} = \boldsymbol{W}^T \boldsymbol{h}^{(t-1)} \tag{10.36}$$
$$= (\boldsymbol{W}^t)^T \boldsymbol{h}^{(0)} \tag{10.37}$$
$$= \boldsymbol{Q}^T \boldsymbol{\Lambda}^t \boldsymbol{Q} \boldsymbol{h}^{(0)} \tag{10.38}$$

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

Any component of $\boldsymbol{h}^{(0)}$ that isn't aligned with the largest eigenvector will eventually be discarded.[26]

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

[26] 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 $\mathcal{O}(v^n)$[27]. *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[28]

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

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

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

The final hidden state can then be computed via

$$h_i^{(t)} = \tanh(s_i^{(t)}) \odot q_i^{(t)} \tag{90}$$

[27] Quick sketch of (my) proof:

$$\text{Var}\left[w^{(i)}\right] = v = \mathbb{E}\left[(w^{(i)})^2\right] - \cancel{\mathbb{E}\left[w^{(i)}\right]^2} \tag{88}$$

$$\text{Var}\left[\prod_t^n w^{(t)}\right] = \mathbb{E}\left[\left(\prod_t^n w^{(t)}\right)^2\right] = \prod_t^n \mathbb{E}\left[(w^{(t)})^2\right] = v^n \tag{89}$$

[28] 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.

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

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

In addition, n-gram models are vulnerable to the curse of dimensionality, since most n-grams won't occur in the training set[29], 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[30] 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*.

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

[30] 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*[31] 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

$$p_{model}(x) = \sum_h \Pr(h)\Pr(x \mid h) = \mathbb{E}_h\left[p_{model}(x \mid h)\right] \tag{91}$$

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[32]. 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:

$$\boldsymbol{x} = \boldsymbol{W}\boldsymbol{h} + \boldsymbol{b} + \text{noise} \tag{92}$$

Probabilistic PCA and Factor Analysis.

- **Factor analysis**:

$$\boldsymbol{h} \sim \mathcal{N}(\boldsymbol{h}; \boldsymbol{0}, \boldsymbol{I}) \tag{93}$$

$$\text{noise} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{\psi} \equiv \text{diag}(\boldsymbol{\sigma^2})) \tag{94}$$

$$\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{x};\ \boldsymbol{b}, \boldsymbol{W}\boldsymbol{W}^T + \boldsymbol{\psi}) \tag{95}$$

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

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

[32] 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 $\Pr(x_i|\boldsymbol{h}, x_{j\neq i}) = \Pr(x_i|\boldsymbol{h})$[33].

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.

$$\begin{aligned}
\boldsymbol{\mu}_x &= \mathbb{E}_{\boldsymbol{h}}\left[\boldsymbol{W}\boldsymbol{h} + \boldsymbol{b}\right] && (96)\\
&= \int p(\boldsymbol{h})(\boldsymbol{W}\boldsymbol{h} + \boldsymbol{b})\mathrm{d}\boldsymbol{h} && (97)\\
&= \boldsymbol{b} + \int \frac{1}{(2\pi)^{n/2}} e^{-\frac{1}{2}\sum_i^n h_i^2}\boldsymbol{W}\boldsymbol{h}\mathrm{d}\boldsymbol{h} && (98)\\
&= \boldsymbol{b} && (99)\\
\text{Cov}(\boldsymbol{x}) &= \mathbb{E}\left[(\boldsymbol{x} - \mathbb{E}\left[\boldsymbol{x}\right])(\boldsymbol{x} - \mathbb{E}\left[\boldsymbol{x}\right])^T\right] && (100)\\
&= \mathbb{E}\left[(\boldsymbol{W}\boldsymbol{h} + \text{noise})(\boldsymbol{h}^T\boldsymbol{W}^T + \text{noise}^T)\right] && (101)\\
&= \mathbb{E}\left[(\boldsymbol{W}\boldsymbol{h}\boldsymbol{h}^T\boldsymbol{W}^T)\right] + \boldsymbol{\psi} && (102)\\
&= \boldsymbol{W}\boldsymbol{W}^T + \boldsymbol{\psi} && (103)
\end{aligned}$$

  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.

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

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

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

Autoencoders (Ch. 14)

Written by Brandon McKinzie

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

$\boldsymbol{r}$ for "reconstruction"

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

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

Greedy Layer-Wise Unsupervised Pretraining. Given:

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

The procedure is as follows:

1. Initialize.

$$f(\cdot) \leftarrow I(\cdot) \tag{104}$$
$$\tilde{\boldsymbol{X}} = \boldsymbol{X} \tag{105}$$

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

$$f^{(k)} = \mathcal{L}(\tilde{\boldsymbol{X}}) \tag{106}$$
$$f(\cdot) \leftarrow f^{(k)}\left(f\left(\cdot\right)\right) \tag{107}$$
$$\tilde{\boldsymbol{X}} \leftarrow f^{(k)}(\tilde{\boldsymbol{X}}) \tag{108}$$

   In English: just apply the regular learning/training process for each layer/stage **sequentially and individually**[34].

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).

[34] 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[35]. 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

$$p(\boldsymbol{x}) = \prod_i p(x_i \mid Pa_{\mathcal{G}}(x_i)) \tag{109}$$

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

Clique potentials are constrained to be nonnegative.

$$\tilde{p}(\boldsymbol{x}) = \prod_{\mathcal{C} \in \mathcal{G}} \phi(\mathcal{C}) \tag{110}$$

[35] 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:

$$p(\boldsymbol{x}) = \frac{1}{Z}\widetilde{p}(\boldsymbol{x}) \tag{111}$$

$$Z = \int \widetilde{p}(\boldsymbol{x})\mathrm{d}\boldsymbol{x} \tag{112}$$

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

$$\widetilde{p}(\boldsymbol{x}) \triangleq \exp\left(-E(\boldsymbol{x})\right) \tag{113}$$

and $E(\boldsymbol{x})$ is known as the **energy function**[36]. 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**:

$$\mathcal{F}(\boldsymbol{x} = x) = -\log \sum_h \exp\left(-E(\boldsymbol{x} = x,\ \boldsymbol{h} = h)\right) \tag{114}$$

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**[37]. 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.

[36] 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.

[37] 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

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\boldsymbol{b}^T\boldsymbol{v} - \boldsymbol{c}^T\boldsymbol{h} - \boldsymbol{v}^T\boldsymbol{W}\boldsymbol{h} \tag{115}$$

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[38].

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

[38] 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:

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

$$p(\boldsymbol{v} \mid \boldsymbol{h}) = \prod_i p(v_i \mid \boldsymbol{h}) \tag{117}$$

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

$$p(h_i = 1 \mid \boldsymbol{v}) = \sigma\left(c_i + \boldsymbol{v}^T \boldsymbol{W}_{:,i}\right) \tag{118}$$

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:

$$s = \int p(\boldsymbol{x}) f(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} = \mathbb{E}_p \left[f(\mathbf{x}) \right] \quad \rightarrow \quad \hat{s}_n = \frac{1}{n} \sum_{i=1,\ \mathbf{x}^{(i)} \sim p}^{n} f(\boldsymbol{x}^{(i)}) \tag{119}$$

As we've seen before, the empirical average is an unbiased[39] 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:

$$p(\boldsymbol{x}) f(\boldsymbol{x}) = q(\boldsymbol{x}) \frac{p(\boldsymbol{x}) f(\boldsymbol{x})}{q(\boldsymbol{x})} \tag{123}$$

- **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 $\hat{s}_q$:

$$\hat{s}_q = \frac{1}{n} \sum_{i=1,\ \mathbf{x}^{(i)} \sim q}^{n} \frac{p(\boldsymbol{x}^{(i)}) f(\boldsymbol{x}^{(i)})}{q(\boldsymbol{x}^{(i)})} \tag{124}$$

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

$$\mathbb{E}_p \left[\hat{s}_n \right] = \frac{1}{n} \sum_{i=1}^{n} \mathbb{E}_p \left[f(\boldsymbol{x}^{(i)}) \right] \tag{120}$$

$$= \frac{1}{n} \sum_{i=1}^{n} s \tag{121}$$

$$= s \tag{122}$$

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:

$$\mathrm{Var}\left[\hat{s}_q\right] = \mathrm{Var}\left[\frac{p(\boldsymbol{x})f(\boldsymbol{x})}{q(\boldsymbol{x})}\right]/n \tag{125}$$

with the optimal (minimum) value of q at:

$$q* = \frac{p(\boldsymbol{x}) \mid f(\boldsymbol{x}) \mid}{Z} \tag{126}$$

- **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:

$$\hat{s}_{BIS} = \frac{\sum_{i=1}^{n} \frac{\tilde{p}(\boldsymbol{x}^{(i)})}{\tilde{q}(\boldsymbol{x}^{(i)})} f(\boldsymbol{x}^{(i)})}{\sum_{i=1}^{n} \frac{\tilde{p}(\boldsymbol{x}^{(i)})}{\tilde{q}(\boldsymbol{x}^{(i)})}} \tag{127}$$

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

$$\log p_{model}(\boldsymbol{x}) = \log \tilde{p}_{model}(\boldsymbol{x}; \boldsymbol{\theta}) + c \tag{128}$$

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

$$p_{joint}(y{=}1) = \frac{1}{2} \tag{129}$$

$$p_{joint}(\boldsymbol{x} \mid y{=}1) = p_{model}(\boldsymbol{x}) \tag{130}$$

$$p_{joint}(\boldsymbol{x} \mid y{=}0) = p_{noise}(\boldsymbol{x}) \tag{131}$$

We can now use MLE on the optimization problem,

$$\boldsymbol{\theta}, c = \arg\max_{\boldsymbol{\theta}, c} \mathbb{E}_{\boldsymbol{x}, y \sim p_{train}} \left[\log p_{joint}(y \mid \boldsymbol{x})\right] \tag{132}$$

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

$$= \frac{1}{1 + p_{noise}(\boldsymbol{x}) / p_{model}(\boldsymbol{x})} \tag{134}$$

$$= \sigma\left(\log p_{model}\left(\boldsymbol{x}\right) - \log p_{noise}\left(\boldsymbol{x}\right)\right) \tag{135}$$

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 $\log p(\boldsymbol{v};\boldsymbol{\theta})$[40]:

$$\mathcal{L}(\boldsymbol{v},\boldsymbol{\theta},q) = \log p(\boldsymbol{v};\boldsymbol{\theta}) - D_{KL}\left(q(\boldsymbol{h}\mid\boldsymbol{v}) \;||\; p(\boldsymbol{h}\mid\boldsymbol{v};\boldsymbol{\theta})\right) \tag{136}$$

$$= -\mathbb{E}_{\boldsymbol{h}\sim q(\boldsymbol{h}|\boldsymbol{v})}\left[\log p(\boldsymbol{h},\boldsymbol{v})\right] + H(q(\boldsymbol{h}\mid\boldsymbol{v})) \tag{137}$$

q is an arbitrary probability distribution over $\boldsymbol{h}$. Note that the book will write q when they really mean $q(\boldsymbol{h}\mid\boldsymbol{v})$.

where the second form is the more canonical definition[41]. 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 $p(\boldsymbol{h}\mid\boldsymbol{v})$.

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

[41] 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

$$q(\boldsymbol{h} \mid \boldsymbol{v}^{(i)}) = p(\boldsymbol{h} \mid \boldsymbol{v}^{(i)}; \boldsymbol{\theta}^{(0)}) \tag{138}$$

   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

$$\arg\max_{\boldsymbol{\theta}} \sum_i \mathcal{L}\left(\boldsymbol{v}^{(i)}, \boldsymbol{\theta}, q(\boldsymbol{h} \mid \boldsymbol{v}^{(i)}; \boldsymbol{\theta}^{(0)})\right) \tag{139}$$

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[42]. 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

$$E(\boldsymbol{x}) = -\sum_{i=1}^{d}\sum_{j=i+1}^{d} x_i U_{i,j} x_j - \sum_{i=1}^{d} b_i x_i \tag{140}$$

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

$$\Delta E_i = \sum_{k=i+1}^{d} U_{i,k} x_k + \sum_{k=1}^{i-1} x_k U_{k,i} + b_i \tag{141}$$

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

$$\Delta E_i = \ln p_{i=on}(\boldsymbol{x}) - \ln p_{i=off}(\boldsymbol{x}) = -\ln\left(\frac{1 - p_{i=on}(\boldsymbol{x})}{p_{i=on}(\boldsymbol{x})}\right) \tag{142}$$

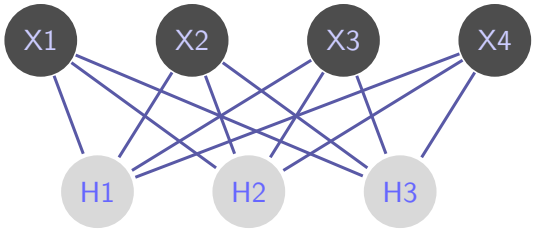
$$\exp(-\Delta E_i) = \frac{1 - p_{i=on}(\boldsymbol{x})}{p_{i=on}(\boldsymbol{x})} = \frac{1}{p_{i=on}(\boldsymbol{x})} - 1 \tag{143}$$

$$\therefore p_{i=on}(\boldsymbol{x}) = \frac{1}{1 + \exp(-\Delta E_i)} \tag{144}$$

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.

[42] 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.



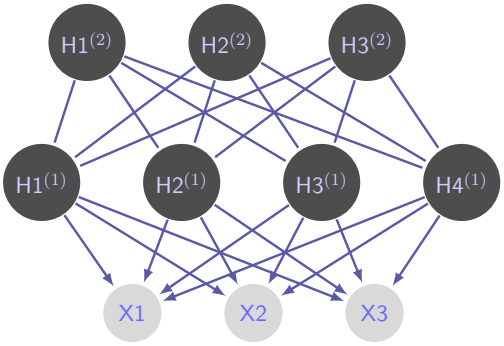
Although the joint distribution $p(\boldsymbol{x}, \boldsymbol{h})$ has a potentially intractable partition function, the conditional distributions can be computed efficiently by exploiting independencies:

$$p(\boldsymbol{h} \mid \boldsymbol{x}) = \prod_{j=1}^{n_h} \sigma\left([2\boldsymbol{h} - 1] \odot [\boldsymbol{c} + \boldsymbol{W}^T \boldsymbol{x}]\right)_j \tag{145}$$

$$p(\boldsymbol{x} \mid \boldsymbol{h}) = \prod_{i=1}^{n_x} \sigma\left([2\boldsymbol{x} - 1] \odot [\boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}]\right)_i \tag{146}$$

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[43].

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:

Case a: interpret $g(\boldsymbol{z})$ as emitting $\boldsymbol{x}$ directly.

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

where $\boldsymbol{L}$ is the *Cholesky decomposition*[44] 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,

$$p_x(\boldsymbol{x}) = \frac{p_z(g^{-1}(\boldsymbol{x}))}{\left|\det \frac{\partial g}{\partial \boldsymbol{z}}\right|} \tag{147}$$

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

Case b: interpret $g(\boldsymbol{z})$ as emitting $p(\boldsymbol{x} \mid \boldsymbol{z})$.

$$p(x_i \mid \boldsymbol{z}) = g(\boldsymbol{z})_i \qquad\qquad p(\boldsymbol{x}) = \mathbb{E}_{\boldsymbol{z}}\left[p(\boldsymbol{x} \mid \boldsymbol{z})\right] \tag{148}$$

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.

[43] 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).

[44] 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 $p_{model}(\boldsymbol{x}; g(\boldsymbol{z})) = p_{model}(\boldsymbol{x} \mid \boldsymbol{z})$.