Contents

Math and Machine Learning

Basics

Contents

Linear Algebra (Quick Review): January 23

Written by Brandon McKinzie

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

where m is the number of data points.

Measure Theory.

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

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

Functions of RVs.

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

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

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

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

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

Gradient-Based Optimization (4.3)

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

.

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

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

The Hessian is the Jacobian of the gradient.

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

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

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

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

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

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

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

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

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

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

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

Capacity, Overfitting, and Underfitting (5.2)

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

Estimators, Bias and Variance (5.4)

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

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

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

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

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

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

Bias-Variance Tradeoff.

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

→ **Summary of main formulas**.

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

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

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

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

and desirable estimators are those with low MSE.

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

[6] Derivation:

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

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

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

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

Bayesian Statistics (5.6)

Distinction between frequentist and bayesian approach:

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

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

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

Maximum Likelihood Estimation (5.5)

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

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

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

[7] The KL divergence is given by

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

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

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

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

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

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

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

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

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

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

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

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

Supervised Learning Algorithms (5.7)

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

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

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

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

Equation 5.81 is the definition of logistic regression

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

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

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

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

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

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

Deep Networks: Modern Practices

Contents

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

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

with ϕ defining a hidden layer.

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

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

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

a.e. is short for "almost everywhere"

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

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

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

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

Back-Propagation (6.5)

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

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

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

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

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

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

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

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

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

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

Regularization for Deep Learning: January 12

Written by Brandon McKinzie

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

Parameter Norm Penalties (7.1)

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

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

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

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

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

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

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

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

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

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

Sparse Representations (7.10)

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

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

Adversarial Training (7.13)

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

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

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

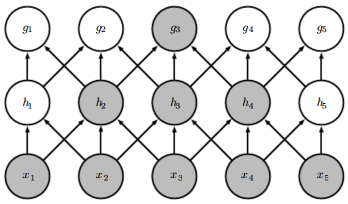
In the context of regularization, one can reduce the error rate on the original i.i.d. test set via **adversarial training** – training on adversarially perturbed training examples.

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

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

Convolution leverages the following three important ideas:

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



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

[10]Technically the convolution output is defined as

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

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

where 9.5 can be asserted due to commutativity of convolution.

Modern Practices **Fall 2016**

Sequence Modeling (RNNs): January 15

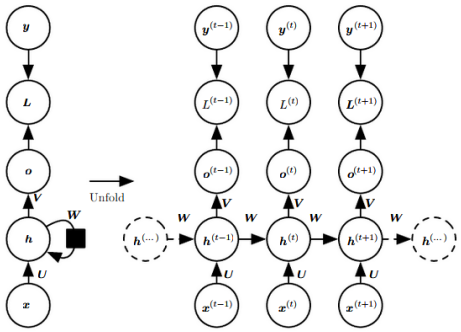
Table of Contents Local *Written by Brandon McKinzie*

Review: The Basics of RNNs

Notation/Architecture Used.

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

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



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

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

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

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

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

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

steps as:

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

Convince yourself this is identical to cross-entropy.

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

Back-Propagation Through Time.

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

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

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

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

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

which leaves all entries of $\boldsymbol{o}^{(t)}$ unchanged *except* for the entry corresponding to the true label, which will become negative in the gradient. All this means is, since we want to increase the probability of this entry, driving this value up will *decrease* the loss (hence negative) and driving any other entries up will *increase* the loss proportional to its current estimated probability (driving up an [incorrect] entry that is already high is "worse" than driving up a small [incorrect entry]).

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

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

Before proceeding, **notice the following useful pattern**: If two nodes a and b, each containing n_a and n_b neurons, are fully connected by parameter matrix $W_{n_b \times n_a}$ and directed like $a \to b \to L$, then[12] $\nabla_a L = W^T \nabla_b L$. Using

[12] More generally,

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

this result, we can then iterate and take gradients back in time from $t = \tau - 1$ to $t = 1$ as follows:

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

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

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

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

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

2. **Parameter Gradients**. Now we can compute the gradients for the parameter matrices/vectors, where it is crucial to remember that a given parameter matrix (e.g. U) is shared across *all* time steps t. We can treat tensor derivatives in the same form as previously done with vectors after a quick abstraction: For any tensor $\mathbf{X}$ of arbitrary rank (e.g. if rank-4 then index like $\mathbf{X}_{ijk\ell}$), use single variable (e.g. i) to represent the complete tuple of indices[13].

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

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

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

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

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

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

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

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

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

$$= \sum_{t}^{\tau} \sum_{i}^{n_{out}} \left(\nabla_{\boldsymbol{o}^{(t)}} L\right)_i \nabla_{\mathbf{V}} \boldsymbol{o}_i^{(t)} \tag{23b}$$

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

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

Note how the summation inside the curly braces only has entries along the ith row of V! It's crucial to understanding why we can go from 23c to 23d.

Deep Learning Research

Contents

Linear Factor Models: January 12

Written by Brandon McKinzie

Overview. Much research is in building a *probabilistic model* 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. Some also have **latent variables**, h, with

$$p_{model}(x) = \sum_h \Pr(h) \Pr(x \mid h) = \mathbb{E}_h \left[p_{model}(x \mid h) \right] \tag{24}$$

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

Condensed Summaries

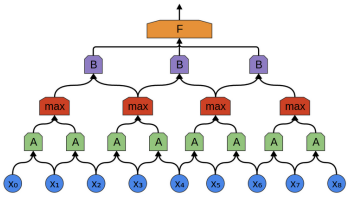
Contents

Conv Nets: A Modular Perspective: December 21

Written by Brandon McKinzie

From this post on Colah's Blog.

The title is inspired by the following figure. Colah mentions how groups of neurons, like A, that appear in multiple places are sometimes called **modules**, and networks that use them are sometimes called modular neural networks. You can feed the output of one convolutional layer into another. With each layer, the network can detect higher-level, more abstract features.



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

$\longrightarrow$ Max pooling layers: kind of "zoom out". They allow later convolutional layers to work on larger sections of the data. They also make us invariant to some very small transformations of the data.

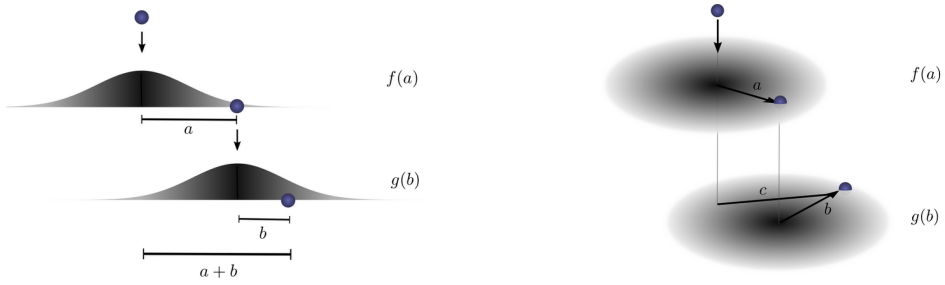
Understanding Convolutions: December 21

 Written by Brandon McKinzie

From Colah's Blog.

Ball-Dropping Example. The posed problem:

> Imagine we drop a ball from some height onto the ground, where it only has one dimension of motion. How likely is it that a ball will go a distance c if you drop it and then drop it again from above the point at which it landed?



From basic probability, we know the result is a sum over possible outcomes, constrained by $a + b = c$. It turns out this is actually the definition of the convolution of f and g.

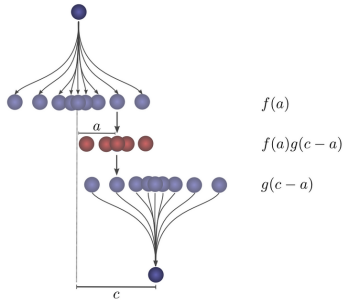
$$\Pr(\text{a} + \text{b} = \text{c}) = \sum_{a+b=c} f(a) \cdot g(b) \tag{25}$$

$$(f * g)(c) = \sum_{a+b=c} f(a) \cdot g(b) \tag{26}$$

$$= \sum_{a} f(a) \cdot g(c - a) \tag{27}$$

Visualizing Convolutions. Keeping the same example in the back of our heads, consider a few interesting facts.

- **Flipping directions**. If $f(x)$ yields the probability of landing a distance x away from where it was dropped, what about the probability that it was dropped a distance x from where it *landed*? Apparently[14] it is $f(-x)$.



- Above is a visualization of one term in the summation of $(f * g)(c)$. It is meant to show how we can move the bottom around to think about evaluating the convolution for different c values.

We can relate these ideas to image recognition. Below are two common kernels used to convolve images with.

0	0	0	0	0
0	1/9	1/9	1/9	0
0	1/9	1/9	1/9	0
0	1/9	1/9	1/9	0
0	0	0	0	0

0	0	0	0	0
0	0	0	0	0
0	-1	1	0	0
0	0	0	0	0
0	0	0	0	0

On the left is a kernel for *blurring* images, accomplished by taking simple averages. On the right is a kernel for *edge detection*, accomplished by taking the difference between two pixels, which will be largest at edges, and essentially zero for similar pixels.

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

Deep Reinforcement Learning: December 23

Written by Brandon McKinzie

Link to tutorial – Part I of "Demystifying deep reinforcement learning."

Reinforcement Learning. Vulnerable to the *credit assignment problem* - i.e. unsure which of the preceding actions was responsible for getting some reward and to what extent. Also need to address the famous *explore-exploit dilemma* when deciding what strategies to use.

Markov Decision Process. Most common method for representing a reinforcement problem. MDPs consist of states, actions, and rewards. Total reward is sum of current (includes previous) and *discounted* future rewards:

$$R_t = r_t \gamma(r_{t+1} + \gamma(r_{t+2} + \ldots)) = r_t + \gamma R_{t+1} \tag{28}$$

Q - learning. Define function $Q(s, a)$ to be best possible score at end of game after performing action a in state s; the "quality" of an action from a given state. The recursive definition of Q (for one transition) is given below in the *Bellman equation*.

$$Q(s, a) = r + \gamma \max_{a'} Q(s', a')$$

and updates are computed with a learning rate α as

$$Q(s_t, a_t) = (1 - \alpha) \cdot Q(s_{t-1}, a_{t-1}) + \alpha \cdot (r + \gamma \max_{a'} Q(s'_{t+1}, a'_{t+1}))$$

Deep Q Network. Deep learning can take deal with issues related to prohibitively large state spaces. The implementation chosen by DeepMind was to represent the Q-function with a neural network, with the states (pixels) as the input and Q-values as output, where the number of output neurons is the number of possible actions from the input state. We can optimize with simple squared loss:

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

and our algorithm from some state s becomes

1. **First forward pass** from s to get all predicted Q-values for each possible action. Choose action corresponding to max output, leading to next s'.
2. **Second forward pass** from s' and again compute $\max_{a'} Q(s', a')$.
3. **Set target output** for each action a' from s'. For the action corresponding to max (from step 2) set its target as $r + \gamma \max_{a'} Q(s', a')$, and for all other actions set target to same as originally returned from step 1, making the error 0 for those outputs. (Interpret as update to our guess for the best Q-value, and keep the others the same.)
4. **Update weights** using backprop.

Experience Replay. This the most important trick for helping convergence of Q-values when approximating with non-linear functions. During gameplay all the experience $< s, a, r, s' >$ are stored in a replay memory. When training the network, random minibatches from the replay memory are used instead of the most recent transition.

Exploration. One could say that initializing the Q-values randomly and then picking the max is essentially a form of exploitation. However, this type of exploration is *greedy*, which can be tamed/fixed with **ε-greedy exploration**. This incorporates a degree of randomness when choosing next action at *all* time-steps, determined by probability ε that we choose the next action randomly. For example, DeepMind decreases ε over time from 1 to 0.1.

Deep Q-Learning Algorithm.

```
initialize replay memory D
initialize action-value function Q with random weights
observe initial state s
repeat
      select an action a
            with probability ε select a random action
            otherwise select a = argmax_a' Q(s,a')
      carry out action a
      observe reward r and new state s'
      store experience <s, a, r, s'> in replay memory D

      sample random transitions <ss, aa, rr, ss'> from replay memory D
      calculate target for each minibatch transition
            if ss' is terminal state then tt = rr
            otherwise tt = rr + γmax_a' Q(ss', aa')
      train the Q network using (tt - Q(ss, aa))^2 as loss

      s = s'
until terminated
```

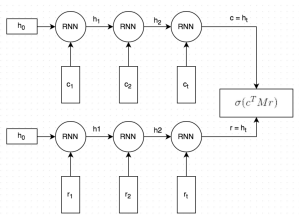
Deep Learning for Chatbots (WildML): January 15

Written by Brandon McKinzie

Overview.

- **Model**. Implementing a retrieval-based model. Input: conversation/context c. Output: response r.
- **Data**. Ubuntu Dialog Corpus (UDC). 1 million examples of form (context, utterance, label). The label can be 1 (utterance was actual response to the context) or a 0 (utterance chosen randomly). Using NLTK, the data has been . . .
  → **Tokenized**: dividing strings into lists of substrings.
  → **Stemmed**. **IDK**
  → **Lemmatized**. **IDK**
  The test/validation set consists (context, ground-truth utterance, [9 distractors (incorrect utterances)]). The distractors are picked at random[15]

Dual-Encoder LSTM.



1. **Inputs**. Both the context and the response text are split by words, and each word is embedded into a vector and fed into the same RNN.

2. **Prediction**. Multiply the [vector representation ("meaning")] c with param matrix M to predict some response r'.

3. **Evaluation**. Measure similarity of predicted r' to actual r via simple dot product. Feed this into sigmoid to obtain a probability [of r' being the correct response]. Use (binary) cross-entropy for loss function:

$$L = -y \cdot \ln(y') - (1 - y) \cdot \ln(1 - y') \tag{29}$$

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

where y' is the predicted probability that r' is correct response r, and $y \in \{0, 1\}$ is the true label for the context-response pair (c, r).

Data Pre-Processing. Courtesy of WildML, we are given 3 files after preprocessing: train.tfrecords, validation.tfrecords, and test.tfrecords, which use TensorFlow's 'Example' format. Each Example consists of . . .

- context: Sequence of word ids.
- context_len: length of the aforementioned sequence.
- utterance: seq of word ids representing utterance (response).
- utterance_len.
- label: only in training data. 0 or 1.
- distractor_[N]: Only in test/validation. N ranges from 0 to 8. Seq of word ids reppin the distractor utterance.
- distractor_[N]_len.

WaveNet (Paper): January 15

Table of Contents Local *Written by Brandon McKinzie*

Abstract.

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

Introduction.

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

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

$$p(x) = \prod_{t=1}^{T} p(x_t \mid x_1, \ldots, x_{t-1}) \tag{30}$$

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

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

Main ingredient of WaveNet is *dilated* causal convolutions, illustrated below. Note the absence of recurrent connections, which makes them faster to train than RNNs, but at the cost of requiring many layers, or large filters to increase the receptive field[16].

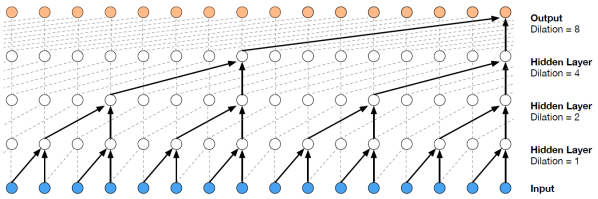


Figure 3: Visualization of a stack of *dilated* causal convolutional layers.

Excellent concise definition from paper:

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

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

$$f(x_t) = \text{sign}(x_t)\frac{\ln(1+\mu|x_t|)}{\ln(1+\mu)} \tag{31}$$

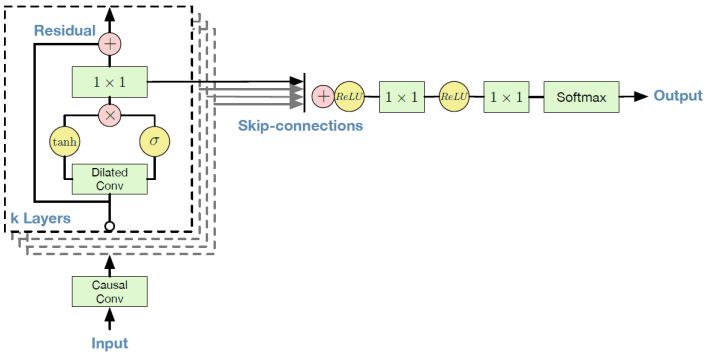
which (after plotting in Wolfram) looks identical to the sigmoid function.

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

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

$$z = \tanh\left(W_{f,k} * x\right) \odot \sigma\left(W_{g,k} * x\right) \tag{32}$$

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



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

$$p(x \mid h) = \prod_{t=1}^{T} p(x_t \mid x_1, \ldots, x_{t-1}, h) \tag{33}$$

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

$$z = \tanh\left(W_{f,k} * x + V_{f,k}^T h\right) \odot \sigma\left(W_{g,k} * x + V_{g,k}^T h\right) \tag{34}$$

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

Experiments.

- **Multi-Speaker Speech Generation**. Dataset: multi-speaker corpus of 44 hours of data from 109 different speakers[17]. Receptive field of 300 milliseconds.
- **Text-to-Speech**. Single-speaker datasets of 24.6 hours (English) and 34.8 hours (Chinese) speech. Locally conditioned on *linguistic features*. Receptive field of 240 milliseconds. Outperformed both LSTM-RNN and HMM.
- **Music**. Trained the WaveNets to model two music datasets: (1) 200 hours of annotated music audio, and (2) 60 hours of solo piano music from youtube. Larger receptive fields sounded more musical.
- **Speech Recognition**. "With WaveNets we have shown that layers of dilated convolutions allow the receptive field to grow longer in a much cheaper way than using LSTM units."

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

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

Neural Style: January 22

 Written by Brandon McKinzie

Notation.

- **Content image**: $\boldsymbol{p}$
- **Filter responses**: the matrix $P^l \in \mathcal{R}^{N_l \times M_l}$ contains the activations of the filters in layer l, where P^l_{ij} would give the activation of the ith filter at position j in layer l. N_l is number of feature maps, each of size M_l (height $\times$ width of the feature map)[18].
- **Reconstructed image**: $\boldsymbol{x}$ (initially random noise). Denote its corresponding filter response matrix at layer l as P^l.

Content Reconstruction.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

APPENDIX A - Questions and Stuff I Always Forget

Questions:

- **Q:** What's the deal with **mixture models**? Why use them? [*Context: Wavenet paper]*
- **Q:** In general, how can one tell if a matrix $\boldsymbol{A}$ has an eigenvalue decomposition? [insert more conceptual matrix-related questions here . . .]
- **Q:** Let $\boldsymbol{A}$ be real-symmetric. What can we say about $\boldsymbol{A}$?
  - Proof that eigendecomposition $A = Q\Lambda Q^T$ exists: Wow this is apparently quite hard to prove according to many online sources. Guess I don't feel so bad now that it wasn't (and still isn't) obvious.
  - Eigendecomposition not unique. This is apparently because two or more eigenvectors may have same eigenvalue.

This is the principal axis theorem: if A symmetric, then orthonorm basis of e-vects exists.

Stuff I Forget:

- Existence of eigenvalues/eigenvectors. Let $\boldsymbol{A} \in \mathbb{R}^{n \times n}$.
  - λ is an eigenvalue of A iff it satisfies $\det(\lambda \boldsymbol{I} - \boldsymbol{A}) = 0$. Why? Because it is an equivalent statement as requiring that $(\lambda \boldsymbol{I} - \boldsymbol{A})\boldsymbol{x} = 0$ has a nonzero solution for $\boldsymbol{x}$.
  - The following statements are equivalent:
    - A is diagonalizable.
    - A has n linearly independent eigenvectors.
  - The **eigenspace** of $\boldsymbol{A}$ corresponding to λ is the solution space of the homogeneous system $(\lambda I - A)x = 0$.
  - $\boldsymbol{A}$ has at most n distinct eigenvalues.

Most info here comes from chapter 5 of your "Elementary Linear Algebra" textbook (around pg305)

- Diagonalizability notes from 5.2 of advanced linear alg. book (261). Recall that $\boldsymbol{A}$ is defined to be diagonalizable if and only if there exists and ordered basis β for the space consisting of eigenvectors of $\boldsymbol{A}$.
  - If the standard way of finding eigenvalues leads to k distinct λ_i, then the corresponding set of k eigenvectors v_i are guaranteed to be linearly independent (but might not span the full space).
  - If $\boldsymbol{A}$ has n linearly independent eigenvectors, then $\boldsymbol{A}$ is diagonalizable.
  - The characteristic polynomial of any diagonalizable linear operator splits (can be factored into product of linear factors). The **algebraic multiplicity** of an eigenvalue λ is the largest positive integer k for which $(t - \lambda)^k$ is a factor of $f(t)$.

Recall that a linear operator is a special case of a linear map where the input space is the same as the output space.

- Logistic regression is called a *linear model* because it classifies based on the linear $\boldsymbol{\theta}^T \boldsymbol{x}$ before feeding that to the logistic sigmoid function.

APPENDIX B - Glossary of Confusing Terms

Gaussian mixture model: each of the components/dists $p(x \mid c = i)$ are Gaussians, each with (possibly) different mean μ and covariance Σ. The params of a GM specify the **prior** probability $p(c = i)$, so-called since expresses belief about c *before* it has observed x.

Latent variable: A random variable that we cannot observe directly (DL pg. 65).

No free lunch theorem:

APPENDIX C - Startup Ideas

- Household-chore robots $\longrightarrow$ world-domination robots.
- MagicMirrors-R-Us
- Deep Reinforcement Learning — 2048
- Website that displays np.info output.