CS 337: Artificial Intelligence

Amit Rajaraman

Last updated October 23, 2021

Contents

§0. Notation

Given a vector $w \in \mathbb{R}^n$, w_i denotes the ith component of w.
Given a vector w, $\|w\|_0$ is equal to the number of non-zero components $|\{i : w_i \neq 0\}|$ of w.

§1. Introduction

1.1. What and Why?

Machine learning involves learning from past experiences to perform a job better.
Broadly, the goal of the field is to make a computer emulate human pattern recognition, which is second nature to us. What do each of these words mean in the context of computer science? "Learning" and "better" usually depend heavily on the context surrounding our goals, and past experiences usually refers to a data set of some form. Broadly, there are two types of learning: *supervised* and *unsupervised*.

In the former, we have access to some *training data*, usually consisting of a set of pairs of input and output. The supervised learning algorithm then analyzes this data to produce an inferred function, which can then be used for determining what unknown inputs (not in the training data) must map to. This requires the algorithm to go from a smaller set of training data to a much broader set of inputs in a way that "makes sense". For example, given the heights and weights of a thousand people, we might be asked to determine the weight of a person of a person of some new height.
In unsupervised learning on the other hand, we do not have access to any prior information, so we must classify them in some sensible manner. For example, given a set of fruits, we may wish to classify them into various groups. If we divide on the basis of colour (alone), we might get two groups – apples and cherries in one and oranges and peaches in the other. If we divide on the basis of both colour and size, we might further divide the first of these groups into two. The computer is forced to create some compact internal representation of the features of these fruits, and from this we might even be able to generate *new* fruits.

How does all this work? It might be simpler to grasp if we dive headlong into an example.
Suppose we are given a data set $\mathcal{D} = \{\langle x_1, y_1\rangle, \ldots, \langle x_k, y_k\rangle\}$. We wish to determine a function f^* such that $f^*(x)$ is the best "predictor" of y with respect to $\mathcal{D}$.
To quantify how good a prediction is, we introduce an *error function* $\varepsilon(f, \mathcal{D})$ that reflects the discrepancy of the function with respect to $\mathcal{D}$. We also assume that our f^* is taken from some base class of functions, say $\mathcal{F}$. We then set

$$f^* = \arg\min_{f \in \mathcal{F}} \varepsilon(f, \mathcal{D}).$$

Typically, we present our function in terms of some "basis functions" $(\phi_i)_{i=1}^n$ each from the set of inputs to $\mathbb{R}$. More compactly, we have a single function ϕ from the set of inputs to $\mathbb{R}^n$ that shows all the relevant things we can glean about any input. Suppose our set of outputs is in $\mathbb{R}^k$. We then learn a function $f : \mathbb{R}^n \to \mathbb{R}^k$ and given a new input x', we predict the corresponding output as $y' = f(\phi(x'))$.
Due to this, it is often helpful to think of the input as an element of $\mathbb{R}^n$, where n is the number of basis functions.

Remark. It is important to note that even though our original input x may be in $\mathbb{R}$, we could have more than just 1 basis function. For instance, we could choose the basis functions as $\{(x \mapsto x^i) : 1 \leq i \leq 100\}$. This is especially important in the coming topic of linear regression, where linearity in the basis functions should not be mixed up with linearity in the original input.

1.2. Linear Regression

We first discuss perhaps the simplest example of machine learning. In this, the class $\mathcal{F}$ of functions we consider is merely the class of all linear functions over the basis functions. That is,

$$\mathcal{F} = \{(x \mapsto w^\top \phi(x)) : w \in \mathbb{R}^n\}.$$

Observe that if one of our basis functions is 1 (or some constant), this is equivalent to the set of all *affine* functions (linear functions plus a constant) on the remaining basis functions.

1.2.1. The method of least squares

Suppose our data set is $\mathcal{D} = \{\langle x_i, y_i\rangle : 1 \le i \le m\}$, where each y_i is in $\mathbb{R}$. In the method of least squares, our choice of error function is given by

$$\varepsilon(f, \mathcal{D}) = \sum_{i=1}^{m} (f(x_i) - y_i)^2.$$

This is an extremely common error function for various reasons – it is convex, non-negative, and well-behaved.
In the case of linear regression, where any function f is uniquely determined by a $w \in \mathbb{R}^n$, we can express the error in terms of this vector as

$$\varepsilon(w) = \sum_{i=1}^{m} (w^\top \phi(x_i) - y_i)^2.$$

Now, set

$$\Phi = \begin{pmatrix} \phi_1(x_1) & \cdots & \phi_n(x_1) \\ \vdots & \ddots & \vdots \\ \phi_1(x_m) & \cdots & \phi_n(x_m) \end{pmatrix} \text{ and } y = \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix}.$$

So,

$$\varepsilon(w) = \|\Phi w - y\|_2^2.$$

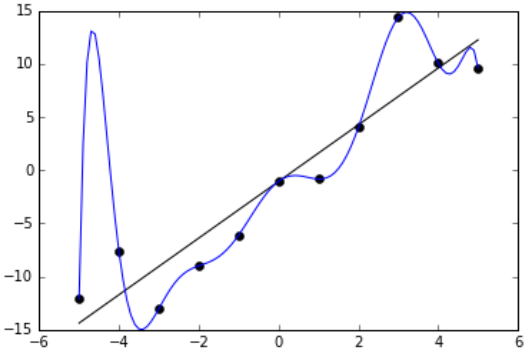
The minimum value of the above is just the distance from y to $\mathcal{C}(\Phi)$, the column space of Φ! In particular, if $y \in \mathcal{C}(\Phi)$, it is possible to get the cost to 0. Let $\hat{w} = \arg\min_w \varepsilon(w)$.
The least square solution is then the distance from y to the projection $\hat{y}$ of y on $\mathcal{C}(\Phi)$. How do we find $\hat{y}$ and $\hat{w}$?
The line joining y and $\hat{y}$ is orthogonal to $\mathcal{C}(\Phi)$. As a result, $(\hat{y} - y)^\top \Phi = 0$. Therefore,

$$\begin{aligned} \hat{y}^\top \Phi &= y^\top \Phi \\ (\Phi\hat{w})^\top \Phi &= y^\top \Phi \\ \hat{w} &= (\Phi^\top \Phi)^{-1} \Phi^\top y. \end{aligned}$$

This is well-defined only if $(\Phi^\top \Phi)$ is invertible (Φ has full column rank). We also have $\hat{y} = \Phi(\Phi^\top \Phi)^{-1}\Phi^\top y$.

Remark (Overfitting). One might think that more basis functions means a better approximation. Indeed, this would mean that we have more "freedom", which should allow us to get a better function. However, this is not the case. While adding more basis functions allows us to emulate the *training* data better, we might mimic it too closely and lose sight of the overall behaviour, which results in bad performance when it comes to the general *testing* data.
This is very clearly seen in the following example where the output is slightly noisy linear data, and we have taken two cases: one wherein the basis functions are just $\phi(x) = (1, x)$ (fitting a degree 1 polynomial), and the second where $\phi(x) = (1, x, \ldots, x^{10})$ (fitting a degree 10 polynomial).

FIGURE 1 – Overfitted data



1.3. Probabilistic Interpretations

Let us move away from regression for a moment and look at what happens if instead of trying to learn a function with inputs and outputs, we instead try to determine the parameters of some random variable given some datapoints drawn from it.
To do this, suppose that any observation z is drawn from the random variable Z with probability density/mass function g_θ (depending on the parameter θ, which we are trying to learn). For example, if Z is a gaussian with mean μ and variance σ^2, we might have $\theta = (\mu, \sigma^2)$ and

$$g_\theta(z) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right).$$

1.3.1. Maximum likelihood estimation

Suppose we are given a dataset $\mathcal{D} = \{z_1, \ldots, z_m\}$.
Also suppose that we have discrete outputs for the moment (g_w is a probability *mass* function). Then, define the *likelihood* function

$$\mathcal{L}(\mathcal{D}; \theta) = \Pr[\mathcal{D} \mid \theta].$$

Making the mild assumption that the various datapoints in $\mathcal{D}$ are drawn iid, this can be rewritten as

$$\mathcal{L}(\mathcal{D}; \theta) = \prod_{i=1}^{m} g_\theta(z_i).$$

Even in the case where we have a continuous output (so a probability density function) instead, this expression remains the same.
As might be expected, in *maximum likelihood estimation* (MLE), we choose that w which maximizes the likelihood. Often, we work with the *log*-likelihood ℓ as well, which is just defined by $\ell(\mathcal{D}; w) = \log \mathcal{L}(\mathcal{D}; w)$.

1.3.2. Bayesian estimation

In all that we have done so far, we have arrived at a single estimate for w or θ. However, this may not necessarily be the best option – might it not be better to arrive at a probability distribution for w which represents how likely various values are?
Bayesian estimation is one simple way of doing this.

We make the assumption that initially, θ is drawn from some *prior* probability distribution p. This allows us to give some input as to how we believe θ behaves. Given the dataset $\mathcal{D} = \{z_1, \ldots, z_m\}$ (and the function g_θ from earlier), we try to improve on this prior to get to a *posterior* probability distribution that better reflects how θ behaves, now that we have knowledge about the dataset. To do this, we use Bayes' law:

$$\begin{aligned} p(\theta \mid \mathcal{D}) &= \frac{\Pr[\mathcal{D} \mid \theta] p(\theta)}{\Pr[\mathcal{D}]} \\ &= \frac{\Pr[\mathcal{D} \mid \theta] p(\theta)}{\int \Pr[\mathcal{D} \mid \alpha] p(\alpha)\, \mathrm{d}\alpha}. \end{aligned} \tag{1.1}$$

Note the contrast between this and MLE; here, we use $\Pr[\theta \mid \mathcal{D}]$, whereas in the latter we use $\Pr[\mathcal{D} \mid \theta]$.
We can then use the posterior to return some estimate of θ.
In *maximum a posteriori estimation* (MAP estimation), we return the mode of the posterior density.
In *Bayesian estimation*, we return the mean of the posterior density.
In the pure Bayesian estimate, we do not return any point estimate for θ at all, and we instead use Bayes' rule to calculate the output given an input as

$$\Pr[x \mid \mathcal{D}] = \int \Pr[x \mid \theta] p(\theta \mid \mathcal{D})\, \mathrm{d}\theta$$

Now, the integral in the denominator of (1.1) can be quite daunting. To deal with this, we introduce the idea of a *conjugate prior*. Such a prior is of the form that both the prior and the posterior are drawn from the same family of distributions. This then enables us to skip the calculation of the denominator, since it merely leads to a constant to normalize the distribution (its integral must be 1), and we know this constant from the structure of the distribution anyway. This can be better understood by an example. Suppose that the datapoints are drawn from a Bernoulli distribution with parameter θ. We then have

$$\Pr[\mathcal{D} \mid \theta] = \theta^r (1-\theta)^{m-r},$$

where r is the number of 1s in $\mathcal{D}$. If we choose our prior to be a Beta distribution with parameters α and β, then the numerator of (1.1) is

$$\theta^r (1-\theta)^{m-r} \cdot \theta^{\alpha-1}(1-\theta)^{\beta-1} \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)}.$$

This is proportional to $\theta^{\alpha+r-1}(1-\theta)^{m-r+\beta-1}$. Since the denominator only gives a constant multiplicative factor, the posterior must just be $\text{Beta}(\alpha+r, \beta+m-r)$.

Some more examples are that the Dirichlet distribution is the conjugate prior to the multivariate Bernoulli and the Gaussian is the conjugate prior to the Gaussian.

1.3.3. Relating probabilistic interpretations and regression

Let us jump back to regression. While we typically assume that our input/output pairs are initially taken from some basic (fixed) function f, this is typically not the case. Indeed, we usually have some amount of error when taking the observations itself. This error is commonly taken to be a Gaussian with 0 mean and some (small) variance σ^2. The output as a whole is then just a random variable, with the parameters involving our base parameter θ *and* the input.

For example, let our training data set be $\mathcal{D} = \{\langle x_1, y_1\rangle, \ldots, \langle x_m, y_m\rangle\}$ where the x_i are in $\mathbb{R}^n$ and the y_i in $\mathbb{R}$. Let us perform normal linear regression, but also add a Gaussian error of known variance σ^2. That is, if the input is x, we predict the distribution of the corresponding output y to be $\mathcal{N}(w^\top x, \sigma^2)$.
Supposing we use MLE, we wish to determine

$$\hat{w} = \arg\max_w \frac{1}{(\sqrt{2\pi}\sigma)^m} \exp\left(-\frac{1}{2\sigma^2}\sum_{i=1}^m (y_i - w^\top x_i)^2\right).$$

It is not too difficult to see that this is in fact the same solution as that given by the discussion in the method of least squares.

1.4. Regularisation

As mentioned in a remark towards the end of Section 1.2.1, overfitting tends to make the error over the test data too large, due to large fluctuations in the model we learn (to emulate the training data better). However, this also leads to large coefficients of the learnt vector, and thus large norm. Regularisation attempts to fix this, by further adding in the constraint that $\|w\|_2$ is not too large.

1.4.1. Regularised regression

This leads to the *penalized regularised least squares regression problem*, where the cost function is replaced with

$$\|\Phi w - y\|_2^2 + \lambda\Omega(w).$$

We are said to be performing

- *ridge regression* if $\Omega(w) = \|w\|_2^2$,

- *lasso*[1] *regression* if $\Omega(w) = \|w\|_1$, and
- *support-based penalty regression* if $\Omega(w) = \|w\|_0$.

The closed-form solution to ridge regression can be computed by setting the gradient to 0, and is given by

$$\hat{w} = (\Phi^\top \Phi + \lambda I)^{-1} \Phi^\top y.$$

Also, this turns out to be the same as performing Bayes/MAP estimation with a multivariate Gaussian prior of mean 0 and variance $(1/\lambda)I$.

Alternatively, we have the *constrained regularised least squares regression problem*, where the cost function is the normal $\|\Phi w - y\|_2^2$, but we have the added constraint that $\Omega(w) \leq \theta$ for some fixed θ.
We again have ridge, lasso, or support-based penalty regression depending on Ω.
It in fact turns out that the above two constraints are equivalent – for any penalized formulation with a λ, there is a corresponding constrained formulation problem with some θ (that depends on both λ and the data) which gives the same solution, and vice-versa.

Why is lasso regression interesting? Observe that the level curves of $\|w\|_1$ (namely cross-polytopes) have many corners on the axes, and as a result, it is likely that many components of w are close to 0 (as opposed to ridge regression where the level curves are rotationally invariant). That is, the learnt vector is likely very sparse, which is desirable.
The closed form solution is the MAP estimate of linear regression subject to the prior being the Laplace distribution with parameters 0 and θ. Note that the Laplace distribution is extremely similar to the Gaussian, except that we use the L^1 norm in the exponential instead of the L^2. The intuition that lasso regression leads to many parameters being close to 0 is strengthened by the sparsity of the Laplace distribution (the density at 0 is high).
However, there is no known closed form solution for the solution to lasso regression. Instead, it is solved iteratively with normal gradient descent. We have

$$\begin{aligned}\hat{w} &= \arg\min_w \|\Phi w - y\|_2^2 + \lambda \|w\|_1 \\ &= \arg\min_w E_{\text{LS}}(w) + \lambda \|w\|_1 .\end{aligned}$$

At the kth step, set

$$\begin{aligned} w_{\text{LS}}^{(k+1)} &= w_{\text{Lasso}}^{(k)} - \eta \nabla E_{\text{LS}}(w_{\text{Lasso}}^{(k)}) \\ (w_{\text{Lasso}}^{(k+1)})_i &= \begin{cases} (w_{\text{LS}}^{(k+1)})_i - \lambda\eta, & (w_{\text{LS}}^{(k+1)})_i > \lambda\eta, \\ (w_{\text{LS}}^{(k+1)})_i + \lambda\eta, & (w_{\text{LS}}^{(k+1)})_i < -\lambda\eta, \\ 0, & \text{otherwise.} \end{cases} \end{aligned}$$

We then estimate $\hat{w}$ as $w_{\text{Lasso}}^{(r)}$ for a suitably large r.

1.4.2. Understanding regularisation

Ridge regression might seem slightly silly from the perspective of just training error. Indeed, the the (minimized) error for the ridge cost function is always greater than the (minimized) error for normal least squares.

Suppose we are performing regression with a true function g and let f be our predicted/fitted function. We sample a bunch of points from g to get a dataset, with possibly some noise that causes small perturbations.
There are three primary sources of test error:

- *Bias*: This occurs due to the difference between the true function and fitted function (or rather, the expected fitted function over some distribution of datasets). It is just equal to $\mathbf{E}[f(x)] - g(x)$.

[1] this is an abbreviation for *l*east *a*bsolute *s*hrinkage and *s*election *o*perator

- *Variance*: This occurs due to the choice of our dataset. It is equal to $\mathbf{Var}(f(\hat{x}))$ (where again, the variance is taken over datasets).
- *Noise*: This occurs due to the implicit noise that is present when we take measurements from the true function. For example, there may be an additive Gaussian noise with mean 0 and variance σ^2. The noise is then just equal to the variance σ^2 of this component.

To reiterate, there are two (independent) sources of randomness: the dataset (we assume that there is some 'nice' distribution over all possible datasets) and the noise.
Now, suppose the noise is some random variable ε with mean 0 and variance σ^2. That is, given an input x, we observe the output y as being drawn from $g(x) + \varepsilon$.
Suppose that for a given dataset $\mathcal{D}$, we predict the function $f_{\mathcal{D}}$.
Fix some new input $\hat{x}$. The question we would like to answer is: what is the expected test error $\mathbf{E}[(f_{\mathcal{D}}(\hat{x}) - (g(\hat{x}) + \varepsilon))^2]$?
The expectation is taken over both the dataset and the noise.
Let $\hat{y} = g(\hat{x}) + \varepsilon$. We then have

$$\begin{aligned}
\mathbf{E}[(f_{\mathcal{D}}(\hat{x}) - \hat{y})^2] &= \mathbf{E}[f_{\mathcal{D}}(\hat{x})^2] + \mathbf{E}[\hat{y}^2] - 2\mathbf{E}[f_{\mathcal{D}}(\hat{x})]\mathbf{E}[\hat{y}] && (\mathcal{D} \text{ and } \varepsilon \text{ are independent}) \\
&= \mathbf{E}[f_{\mathcal{D}}(\hat{x})]^2 + \mathbf{Var}(f_{\mathcal{D}}(\hat{x})) + \mathbf{E}[\hat{y}^2] - 2\mathbf{E}[f_{\mathcal{D}}(x)]g(\hat{x}) \\
&= \mathbf{E}[f_{\mathcal{D}}(\hat{x})]^2 + \mathbf{Var}(f_{\mathcal{D}}(\hat{x})) + \mathbf{E}[g(\hat{x})^2 + \varepsilon^2] - 2\mathbf{E}[f_{\mathcal{D}}(x)]g(\hat{x}) \\
&= \mathbf{E}[f_{\mathcal{D}}(\hat{x})]^2 + g(\hat{x})^2 - 2\mathbf{E}[f_{\mathcal{D}}(x)]g(\hat{x}) + \mathbf{Var}(f_{\mathcal{D}}(\hat{x})) + \sigma^2 \\
&= \underbrace{(\mathbf{E}[f_{\mathcal{D}}(\hat{x})] - g(\hat{x}))^2}_{\text{Bias}^2} + \underbrace{\mathbf{Var}(f_{\mathcal{D}}(\hat{x}))}_{\text{Variance}} + \underbrace{\sigma^2}_{\text{Noise}} .
\end{aligned}$$

In this context, all regularization does is decrease the variance at the expense of possibly increasing the bias.

Given the bias-variance tradeoff, how do we choose the best predictor? How do we set the model's parameters?
In *bootstrap sampling*, we repeatedly sample observations from a dataset with replacement. Given dataset $\mathcal{D}_b$, we repeatedly choose a V_b, train on $\mathcal{D}_b \setminus V_b$, then test on V_b.
So, the training set is that we use to actually train the algorithm. V_b is the *validation* or *development* set, and is used for model selection and tuning hyperparameters. The test/evaluation set is that we use for testing finally.

§2. Classification

Suppose we wish to intelligently classify emails into two categories depending on whether they are spam or not. How do we learn a method to classify an email as spam?
It is not similar to regression where the function maps to $\mathbb{R}$, but we instead map to a discrete set $\{c_1, c_2, \ldots, c_k\}$. The simple hack of mapping c_i to i, say, and then performing usual regression. However, the ordering of the c_i influences it significantly then, which is very undesirable, since the ordering doesn't mean anything.
A better idea would be to take c_i as

$$y_i = e_i \in \mathbb{R}^k,$$

to encode the idea that the various components are 'independent'.
We present these ideas over the next few sections.

2.1. Perceptron Classification

2.1.1. The basic idea

Rather than mapping directly to $\mathcal{C} = \{c_1, \ldots, c_k\}$, we could map to some intermediate space which in turn maps to $\mathcal{C}$ (or a sequence of spaces and functions, the last of which is $\mathcal{C}$).

First, consider the binary classification problem, where $\mathcal{C} = \{-1, 1\}$. Let $\mathcal{D} = \{\langle x_i, y_i\rangle : 1 \leq i \leq m\}$, where each $y_i \in \{-1, 1\}$. The ordering does not matter then (both orderings are equivalent), so we can learn a linear classifier. Suppose $f(x) = w^\top \phi(x)$.
Given a new input x, we predict the new output as 1 if $f(x) \geq 0$ and -1 if $f(x) < 0$. So, we want

$$y \cdot w^\top \phi(x) \geq 0.$$

We learn w as follows. Starting with $w^{(0)} = 0$, we iterate over the dataset. If a point is correctly classified, we leave $w^{(r+1)} = w^{(r)}$. Otherwise, we marginally correct the weights by performing

$$w^{(r+1)} = w^{(r)} + \eta y_{r+1} \phi(x_{r+1}),$$

for some (typically small) learning rate η.
Observe that the quantity $w^\top \phi(x_r)$ is the distance of x_r from the hyperplane $\{x : w^\top \phi(x) = 0\}$.
So,

$$y_{r+1}(w^{(r+1)})^\top \phi(x_r) = y_{r+1}(w^{(r)})^\top \phi(x_{r+1}) + \eta \left\|\phi(x_{r+1})\right\|^2.$$

This makes the quantity $y_{r+1} w^\top \phi(x_r)$ less negative. We repeat the above process until $y_i w^\top \phi(x_i) \geq 0$ for all i (or until we terminate).
This *perceptron* algorithm does not attempt to find the best separating hyperplane, it merely tries to find any separating hyperplane – it need not provide any 'breathing room' for the inputs. This is addressed by support vector machines, which we shall read later.
One issue with perceptron classification is that it does not acknowledge how much the estimate differs from the expected ± 1 – we only care that $yw^\top \phi(x) \geq 0$, not how positive it is.

2.1.2. Convergence

Now, suppose there exists some u such that for all i, $y_i u^\top \phi(x_i) \geq 0$. It turns out that the perceptron update rule will then converge in a finite number of iterations. That is, perceptrons can model linearly separable functions.

Proof. Since we can normalize u, suppose that $\|u\| = 1$. We can also normalize the dataset itself, so assume $\left\|\phi(x_i)\right\| \leq 1$ for all i.
Define the *margin* $\gamma = \min_i |u^\top \phi(x_i)| = \min_i y_i u^\top \phi(x_i)$.
Further make the assumption that $\eta = 1$ (the general case follows similarly.)
We claim that in at most $t = 1/\gamma^2$ updates, we will have $y_i (w^{(t)})^\top x_i > 0$ for all i.
If we update $w^{(k)}$ to $w^{(k+1)} = w^{(k)} + y\phi(x)$ on reading the input $\langle x, y\rangle$,

$$(w^{(k+1)})^\top u = (w^{(k)} + y\phi(x))^\top u \geq (w^{(k)})^\top u + \gamma.$$

and

$$\left\|w^{(k+1)}\right\|^2 = \left\|w^{(k)} + y\phi(x)\right\|^2 < \left\|w^{(k)}\right\|^2 + 1.$$

As a result, after k updates,

$$(w^{(k)})^\top u \geq k\gamma \text{ and } \left\|w^{(k)}\right\|^2 \leq k$$

so

$$\sqrt{k} \geq \underbrace{\left\|w^{(k)}\right\| \geq u^\top w^{(k)}}_{\text{Using Cauchy-Schwarz}} \geq k\gamma.$$

This gives $k \leq 1/\gamma^2$, completing the proof. ■

2.1.3. Viewing the algorithm with a loss function

Consider the loss function

$$L_P(w^\top(\phi(x)), y) = \max\{0, -yw^\top\phi(x)\}.$$

We can then apply the *stochastic gradient descent* algorithm. Let $f_w(x) = w^\top\phi(x)$ and choose some example $\langle x_t, y_t\rangle$. We then compute the gradient of the loss of $f_w(x_t)$ with respect to w, then update the weight vector using it as

$$w \leftarrow w - \nabla_w L_P(f_w(x_t), y).$$

Perceptron update is then just stochastic gradient descent on this "hinge loss" function.
Similarly, we could use different loss functions.

Now, what w should we use at the time of testing? Using the final w is not always a good idea due to overfitting and the stochastic nature of the algorithm.
So, we usually use an intermediate w. The *voted perceptron* algorithm counts votes for weight vectors tallying how many examples they have correctly classified. The *averaged perceptron* algorithm uses the averaged weight vector (not necessarily the average of all weight vectors, but a suitable subset).

Typically, rather than doing stochastic gradient descent (datapoint by datapoint), we perform a similar operation using batches of datapoints. That is, we do

$$w \leftarrow w - \frac{1}{|\mathcal{B}|}\sum_{(x,y)\in\mathcal{B}} \nabla_w L_P(f_w(x), y)$$

for a suitable batch $\mathcal{B}$, which is intermediate between normal gradient descent and stochastic gradient descent. Observe that if $\mathcal{B}$ is the entire dataset, we get ordinary gradient descent and if $\mathcal{B}$ is a singleton, we get stochastic gradient descent.

2.2. Logistic Regression

Recall that the function we use in perceptron classification is just a step function. Is it possible to do better, and possibly take care of smoothness issues? That is, instead of $\text{sign}(w^\top\phi(x))$, can we use some more general $\Omega(w^\top\phi(x))$? In logistic regression, we use the *sigmoid function* defined by

$$\sigma(t) = \frac{1}{1+e^{-t}}.$$

Rather than looking at whether the value of the function is positive or negative, we instead see here whether it is greater than or less than $1/2$. Now, let us return to the binary classification problem, taking $\mathcal{C} = \{0, 1\}$ instead of $\{-1, 1\}$. Since the sigmoid maps into $(0, 1)$, this also allows us a probabilistic interpretation as

$$\Pr[y = 1 \mid x] = \frac{1}{1+e^{-w^\top\phi(x)}} \text{ and } \Pr[y = 0 \mid x] = \frac{e^{-w^\top\phi(x)}}{1+e^{-w^\top\phi(x)}}.$$

That is, we can view it as a Bernoulli model with parameter $\sigma(w^\top \phi(x))$.
We can then do maximum likelihood estimation for the parameter of the Bernoulli (or minimize the error), and thus arrive at an estimate for w! We can also do something along the lines of Bayesian or MAP estimation.
To be more explicit, the maximum likelihood estimate is is

$$\begin{aligned}
\hat{w} &= \arg\max_w \mathcal{L}(\mathcal{D}; w) \\
&= \arg\max_w \Pr\left[y_1, \ldots, y_m \mid w, x_1, \ldots, x_m\right] \\
&= \arg\max_w \prod_{i=1}^m \Pr\left[y_i \mid w, x_i\right] \\
&= \arg\max_w \prod_{i=1}^m \sigma(w^\top \phi(x_i))^{y_i} (1 - \sigma(w^\top \phi(x_i)))^{1-y_i} \\
&= \arg\max_w \sum_{i=1}^m y_i \log(w^\top \phi(x_i)) + (1 - y_i) \log(1 - \sigma(w^\top \phi(x_i))) \\
&= \arg\min_w \underbrace{-\sum_{i=1}^m y_i \log(w^\top \phi(x_i)) + (1 - y_i) \log(1 - \sigma(w^\top \phi(x_i)))}_{-\ell(\mathcal{D}; w)} .
\end{aligned}$$

The average negative log-likelihood ($-1/m \log(\mathcal{L}(\mathcal{D}; w))$) is often referred to as the *cross-entropy* loss function $E(w)$, and represents a "distance" between the data distribution (the y_i) and the model distribution (the σ). Cross-entropy is the average number of bits required to identify an event drawn from the dataset, if a coding scheme is used that is optimized for a modelled probability distribution $\Pr[y \mid \mathcal{M}]$ (as opposed to the 'true' distribution $\Pr[y \mid \mathcal{D}]$).
Alternatively, we have

$$E(w) = -\frac{1}{m} \sum_{i=1}^m y_i w^\top \phi(x_i) - \log(1 + e^{w^\top \phi(x_i)}).$$

The first component here resembles the loss in perceptron classification.
We do not have a simple closed-form solution for the cross-entropy loss function, so we apply gradient descent. The update rule looks like

$$w^{(k+1)} = w^{(k)} + \eta \left(\frac{1}{m} \sum_{i=1}^m (y_i - \sigma((w^{(k)})^\top \phi(x_i))) \phi(x_i) \right).$$

The stochastic version of this is

$$w^{(k+1)} = w^{(k)} + \eta (y_i - \sigma((w^{(k)})^\top \phi(x_i))) \phi(x_i).$$

We shall return to something similar when we study generalized linear models later in the course, of which perceptron, linear regression, and logistic regression are special cases.
The *regularized* (logistic) cross-entropy loss function is

$$E_r(w) = -\frac{1}{m} \sum_{i=1}^m y_i \log(w^\top \phi(x_i)) + (1 - y_i) \log(1 - \sigma(w^\top \phi(x_i))) + \frac{\lambda}{2m} \|w\|_2^2 ,$$

which attempts to avoid overfitting. The probabilistic interpretation of this is that this is the MAP estimate given a Gaussian prior on w with mean 0 and variance $1/\lambda$. In this case, the gradient descent update rule is

$$w^{(k+1)} = w^{(k)} + \eta \left(\frac{1}{m} \sum_{i=1}^m (y_i - \sigma((w^{(k)})^\top \phi(x_i))) \phi(x_i) - \lambda w^{(k)} \right).$$

Logistic regression is easily extended to even the multi-class scenario (say k classes $1, \ldots, k$), by keeping a different weight vector w_c for each $1 \leq c \leq k-1$, then letting

$$\Pr[y = c \mid x] = \frac{e^{-w_c^\top \phi(x)}}{1 + \sum_{i=1}^{k-1} e^{-w_i^\top \phi(x)}}$$

for $1 \leq c \leq k-1$ and

$$\Pr[y = k \mid x] = \frac{1}{1 + \sum_{i=1}^{k-1} e^{-w_i^\top \phi(x)}}.$$

We can have parameters for all k classes, but this is not necessary since the kth class occurs iff none of the other $k-1$ classes occurs.

2.3. Kernels

2.3.1. Introduction

Recall the perceptron update rule. If we update $w^{(k)}$ to $w^{(k+1)}$ on reading the misclassified point (x', y'), we have for any datapoint (x, y) that

$$(w^{(k+1)})^\top \phi(x) = (w^{(k)})^\top \phi(x) + y' \phi(x')^\top \phi(x).$$

The quantity $\phi(x')^\top \phi(x)$ represents some sort of similarity between $\phi(x')$ and $\phi(x)$. Let $f^{(k)}(x) = (w^{(k)})^\top \phi(x)$. We then have

$$f^{(k+1)}(x) = f^{(k)}(x) + y' \phi(x')^\top \phi(x).$$

Now, if we obtained $w^{(k)}$ from $w^{(k-1)}$ on reading the datapoint (x'', y''), we can perform a similar substitution to get

$$f^{(k+1)}(x) = f^{(k-1)}(x) + y' \phi(x')^\top \phi(x) + y'' \phi(x'')^\top \phi(x).$$

More generally, if we obtained $w^{(i+1)}$ from $w^{(i)}$ on reading the datapoint (x_i, y_i) for each i,

$$f^{(k+1)}(x) = f^{(0)}(x) + \sum_{i=0}^{k} y_i \phi(x_i)^\top \phi(x).$$

If for each datapoint (x', y'), $\alpha_{x'}$ represents the number of times it was misclassified in the first k updates, then

$$f^{(k+1)}(x) = f^{(0)} x + \sum_{(x', y')} y' \phi(x')^\top \phi(x) \alpha_{x'}.$$

So, the final score is the initial score plus some weighted sum of the similarities of each of the misclassified points with the test point. Let $K(x, x') = \phi(x')^\top \phi(x)$ for each pair (x, x') represent a notion of similarity (as we mentioned earlier). We then just have

$$f^{(k+1)}(x) = f^{(0)}(x) + \sum_{(x', y')} y' K(x, x') \alpha_{x'}.$$

Misclassification of a point x indirectly computes similarity $K(x, x')$ for each of the previously misclassified points through the classifier, since we have 'folded in' the information about x' into the classifier. Observe that here, the classifier is parametrized using the α_x, *not* the w as earlier.

Based on this, we define the *kernelized perceptron* as follows. We have a notion of similarity given by $K(x, y) = \phi(x)^\top \phi(y)$ and initialize all the α_x as 0. Let $f^{(0)}$ be identically zero. Given $f^{(k)}$ and a misclassified point (x, y), a point such that

$$y f^{(k)}(x) = \sum_{(x', y')} y y' K(x, x') \alpha_{x'} < 0,$$

we update $\alpha_x \leftarrow \alpha_x + 1$. We could also add a bias b to f (essentially initializing $f^{(0)}$ to be the constant b), which represents' a non-zero initialization of w.

Based on how we choose ϕ, and thus $K(\cdot, \cdot)$, we can adapt to datasets that have separators far more complicated than the simple linear separator we have been dealing with thus far! For example, the choice of the *radial basis function* given by $K(x, y) = \exp(-\|x - y\|_2^2 / 2\sigma^2)$ can deal with spherical separators.
Kernels typically operate in high-dimensional implicit feature spaces. What this means is that if our original dataset has the x from, say, $\mathbb{R}^d$ but our actual ϕ maps to $\mathbb{R}^n$ for some high n, it might often be possible (for nice choices of ϕ)

to express $K(x, y)$ in terms of some simpler function of x and y itself. As a result, we don't necessarily need to compute the coordinates of $\phi(x)$ and can get away with just manipulating the coordinates of the datapoint to compute the kernel function. This tends to be computationally cheaper.

For a dataset $\{x_1, \ldots, x_m\}$, the *Gram matrix* $\mathcal{K}$ is defined as

$$\mathcal{K} = \begin{pmatrix} K(x_1, x_1) & \cdots & K(x_1, x_m) \\ \vdots & \ddots & \vdots \\ K(x_m, x_1) & \cdots & K(x_m, x_m) \end{pmatrix}.$$

2.3.2. Positive semi-definite kernels

What are some properties we desire from K, and so $\mathcal{K}$? A few that come to mind are:

- $K(\cdot, \cdot)$ is symmetric in the two variables, that is, $\mathcal{K}$ is symmetric.
- $\mathcal{K}$ is positive-semidefinite. Indeed, for any $v \in \mathbb{R}^m$,

$$\begin{aligned} v^\top \mathcal{K} v &= \sum_{i,j} v_i \mathcal{K}_{ij} v_j \\ &= \sum_{i,j} v_i v_j \phi(x_i)^\top \phi(x_j)^\top \\ &= \left(\sum_i v_i \phi(x_i) \right)^\top \left(\sum_j v_j \phi(x_j) \right) \\ &= \left\| \sum_i v_i \phi(x_i) \right\|_2^2 \geq 0. \end{aligned}$$

A kernel $K(\cdot, \cdot)$ satisfying the above two conditions is known as a positive semi-definite kernel.

We clearly have that for any choice of ϕ, a positive semi-definite kernel is obtained. Is the converse true as well? Yes, we claim that for any positive semi-definite kernel, there exists a Hilbert space H and a $\phi : \mathbb{R}^n \to H$ such that $K(x_i, x_j) = \phi(x_i)^\top \phi(x_j)$. Indeed, if K is symmetric and positive semi-definite, we can perform the Cholsky(/singular value) decomposition on $\mathcal{K}$ to get

$$\mathcal{K} = U \Sigma U^\top = (U \Sigma^{1/2})(U \Sigma^{1/2})^\top = R R^\top,$$

where the rows of U are linearly independent and Σ is a diagonal matrix with non-negative entries.
Mapping each datapoint x_i to the ith row of $\mathcal{K}$ gives the required.

2.3.3. Mercer kernels

What about when our set of features is infinite-dimensional? Indeed, in this case we can extend it to an eigen*function* decomposition and write

$$K(x_1, x_2) = \sum_{i=1}^{\infty} \alpha_i \phi_i(x_1) \phi_i(x_2),$$

where each $\alpha_i \geq 0$ and $\sum_{i=1}^{\infty} \alpha_i^2 < \infty$ (the α_i are eigenvalues and the ϕ_i are eigenfunctions). This is known as the *Mercer kernel* decomposition.

Equivalently, $K(x_1, x_2)$ is a Mercer kernel (and is said to satisfy Mercer's condition) if for all square integrable functions g,

$$\iint K(x_1, x_2)g(x_1)g(x_2)\,\mathrm{d}x_1\,\mathrm{d}x_2.$$

When the input space of x is compact, the Mercer kernel and positive semi-definite kernel turn out to be equivalent.

Again, as before, for any choice of ϕ, we obtain a Mercer kernel. The converse is true here as well.
That is, for any Mercer Kernel $K(x_1, x_2)$, there exists a $\phi(x) : \mathbb{R}^n \to H$ such that $K(x_1, x_2) = \phi(x_1)^\top \phi(x_2)$, where H is a Hilbert space.

Thus, we don't really need to deal with basis functions when dealing with this method, since all we need is a (positive semi-definite or Mercer) kernel.
Now, let us discuss a couple of properties of positive semi-definite/Mercer kernels.

- If K_1 and K_2 are kernels and $\alpha_1, \alpha_2 \geq 0$, then $\alpha_1 K_1 + \alpha_2 K_2$ is a kernel. (think of appending the two basis functions)
- If K_1 and K_2 are kernels, then $K_1 K_2$ is a kernel. (think of converting the product of sums to a product of sums)

2.3.4. Example: kernelization of regression

Now, let us try to kernelize regularized logistic regression. We had defined the cross-entropy loss function

$$E(w) = -\frac{1}{m}\sum_{i=1}^{m}\left(y_i w^\top \phi(x_i) - \log\left(1 + \exp(w^\top \phi(x_i))\right)\right) + \frac{\lambda}{2m}\|w\|_2^2.$$

The equivalent dual kernelized objective, which comes from the *representer theorem* (we shall study this later) is given by
In place of the 'hard' thresholding we had done in ordinary perceptron with the hinge loss, here we have a 'soft' thresholding which has probabilites instead. As a result, the decision function is instead given by

$$\sigma_w(x) = \frac{1}{1 + \exp(-\sum_j \alpha_j K(x, x_j))}$$

and the dual kernelized objective (which we must minimize with respect to $(\alpha_i) \in \mathbb{R}^m$) is

$$E_D(\alpha) = -\sum_{i=1}^{m}\left(\left(\sum_{j=1}^{m} y_i K(x_i, x_j)\alpha_j - \frac{\lambda}{2}\alpha_i K(x_i, x_j)\alpha_j\right) - \log\left(1 + \exp\left(-\sum_{i=1}^{m}\alpha_j K(x_i, x_j)\right)\right)\right).$$

Recall that in ridge regression,

$$\begin{aligned}\hat{w} &= \arg\min_w \|\Phi w - y\|_2^2 + \lambda\|w\|_2^2 \\ &= (\Phi^\top\Phi + \lambda I)^{-1}\Phi^\top y.\end{aligned}$$

Consider the ridge regression function

$$f(x) = \phi^\top(x)w = \phi^\top(x)(\Phi^\top\Phi + \lambda I)^{-1}\phi^\top y.$$

Consider the following matrix identity.

$$(P^{-1} + B^\top R^{-1} B)^{-1} B^\top R^{-1} = PB^\top(BPB^\top + R)^{-1}.$$

Setting $R = I$, $P = (1/\lambda I)$, and $B = \Phi$,

$$(\lambda I + \Phi^\top\Phi)^{-1}\Phi^\top = (1/\lambda)\Phi^\top((1/\lambda)\Phi\Phi^\top + I)^{-1}.$$

So,

$$f(x) = \phi(x)^\top \Phi^\top (\Phi\Phi^\top + \lambda I)^{-1} y.$$

$\Phi^\top\Phi$ is exactly what we want, since $(\Phi^\top\Phi)_{ij} = K(x_i, x_j)$. Further, the $\phi(x)^\top\Phi^\top$ combines in terms of the kernel function as well! Overall, we have

$$f(x) = \sum_i \alpha_i K(x, x_i),$$

where $\alpha_i = ((\Phi\Phi^\top + \lambda I)^{-1} y)_i$.

2.4. Support Vector Machines

A drawback of perceptron classification we looked at earlier was that it does not find the *best* separating hyperplane. This can be solved using 'large margins' in support vector classification.
Also, perceptrons only deal with linearly separable data. Of course, we fixed this using kernel methods, but we still need to design the kernel properly. To deal with this, we design *neural networks*, which are a cascade of (sigmoidal) perceptrons.

Support Vector Machines (SVMs) address the first issue of breathing room. Suppose that we have 'strong separation' for a separating vector w with

$$yw^\top \phi(x) \geq 1$$

instead of just 0. Let x^+, x^- with $w^\top\phi(x^+) = 1$ and $w^\top\phi(x^-) = -1$. Then,

$$\phi(x^+) - \phi(x^-) = m\frac{w}{\|w\|}$$

where m is the margin, and

$$w^\top(\phi(x^+) - \phi(x^-)) = 2.$$

Combining the two,

$$m\|w\| = 2.$$

So, the margin is proportional to $1/\|w\|$.

But of course, we need not have this strong separation. Suppose we instead have that

$$y_i w^\top \phi(x_i) \geq 1 - \xi_i,$$

where ξ_i is some 'slackness'. SVM tries to maximize the margin (proportional to $1/\|w\|$), and minimize the overall slackness $\sum_i \xi_i$.
So, we typically use the cost function

$$w^*, (\xi_i^*) = \arg\min_{w,\xi_i} \frac{1}{2}\|w\|^2 + C\sum_i \xi_i$$

under the constraint that

$$y_i w^\top \phi(x_i) \geq 1 - \xi_i$$

and $\xi_i \geq 0$ for each i.
The constant C determines the tradeoff between slackness minimization and margin maximization.

2.5. Neural Networks

As we saw earlier, one issue with perceptrons was that it does not ensure a large margin. We dealt with this by introducing support vector machines.
We also saw how non-linear classification could be attained using kernels. Neural networks are another way of achieving this.

Neural networks are interesting because they have *universal* approximation properties, with regards to both regression and classification. The greater the required accuracy, the more hidden units we require. This is codified by the no free lunch theorems.

We first focus on classification. Suppose we have m data points $(x_i)_{i=1}^m$, and k classes $(c_i)_{i=1}^k$. In general, we come up with a sequence of mappings between intermediate 'layers' of spaces, the first being that of x and the last being the set of classes.
Consider classification, where we have some activation function $g(w^\top \phi(x))$. We are familiar with the sigmoid or linear functions. We also often use the $\tanh$ activation function and the ReLU function which is just defined by $g(s) = \max\{s, 0\}$. We also sometimes use the Softplus function $g(s) = \ln(1 + e^s)$. We have also seen the multiclass logistic regression with

$$\Pr[y = c \mid \phi(x)] = \frac{e^{-(w_c)^\top \phi(x)}}{1 + \sum_{k=1}^{K-1} e^{-(w_k)^\top \phi(x)}}$$

and softmax which has k parameters instead (one of the parameters is redundant).

Now, a question one can ask is: how does one measure the separability that can be attained by a classifier?
In pursuit of this, we define the **VC-dimension** (**V**apnik-**C**hervonenkis), which represents the richness of the space of functions expressible by a classifier. A classification function f_w is said to *shatter* a set of datapoints $(x_1, \ldots, x_n)$ if for all assignments of labels to these points, there exists some w such that f_w makes no errors when evaluating that set of datapoints. The cardinality of the largest set of points that $f(w)$ can shatter is its VC-dimension.
For example, one can show that the VC-dimension of the linear classifier in $\mathbb{R}^2$ is 3 (Why?). More generally, the VC-dimension of the linear classifier in $\mathbb{R}^n$ is $n + 1$.

The VC-dimension of neural networks increases rapidly with more layers in between and more perceptrons in each layer. We cascade *non-linear*[2] activation functions, which allows more expressive power.

Neural networks are easier to understand with the context of pursuit projection regression, where we have some M, and we predict the output of input X as

$$f(X) = \sum_{m=1}^{M} g_m(w_m^\top X)$$

for some functions g_m and suitable unit vectors $(w_m)_{m=1}^M$.
To put it in more familiar terms, one may try to minimize the least squares error

$$\sum_{i=1}^{N} \left(y_i - \sum_{m=1}^{M} g_m(w_m^\top x_i) \right)^2.$$

Consider the scenario where $M = 1$ and we have $w = w_m$, $g = g_m$. Given a choice of w, one can estimate the optimal g through various means, and given a choice of g, one can estimate the optimal w through a combination of gradient descent and least squares regression.

Now, consider the two-level neural network model. As mentioned earlier, the basic idea is to choose an intermediate space and allow *two* functions, one from the input space to the intermediate space and another from the intermediate space to the final output space, thus allowing more expressiveness.
For starters, suppose we are in the classification setting with K classes. Given input X, one may define the M-dimensional vector Z by

$$Z_m = \sigma(\alpha_m^\top X)$$

and the K-dimensional vector T by

$$T_k = \beta_k^\top Z$$

[2] if we use linear everywhere, we do not gain any expressive power by adding more layers.

for some vectors $(\alpha_m)_{m=1}^M$ and $(\beta_k)_{k=1}^K$. σ is an activation function, that may be chosen to be the *sigmoid* function we looked at earlier.
So, overall, we get a number $f_k(X) = g_k(T)$ corresponding to each class k. In the final step, one may take g_k to be something like the softmax function that we used in logistic regression, for instance.
The idea of neural networks is largely similar to that of the usual basis functions we have used in the past, except that the manner in which we 'expand' the basis is learned from the data as well.
Observe that the power of the above two-level network collapses if we choose σ to be linear. Also note that the two-level neural network model is a form of pursuit projection regression, where all the g_m are σ.

While designing a neural network, we typically use one hidden layer, and if otherwise, we maintain the same number of hidden units in each layer (this refers to the dimension of that layer). The number of hidden units in each layer is also usually kept a constant factor times the dimension of the input.
Logistic regression is essentially just a single-node sigmoidal neural network (with no hidden layers).
For neural networks, the level curves of our functions can be highly non-convex as well. How do we ensure that we go to the true global minimum and not a local minimum in gradient descent?
We also have the algorithmic challenge of handling 'unknown' activations in the hidden layer.

What we do is back-propagation. Recall how in pursuit projection regression, if one of w or g is fixed, we can improve the other. This allows us to repeatedly improve each of w and g using the other. As an example, suppose we have

$$f(x, y, z) = g_f(w_q g_q(w_x x + w_y y) + w_z z).$$

We then have

$$\frac{\partial f}{\partial w_x} = \frac{\partial g_f}{\partial g_q} \cdot \frac{\partial g_q}{\partial w_x}.$$

This application of the chain rule is the first part of backpropagation.
So, in all, we randomly initialize the weights $w_{i,j}^\ell$. We then do forward propagation to get the intermediate activation functions f_w. We then execute backpropagation by computing the partial derivatives of the *overall* cost function that depends on all the $w_{i,j}^\ell$ with respect to each parameter, and use gradient descent in an attempt to minimize the (possibly non-convex) cost. We must then verify that the cost has actually reduced, and if not we apply some random perturbation of the weights (or we could start from the beginning again with different initializations).
When calculating the gradients, certain partial derivatives in earlier stages of the network (for different function) may be used in partial derivatives with respect to many variables, so memoization can significantly speed up the back-propagation step. Our notation is that the node corresponding to function $\sigma_i^{\ell-1}$ is connected to the node (in the next layer) corresponding to $\sigma_j^{\ell-1}$ by the weight $w_{i,j}^\ell$.
Let us denote by sum_j^ℓ the weighted sum of all the functions coming into σ_j^ℓ. That is, $\text{sum}_j^\ell = \sum_{i=1}^{s_{\ell-1}} \sigma_i^{\ell-1} w_{i,j}^\ell$, where $s_{\ell-1}$ is the number of nodes in the $(\ell-1)$th layer. The neural network objective to be minimized is

$$E(w) = -\frac{1}{M}\left(\sum_{i=1}^{M}\sum_{k=1}^{K}(y_i)_k \log(\sigma_k^L(x_i)) + (1-(y_i)_k)\log(1-\sigma_k^L(x_i))\right) + \frac{\lambda}{2M}\sum_{\ell=1}^{L}\sum_{i=1}^{s_{\ell-1}}\sum_{j=1}^{s_\ell}\left(w_{i,j}^\ell\right)^2.$$

The 0th layer is considered as the input layer. We have

$$\frac{\partial E}{\partial w_{i,j}^\ell} = \frac{\partial E}{\partial \sigma_j^\ell}\frac{\partial \sigma_j^\ell}{\partial \text{sum}_j^\ell}\frac{\partial \text{sum}_j^\ell}{w_{i,j}^\ell}.$$

Calculating the first of the three derivatives above is only problematic part.

$$\frac{\partial E}{\partial \sigma_j^\ell} = \sum_{p=1}^{s_{\ell+1}} \frac{\partial E}{\partial \sigma_p^{\ell+1}}\frac{\partial \sigma_p^{\ell+1}}{\partial \text{sum}_j^{\ell+1}} w_{j,p}^{\ell+1}$$

The above is simple for $\ell = L$, but when $\ell < L$, we must recursively calculate it using higher layers, which is where memoization comes into play.
The second derivative is just $\sigma_j^\ell(1-\sigma_j^\ell)$, and the third component is $\sigma_i^{\ell-1}$.

Now, what are some issues with neural networks?

- The actual evaluation function is not equal to the surrogate loss function.
- Finding global minima might be problematic due to non-convexity, and we might only arrive at a local minimum. This is resolved to an extent by stochastic gradient descent.
- We require a large number of training instances for good precision. Sometimes, 'pre-training' is done where we perform unsupervised learning in the first few layers.
- There are extensive computational requirements and numerical precision is needed.
- How many nodes/edges do we choose in each layer? How many layers do we choose? To account for this, we perform regularization after overestimation using *Dropout*, where we set some weights to 0 (in a random fashion).
- Overfitting is a bigger issue here due to the universal approximation properties of neural networks.

In general, bias is lowered by

- increasing the number of epochs or
- increasing the size of the network

and variance is lowered by

- increasing the size of the training set,
- designing the network more intelligently, or
- regularization. Some methods o regularization are norm penalties, dropout, bagging and other ensemble methods, dataset augmentation, manifold tangent classification, multi-task learning, semi-supervised learning, adversarial learning, parameter tying and sharing, and sparse representations.

We want to 'tame' neural networks intelligently. One way to do this is using *convolutional neural networks*.
The basic idea of this is that we attempt to recognize locally connected common features. For example, if we wish to recognize a car, we might begin by checking if a door exists, then a windshield. We can then do weight sharing to detect these recurring components, say multiple wheels. Finally, we try to shift the image to bring it into some form.
First off, the *depth* or number of *feature-maps* is the number of quantities at each pixel. For example, if we have an RGB image, the depth is 3.
To bring about the local components, instead of having a weight for every single pair in adjacent layers, we only have weights between 'nearby' neurons. This ensures sparse interactions (and thus fewer parameters to learn) using a filter of sorts. For example, if we have 2-dimensional inputs, neighbouring pixels are more relevant than those further away. This can be viewed as multiplication with a Toeplitz matrix. Further, we could have weights only depending on the distance between the pixels. So, we have something like

$$h_i = \sum_m x_m w_{i-m} K(i-m).$$

This indicates that neighbouring signals interact in similar ways *irrespective of dimension*. This can be thought of as moving a 'patch' around.
Now, we don't need to move this patch aroudn one step at a time, we can do so by s steps at a time instead. The *stride* s corresponds to downsampling it by s. We also perform some *padding* at the edges.
Some notable examples of convolutions in image processing (the weights that contribute to a given pixel) are the Sobel vertical edge detector

$$\begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix},$$

the Sobel horizontal edge detector

$$\begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix},$$

the image blurring filter

$$\frac{1}{9}\begin{pmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{pmatrix},$$

and the image sharpening filter

$$\begin{pmatrix}0 & -1 & 0\\ -1 & 3 & -1\\ 0 & -1 & 0\end{pmatrix}.$$

The three main ideas leveraged are sparse interactions, parameter sharing, and equivariant representations ($f(g(x)) = g(f(x))$ where f is a convolution and g is a shift function)
It is possible to increase the depth after performing something called *pooling*. The increase in depth is primarily due to the application of multiple fitters. We learn all the fitters by training. In striding, we just perform downsampling. In pooling on the other hand, we perform non-linear downsampling.
What if we want to decrease the number of weights/neurons even more? A common example of this is to use the max-pooling filter where we just choose the maximum weight from each patch of a certain size.
To summarize, the convolutional layer can be thought of as consisting of the initial convolution stage with an affine transform, a detector stage which introduces non-linearity, and the pooling layer.[3]
Is it possible to do deconvolution? That is, we want to enhance the image from blurry features. Using convolution and deconvolution together gives a PCA-like effect (this is done in unsupervised learning).

Next, what if we want to predict a sequential output? That is, later parts of the output depend on other parts somehow. These are known as recurrent neural networks (RNNs). We define this using Markov/conditional independence. We also have something known as the factorization property. It turns out that under suitable conditions, these two are equivalent, as given by the Hammersley-Clifford Theorem.
We combine probability theory and graph theory to get probabilistic graphical models. These provide new insights into existing models and motivation for new models. Each node in a graphical model represents a random variable (or a vector of random variables). There are edges between nodes, and certain edges are required to be *absent* to encode independence. An example of this is acyclicity of the underlying graph, which gives rise to Bayesian networks. In undirected graphs on the other hand, we look at cliques to encode the conditional independence and factorization.

[3] Some authors take the convolutional layer to only consist of the affine transform part.

§3. Unsupervised Learning

Up until now, we had a y_i for each x_i. Now, we move onto unsupervised learning, where we only have a bunch of x_i that we must classify or interpret in some meaningful way.

3.1. Dimensionality reduction

Essentially, dimensionality reduction captures some hidden relationship between the coordinates of the datapoints.

Occam's Razor says that all other things equal, the simplest solution should be picked. We have already been using this extensively, for example when picking a linear model unless it is clear that a linear model will not do. Simple solutions save on training and inference time, and slightly prevent overfitting.
Our data might lie on a simpler lower-dimensional affine space, or at least close to one. In two dimensions for instance, we might have $x_1 \approx \alpha x_2$. Dimensionality reduction tries to find the true intrinsic dimensionality of thed ata. Fewer dimensions makes the data easier to learn and we are less prone to overfitting.
This is also related to something called "VC-dimension", which we shall not study.

Given a set of m datapoints $x \in \mathbb{R}^d$, suppose we want to reduce dimensionality by projecting them to some k-dimensional space. We then have that the result projected vector is $z = U^\top x$, where U is a $d \times k$ projection matrix. How do we find the optimal U?
Let us further suppose that $U^\top U = I_k$. Given $z = U^\top x$, we may approximate x as $\tilde{x} = Uz$. There are two sources of error here: the vectors might not lie perfectly on the subspace and $UU^\top$ might not be the identity.
To make $\tilde{x}$ and x as close as possible, we could try to minimize some form of error.

3.1.1. Principal component analysis

Principal component analysis (PCA) chooses the error as $\sum \|x - \tilde{x}\|_2^2$. That is, we choose

$$U = \underset{\substack{\hat{U} \in \mathbb{R}^{d \times k}: \\ \hat{U}^\top \hat{U} = I}}{\arg\min} \sum_x \left\| x - \hat{U}\hat{U}^\top x \right\|_2^2.$$

This feels quite difficult to optimize, is there an alternate perspective?
Assume that the datapoints are centered, that is, $\mathbf{E}[x] = 0$. We can view the (optimal) projection as that which has maximum variance in the projected space. The variance of the projected data $\mathbf{Var}(U^\top x)$ is

$$\mathbf{E}\left[\left\| U^\top x - \mathbf{E}[U^\top x] \right\|_2^2 \right] = \mathbf{E}\left[\left\| U^\top x \right\|_2^2 \right]$$

That is, we want to find

$$U = \underset{\substack{\hat{U} \in \mathbb{R}^{d \times k}: \\ U^\top U = I}}{\arg\max} \mathbf{E}\left[\left\| U^\top x \right\|^2 \right].$$

It turns out that this U and the U we looked at above are the same!

Now, the knowledgeable reader might know that the above form of U can be dealt with quite easily if we use eigenvalue decomposition! We can just take the k-dimensional subspace as the span of the first k eigenvectors of the covariance matrix of the original data (where the eigenvectors are arranged in decreasing eigenvalues).
To demonstrate this, the first *principal component* is given by

$$\begin{aligned} \max_{\|u\|=1} \mathbf{E}[(u^\top x)^2] &= \max_{\|u\|=1} \frac{1}{m} \left\| u^\top X \right\|^2 \\ &= \max_{\|u\|=1} u^\top \underbrace{\left(\frac{1}{m} X X^\top \right)}_{\text{covariance matrix}} u, \end{aligned}$$

which is just the operator norm of the covariance matrix, that is, its largest eigenvector! Further, this maximum is attained when u is the corresponding eigenvector.

To summarize, if we let S be the covariance matrix of the (centered) data and perform eigenvalue decomposition on it to get $S = U\Sigma U^\top$ (where U is orthonormal and Σ is diagonal), we take U to be the matrix consisting of the first k eigenvectors (columns) of U.

3.1.2. Kernelization of PCA

A natural next question is: is there a way to 'kernelize' PCA and generalize it? Suppose we perform singular-value decomposition on X to get

$$X = ADB^\top.$$

The covariance matrix is then given by

$$mS = XX^\top = (ADB^\top)(BD^\top A^\top) = AD^2A^\top.$$

The kernel matrix we looked at earlier was just

$$\mathcal{K} = X^\top X = (BDA^\top)(ADB^\top) = BD^2B^\top.$$

The matrix of principal component variables we desire is $U = AD$. From the eigendecomposition of $\mathcal{K}$, we can get the projections XU of our data X onto these components.
That is, the top k eigenvectors of $\mathcal{K}$ gives us not the principal components, but the projections of our data itself onto these components.

3.2. Clustering

In clustering on the other hand, we aim to group together chunks of the data in some way.
Clustering is heavily used in search engines, since it attempts to give suggestions similar to our search. It should be understood that clustering is quite different from classification since we do not have pre-defined classes for the former (it is unsupervised)!

One way to cluster is to consider pairs of points, and then 'cluster upwards' in a bottom-up manner.
Top-down clustering on the other hand, chooses a couple of means initially. If the data is very different from the means, then we adjust the means slightly.
A few more questions are: how many clusters do we want? How do we consider similarity? There might also be a lot of noisy dimensions in the data.

3.2.1. Notions of measure

Let us denote by $\rho(\cdot,\cdot)$ a similarity measure (such as cosine-similarity) and by $\delta(\cdot,\cdot)$ a distance measure (such as Euclidean distance).
There are two notions of measure we care about: *intra*cluster (within the cluster) and *inter*cluster (between different clusters). We desire that the former has low δ and high ρ, and the latter has high δ and low ρ.

3.2.2. Partitioning Approaches

Suppose we want to partition our data into k clusters $\{D_1, \ldots, D_k\}$. A couple of choices are:

- Minimizing intracluster distance $\sum_i \sum_{d_1,d_2 \in D_i} \delta(d_1, d_2)$.
- Maximizing intracluster resemblance $\sum_i \sum_{d_1,d_2 \in D_i} \rho(d_1, d_2)$.
- If a 'cluster representation' $\hat{D}_i$ of D_i is available (such as the mean, for instance),

  - Minimize $\sum_i \sum_{d \in D_i} \delta(d, \hat{D}_i)$.
  - Maximize $\sum_i \sum_{d \in D_i} \rho(d, \hat{D}_i)$.
- Soft clustering, where we assign d to D_i with 'confidence' $z_{d,i}$, then finding the $z_{d,i}$ so as to minimize $\sum_i \sum_{d \in D_i} z_{d,i} \delta(d, \hat{D}_i)$ or maximize $\sum_i \sum_{d \in D_i} z_{d,i} \rho(d, \hat{D}_i)$.

Broadly, there are two ways to get partitions – *bottom-up* or *top-down* clustering.

In top-down clustering, we again have two variants: 'hard' (we assign point to clusters) or 'soft' (we assign fractional scores for each point-cluster pair).
In *hard k-means* clustering, we start off by choosing k arbitrary centroids. We assign each point to the cluster corresponding to the nearest centroid, then recompute centroids, repeating until the change becomes negligible.
In soft k-means on the other hand, we don't need to break close ties between point-cluster assignments anymore.

In bottom-up clustering, we could just choose the two points among those to be clustered that are "closest", dump them in the same cluster, and replace the two points with their mean. We then perform the algorithm again, breaking off at a suitable point.

We can in fact use bottom-up clustering in k-means, by initializing the centroids to the clusters at the point in bottom-up when there are k clusters.

Our choice of k is typically problem-driven. Data-driven approaches usually only work when we have massive amounts of data.
We could also use hypothesis testing or Bayesian estimation.

3.2.3. Kernelization of k-means

Denote by μ_i the centroid of the ith cluster and by $P_{j,i} \in \{0, 1\}$ the membership of the jth point in the ith cluster.
In k-means, we initialize all the μ_i arbitrarily to $\mu_i^{(0)}$. Our goal is to minimize

$$\sum_i \sum_j P_{j,i} \left\| \phi(x_j) - \mu_i \right\|_2^2.$$

We do so by alternately changing the μ_i (by setting $\mu_i^{(t+1)}$ as the mean of all points $\phi(x_j)$ with $P_{j,i}^{(t)} = 1$) and $P_{j,i}$ (by setting $P_{j,i}^{(t+1)}$ as 1 iff $\mu_i^{(t+1)}$ is the centroid closest to $\phi(x_j)$). That is,

$$\mu_i^{(t+1)} = \frac{\sum_j P_{j,i}^{(t)} \phi(x_j)}{\sum_j P_{j,i}^{(t)}}$$

and

$$P_{j,i}^{(t+1)} = 1 \text{ iff } \mu_i^{(t+1)} = \arg\min_k \left\| \phi(x_j) - \mu_k^{(t+1)} \right\|.$$

We repeat if any $P_{j,i}$ changes.

We claim that k-means converges in a finite number of iterations. Indeed, there are only finitely many configurations for the $(P_{j,i})$, and given a set of $P_{j,i}^{(t)}$, the choice of $\mu_i^{(t+1)}$ is the exact set of corresponding minimizers. That is, if there are ℓ clusters and m points, we will converge in at most ℓ^m steps.

So now, how do we kernelize k-means? Instead of μ_i, suppose we have $\phi(\mu_i)$, so we can talk about inner products. We then have

$$\left\| \phi(x_j) - \phi(\mu_i) \right\|_2^2 = K(x_j, x_j) + K(\mu_i, \mu_i) - 2K(x_j, \mu_i).$$

So, we want to minimize

$$\sum_i \sum_j P_{j,i} K(\mu_i, \mu_i) + K(x_j, x_j) - 2K(\mu_i, x_j).$$

$\phi(\mu_i)$ is the average of ϕs of datapoints assigned to the ith cluster.