CS 194-129 Deep Neural Networks
Spring 2018 Canny Lecture 2

1 Reading

1.1. Machine Learning Principles

In ML, we strive to find patterns in data, then make predictions using those patterns.

1.2. Ordinary Least Squares

Linear Regression (2D case of OLS)

Applications & Data: We have n data points (a_i, b_i) and want to know how the a_i's determine the b_i's.
Model: We assume that a_i and b_i are linearly related, and wish to find a model m, c such that $b_i \approx ma_i + c$.
Optimization Problem: $\min_{m,c} \|(ma + c\mathbb{1}) - b\|_2^2$
Optimization Algorithm: We can derive a closed-form solution for x^* in a number of ways.

If our samples are vectors, our model becomes a matrix $\mathbf{X}$:

Linear Regression

Applications & Data: We have n data points (a_i, b_i) and want to know how the a_i's determine the b_i's.
Model: We wish to find $\mathbf{X}$ s.t. $b_i \approx \mathbf{X}a_i$, where $b_i \in \mathbb{R}^m, a_i \in \mathbb{R}^l, \mathbf{X} \in \mathbb{R}^{m \times l}$.
Optimization Problem: After flattening $\mathbf{X}$ and turning the a_i's into diagonal block matrices, we have

$$\min_{\mathbf{x}} \|\mathbf{A}_{mn \times ml}\mathbf{x}_{ml \times 1} - \mathbf{b}_{mn \times 1}\|^2$$

Optimization Algorithm: We can derive a closed-form solution for $\mathbf{x}^*$ in a number of ways.

We can use either a vector calculus approach or a projection approach to derive $\mathbf{x}^* = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b}$.

1.3. Feature Engineering

The least-squares optimization problem $\min_{\mathbf{x}} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|$ represents the best-fit linear model. However, sometimes we would like to fit nonlinear models (e.g. ellipses and cubic functions) as well. To do this under the framework of least-squares, we can add arbitrary new features to the data. The resulting models will then be linear with respect to the augmented features, but nonlinear with respect to the original features.

Model Notation

A model has the following form: $y = \sum_{i=1}^{P} \alpha_i \Phi_i(x)$.

- y is the prediction from the model (a linear combination of the features).
- Φ_i represents the ith feature, which depends upon the raw features of the data point x.
- α_i is the fixed weight corresponding to the ith feature.

Polynomial Features

Polynomial features can be interpreted as polynomials of the raw features. They are important because under Taylor's Theorem, any sufficiently smooth function can be approximated arbitrarily closely by a polynomial of sufficient degree – thereby giving polynomials the title of **universal approximators**.

6.6. Discriminative Models

In generative models, we explicitly model the probability of the data $P(\mathbf{X}, Y)$. This approach is flexible and allows us to generate new samples using our model, but inefficient in that it requires estimation of a large number of parameters (i.e. the covariance matrices). Often, there is a large discrepancy between the number of parameters needed to specify the generative model and the number of parameters needed to actually perform classification.

Luckily, we also have **discriminative models**, where we can directly learn a decision boundary for the purpose of classification. The process of constructing a discriminative model consists mainly of the following two design choices:

1. **Representation:** how we represent the output of the model.
2. **Loss function:** how we train and penalize errors.

6.7. Least-Squares Support Vector Machine

Model and Training

As the first example of a discriminative model, we discuss the least-squares SVM. Consider the two-class version: a binary classification problem in which classes are represented by -1 and $+1$. We would like to classify a data point x by estimating parameters $\mathbf{w}$, computing $\mathbf{w}^T x$, and classifying x as $\text{sign}(\mathbf{w}^T x)$. The decision boundary can then be represented as the hyperplane $\mathbf{w}^T x = 0$.

For training, we can try to fit either the least-squares objective or a relaxed version of that objective:

$$\arg\min_{\mathbf{w}} \sum_{i=1}^{n} \|y_i - \text{sign}(\mathbf{w}^T x_i)\|^2 + \lambda\|\mathbf{w}\|^2 \quad \text{or} \quad \arg\min_{\mathbf{w}} \sum_{i=1}^{n} \|y_i - \mathbf{w}^T x_i\|^2 + \lambda\|\mathbf{w}\|^2$$

The non-relaxed version is computationally intractable.

Feature Extension

We may consider adding features that would enable us to come up with nonlinear classifiers, but still work with linear classifiers in the augmented feature space. To do so, we would re-express our objective as

$$\arg\min_{\mathbf{w}} \sum_{i=1}^{n} \|y_i - \mathbf{w}^T \phi(x_i)\|^2 + \lambda\|\mathbf{w}\|^2$$

where ϕ is a function that takes data in raw feature space and returns data in augmented feature space.

Neural Network Extension

We can also directly use a nonlinear function in the original feature space (for instance a neural net):

$$\arg\min_{\mathbf{w}} \sum_{i=1}^{n} \|y_i - g_{\mathbf{w}}(x_i)\|^2 + \lambda\|\mathbf{w}\|^2$$

where $g_{\mathbf{w}}$ is any function that can be easily optimized. We can then classify using the rule

$$\hat{y}_i = \begin{cases} 1 & g_{\mathbf{w}}(x_i) > \theta \\ -1 & g_{\mathbf{w}}(x_i) \leq \theta \end{cases}$$

where θ is some threshold. In LS-SVM, $g_{\mathbf{w}}(x_i) = \mathbf{w}^T x_i$ and $\theta = 0$. Using a neural network with nonlinearities such as $g_{\mathbf{w}}$ can produce nonlinear decision boundaries.

Multiclass Extension

We can extend this approach to the case where we have multiple classes. Suppose there are K classes. Then, if the ith observation has class k, we can represent it as $y_i = e_k$, where e_k is the kth canonical basis vector (this is known as one-hot vector encoding).

When we have multiple classes, each y_i is a K-dimensional one-hot vector, so for LS-SVM we are optimizing over a $K \times (d+1)$ weight matrix:

$$\arg\min_{\mathbf{W}} \sum_{i=1}^{n} \|y_i - \mathbf{W}x_i\|^2 + \lambda \|\mathbf{w}\|^2$$

To classify a data point x_i, we compute $\mathbf{W}x_i$ and choose the largest component j as the class.

6.8. Logistic Regression

Logistic regression is a discriminative classification technique that directly outputs a probability in the range $[0, 1]$. Suppose we have a binary classification problem where classes are represented by 0 and 1. We would like our model to output an estimate of the probability that a data point is in class 1. We can start with the linear function $\mathbf{w}^T x$ and convert it to a number between 0 and 1 by applying a sigmoid transformation $s(\mathbf{w}^T x)$, where $s = \frac{1}{1+e^{-x}}$. After learning the weights w and estimating the probability as

$$P(y_i = 1 \mid x_i) = s(\mathbf{w}^T x_i)$$

we could therefore classify x_i as

$$\hat{y}_i = \begin{cases} 1 & \text{if } s(\mathbf{w}^T x_i) \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

To train our model, we can optimize the log-loss, otherwise known as the **cross-entropy** loss:

$$L(\mathbf{w}) = -\sum_{i=1}^{n} y_i \ln \frac{1}{s(\mathbf{w}^T x_i)} + (1 - y_i) \ln \frac{1}{1 - s(\mathbf{w}^T x_i)}$$

If we define $p_i = s(\mathbf{w}^T x_i)$, we can condense this to

$$L(\mathbf{w}) = -\sum_{i=1}^{n} y_i \ln p_i + (1 - y_i) \ln (1 - p_i)$$

For each x_i, p_i represents the predicted probability that its corresponding class is 1. Since $y_i \in \{0, 1\}$, the loss corresponding to the ith data point is $-\ln p_i$ if $y_i = 1$, and $-\ln (1 - p_i)$ if $y_i = 0$.

Logistic Loss: Maximum Likelihood Approach

We can derive logistic regression from a maximum likelihood model. See *A Comprehensive Guide to Machine Learning* (Nasiriany et al.) for an excellent walkthrough of this.

6.9. Multiclass Logistic Regression

In logistic regression, we tune a weight vector $\mathbf{w} \in \mathbb{R}^{d+1}$, which leads to a posterior distribution $Q_i \sim$ Bernoulli(p_i) over the binary classes 0 and 1:

$$P(Q_i = 1) = p_i = s(\mathbf{w}^T x_i) = \frac{1}{1 + e^{-\mathbf{w}^T x_i}}$$
$$P(Q_i = 0) = 1 - p_i = 1 - s(\mathbf{w}^T x_i) = \frac{e^{-\mathbf{w}^T x_i}}{1 + e^{-\mathbf{w}^T x_i}}$$

We can generalize this concept to **multiclass logistic regression**, where there are K classes. Again we use a one-hot encoding to represent all of our labels (in order to eliminate any relative ordering in the class representations). There must now be a set of $d + 1$ weights associated with every class, amounting to a matrix $\mathbf{W} \in \mathbb{R}^{K \times (d+1)}$. For each input $x_i \in \mathbb{R}^{d+1}$, each class k is given a score $z_k = \mathbf{w}_k^T x_i$ where $\mathbf{w}_k$ is the kth row of the $\mathbf{W}$ matrix. In total there are K scores for an input x_i:

$$\begin{bmatrix} \mathbf{w}_1^T x_i & \mathbf{w}_2^T x_i & ... & \mathbf{w}_K^T x_i \end{bmatrix}$$

The higher the score for a class, the more likely logistic regression is to pick that class. We can transform all of these scores into a posterior probability distribution Q via the softmax function. If the logistic function takes the value $\mathbf{w}^T x_i$ and squashes it to a value between 0 and 1, then the **softmax function** is a multiclass generalization of the logistic function. It takes as input all K scores (formally known as **logits**) and an index j, and outputs the probability that the corresponding softmax distribution takes value j:

$$\text{softmax}(j, \{z_1, ..., z_K\}) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

Thus the logits induce a softmax distribution, which is itself a probability distribution because the entries are between 0 and 1 and add up to 1.

Loss Function

The target distribution is $P(Q_i = j) = y_i[j]$ (i.e. the entire distribution is concentrated on the label for the training example). The estimated distribution Q comes from multiclass logistic regression, and in this case is the softmax distribution:

$$P(Q_i = j) = \frac{e^{\mathbf{w}_j^T x_i}}{\sum_{k=1}^{K} e^{\mathbf{w}_k^T x_i}}$$

The loss function can be derived by minimizing the sum of the KL divergences contributed by the training examples, or by again using MLE. It is

$$L(\mathbf{W}) = -\sum_{i=1}^{n} \sum_{j=1}^{K} y_i[j] \cdot \ln P(Q_i = j)$$

2 Lecture

Recap

The second layer image (and beyond) in a convolutional network has a depth dimension in addition to the x- and y-dimensions. Each component of the depth dimension is a different learned filter.

The early layers of a neural network are often generic and/or task-independent. Thus we can reuse models for different but similar tasks, initializing with the pre-trained weights from before and then only training the output layers (which are really the task-dependent ones). Say we want to train a network on a task with

a small dataset. We can pre-train a model on a similar task with a large dataset, then repurpose that model (training just the last few layers, or slowly adapting the beginning of the network) to cheaply acquire an end-to-end model for the desired original task.

Multi-task learning: sharing model weights across tasks often improves performance on both (e.g. training for image classification and image captioning in alternation). The beginning of the model can be shared, and then there can be different task-specific tails.

Summary of Deep Nets

- Depth and complexity seem to be only limited by the amount of data needed to train without overfitting. More complex models → better accuracy.
- Inner layer representations are typically task-independent → easy to reuse models in other applications that don't have large training datasets.
- Building blocks (layer types) are standard across different toolkits and domains (image processing, speech, text, robotics), meaning transfer (/reuse of ideas) is easier.
- Real-world problems addressed by deep networks are usually "easy" to learn (typically not fraught with lots of local optima, and if they are, the local optima tend to have decent performance).

Machine Learning Background I

We start with a space of "observations" $\mathcal{X}$ and a space of "targets" or "labels" $\mathcal{Y}$. We are interested in how the observations determine the targets. Our data is a collection of pairs (x_i, y_i), with $x_i \in \mathcal{X}$ and $y_i \in \mathcal{Y}$. Given a new observation x, we should be able to predict the corresponding y.

Typically, observations are cheap, while targets are expensive and non-observable. Thus predicting those things has value.

Probabilistically, it is assumed that x and y are observations of random variables X and Y, which have a joint distribution $P(X, Y)$ (specifying the probability of any given pair). For prediction, we want at least the conditional dist. $P(Y \mid X)$ so that we can determine the dist. of targets y given an observation x.

An approximation $\hat{P}(Y \mid X)$ to the true distribution $P(Y \mid X)$ is called a **model**. The goal of machine learning is to construct models that are computable (i.e. we can compute a probability distribution from it) and which provide predictions that are close to those from the true distribution $P(Y \mid X)$.

Discriminative Models

With predictive models in ML, the main distinction is between generative and discriminative models. **Generative models** attempt to acquire a complete model of how the data is distributed – typically the joint distribution $P(X, Y)$, although at a minimum the data distribution $P(X)$. If we have that distribution, we can generate synthetic data pairs (x_i, y_i).

A **discriminative model** tries only to generate an approximation of the conditional data distribution $P(Y \mid X)$, i.e. a model of target values conditioned on the data. In this case, we can't get the data distribution; these models are capable of labeling or predicting but incapable of generating data. (In such a sense, generative models are overkill.)

Examples of generative models are Gaussian mixture models and hidden Markov models. Examples of discriminative models are support vector machines, decision trees, and neural networks.

An Example of a Generative Model: Naive Bayes

Let's examine a naive Bayes model for images. Say our image is binary (black and white), meaning our observation is a bunch of pixels that are either 0 or 1. A naive Bayes model would be specified by

$$\begin{cases} P_q(X = \text{ white} \mid Y = \text{ “woman”}) & \text{for each pixel } q, \\ P_q(X = \text{ white} \mid Y = \text{ “man”}) & \text{for each pixel } q, \\ P_q(X = \text{ white} \mid Y = \text{ “cat”}) & \text{for each pixel } q, \\ ... \end{cases}$$

This is a conditional model of the data given the label.

To generate an image for the class "woman," we could then sample each pixel q *independently* according to $P_q(X = \text{ white} \mid Y = \text{ “woman”})$. However, the independence assumption is not very realistic for images (it's saying that all the pixels are independent of each other). Thus naive Bayes is a very bad generator, and also a bad classifier!

(The assumption of naive Bayes is that data values are independent conditioned on the class label.)

Generative vs. Discriminative Tradeoffs

Generally, in order to be tractable, generative models make very strong assumptions about the data e.g. independence. However, if these assumptions are not really true, the models can have a lot of bias (and asymptotic error may be high). On the plus side, training is usually faster for generative models, and they often have better performance if the data is sparse.

In comparison, discriminative models typically make weaker assumptions about the data and involve fewer forms of bias. They might also require more training data for a modest level of accuracy, because they are less constrained in their assumptions.

As an example: a naive Bayes model approaches its asymptotic error much faster, i.e. logarithmically in the number of model parameters, than the logistic model which learns at a linear rate!

In terms of understanding the data, generative models are favored as true models of the dataset, as they illuminate the physical processes that created the data. Nevertheless, large real-world datasets are rarely well-explained by these simple models. And in terms of fitting the data, discriminative models are typically able to model more complex dataset dependencies because they make fewer assumptions. Ultimately, discriminative models have fewer limits on performance. (Deep networks are discriminative models.)

Machine Learning Background I (Continued)

Prediction Functions

We can simplify the modeling task by making a strong assumption about the model $\hat{P}(X, Y)$ – namely that $y = f(x)$, i.e. *y takes a single value* given x. This is done instead of worrying about the full conditional distribution of Y given X; in other words, we assume only a single label y instead of allowing for multiple labels.

For example, in linear regression, $y = f(x)$ is a linear function, where $y \in \mathbb{R}$ is a real value.

Loss Functions

We don't expect very often, if ever, to get a perfectly accurate prediction. The dataset might have many translations, but we're forcing only one. Typically there's also a lot of noise in the data. Therefore we need a metric of error, which we get from the loss function. A loss function characterizes the difference between a target prediction and a target data value.

Linear Regression

The total loss across all points is

$$L = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 = \sum_{i=1}^{n} (ax_i + b - y_i)^2$$

We want the optimum values of a and b, and since the loss is differentiable we simply set the derivatives dL/da and dL/db to 0 and solve for a and b. The least-squares loss is convex, which makes this work.

When training, we minimize the squared loss on some data we already have. But what we really want to do is minimize expected loss on some new data: $\mathbb{E}[(\hat{y} - y)^2]$. The expected loss is called **risk**. Meanwhile, the averaged loss (over a finite number of data points) is called **empirical risk**, i.e. risk over a specific dataset. ML approximates true risk-minimizing models with empirical risk-minimizing ones. (True risk isn't really an accessible quantity. We tend to only have observations of the data.)

If the data is unbiased, the empirical risk-minimizing model should converge to the true risk-minimizing model! However, empirical risk minimization can fail if the dataset is biased (not sampled according the true joint distribution), or if there is not enough data.

Multivariate Linear Regression

This time, $x \in \mathbb{R}^m$ and $y \in \mathbb{R}^k$ and the model is $y = Ax$ for a $k \times m$ matrix A. We assume that the last coordinate of each x vector is 1, meaning the last column of A can act as the bias vector b we had before. The empirical risk that we want to minimize is

$$L = \sum_{i=1}^{n} (Ax_i - y_i)^2 = \sum_{i=1}^{n} (x_i^T A^T - y_i^T)(Ax_i - y_i)$$

We want to optimize L with respect to the matrix A, so this time we need a gradient. We set said gradient $\nabla_A L$ to 0 and solve for A. (Note: the risk was a quadratic function of A, so the gradient is a linear function of A.)

As an aside, when we write $\nabla_A L(A)$, we mean the vector of partial derivatives

$$\nabla_A L(A) = \left[\frac{\partial L}{\partial A_{11}}, \frac{\partial L}{\partial A_{12}}, \frac{\partial L}{\partial A_{21}}, \frac{\partial L}{\partial A_{22}} \right]^T$$

where $\partial L/\partial A_{11}$ measures how quickly the loss changes with respect to change in A_{11}. When $\nabla_A L(A) = 0$, it means that all the partials are zero and thus the loss is not changing in any direction. (Hence we are at a local optimum, or at least a saddle point.)

Logistic Regression

Setup: x is an m-dimensional binary vector, i.e. $x \in \{0,1\}^m$, for example the pixels in a binary image. The target values y will also be binary, i.e. $y \in \{0,1\}$, but our prediction $\hat{y} = f(x)$ will actually be a real value in the interval $[0,1]$. We can apply a threshold $\hat{y} > y_{threshold}$ later if we want a class assignment $\{0,1\}$.

Say we *are* working with binary images, and the target class is "cat." Then the data is made up of pairs (x, y) of images and labels, where $y = 1$ means x is a cat image and $y = 0$ means x is no cat.

We can think of $\hat{y} = f(x)$, the output of our model, as being the probability that the image x is a cat. Thus the model tells us *how sure* it is that an image is a cat.

Let $w \in \mathbb{R}^m$ be a weight vector (the logistic model). Then $f(x) = 1/(1 + \exp(-w^T x))$ is the probability that the output should be "cat."

We can write this as $f(x) = g(w^T x)$, where $g(u) = 1/(1 + \exp(-u))$ is the **logistic function**. Notice that the output is

$$f(x) = \begin{cases} < 0.5 & \text{if } w^T x < 0 \\ 0.5 & \text{if } w^T x = 0 \\ > 0.5 & \text{if } w^T x > 0 \end{cases}$$

Thus, choosing a threshold of $y_{threshold} = 0.5$ gives the output of the linear classifier $w^T x > 0$. In general, choosing a threshold gives the output of the linear classifier $w^T x >$ (some v).

Loss for Logistic Regression

Assuming that $\hat{y} = f(x)$ is the *probability that x is in the target class*, we can compute the *probability of correct classification* and maximize that. The probability of correct classification (for one input) is

$$p_{correct} = \begin{cases} \hat{y} & \text{if } y = 1 \text{ (true positive probability)} \\ 1 - \hat{y} & \text{if } y = 0 \text{ (true negative probability)} \end{cases}$$

which we can turn into a simple expression as $p_{correct} = y\hat{y} + (1 - y)(1 - \hat{y})$. We could use $-p_{correct}$ as the loss for each observation, summing over all observations and minimizing. However, we'll more commonly use the negative log of the probability of a correct result:

$$-\log p_{correct} = -y \log \hat{y} - (1 - y) \log (1 - \hat{y})$$

which overall creates the **cross-entropy loss**:

$$L = -\sum_{i=1}^{n} y_i \log \hat{y}_i + (1 - y_i) \log (1 - \hat{y}_i)$$

Cross-entropy loss is the *negative log probability that every label is correct* (as opposed to simply the negative probability that every label is correct). It corresponds to the log of the product of the probabilities of each label being correct: since label errors are independent, we should multiply them to get the overall probability that everything is correct. Taking a log turns this product into a sum.

Why do we use this particular function, the logistic function, for regression? Apart from a number of nice properties, it can exactly learn a naive Bayes classifier – i.e. it is able to find weights w such that $f(x) =$ the naive Bayes probability that x is in the class.

References

[1] S. Nasiriany, G. Thomas, W. Wang, and A. Yang.
A Comprehensive Guide to Machine Learning.
`http://snasiriany.me/files/ml-book.pdf`