Lecture 7: Decision Theory and Regression

- The previous lecture formulated edge detection as statistical inference. We learned probability distributions $P(x|y)$ where x is a filtered image (we drop $f(.)$ and $I(.)$ for simplicity) and $y \in \pm 1$ (where $y = \pm 1$ denotes edge and non-edge respectively. We then performed edge detection by thresholding the log-likelihood function $\log \frac{P(x|y=1)}{P(x|y=-1)} > T$.
- This can be reformulated in terms of Bayes Decision Theory. This assumes a set of decision rules $\alpha()$ (with $\alpha(x) \in \{\pm 1\}$), a loss function $L(\alpha(x), y)$ (the penalty for making decision $\alpha(x)$ when the true decision is y), and a distribution $P(x, y)$ over the domain.
- The Bayes rule is the decision rule $\alpha^*(x)$ which minimizes the risk, or expected loss, $R(\alpha) = \sum_{x,y} L(\alpha(x), y)P(x, y)$. The Bayes risk is $R(\alpha^*)$ (caveat: almost always).
- Bayes Decision Theory is a theory for how to make decisions in presence of uncertainty (developed in WW2 for making decision from noisy radar signals, for decoding encrypted messages, and so on). It has been suggested (and disproved) as a theory for how humans make rational decisions. Advantages: it minimizes expected risk and lead to a conceptually attractive and often very useful theory. Disadvantages, expected risk is arguable (e.g., why not "worst case"?), it assumes the probabilities are known. Machine Learning (discussed later) minimizes the empirical risk.

Statistical Edge Detection and Bayes Decision Theory

- We can derive statistical edge detection from Bayes Decision Theory. It is the Bayes rule where the threshold T is determined by the prior $P(y)$ and the loss function $L(\alpha(x), y)$. (This is a standard result for binary classification, where y only takes two possible values).
- To see this, we can re-express the risk as $\sum_x P(x) \sum_y P(y|x) L(\alpha(x), y)$. Hence it is equivalent to picking the rule $\alpha(x)$ that minimizes $\sum_y P(y|x) L(\alpha(x), y)$ for each x (we have used the identity $P(x, y) = P(x)P(y|x)$).
- This can be re-expressed as $\frac{1}{P(x)} \sum_y P(x|y)P(y)L(\alpha(x), y)$ (using Bayes Theorem $P(y|x) = \frac{P(x|y)P(y)}{P(x)}$). It follows that the optimal decision is made by thresholding the log-likelihood (see background slides for the details).
- In practice it is often hard to specify a loss function (and often to determine the prior). Instead we can allow T ro vary and see how the false-positives and false-negatives vary using receiver operating characteristic (ROC) curves and other measures (False positives and false negatives are the two types of errors – data x which is incorrectly classified as $y = 1$ or $y = -1$ respectively).
- Note: for edge detection Bayes Decision Theory requires specifying a class of decision rules. For statistical edge detection this means specifying a class of filters $f(I(x))$. For each filter $f(I(x))$ we obtain a decision rule (BDT specifies the optimal decision rule for each filter after learning the probability distributions). Then we should compare the decision rules for different filters.

Regression and Bayes Decision Theory

- An alternative is to learn the distribution $P(y|x)$ directly. This is called regression. Some authors use "regression" only if y is a continuous variable. But in this course we will use "regression" for binary/multiclass as well as for continuous.
- From a Bayesian perspective $P(y|x)$ is the posterior distribution and it is derived from the likelihood function $P(x|y)$ and the prior $P(y)$ by Bayes Rule: $P(y|x) = \frac{P(x|y)P(y)}{P(x)}$. In other worlds, the likelihood and the prior are basic but the posterior is derived from them.
- But from the perspective of regression, or discriminative approaches, there is no need to have a prior or a likelihood function. After all, the decision should be based on $P(y|x)$. Learning $P(y)$ and $P(x|y)$ may be difficult/impossible and a waste of effort. Inevitably we will need to use simplifications when learning $P(y)$ and $P(x|y)$ (due to limited amounts of training data, needing to guess the types of models for the distributions, etc.).
- The situation is even worse for vision. Because it is extremely hard to put probability distributions on images since they are so high-dimensional. But, for many visual tasks, the dimension of y is much smaller. So it is practical to learn $P(y|x)$ but much harder to learn $P(x|y)$.

Linear Regression

- This was invented by Gauss (roughly in 1800) when trying to find the planetoid Ceres by predicting its position from previous estimates. This could be formulated as expressing $y = ax + b + \epsilon$, where a, b are unknown parameters and ϵ is zero mean Gaussian noise. This can be written as a probabilistic model $P(y|x : a, b) = \frac{1}{\sqrt{2\pi}\sigma} \exp\{-(y - ax - b)^2/(2\sigma^2)\}$.
- Given training data $\{(x_n, y_n) : n = 1, ..., N\}$ the variables a, b, σ can be estimated by $(a^*, b^*, \sigma^*) = \arg\max \prod_{n=1}^{N} P(y_n|x_n; a, b, \sigma)$. For estimating a, b this is equivalent to minimizing $\sum_{n=1}^{N}(y_n - ax_n - b)^2$ which gives linear equations for a^*, b^*, hence linear regression. There is also a closed form solution for σ^*.
- This can be extended in many ways, e.g. by allowing y and x to be vector-valued. It is straightforward to generalize this approach so that y is a non-linear parametric function of x (but for some non-linear functions estimating the parameters can be non-trivial).

The Perceptron (1)

- The Perceptron was developed by Rosenblatt in the 1950's and started the first wave of neural networks.
- The Perceptron specified a classification rule $y^*(x) = \arg\max_{y \in \{\pm 1\}} y(\vec{w} \cdot \vec{x} + w_0)$. I.e. $y = 1$ if $\vec{w} \cdot \vec{x} + w_0 > 0$ and $y = -1$ if $\vec{w} \cdot \vec{x} + w_0 < 0$.
- Geometrically, $\vec{w} \cdot \vec{x} + w_0 = 0$ specifies a hyperplane and points $\vec{x}$ which are above the hyperplane – i.e. $\vec{w} \cdot \vec{x} + w_0 > 0$ are classified as positive $(y = 1)$ – while points $\vec{x}$ which are above the hyperplane are classified as negative.
- This was based on a (extremely) simplified model of a neuron, where $\vec{x}$ denotes the input at different synapses, $\vec{w}$ correspond to synaptic weights, $-w_0$ is a threshold, and the neuron fires an action potential if total stimulation to the cell – given by the weighted sum $\vec{w} \cdot \vec{x}$ of the inputs – is bigger than the threshold.

The Perceptron (2)

- The Perceptron can be learnt from a set of trained data $\{(\vec{x}_n, y_n) : n = 1, ..., N\}$. Rosenblatt specified the Perceptron algorithm that was guaranteed to converge to a hyper-plane $\vec{w} \cdot \vec{x} + w_0$ which separated the positive from the negative examples, provided a separating hyper-plane existed. If the positive and negative examples cannot be separated by a hyper-plane then the algorithm will not converge. Later researchers showed that, in this case, the average of the weights $\vec{w}$ and the threshold w_0 converge to reasonable result, which separates the positive and negative examples as well as possible.
- The limitation of the Perceptron is that a separating hyperplane is only useful for a limited class of problems. To make it useful you would have to find a principled way to find features $\vec{\phi}(\vec{x})$ and apply the Perceptron to those features.

The Perceptron as logistic regression

- We can reformulate the problem as (logistic) regression by specifying a conditional probability distribution $P(y|\vec{x}) = \frac{\exp\{y\{\vec{w}\cdot\vec{x}+w_0\}\}}{\exp\{\vec{w}\cdot\vec{x}+w_0\}+\exp\{-\vec{w}\cdot\vec{x}+w_0\}}$.
- The parameters can be estimated from the training set $\{(\vec{x}_n, y_n) : n = 1, ..., N\}$ by $(\vec{w}^*, w_0^*) = \arg\min -\sum_{n=1}^{N} \log P(y_n|\vec{x}_n)$.
- This can be performed by gradient descent: $(\vec{w}^{t+1}, w_0^{t+1}) = (\vec{w}^t, w_0^t) - \zeta_t(\frac{\partial}{\partial \vec{w}}, \frac{\partial}{\partial w_0})\{-\sum_{n=1}^{N} \log P(y_n|\vec{x}_n)\}$.
- This is guaranteed to converge to the global optimum (if ζ_t is well chosen) because $-\sum_{n=1}^{N} \log P(y_n|\vec{x}_n)$ is a convex function of $\vec{w}, w_0$. There is also a discrete update rule (see lecture on variational bounding and CCCP) which converges (without needing to specify a learning rate ζ_n).
- Observe that, after learning, data points which lie above the hyper-plane $\vec{w}\cdot\vec{x} + w_0 = 0$ will have $P(y = 1|\vec{x}) > 1/2$, while points below the hyperplane will have probability $P(y = -1|\vec{x}) > 1/2$. So the hyper-plane still separates the positive and negative examples by a hyper-plane. But the separation is [soft] and specified by a number $P(y = 1|\vec{x} \in [0, 1]$ which tends to 1 for data points which are infinitely above the hyperplane and to 0 for points infinitely below.
- Note that the original Perceptron assigned output values ± 1, but these can be modified to be $\{0, 1\}$ by a simple transformation.

Multi-layer Perceptrons (1)

- "Soft" Perceptrons represent the output by a continuous function – $P(y|\vec{x})$ or $\sigma(\vec{w} \cdot \vec{x} + w_0)$ – which is *differentiable* as a function of $\vec{w}, w_0$. This makes it possible to build a network, called a multi-layer perceptron, by stacking perceptrons on top of each other so that the input to a perceptron is the output of another perceptron. The differentiability of the individual perceptrons ensures that the final output is a differentiable function of the parameters of all the neurons. This gives a natural learning rule which performs steepest descent on the input-output function.
- A simple example has output $y = f(\vec{w}, \{\vec{\Omega}^i\}, \vec{x})$, where $y = \sigma(\vec{w} \cdot \vec{z})$ with $\vec{z} = (z_1, z_2, z_3)$ and $z_i = \sigma(\vec{\Omega}^i \cdot \vec{x})$, for $i = 1, 2, 3$. Here the $\{z_i\}$ are called *hidden variables* since they cannot be observed directly.
- The weights $\vec{w}, \{\vec{\Omega}^i\}$ can be learnt from training data $\{(\vec{x}_n, y_n) : n = 1, ..., N\}$ by minimizing an energy function. $E(\vec{w}, \{\vec{\Omega}^i\}) = \sum_{i=1}^{N} E(\vec{x}_n, y_n : \vec{w}, \{\vec{\Omega}^i\})$. Here $E(.)$ is a measure of similarity between the true output y_n and the predicted output $f(\vec{w}, \{\vec{\Omega}^i\}, \vec{x}_n)$. From the regression perspective, this can be formulated as $P(y, \vec{x}) = (1/Z) \exp\{-E(\vec{x}_n, y_n : \vec{w}, \{\vec{\Omega}^i\})\}$ (strictly speaking you must require the condition that $\int dy \exp\{-E(\vec{n}, y_n : \vec{w}, \{\vec{\Omega}^i\})\}$ is independent of $\vec{w}, \{\vec{\Omega}^i\}$).

Multi-layer Perceptrons (2)

- The learning rule requires differentiating with respect to the weights. This is straightfroward to do for $\vec{w}$ (since the output fnction depends directly in $\vec{w}$). It is harder for $\{\vec{\Omega}^i\}$ but can be done by the chain rule – known as backpropagation because it propagates the error back so as to reward the weights of the neurons in the early layers of the network (this solves the *credit assignment problem*).
- Solving for the weights is a non-convex optimization problem. This can be easily realized because the network has hidden symmetries – there are several different ways to get the same input-output function by permuting the hidden units (and making changes to the output weights). This means that for any minimum of the energy function there exist several other minima with exactly the same values. Hence there cannot be any single global minimum (unless these minima are all connected by a valley).
- This symmetry means that there are a set of minima which are equally good as solutions and we only need to converge to one of them. We can perform steepest descent on the energy function – which riaks getting stuck in a local minima or a saddle point before reaching one of the "good" minima.

Multi-layer Perceptrons (3)

- An alternative is stochastic gradient descent. This selects a single training example at random, performing one iteration of steepest descent, then selecting another example, and so on. If the learning rate ζ_t satisfies certain fall-off conditions for large t then in some cases (not multi-layer perceptrons) the learning algorithm can be guaranteed to converge to the global optimum. This is known as Robbins-Monroe. Intuitively, stochastic gradient descent will prevent the algorithm from getting stuck in local minima (because these are unlikely to be at the same positions for all training data).
- In the second wave of neural networks, batch-processing meant using all your training examples at the same time and online learning meant doing stochastic gradient descent. But in the third wave datasets are so big that it is impractical to use the full batch. Instead we select sub-batches at random, which means that we do stochastic gradient descent.