Regression and Neural Networks

Alan Yuille

Feb 5   2024

Regression versus Bayes

▶ The Bayesian approach to vision involves learning a probability distribution $P(x|y)$ for generating the data x (e.g., images) conditioned on the state y of the world and a prior distribution $P(y)$. Then we estimate y by maximizing the posterior distribution $p(y|x) = p(x|y)p(y)/p(x)$. For simplicity, loss functions are ignored but it is easy to introduce them.

▶ For images this requires learning probability distributions $p(x|y)$, a generative model, for images. This is very difficult because images are complicated and very high dimensional. Typically y is much lower dimensional than x. E.g., y is binary valued for edge detection and for object classification can take 1,000 different values.

▶ So why not simply the problem by directly learning the probability distribution $p(y|x)$? Why learn a generative model at all? 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. In many situations particularly those used in the AI community to evaluate algorithms there is no need to learn a generative model and we only need $p(y|x)$.

▶ Modeling $p(y|x)$ directly is considered to be a discriminative approach. In statistics terminology this is called regression. Note: in neural networks terminology it is only regression if y is continuous-valued.

Regression versus Bayes

▶ To learn $p(y|x)$ we need to specify a family of distributions $p_\omega(y|x)$ and then estimate the best parameters ω from the training data $\mathcal{X} = \{(x_i, y_i) : i = 1, ..., N\}$. This can be done by MLE: find $\hat{\omega} = \arg\min \sum_{i=1}^{N} \{-\log p_\omega(y_i|x_i)\}$.

▶ How to specify the form of the probability distribution $p_\omega(y|x)$? The exponential models can be used, examples will be given later in the course, but there are difficulties in using them because it is not clear what type of statistics should be used.

▶ $p_\omega(y|x)$ is specified by a neural network where the ω are the weights of the neural network. This gives an alternative to neural network models. They have the advantages that they scale to large amounts of data and also can enable probability models with long range context.

▶ The original neural network models, or architectures, are convolutional neural networks (CNNs). Other important architectures, like transformers, have other important properties and advantages and will be discussed later.

▶ We will start by discussion the basic ideas of regression and then proceed to convolutional neural networks.

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 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 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 risks getting stuck in a local minima or a saddle point before reaching one of the "good" minima.

- 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.

Deep Networks and the Chain Rule

▶ Deep Networks were first shown to be successful for object classification. Their structure was well suited for this task and is motivated by conjectures about the structure of the human/mamalian visual system, See the slides aboyt AlexNet. This is "course" task where the aim is to classify an image as a single object. This is naturally suited to a hierarchical structure which learns features which are invariant to irrelevant details about the object.

▶ But deep networks could rapidly be adapted to fine-detail tasks like estimating edges in image or for semantic segmentation. Both tasks require specify results per-pixel (hence fine-detail). For edge detection this is addressed by having loss functions at different levels of the network.

▶ Nueral networks are learnt using the chain rule. The output of a deep network can be expressed in terms of compositions of elementary functions, where each elementary function depends on weights. This compositional structure can be exploited to calculate the derivatives of all the weights in the network, hence enabling an efficient learning algorithm. We illustrate this for a network with two layers of hidden variables (but the same procedure can be applied to networks with any numbers of layers).

Deep Networks and the Chain Rule

- Formally, $\vec{O} = O(\vec{W}_3, \vec{H}_2)$, $\vec{H}_2 = H_2(\vec{W}_2, \vec{H}_1)$, $\vec{H}_1 = H_1(\vec{W}_1, \vec{I})$. A special case is $\vec{H}_2 = \mathrm{ReLu}(\vec{W}_2 \cdot \vec{H}_1)$ (etc). The input-output can be expressed as $\vec{O} = O(\vec{W}_3, \vec{W}_2, \vec{W}_1, \vec{I})$.

- The loss function takes the form $L(O(\vec{W}_3, \vec{W}_2, \vec{W}_1, \vec{I}), T)$, where T is the groundtruth.

- The learning update rule – to estimate the ML estimate – requires computing the partial derivatives of the weights. This can be done efficiently by exploiting the chain rule of differentiation. This is known as the backpropagation algorithm (because errors are propagated backwards done the hierarchy).

Backpropagation: The Chain Rule of Differentiation

- For $\vec{W}_3$ the derivative is simple: $\frac{\partial L}{\partial \vec{W}_3} = \frac{\partial L}{\partial \vec{W}_2}, \frac{\partial L}{\partial \vec{W}_1}$. (because the weights $\vec{W}_3$ directly affect the output O so it easy to assign them credit).

- For $\vec{W}_2$, the derivative is given by $\frac{\partial L}{\partial \vec{W}_2} = \frac{\partial L}{\partial \vec{H}_2} \frac{\partial \vec{H}_2}{\partial W_2}$ with $\frac{\partial L}{\partial \vec{H}_2} = \frac{\partial L}{\partial O} \frac{\partial O}{\partial H_2}$. (This requires backpropagating information back to the earlier layers so to give credit to weights that do not directly affect the output).

- For $\vec{W}_1$, the derivative is given by $\frac{\partial L}{\partial \vec{W}_1} = \frac{\partial L}{\partial \vec{H}_1} \frac{\partial \vec{H}_1}{\partial W_1}$ with $\frac{\partial L}{\partial \vec{H}_1} = \frac{\partial L}{\partial \vec{H}_2} \frac{\partial O}{\partial H_1}$. (This requires backpropagating information back to the earlier layers so to give credit to weights that do not directly affect the output).

- These expressions yield the derivatives of the loss function with respect to the weights in terms of the derivatives of the compositional functions $\vec{O} = O(\vec{W}_3, \vec{H}_2)$, $\vec{H}_2 = H_2(\vec{W}_2, \vec{H}_1)$, $\vec{H}_1 = H_1(\vec{W}_1, \vec{I})$.

Edge Detection: HED

- ▶ The HED edge detector exploits the fact that edges occur at different scales in the image. Intuitively, we could detect edges at different scales by convolving the image with a Gaussian at different scales (scale corresponds to the standard deviation of the Gaussian) and taking the derivative. Small edges (i.e., small changes in intensity across the edge) will be blurred out by Gaussians with large variances and so will only give cues for edges at the smallest scales (e.g., texture edges, which HED does not want to detect). By contrast, large edges will remain as we blur the image with larger Gaussians. There will be information about edges at all scales (as exploited by statistical edge detectors).

- ▶ HED captures this intuition by having side layers which give evidence for edges at different levels of the hierarchy. These side layers have access to the ground truth and have a corresponding loss function. The side layers are combined together to yield the final decision which combines information from all scales. This enables the lower-layers, which have higher spatial resolution, to give accurate estimates for the local positions of edges while the higher-layers validate that there is an edge (HED wants to detect boundary edges which are large and which have evidence for them at most scales of the network).

- ▶ In summary, HED has a standard deep network architecture with side layers and a sum of loss functions for detecting edges at different scales.

- ▶ Note that, like statistical edge detection, the loss functions must stromgly penalize false negatives.