IIST, Thiruvananthapuram | AVD624 - Computer Vision

Lecture Notes: Weeks - 12 to 15

12/04/2022 - 03/05/2022

Prof. Dr. Deepak Mishra | *Summarised by: Vatsalya Gupta*

This is a weekly summary of the lectures given by *Dr. Deepak Mishra*, Professor and Head of Department of Avionics, for the course on *Computer Vision* conducted during the even semester 2021-22.

1 Epipolar Geometry

Epipolar geometry is the geometry of stereo vision. When two cameras view a 3D scene from two distinct positions, there are a number of geometric relations between the 3D points and their projections onto the 2D images that lead to constraints between the image points. We want to avoid search over entire image. Epipolar constraint reduces search to a single line.

1.1 Essential Matrix

The Essential Matrix is a 3×3 matrix that encodes epipolar geometry. Given a point in one image, multiplying by the essential matrix will tell us the epipolar line in the second view. If the point *x* is on the epipolar line *l* then $x^T l = 0$. So if $x'^T l = 0$ and $Ex = l'$ then $x^T Ex = 0$. Using the rigid motion and coplanarity constraints, $E = R[t_\times]$. The epipoles are $e'^T E = 0$ and $Ee = 0$. The fundamental matrix is a generalization of the essential matrix, where the assumption of Identity matrices is removed, $F = K'^{-T}[t_\times]RK^{-1}$, i.e. depends on both intrinsic and extrinsic parameters. The epipole is in the *right null space* of *F*.

Essential matrix and homography matrix are both 3×3 matrices but Essential matrix maps a **point** to a **line**, while Homography maps a **point** to a **point**.

1.2 8-point Algorithm

The 8-point algorithm for finding the fundamental matrix is as follows.

1. Normalise *M* matched image points. Construct the $M \times 9$ matrix **A**.
2. Find SVD of **A**. Entries of **F** are the elements of column of **V** corresponding to least singular value.
3. Enforce rank 2 constraint on **F** using SVD. Un-normalise **F**.

2 Classification

Given a feature representation for images, we want to learn a model for distinguishing features from different classes. A primitive way is to assign input vector to one of two or more classes. Input space is divided into decision regions separated by decision boundaries.

Training: Given a training set of labeled examples $(x_1, y_1), ..., (x_N, y_N)$, estimate the prediction function f by minimizing the prediction error on the training set.

Testing: apply f to a never before seen test example x and output the predicted value $y = f(x)$.

2.1 K-Nearest Neighbor (KNN) Classifier

It is a *non-parametric pattern* classification approach. Consider a two class problem where each sample consists of two measurements (x, y). For a given query point q, assign the class of the nearest neighbor. Compute the k nearest neighbors and assign the class by majority vote. For a new point, find the k closest points from training data. Labels of the k points 'vote' to classify. This method is simple yet effective. But the search is expensive (can be sped-up), storage requirements can be an issue and there can be difficulties with high-dimensional data.

2.2 Naive Bayes' Classifier

A naive Bayes classifier assumes all features are conditionally independent. The naive Bayes classifier is solving this optimisation, MAP (maximum a posteriori) estimate.

$$\hat{z} = \arg\max_{z \in Z} p(z|X)$$

To compute the MAP estimate, Given a set of known parameters $(p(z), p(x|z))$ and observations $(\{x_1, x_2, ..., x_N\})$, compute which z has the largest probability.

$$\hat{z} = \arg\max_{z \in Z} p(z) \prod_n p(x_n|z)$$

2.3 Support Vector Machines (SVMs)

It is a *discriminative classifier* based on optimal separating line (for 2D case). The objective is to maximise the margin between the positive and negative training examples, i.e. $\min_w ||w||$, subject to $y_i(w \cdot x_i + b) \geq 1$ for $i = 1, ..., N$. This is a convex quadratic programming (QP) problem with a unique solution. The solution is $\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i$ and $b = y_i - \mathbf{w} \cdot \mathbf{x}_i$. The classification function can be considered as follows.

$$f(x) = \text{sign}(\mathbf{w} \cdot \mathbf{x} + b) = \text{sign}(\sum_i \alpha_i y_i \mathbf{x}_i \cdot \mathbf{x} + b)$$

If $f(x) < 0$, classify as negative, otherwise classify as positive.

Non-linear SVMs: The original input space can be mapped to some higher-dimensional feature space where the training set is separable.

Kernel trick: The linear classifier relies on dot product between vectors $K(x_i, x_j) = x_i \cdot x_j$ If every data point is mapped into high-dimensional space via some transformation $\Phi : x_i \rightarrow \phi(x_i)$, the dot product becomes: $K(x_i, x_j) = \phi(x_i) \cdot \phi(x_j)$. Some examples of kernel functions are linear, polynomials of degree up to d, Gaussian RBF and histogram intersection.

Soft-margin SVMs: $\min_{\mathbf{w}, \boldsymbol{\xi}} ||\mathbf{w}||^2 + C \sum_i \xi_i$ subject to $y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 - \xi_i$ for $i = 1, ..., N$.

2.4 Training and Testing

Underfitting: Model is too 'simple' to represent all the relevant class characteristics. So, it has high bias and low variance as well as high training error and high test error.

Overfitting: Model is too 'complex' and fits irrelevant characteristics (noise) in the data. So, it has low bias and high variance as well as low training error and high test error.

It is better to have smart features and simple classifiers than simple features and smart classifiers. We can use increasingly powerful classifiers with more training data. As an additional technique for reducing variance, try regularising the parameters (penalise large coefficient values).

3 Convolutional Neural Networks (CNNs)

Convolutional neural networks are a type of neural network with layers that perform special operations. The network learns a feature hierarchy all the way from pixels to classifier. Each layer extracts features from the output of previous layer. All the layers are trained jointly. Some of the **activation** functions used for this task are sigmoid, tanh, ReLU, leaky ReLU and maxout ELU.

Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$ **tanh:** $\tanh(x)$ **ReLU:** $\max(0, x)$ **Leaky ReLU:** $\max(0.1x, x)$

Maxout ELU: $\max(w_1^T x + b_1, w_2^T x + b)$, $f(x) = x$ if $x > 0$ else $\alpha(e^x - 1)$

The goal is to iteratively find such a set of weights that allow the activations/outputs to match the desired output. We want to minimise a **loss** function. The loss function is a function of the weights in the network, for example hinge loss, softmax (cross-entropy) and triplet loss.

Hinge loss: $L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$

Softmax (cross-entropy): $L_i = -\log P(Y = y_i | X = x_i) = -\log\left(\frac{e^s k}{\sum_j e^{s_j}}\right)$, where $s = f(x_i; W)$

Triplet loss: $L_i = \sum_i^N \left[||f(x_i^a) - f(x_i^p)||_2^2 - ||f(x_i^a) - f(x_i^n)||_2^2 + \alpha\right]_+$, *a* anchor, *p* positive, *n* negative

We want to change the weights in such a way that makes the loss decrease as fast as possible. This can be done using **gradient descent**. Another way is to use **mini-batch** gradient descent, i.e. only use *some* of the data for each gradient update, instead of all training examples, and cycle through multiple times. Each time we have cycled through all of the training examples once is called an 'epoch. To update weights at all layers, we can use backpropagation of error from higher layers to lower layers. This allows faster training (e.g. on GPUs) and parallelisation.

■ ■ ■