Modeling Video Dynamics with Deep Dynencoder

Xing Yan, Hong Chang, Shiguang Shan, and Xilin Chen

Key Lab of Intelligent Information Processing of Chinese Academy of Sciences (CAS),
Institute of Computing Technology, CAS, Beijing, 100190, China
{xing.yan,hong.chang,shiguang.shan,xilin.chen}@vipl.ict.ac.cn

Abstract. Videos always exhibit various pattern motions, which can be modeled according to dynamics between adjacent frames. Previous methods based on linear dynamic system can model dynamic textures but have limited capacity of representing sophisticated nonlinear dynamics. Inspired by the nonlinear expression power of deep autoencoders, we propose a novel model named dynencoder which has an autoencoder at the bottom and a variant of it at the top (named as dynpredictor). It generates hidden states from raw pixel inputs via the autoencoder and then encodes the dynamic of state transition over time via the dynpredictor. Deep dynencoder can be constructed by proper stacking strategy and trained by layer-wise pre-training and joint fine-tuning. Experiments verify that our model can describe sophisticated video dynamics and synthesize endless video texture sequences with high visual quality. We also design classification and clustering methods based on our model and demonstrate the efficacy of them on traffic scene classification and motion segmentation. ...

Keywords: Video Dynamics, Deep Model, Autoencoder, Time Series, Dynamic Textures.

1 Introduction

Video dynamics, representing as various object motions, widely exist in real-world video data, e.g., regular rigid motion like moving escalator and windmill, chaotic motion like smoke and water waves, sophisticated motion caused by camera panning or zooming, etc. Modeling video dynamics is challenging but very important for subsequent vision tasks, such as dynamic texture (DT) synthesis, video classification, motion segmentation and so on.

One popular way to address this challenge is via linear dynamic system (LDS), a probabilistic generative model defined over space and time, which estimates hidden states using Principal Component Analysis (PCA) and describes their trajectory as time evolves. LDS can model videos with smooth motions and has been applied to computer vision tasks such as DT synthesis [9], video segmentation [10][7] and video classification [18][5][17]. However, LDS has obvious disadvantages due to its simplistic linearity so that many variants have been

D. Fleet et al. (Eds.): ECCV 2014, Part IV, LNCS 8692, pp. 215–230, 2014.

proposed. To overcome visible decay and discontinuities of sequences synthesized by LDS, closed-loop LDS (CLDS) [28] is proposed. However, the model is still linear and may fail to model some discontinuous rigid motions. Kernel-DT model is proposed in [6], where the nonlinear observation function is learned using kernel-PCA while the hidden states change linearly. kernel-DT can capture more motions such as camera panning, but may have the same weaknesses as LDS due to the linear state transition. By treating nonlinear DT as temporally multiple linear DTs, piecewise linear dynamic systems (PLDS) [27] automatically divides the video sequence and models each segment with an LDS. It can tackle videos with camera switching. But piecewise linearity is too different from nonlinearity. In short, above LDS-based approaches try to model video dynamics, but do have weaknesses more or less.

Other ways to model video dynamics for texture synthesis is via nonparametric methods. Masiero and Chiuso [15] propose to estimate the state distribution and generate samples from the implicit model. Liu et al. [13] assume the spectral parameters of image sequences lie on a low-dimensional manifold and use a mixture of linear subspaces to model it. In [2], the observation data are embedded into a higher dimensional phase space, where the predictions computed through kernel regression. Unlike LDS-based models, these nonparametric methods have no application reported such as video classification.

Once each video has been modeled by a dynamic system, we can not only synthesize dynamic textures, but also perform video classification by defining distance or kernel between pair of models. Some researchers [18][6] use the Martin distance [14] between two LDSs (or the kernel version) and achieve good performance in DT recognition. They also use probabilistic kernel based on the Kullback-Leibler divergence [5] for traffic scene classification. With proper distance definition, video clustering and segmentation can be done as well [10].

Deep learning methods have recently been proposed with notable successes in some areas including computer vision, beating the state-of-the-art [3]. Deep models, such as Deep Belief Network and stacked autoencoders, have much more expressive power than traditional shallow models and can be efficiently trained with layer-wise pre-training and fine-tuning [3][11]. Stacked autoencoders have been successfully used as a robust feature extractor [24]. Besides, they can also be used to model complex relationships between variables due to the composition of several levels of nonlinearity [24]. For example, Xie et al. [26] model relationship between noisy and clean images using stacked denoising autoencoders. Their method achieves state-of-the-art performance in image denoising and inpainting tasks. However, deep autoencoders are rarely used to model time series data, although there have been some works on using variants of Restricted Boltzmann Machine (RBM) for specific time series data, i.e., human motion [23][22]. Some other deep models address video data with convolutional learning of spatio-temporal features [21][12].

In this paper, we propose a novel hierarchical model named deep dynencoder to model video dynamics, i.e., the relationship between all pairs of adjacent frames in an image sequence. We stack an ordinary autoencoder and a variant

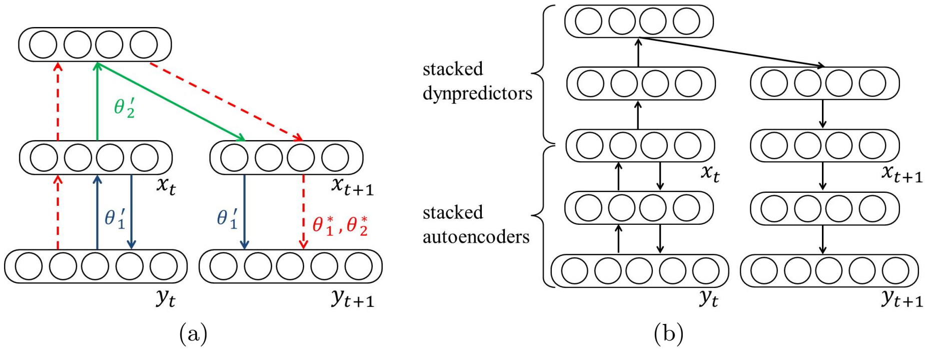


Fig. 1. Model architecture: (a) the basic dynencoder (the parameters θ_1' and θ_2' are obtained from layer-wise pre-training, while the red dashed arrows represent the fine-tuning process of the model parameters); (b) deep dynencoder

of it to form a basic dynencoder, which can be further deepened through proper stacking strategy. After layer-wise pre-training and fine-tuning, the model can capture various video dynamics including regular rigid motions, chaotic motions and camera motions such as panning or zooming. Given a DT sequence for training and an initial frame, our model can synthesize an endless DT sequence with impressive high visual quality. Similar with [6][18][10], we define a distance measure between two videos based on our model and apply it to traffic scene classification and motion segmentation. The performances of classification and segmentation are close to or higher than the state-of-the-art.

To summarize, our contributions are threefold. First, we propose deep dynencoder for modeling video dynamics and effective algorithm for training. Second, our method performs outstandingly in DT synthesis, showing its ability of describing sophisticated video dynamics. Third, we associate our model with a distance measure definition and demonstrate its usefulness on vision tasks including classification and segmentation.

2 Model Description

In this section, we first give the formulation and training method of the basic dynencoder for video modeling. Then we introduce how to construct the deep dynencoder with a stacking strategy. Notice that a video sequence is denoted by $Y = \{y_t\}_{t=1}^{T}$, where $y_t \in \mathbb{R}^m$ contains raw pixel values of the frame at time t.

2.1 Dynencoder

Dynencoder is a three-layer network constructed by stacking a variant of autoencoder on top of an ordinary one, as shown in Figure 1(a). For a video sequence, the autoencoder at the bottom generates hidden states (or compresses the input) from raw frames, while its variant (we name it *dynpredictor*) at the top encodes the dynamic of the hidden states and predicts new states.

Step 1. Autoencoder Pre-training. Let $\{y_t\}$ be the training data, the ordinary autoencoder is defined as:

$$h_{\theta_1}(y_t) = s(W_1^1 y_t + b_1^1), \tag{1}$$

$$f_{\theta_1}(h_{\theta_1}(y_t)) = s(W_2^1 h_{\theta_1}(y_t) + b_2^1). \tag{2}$$

Here $\theta_1 = \{W_1^1, b_1^1, W_2^1, b_2^1\}$ are model parameters consisting of connection weights and biases. $s(x) = (1 + \exp(-x))^{-1}$ is the sigmoid activation function. $h_{\theta_1}(y_t)$ is the hidden layer activation and $f_{\theta_1}(h_{\theta_1}(y_t))$ is the output layer activation. The parameters θ_1 are optimized by minimizing the average reconstruction error:

$$\theta_1' = \arg\min_{\theta_1} \frac{1}{T} \sum_{t=1}^{T} \|f_{\theta_1}(h_{\theta_1}(y_t)) - y_t\|_2^2. \tag{3}$$

It reveals that the hidden layer activation $h_{\theta_1}(y_t)$ can be seen as a compressed representation of y_t, as well as the hidden state of y_t.

Step 2. Dynpredictor Pre-training. After training the autoencoder, we can take hidden layer activations $\{h_{\theta_1}(y_t)\}$ as the input of a dynpredictor and train it. This means that the dynpredictor is put on the top of the autoencoder, as shown in Figure 1(a). Let $x_t = h_{\theta_1}(y_t)$, the layer activations of dynpredictor are computed similarly:

$$h_{\theta_2}(x_t) = s(W_1^2 x_t + b_1^2), \tag{4}$$

$$f_{\theta_2}(h_{\theta_2}(x_t)) = s(W_2^2 h_{\theta_2}(x_t) + b_2^2). \tag{5}$$

Here $\theta_2 = \{W_1^2, b_1^2, W_2^2, b_2^2\}$ are model parameters consisting of connection weights and biases. $h_{\theta_2}(x_t)$ and $f_{\theta_2}(h_{\theta_2}(x_t))$ are the hidden and output layer activations respectively. Note that the dynpredictor is different from autoencoder in that we optimize the parameters θ_2 by minimizing the following error:

$$\theta_2' = \arg\min_{\theta_2} \frac{1}{T-1} \sum_{t=1}^{T-1} \|f_{\theta_2}(h_{\theta_2}(x_t)) - x_{t+1}\|_2^2. \tag{6}$$

That is, dynpredictor is not used for reconstructing the input, but predicting the input at next time step.

After that, the output layer activations $\{f_{\theta_2}(h_{\theta_2}(x_t))\}$ are mapped back to the output layer of the autoencoder below. Therefore, when given y_t, the calculation process of dynencoder is:

$$F(y_t) = f_{\theta_1}(f_{\theta_2}(h_{\theta_2}(h_{\theta_1}(y_t)))). \tag{7}$$

Step 3. Joint Fine-tunning. The above two steps can be considered as the layer-wise pre-training strategy in deep learning. After pre-training with $\{y_t\}$, it is obvious that the dynencoder with parameters $\{\theta_1', \theta_2'\}$ tends to output video frame of the next time step when given the current one, i.e., $F(y_t) \approx y_{t+1}$. Similar to deep learning, fine-tuning can be performed to make the prediction more accurate. With initialized parameters $\{\theta_1', \theta_2'\}$ and target sequence $\{y_{t+1}\}$, we fine-tune the parameters as:

$$\{\theta_1^\star, \theta_2^\star\} = \arg \min_{\{\theta_1, \theta_2\}} \frac{1}{T-1} \sum_{t=1}^{T-1} \|F(y_t) - y_{t+1}\|_2^2. \tag{8}$$

After joint fine-tuning, the dynencoder represents the mapping $y_t \rightarrow y_{t+1}$ more precisely. Actually, it can model the underlying relationship between temporally adjacent two inputs and encode the video dynamic into parameters. More specifically, the dynencoder maps the observation data to hidden states (and vice versa) via autoencoder and describes the dynamic of hidden states via dynpredictor. The mapping line is: $y_t \rightarrow x_t \rightarrow x_{t+1} \rightarrow y_{t+1}$. So given an initial frame, a trained dynencoder can generate an endless video of the same dynamic.

2.2 Deep Dynencoder

Like deep autoencoders, we can also stack many building blocks to get deep dynencoder. In the architecture of dynencoder, there are two kinds of building blocks: autoencoder and dynpredictor. So, the model can be deepened by adding more autoencoders at the bottom and more dynpredictors on the top, as shown in Figure 1(b). If we have stacked M autoencoders with parameters $\{\theta_i\}_{i=1}^{M}$ and N dynpredictors with parameters $\{\theta_i\}_{i=M+1}^{M+N}$ from bottom to top, the resulting deep dynencoder maps input y_t to output $F(y_t)$ by:

$$F(y_t) = f_{\theta_1} \circ \cdots \circ f_{\theta_{M+N}} \circ h_{\theta_{M+N}} \circ \cdots \circ h_{\theta_1}(y_t), \tag{9}$$

where $\circ$ represents function composition. In other words, the layers are activated bottom-top-bottom.

Given $\{y_t\}$, the $M+N$ building blocks are pre-trained one by one from bottom to top in the same way as stacked autoencoders. For example, $\{h_{\theta_1}(y_t)\}$ are used to train the second autoencoder, and so on. The hidden layer activations of the top autoencoder $\{h_{\theta_M} \circ \cdots \circ h_{\theta_1}(y_t)\}$, i.e., the hidden states $\{x_t\}$, are used to train the first dynpredictor . Then $\{h_{\theta_{M+1}}(x_t)\}$ are used to train the second dynpredictor, and so on. Finally, we get the pre-training parameters $\{\theta_1', \ldots, \theta_{M+N}'\}$ and use them as initializations in the following fine-tuning step:

$$\{\theta_i^\star\} = \arg \min_{\{\theta_i\}} \frac{1}{T-1} \sum_{t=1}^{T-1} \|F(y_t) - y_{t+1}\|_2^2. \tag{10}$$

The deep dynencoder can model more complex dynamics $y_t \rightarrow y_{t+1}$.

It is worth noting that the training strategy need to be modified slightly when stacking two dynpredictors. At the pre-training step, unlike autoencoder aiming at reconstructing input, dynpredictor tries to predict the hidden states of next time $\{x_{t+1}\}$. If we train the second dynpredictor using the same method as the first one and stack them, the network will tend to output x_{t+2} when inputting x_t. After pre-training, the deep dynencoder tends to output y_{t+2} when inputting y_t, which is not what we want. A simple solution is stacking only one dynpredictor on top of several autoencoders. A better solution of stacking multiple dynpredictors is interpolating between two adjacent inputs to form a new data set $\{x_t, x_{t+\frac{1}{2}}\}_{t=1}^{T}$. Then, two stacked dynpredictors tends to represent the mapping $x_t \to x_{t+1}$, and $x_{t+\frac{1}{2}} \to x_{t+\frac{3}{2}}$ simultaneously. Since interpolating between $\{x_t\}$ directly may lack some sense, we can adopt video frame interpolation first to get $\{y_{t+\frac{1}{2}}\}$ and treat $\{y_t, y_{t+\frac{1}{2}}\}$ as training data of bottom autoencoders to get $\{x_t, x_{t+\frac{1}{2}}\}$. After pre-training, $\{y_{t+\frac{1}{2}}\}$ and $\{x_{t+\frac{1}{2}}\}$ are abandoned to eliminate their influence, i.e., we still perform the optimization according to Equation (10). This strategy can be adopted when stacking more dynpredictors. In fact, the basic dynencoder including one autoencoder and one dynpredictor can model complex videos well. We will verify this in the experiment section.

2.3 Discussion

Dynencoder preserves some similarities with LDS. The formulation of LDS contains a couple of equations representing the observation and state transition processes, which are just the objectives of autoencoder and dynpredictor respectively, in our model. But they are significantly different as dynencoder is completely nonlinear while LDS is linear. Even some variants of LDS remain linear to some extent, e.g., in kernel-DT, hidden states change linearly. With the help of nonlinear representation capability, dynencoder can model video dynamics better than LDS-based models do.

It is clear that computational complexity of training our model is the same as training stacked autoencoders, which mainly depends on the optimization algorithm adopted in each optimization problem described above. Two popular algorithms can be adopted: L-BFGS or nonlinear conjugate gradient method. The latter is faster in each iteration and the L-BFGS needs less iterations. It is well known that the computational complexity of training a deep model is high when the input dimension is high (all pixel intensities of an image). Training with multicore computing or GPU acceleration will save a lot of time.

One may doubt why the regularization term that tends to decrease the magnitude of the weights is absent in our formulations. Actually, regularization term is to prevent overfitting and improve predictive performance. In some cases (e.g., DT synthesis), the model may be required to reconstruct the original sequence with small error. The regularization term may cause opposite effect so that it is absent in this paper. Furthermore, we can set the dimensions of hidden layers of dynencoder to be small to prevent overfitting.

Our method has potential to model any time series data beyond computer vision field, e.g., large amounts of financial data or medical time series. It may need some modifications to match different data types and tasks, e.g., it might be necessary to add regularization term in some cases. We do not give much discussion here because our focus in this paper is modeling video dynamics.

3 Vision Applications of Deep Dynencoder

We give approaches to DT synthesis, video classification and video segmentation using our proposed deep dynencoder. For classification and segmentation, a distance definition between two videos is given and the effectiveness of it is verified in the experiment section.

3.1 DT Synthesis

DT synthesis is a classical problem in computer vision and computer graphics. Our model can naturally be used to synthesize DT of any length. Given a training sequence $\{y_t\}_{t=1}^{T}$, we learn deep dynencoder with the training strategy described above and get the output mapping F of the model according to Equation (9). Given an initial frame y'_1, an endless sequence $\{y'_t\}$ can be obtained iteratively:

$$y'_{t+1} = F(y'_t). \tag{11}$$

The synthesis process is real-time as many previous methods are.

3.2 Video Classification

In order to do video classification, we need to define a distance measure, which many classification algorithms, such as kNN and SVM, rely on. In LDS-based classification methods [18][6], Martin distance is adopted which is originally a metric for autoregressive moving average process (ARMA) [14]. Unfortunately, it seems that no distance has been defined between two neural networks.

Wolf et al. [25] introduce an one-shot similarity (OSS) kernel which compares two vectors representing images. They train two classifiers for each vector and apply them on the opposite vector. Two classification confidence scores are obtained and averaged to get the similarity. Inspired by OSS switching classifiers, we develop a distance definition between two videos $Y_a = \{y_t^a\}_{t=1}^{T_a}$ and $Y_b = \{y_t^b\}_{t=1}^{T_b}$ by switching models. Treating Y_a and Y_b as training data respectively, we learn two deep dynencoders M_a and M_b with output mapping F_a and F_b. Then we switch the models and measure how M_a fit Y_b well and M_b fit Y_a well. The average fitting errors are adopted as:

$$d_a^2 = \frac{1}{T_a - 1} \sum_{t=1}^{T_a - 1} \|F_b(y_t^a) - y_{t+1}^a\|_2^2, \tag{12}$$

$$d_b^2 = \frac{1}{T_b - 1} \sum_{t=1}^{T_b - 1} \|F_a(y_t^b) - y_{t+1}^b\|_2^2. \tag{13}$$

The distance measure between Y_a and Y_b is then defined as:

$$d(Y_a, Y_b) = \frac{d_a + d_b}{2}. \tag{14}$$

Because video dynamics are represented by our models, this distance can reflect difference of two videos in dynamic nature. After calculating all distances between videos in a video database, any classification algorithms that rely on distance measure, e.g., kernel-SVM, can be adopted for classifying videos. Although the distance defined above is not a distance metric and the kernel using it cannot be proved positive definite, this does not affect its usefulness for classifying videos. Experiment in video classification in the next section will show the efficacy of our model together with the distance.

3.3 Video Segmentation

Distance definition also makes video clustering feasible and video segmentation can be done through the clustering of spatiotemporal patches, as [10] and [7] do. Because our model represents video dynamics or motions well, our method can segment a video into regions of homogeneous motion, i.e., can do motion segmentation. Specifically, a collection of overlapped video tubes of dimensions $p \times p \times q$ are extracted from each location in the video. Then we cluster them using algorithm like K-medoids with distance defined above. If the segmentation boundaries need not change over time, q is equal to the length of the video sequence. The segmentation result is then obtained using a voting scheme: each pixel in a tube receives a vote for its clustering result. The pixel is then assigned to the cluster with the most votes.

4 Experiments

We evaluate our method on the tasks of DT synthesis, traffic scene classification and motion segmentation, and compare its performance against popular methods in each task. In all the experiments, the raw grey scale pixel values of each video are used to train a deep dynencoder. Each frame is reshaped to a vector. Dimensions of hidden layers of dynencoder are set to be small factors times the dimension of the input, e.g., 0.1, 0.05 etc. Too high dimensions may cause overfitting while too low ones may decrease the representation ability of the model. It is easy to choose suitable dimensions. All the optimizations in our training are solved with non-linear conjugate gradient algorithm.

In the tasks related to classification, we train deep dynencoders for all videos in the video database $\mathcal{Y}$ and calculate distances between them. Then the distances are used to train a kernel-SVM classifier with an RBF-kernel. The bandwidth parameter σ^2 is estimated as:

$$\sigma^2 = \frac{1}{2}\text{median}\{d^2(Y_a, Y_b)\}_{Y_a, Y_b \in \mathcal{Y}}, \tag{15}$$

where Y_a and Y_b are two videos in the database $\mathcal{Y}$. Then we get the kernel $k(Y_a, Y_b) = \exp(-\frac{1}{2\sigma^2} d^2(Y_a, Y_b))$. We train kernel-SVM using LIBSVM [8].

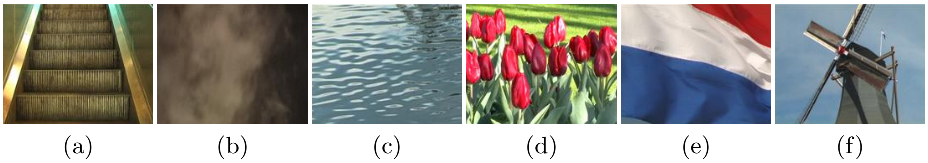


Fig. 2. Six examples in the DynTex database: (a) Escalator, (b) Steam, (c) River, (d) Flowers, (e) Flag (f) Windmill

Table 1. Comparison between mean squared errors given by five synthesis methods

Sequence\Method	LDS	Kernel-DT	Stable LDS	CLDS	Basic Dynencoder
Escalator	5.83	5.47	7.54	**4.87**	13.95
Steam	431.91	118.33	91.59	95.41	**89.88**
River	90.37	92.45	81.60	126.86	**35.65**

4.1 DT Synthesis

In this experiment, we evaluate the performance of our method in DT synthesis. We use DT sequences from the DynTex database [16] as training data respectively. Figure 2 shows some examples such as Escalator, Steam and River. Original sequences all have 250 frames of size 720×576 and we down sample them to 120×96 grey scale images. For each of these sequences, we train a dynencoder using 100 or 120 frames and synthesize a new DT sequence which is 10 times as long as the original one. Both qualitative images and quantitative measurements between synthesized and original videos are provided for comparisons.

We first compare basic dynencoder to some well-known LDS-based methods: LDS, kernel-DT, and CLDS, on some sequences in DynTex database. We implement these comparative methods and select parameters of each to produce best results. Qualitative results over two challenging sequences Steam and River are shown in Figure 3. One can see that LDS and kernel-DT produce unsatisfactory DT sequences which tend to explode or converge over time, while CLDS and dynencoder produce DT sequences with high quality. The reason why the sequence produced by LDS or kernel-DT explodes or converges is because of the eigenvalue property of the transition matrix, which is analyzed in [28] where CLDS is proposed. The sequence tends to explode when the transition matrix has eigenvalues greater than 1 and tends to converge if all the eigenvalues are less than 1. Stable LDS [4] tries to solve this by adding constraints on the eigenvalues. We apply it to the DT sequences using the released code and add the results into comparison (no images shown due to space limit). Quantitative results of all methods through mean squared errors between original frames and new synthesized frames are provided in Table 1. It reveals that our method gives close results comparing to other methods over simple sequences (such as Escalator) and achieves better performance over challenging sequences (Steam and River).

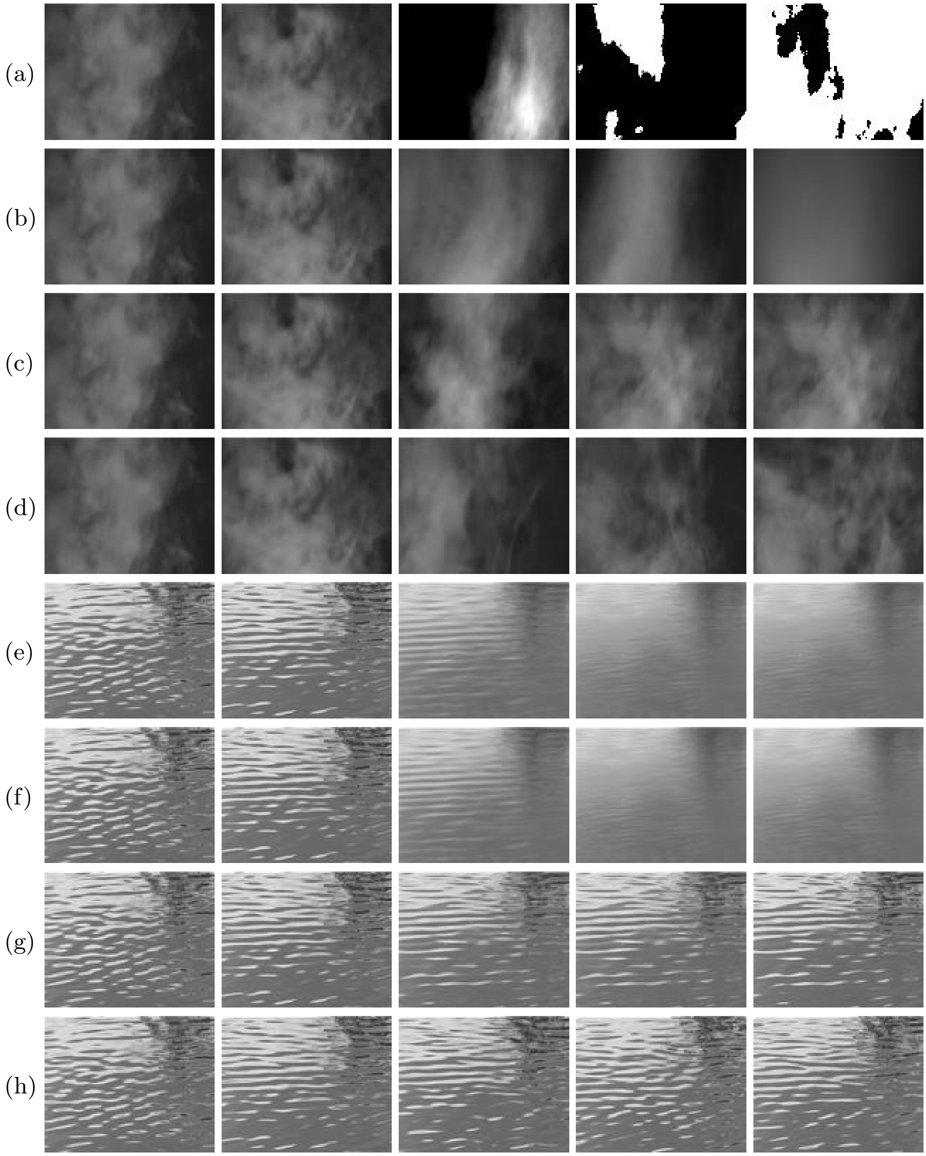


Fig. 3. Comparison between four synthesis methods. 120 frames are used for training and 1200 frames are synthesized. From left to right, the columns are the 1-st, 80-th, 200-th, 600-th, 1200-th frames of synthesized sequences. The results in each row are given by: (a) LDS, (b) Kernel-DT, (c) CLDS, (d) Dynencoder, (e) LDS, (f) Kernel-DT, (g) CLDS, (h) Dynencoder.

It is worth noting that sometimes the quantitative result given by CLDS may be unfair for itself (e.g., the result on sequence River in Table 1) because CLDS is unlike other three methods that it is not based on predicting new frames but on

Table 2. Comparison between CLDS and multilayered dynencoders through mean squared errors

Sequence\Method	CLDS	Dynencoder 1+1	Dynencoder 2+1	Dynencoder 2+2
Flowers	60.51	17.68	15.68	**11.88**
Flag	129.73	144.13	128.20	**118.82**
Windmill	21.97	73.04	47.66	**21.16**

Table 3. Comparison in traffic scene classification task. We achieve the best performance than others.

Method	Accuracy
LDS [19]	87.50% ± 0.87
CS-LDS [19]	89.06% ± 2.16
KL-SVM [5]	95%
NLSSA [1]	94.49% ± 2.02
Dynencoder 1+1	94.09% ± 1.71
Dynencoder 2+1	94.87% ± 1.75
Dynencoder 2+2	**96.06% ± 1.39**

stitching video clips getting from the original video. It is not proper to calculate predicting error for it sometimes.

CLDS achieves good performance over many sequences in our experience. It is a technique combining two steps: concatenating video clips that have similar boundaries and smoothing between the boundaries. The reason why CLDS works well may be that in most cases it can find clips that have similar boundaries. If not, the smooth process may fail and discontinuities may occur. To show this, we handcrafted a new sequence with camera motion in it. The sequence is obtained by applying a sliding window of changeable size over the sequence Flowers, which changes itself through zooming and panning in turn and will not return to the starting location exactly. Over this sequence, we compare the CLDS method not only to basic dynencoder, but also deep dynencoders with more layers to see the influence of the number of layers. The synthesis results are shown in Figure 4, in which Dynencoder $M + N$ represents the deep dynencoder consisting of M autoencoders and N dynpredictors. We can infer from the figure that CLDS fails to find clips that have similar boundaries and the smooth process does not work well. So discontinuities occur near the boundaries while our method generates more continuous sequences. Only there are some obvious discontinuities in sequence synthesized by dynencoder 1+1. Besides, we implement multilayered dynencoders over other two sequences and quantitatively compare their mean squared errors, as shown in Table 2. As we expected, dynencoder with more layers yields better performance.

4.2 Traffic Scene Classification

In this experiment, we use the UCSD traffic video database [5] to classify videos based on the density of traffic and evaluate the dynencoder and the distance

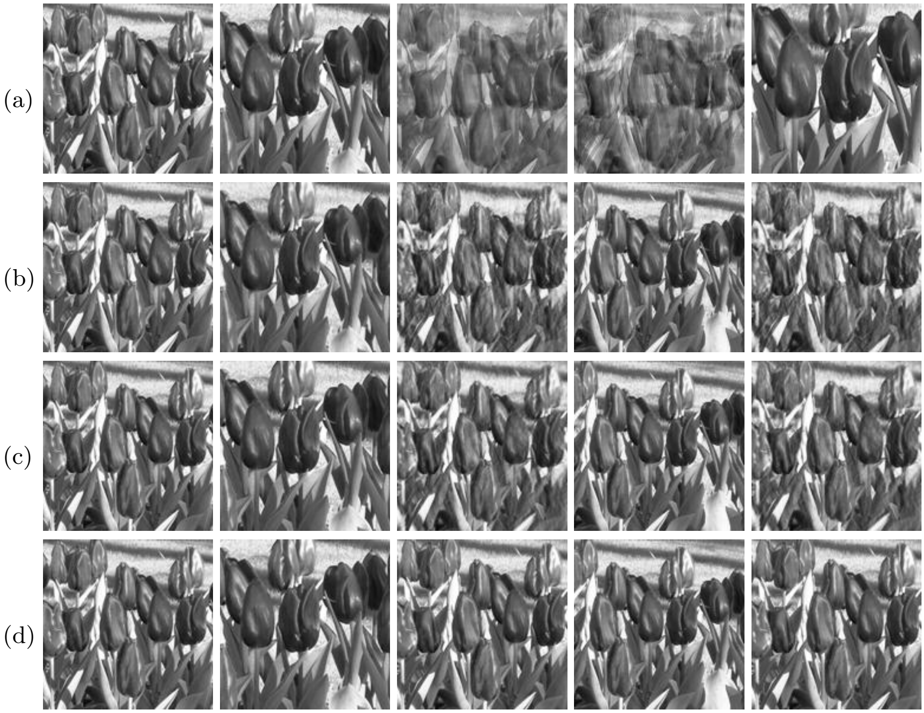


Fig. 4. Comparison between CLDS and multilayered dynencoders. 100 frames are used for training and 1000 frames are synthesized. From left to right, the columns are the 1-st, 60-th, 102-th, 380-th, 690-th, 918-th frames of synthesized sequences. The results in each row are given by: (a) CLDS, (b) Dynencoder 1+1, (c) Dynencoder 2+1, (d) Dynencoder 2+2.

definition. The database consists of 254 videos of highway traffic which are partitioned into 3 classes corresponding to the amount of traffic congestion (Figure 5 shows some examples). There are 44 sequences of heavy traffic, 45 of medium traffic, and 165 of light traffic. Each video contains between 42 to 52 frames of size 320×240 and is converted, resized and clipped to 48×48 grey scale images. All the data after preprocessing are provided in the database, and four trials of train/test splits (75% for training and 25% for testing) are suggested. We set the parameter C of SVM to be 2 without selection and report the average classification accuracy and standard deviation over the four trials.

We compare the performance of our method with three LDS-based methods: LDS [19], compressive sensing LDS (CS-LDS) [19], and probabilistic kernels (KL-SVM) [5]. We also compare it to another method named Non-Linear Stationary Subspace Analysis (NLSSA) [1]. All the results are listed in Table 3, in which Dynencoder $M + N$ represents the deep dynencoder consisting of M autoencoders and N dynpredictors. It can be seen that deeper model gets better results and Dynencoder $2 + 2$ outperforms all other methods and achieves the

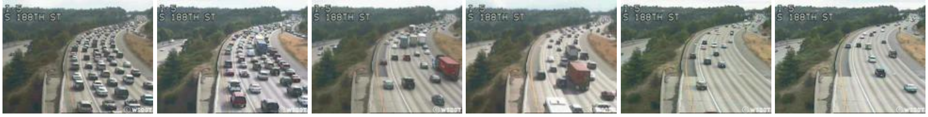
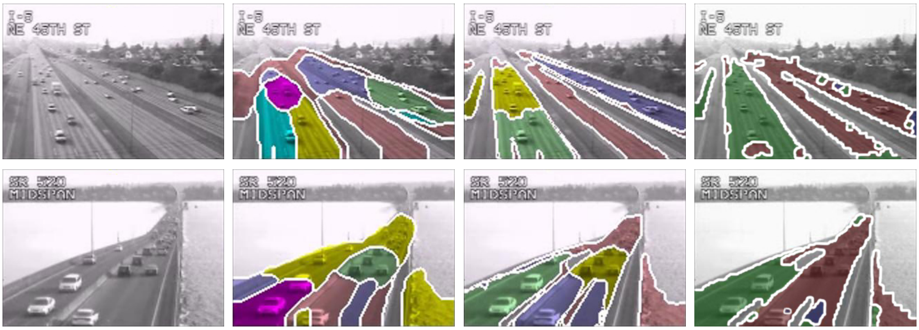


Fig. 5. Examples of traffic videos in the UCSD database. From left to right, the amounts of traffic congestion are: heavy, heavy, medium, medium, light, light.

highest accuracy 96.06%. It confirms the fact that deep dynencoder represents video dynamics well and the distance definition is efficacious.

4.3 Motion Segmentation

We do segmentation experiment on real-world video sequences depicting vehicle traffic on a bridge or highway using our method described in Section 3. Each sequence has 51 grey scale frames of size 160×113 and the size of spatiotemporal patches is set to be 5×5×51. Patches with pixel-level temporal variances of less than 10 are marked as static background. We use K-medoids algorithm for clustering with $K = 3$ or 4. Finally, the segmentation maps are postprocessed with a 5×5 majority smoothing filter.



(a) Original Sequences (b) NormCuts (c) DytexMixCS (d) Dynencoder

Fig. 6. Original video sequences and segmentation results given by three methods

Segmentation results of the basic dynencoder are compared with those produced by an LDS-based method DytexMixCS [7] and a traditional optical-flow-based method NormCuts [20]. It is difficult to give quantitative evaluation on the results, so we qualitatively show the segmented video frames[1] in Figure 6. As we can see, our method segments the videos into regions of traffic which are moving away from the camera and moving towards the camera. Although there are some small incorrect segmented regions, our results are better than those

[1] The original videos and the results of comparison methods are from the companion web site of [7].

produced by other methods in some ways. DytexMixCS and NormCuts segment the region with traffic moving away from (or moving towards) the camera into more than one main region because of perspective effects while our method does not. It means that our method can handle strong perspective effects.

5 Conclusion and Discussion

We have proposed a hierarchical model named deep dynencoder to model video dynamics, that relies on the combination of autoencoder and dynpredictor. We describe the architecture of basic dynencoder and show how to construct deep dynencoder by proper stacking strategy. Effective training algorithm is also given, which includes layer-wise pre-training and joint fine-tuning. Our model can be applied to DT synthesis and the outstanding synthesis performance verify its ability of capturing various video dynamics. With a defined distance measure between videos, our model can also be applied to video classification and clustering. Experimental results on traffic scene classification and motion segmentation confirm the effectiveness of our model and the distance definition as well.

In our experimental comparisons, we only report results on small-scale data. In our future research, we will set up more complex deep models with large-scale training data, from which more attractive results are expected. The computational efficiency is then an issue to be considered. Other possible directions include deep probabilistic models for video dynamics and more vision applications, like some general video segmentation and classification tasks.

Acknowledgement. This work is partially supported by the National Natural Science Foundation of China under contract No. 61390510, 61025010, 61379083, and 61272319.

References

1. Baktashmotlagh, M., Harandi, M., Bigdeli, A., Lovell, B., Salzmann, M.: Non-linear stationary subspace analysis with application to video classification. In: International Conference on Machine Learning, pp. 450–458 (2013)
2. Basharat, A., Shah, M.: Time series prediction by chaotic modeling of nonlinear dynamical systems. In: IEEE International Conference on Computer Vision, pp. 1941–1948 (2009)
3. Bengio, Y.: Learning deep architectures for AI. Foundations and Trends in Machine Learning 2(1), 1–127 (2009)
4. Boots, B., Gordon, G.J., Siddiqi, S.M.: A constraint generation approach to learning stable linear dynamical systems. In: Advances in Neural Information Processing Systems 20, pp. 1329–1336 (2008)
5. Chan, A., Vasconcelos, N.: Probabilistic kernels for the classification of auto-regressive visual processes. In: IEEE Conference on Computer Vision and Pattern Recognition, vol. 1, pp. 846–851 (2005)
6. Chan, A., Vasconcelos, N.: Classifying video with kernel dynamic textures. In: IEEE Conference on Computer Vision and Pattern Recognition, pp. 1–6 (2007)

7. Chan, A., Vasconcelos, N.: Modeling, clustering, and segmenting video with mixtures of dynamic textures. IEEE Transactions on Pattern Analysis and Machine Intelligence 30(5), 909–926 (2008)
8. Chang, C.C., Lin, C.J.: Libsvm: a library for support vector machines. ACM Transactions on Intelligent Systems and Technology 2(3), 27 (2011)
9. Doretto, G., Chiuso, A., Wu, Y., Soatto, S.: Dynamic textures. International Journal of Computer Vision 51(2), 91–109 (2003)
10. Doretto, G., Cremers, D., Favaro, P., Soatto, S.: Dynamic texture segmentation. In: IEEE International Conference on Computer Vision (October 2003)
11. Erhan, D., Bengio, Y., Courville, A., Manzagol, P.A., Vincent, P., Bengio, S.: Why does unsupervised pre-training help deep learning? Journal of Machine Learning Research 11, 625–660 (2010)
12. Le, Q.V., Zou, W.Y., Yeung, S.Y., Ng, A.Y.: Learning hierarchical invariant spatio-temporal features for action recognition with independent subspace analysis. In: IEEE Conference on Computer Vision and Pattern Recognition (2011)
13. Liu, C.B., Lin, R.S., Ahuja, N., Yang, M.H.: Dynamic textures synthesis as nonlinear manifold learning and traversing. In: BMVC, pp. 859–868 (2006)
14. Martin, R.: A metric for ARMA processes. IEEE Transactions on Signal Processing 48(4), 1164–1170 (2000)
15. Masiero, A., Chiuso, A.: Nonlinear temporal textures synthesis: a monte carlo approach. In: Leonardis, A., Bischof, H., Pinz, A. (eds.) ECCV 2006, Part II. LNCS, vol. 3952, pp. 283–294. Springer, Heidelberg (2006)
16. Péteri, R., Fazekas, S., Huiskes, M.: Dyntex: A comprehensive database of dynamic textures. Pattern Recognition Letters 31(12), 1627–1632 (2010)
17. Ravichandran, A., Chaudhry, R., Vidal, R.: Categorizing dynamic textures using a bag of dynamical systems. IEEE Transactions on Pattern Analysis and Machine Intelligence 35(2), 342–353 (2013)
18. Saisan, P., Doretto, G., Wu, Y.N., Soatto, S.: Dynamic texture recognition. In: IEEE Conference on Computer Vision and Pattern Recognition, vol. 2 (2001)
19. Sankaranarayanan, A.C., Turaga, P.K., Baraniuk, R.G., Chellappa, R.: Compressive acquisition of dynamic scenes. In: Daniilidis, K., Maragos, P., Paragios, N. (eds.) ECCV 2010, Part I. LNCS, vol. 6311, pp. 129–142. Springer, Heidelberg (2010)
20. Shi, J., Malik, J.: Normalized cuts and image segmentation. IEEE Transactions on Pattern Analysis and Machine Intelligence 22(8), 888–905 (2000)
21. Taylor, G.W., Fergus, R., LeCun, Y., Bregler, C.: Convolutional learning of spatio-temporal features. In: Daniilidis, K., Maragos, P., Paragios, N. (eds.) ECCV 2010, Part VI. LNCS, vol. 6316, pp. 140–153. Springer, Heidelberg (2010)
22. Taylor, G.W., Hinton, G.E.: Factored conditional restricted boltzmann machines for modeling motion style. In: Annual International Conference on Machine Learning, pp. 1025–1032. ACM (2009)
23. Taylor, G.W., Hinton, G.E., Roweis, S.T.: Modeling human motion using binary latent variables. In: Advances in Neural Information Processing Systems 19, p. 1345 (2007)
24. Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., Manzagol, P.A.: Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. Journal of Machine Learning Research 11, 3371–3408 (2010)
25. Wolf, L., Hassner, T., Taigman, Y.: The one-shot similarity kernel. In: IEEE International Conference on Computer Vision, pp. 897–902 (2009)
26. Xie, J., Xu, L., Chen, E.: Image denoising and inpainting with deep neural networks. In: Advances in Neural Information Processing Systems, pp. 350–358 (2012)

27. Yan, X., Chang, H., Chen, X.: Temporally multiple dynamic textures synthesis using piecewise linear dynamic systems. In: IEEE International Conference on Image Processing (2013)
28. Yuan, L., Wen, F., Liu, C., Shum, H.-Y.: Synthesizing dynamic texture with closed-loop linear dynamic system. In: Pajdla, T., Matas, J(G.) (eds.) ECCV 2004. LNCS, vol. 3022, pp. 603–616. Springer, Heidelberg (2004)