A Comparison of RCN, CNN, SVM, KNN on EMNIST-letters Dataset

Rui Lin, Huajing Zhao

December 18, 2017

1 Motivation

2 Related works

3 Problem Statement

4 RCN Architecture (Brief)

4.1 Model

Recursive Cortical Network(RCN)[1] is a generative model. More precisely, it is a probabilistic graphical model describing how an image is generated from a set of latent variables.

RCN model is factorized into two parts: shape model and appearance model. The shape model is hierarchical. It describes how the shape (the edge map) of an object is generated at first. Appearance is modeled using Conditional Random Field (CRF) and describes how shapes can then be combined with colors or textures to generate the final image.

Since we are using EMNIST-letters dataset, which only contains grayscale handwritten alphabet characters, appearance model is not necessary for our setup. And because the complete RCN model is too complicate to describe in such a short report[1], here we only give a brief description of RCN shape model. Instead, we will emphasize more on comparing RCN shape architecture to CNN architecture.

4.1.1 Shape Model Overview

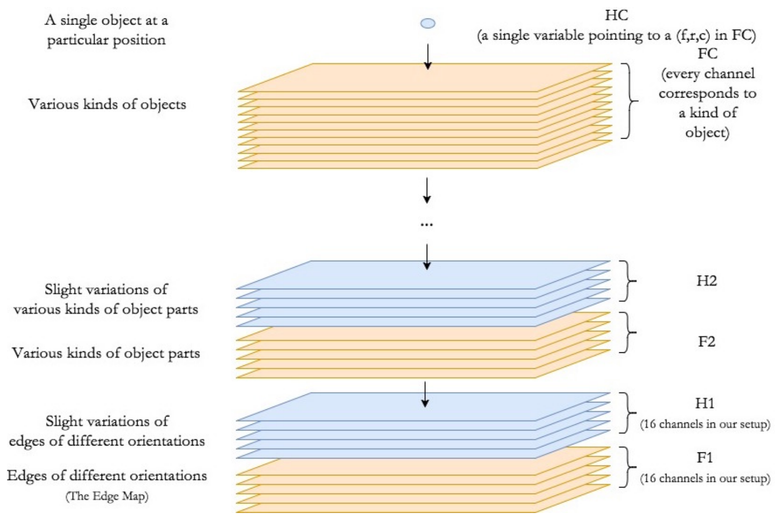
A single object at a particular position
HC
(a single variable pointing to a (f,r,c) in FC)
Various kinds of objects
FC
(every channel corresponds to a kind of object)
...
Slight variations of various kinds of object parts
H2
Various kinds of object parts
F2
Slight variations of edges of different orientations
H1
(16 channels in our setup)
Edges of different orientations
(The Edge Map)
F1
(16 channels in our setup)

Figure 1: RCN shape model

RCN shape model is hierachical and consists of alternating *feature layers* and *pooling layers* from bottom to top, as shown in figure 1. Let F^l and H^l be the l's feature and pooling layer. F^1 is thus

[1]The original paper[1] has a 74-page supplementary material describing the method.

the bottom feature layer, which corresponds to the generated edge map. H^C would be the top layer, given there is a total number of C layers for each type. Similar to CNN, each layer in this model is 3-dimensional, with elements denoted as F^l_{frc} or H^l_{frc}. The subscripts f, r, c are *feature (aka. channel)*, *row*, *column* respectively. RCN shape model is also a probabilistic graphical model in itself, and every layer is assumed to only depend on the layer above. Thus, the joint PDF can be written as:

$$p(F^1, H^1, F^2, H^2, ..., F^C, H^C) = p(F^1|H^1)p(H^1|F^2)p(F^2|H^2)...p(F^C|H^C)p(H^C) \tag{1}$$

4.1.2 F^1 layer

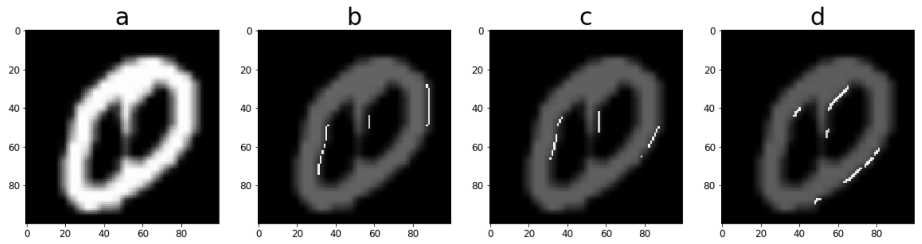

Figure 2: Channels in F^1 (aka. the edge map). a) the real image. b), c) and d): 3 channels of different orientations. The white dots correspond to where $F^1_{frc} = 1$. The real image is provided as a background.

Just like in CNN, each feature layer in the shape model describes some features of the underlying image, with the lower feature layers describing more detailed features such as edges, and higher feature layers describing more general features such as an angle, a square or, in the top feature layer, an entire complex object. In F^1 for example, each f describes an edge orientation; if F_{frc_1} is 1, there is an edge of orientation f at location (r, c), as shown in Figure 2. (This is exactly why F^1 is also referred to as the edge map.) Every variable in feature layers are binary; $F^l_{frc} = 1$ means the feature f of layer l exists in the image at position (r, c).

4.1.3 Intermediate Pooling Layers

We refer to pooling layers other than H^C as intermediate pooling layers. This section briefly describes their architecture.

Different from CNN, pooling layers here are neither max pooling nor average pooling. Moreover, pooling layers[2] in RCN are of the same dimension as their corresponding feature layers. Every pooling layer variable H^l_{frc} is associated with a set of *pool members* in its corresponding feature layer. Pool members of H^l_{frc} are $\{F^l_{f'r'c'} : |r' - r| < ph, |c' - c| < pw, f' = f\}$, where ph and pw are pool height and pool width respectively. They are both hyper-parameters of the model and are shared by all pooling layer channels in our setup. Two things worth mentioning are that, 1) pooling layers always have the same number of channels as its corresponding feature layers, and there is a 1-1 mapping between a pool channel and a feature channel, 2) a feature layer variable may be a member of multiple pooling variables.

Pooling layer variables are multinomial; if H^l_{frc} is ON (not 0), one of its pool members in the feature layer should also be ON. Thus, given H^l, F^l is deterministic. Formally, we have:

$$p(F^l_{f'r'c'} = 1|H^l) = \begin{cases} 1, & \text{if any pool in } H^l \text{ has a state corresponding to } F^l_{f'r'c'} \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

A pooling layer in RCN can thus be used to allow variations of its corresponding feature. While $F^l_{frc} = 1$ means the feature f of layer l exists in the image at position (r, c), $H^l_{frc} = ON$ means the same feature f exists in the image *in the neighborhood* of position (r, c).

4.1.4 Intermediate Feature Layers

Feature layers other than F^1 is called intermediate feature layers. They describe higher level image features such as object parts or even entire objects. Every feature layer variable F^l_{frc} is associated with several pooling layer variables below. Let the coordinates of those pooling variables at H^{l-1} be $\{H_1, H_2, H_3, ..., H_k\}$. When there is no mutual constraint (aka. *lateral constraints*) between these pooling

[2]Here we mean translational pooling layers. Other pooling layers may exist.

variables, if $F^l_{frc} = 1$, all these pooling variables should be ON, with their exact values randomly selected independently. When lateral constraints are used, the assignment of these variables is randomly selected from the set of legal assignments that satisfy the constraints. Formally,

$$p(H_1 = 0, H_2 = 0, H_3 = 0, ..., H_k = 0 | F_{frc} = 0) = 1 \tag{3}$$

$$p(H_1 = s_1, H_2 = s_2, H_3 = s_3, ..., H_k = s_k | F_{frc} = 1) = \begin{cases} 1/N^F_f, & \text{if } (s_1, s_2, s_3, ..., s_k) \text{ is legal} \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

where N^F_f is the total number of legal assignments.

Thus, $H_{l-1}|F_l$ is stochastic. The relative coordinates[3] of $\{H_1, H_2, H_3, ..., H_k\}$, denoted as $\{(f_1, \Delta r_1, \Delta c_1), (f_2, \Delta r_2, \Delta c_2), ..., (f_k, \Delta r_k, \Delta c_k)\}$ and the legal assignment set are the parameters to be learned.

More details of feature layers are as follows:

1. Every channel of every layer has its individual parameter set, but all variables within the same channel use the same set of parameters. This is convolutional in nature (another similarity to CNN), and guarantees the translation invariance of the model.

2. For each channel, $f_1, f_2, ..., f_k$ can be different. This allows higher layers to combine different low-level features to represent more complex geometric shapes. In fact, in F^C, every channel corresponds to exactly one type of object to recognize.

3. In RCN, each pooling variable can only be associated with one feature variable above; if multiple feature variables also need to activate the same pool below, more pool layers can be created and used instead.

4.1.5 H^C layer

H^C layer is the same as other pooling layers. The only difference is that there is just a single variable in this layer. This multinomial variable H^C has a domain corresponds to all feature variables in F^C. If H^C activates F^C_{frc}, it means that object f is detected in the image at location (r, c).

4.2 Inference with Loopy Belief Propagation

After an RCN model is learned, recognition on the input image is thus done by performing MAP inference on the probabilistic graph. The standard way of doing this on a loopy graph is to use the loopy max-product belief propagation.

Max-product belief propagation on non-loopy graphs is just like Viterbi decoding, and the complexity is linear in the number of nodes. When applied on loopy graphs, the method still works reasonably well, but at the cost of lots of iterations of message passing across the network, before it converges.

Given a joint probability distribution can be factorized into a product form: $p(\mathbf{x}) = p(x_1, x_2, ..., x_N) = \prod_c \phi_c(\mathbf{x}_c)$, where $\phi_c(\mathbf{x}_c)$ is called a factor function and $\mathbf{x}_c$ is a subset of $\mathbf{x}$, the message updating rule for loopy max-product belief propagation is as follows:[4]

$$\hat{m}_{c\to i}(x_i)^{new} = \max_{\mathbf{x}_{c\setminus i}}[\log \phi_c(x_i, \mathbf{x}_{c\setminus i}) + \sum_{j\in c\setminus i} (\mu(x_j) - \hat{m}_{c\to j}(x_j))] \tag{5}$$

where $\mathbf{x}_{c\setminus i}$ are all variables in factor c excluding x_i itself, and $\hat{m}_{c\to i}(x_i)$ represents the messages from each factor c to each of the variable i, updated in each iterations using max-product updating rule.

Detailed explanation of loopy belief propagation is beyond this paper, and can be found in various resources such as [2].

Iterative methods can be computationally intensive. Thus, a modified version of loopy belief propagation is used for inference on RCN model. A single bottom-up forward pass together with a subsequent top-down backward pass is considered enough to find a good approximation[5][1]. Detailed descriptions can be found in the supplementary materials of [1] in its Section 4.

[3]Relative to coordinates of F^l_{frc}, that is (f, r, c)

[4]Note the *max* and the *product* operations (*sum* in log space).

[5]However, this is not how it is done in their reference implementation. In the code, they first apply a non-loopy belief propagation on the minimal spanning tree of the factor graph. This is an approximation, and should generate several candidates. They then apply the standard loopy belief propagation with at most 300 iterations for each candidate.

4.3 Learning

Here we provide an abstract of how learning is done in RCN. More details can be found in the original paper [1].

In the learning process, we aim to learn a hierarchical model from data for both features and latent connections. Features are learned from a training set with images preprocessed to extract the contours, using unsupervised dictionary learning and sparse coding. Lateral connections for pooling layers are learned from the contours of the input. Learning is performed layerwise in a bottom-up manner, i.e., we learn the parameters for feature layer F^1 followed by F^2 and F^3. An example of lateral constraints learned is shown in figure 3.

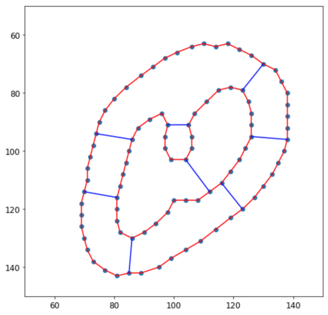

Figure 3: An example of lateral constraints learned. All red and blue lines are lateral connections between pools.

A *LearnDictionary* algorithm is executed on the training set. In dictionary learning, we consider a partially learned model; new features are to be learned at level k, where all features up to level $k-1$ and some features of level k have already been learned and finalized. A pseudo algorithm for the feature learning outer loop is given as below:

Algorithm 1: Feature learning outer loop

```
1  procedure LearnDictionary($X, HierarchyBelow$);
2  Dictionary $\longleftarrow$ $\Phi$ ;
3  Sparsifications $\longleftarrow$ $(\Phi, ..., \Phi)$;
4  while True do
5      Candidates $\longleftarrow$PickGoodCandidates($Candidates$);
6      Chosen $\longleftarrow$ProposeFeatures($Dictionary, Sparsifications, X$);
7      Dictionary $\longleftarrow$ Dictionary $\cup$ Chosen;
8      if Chosen $= \Phi$ then
9          return Dictionary;
10     end
11 end
```

Given a scenario in which F^1 has already been learned and we are learning F^2, we first express each training image as a combination of already learned F^2 features, the parts of the image not captured by the existing F^2 features will be expressed using F^1 features. This is the so-called sparsification or sparse coding. F^1 features that are not part of any F^2 feature can then be grouped based on contour continuity to form new F^2 features. This process is repeated until all training images can be explained reasonably well by only F^2 features. Trade-off is made between the number of features and reconstruction error.

The same procedure can be applied recursively to learn any number of layers. Detailed descriptions involving sparsification, intermediate feature layers and top-level feature layer learning and laterals learning can be found in the supplementary materials of [1] in its Section 5.

4.4 Comparison to CNN

Here we briefly compare the architecture of RCN and CNN, to highlight some of the key similarities and differences between these two models. (Table 1)

	RCN	CNN
Model Type	Generative	Discriminative
	Probabilistic graphical model	Neural net
Architecture	Layered architecture	
	Using F^C and H^C to do final classification	Using fully-connected layer to do final classification
	3D-feature layers, convolutional in nature, translation invariance	
	3D-pooling layers, robust to shape variation or distortion, scaling invariance	
	Translational pooling	Normally max-pooling or average-pooling, with down-sampling
Learning (Training)	Unsupervised dictionary learning and sparse coding, supervised learning only for the top layer	Supervised learning (back-propagation, gradient descent)
Inference (Prediction)	MAP inference using max-product loopy belief propagation on factor graph	Forward propagation on neural net

Table 1: RCN vs CNN

5 Evaluation

To better understand the strength and weakness of RCN, we want to compare the performance of RCN model with other models on a real-world dataset. The group of other models we select contains CNN, SVM, and KNN. As mentioned above, RCN has many similarities to CNN, and they are both specialized in visual recognition. SVM is a versatile thus widely used machine learning method, and it has proved to perform very well with Histogram of Oriented Gradient (HOG) [6] feature in visual perception areas. KNN is yet another simple but powerful machine learning method that has also been successfully applied to visual perception, which is not only easy to implement but also perform very good classification accuracy given large training dataset.

The methods we choose are different in nature: RCN is a generative probabilistic graphical model, CNN is a discriminative neural net, SVM is a kernelizable large margin classifier, and KNN relies on the smoothness of the underlying problem. Therefore it is interesting to see how they compare in the real world.

5.1 Dataset

We use the EMNIST-letters dataset to compare all four models. EMNIST-letters [4] is a handwritten English characters dataset derived from the NIST Special Database 19 and converted to a 28x28 pixel image format. It contains 145,600 characters and 26 classes, all balanced. To evaluate the performance of each classifier on relatively limited training samples, we randomly selected 10, 20, and 50 samples for each character from the original training set to form our 3 training sets. Our test set contains 50 samples randomly selected from the original test set for each character.

5.2 Model Setup

Here we briefly describe the setup for each model.

RCN is a probabilistic graphical model. Though it seems not extremely difficult to implement the shape model, fully construction of the model requires doing max-product loopy belief propagation on this complex factor graph[7] and unsupervised dictionary learning. Unfortunately, this is too difficult for a team of only two members. Thus we rely mainly on the reference example code for MNIST digit recognition dataset, and apply some parameter tuning for our own task. Some details are:

- number of layers (parameter C): 2 (i.e. F^1, H^1, F^2, H^2).
- F^1, H^1 are predetermined as edges of 16 orientations. Only F^2 needs to be learned.
- pool shape: (25, 25).
- perturb factor: 2.0.

[6] Histogram of Oriented Gradient (HOG) feature can be viewed as an affine weighting on the margin of a quadratic kernel SVM, which connects feature extraction and learning processes. HOG-SVM pipeline adds prior to a linear SVM trained on pixels, preserves second-order statistics and locality of interactions [3].

[7] The original author applied several approximations during the process.

- number or candidates[8] : 5.

Our 1-10-P-20-P-320-50-26 CNN model[9] is implemented in Pytorch. It has two convolution layers, two corresponding pooling layers, and two fully-connected layers. ReLU unit is used after both pooling layers and the first fully-connected layer. Softmax is used for the final output. Negative log-likelihood is used as the loss function. We also apply *random dropout* after the second convolution layer and the first fully-connected layer, to avoid overfitting. The model is trained using mini-batch SGD with Adam on both CPU (for the smaller datasets, 100 epochs) and GPU (for the full dataset, 200 epochs).

Our SVM classifier is implemented in Matlab with Statistics and Machine Learning Toolbox, using HOG features to perform a multiclass classification. Prior to training and testing a classifier, a pre-processing step is applied to remove noise artifacts introduced while collecting the image samples. Gradient computation is achieved through convolution with a bank of oriented edge filters, where the nonlinear transform is the pointwise squaring of the gradient responses [3]. The data used to train the classifier are HOG feature vectors extracted from the training images, and then evaluated on the test set where test images were classified with extracted HOG features of the same procedure.

Besides that, we also develop a KNN model in Matlab. Though KNN with L3 distance or more advance techniques such as Tangent Distance and non-linear deformation (IDM) may perform better, we implement with L1 distance as a baseline method. Also, it is worth noting that the increasing values of K cause the accuracy to drop, perhaps due to the fact that some handwritten characters such as t and f, finding additional nearest neighbors may cause the classifier to include more similar-looking but different characters. Here we use $K = 1$ for comparison.

All code is published on github.[10]

5.3 Results & Analysis

The results we get are shown in table 2. Clearly, RCN outperforms all other three models when the training set is small. This is what we expect. In RCN, lower layer features (such as basic geometric shapes) are already learned through unsupervised dictionary learning beforehand. This means the model does not need to learn from scratch, and this is the main reason behind its excellent performance.

To show that our implementation of CNN, SVM, and KNN has no significant issue, and can be used as good baselines for RCN, we also run those three algorithms with the full dataset. The result is shown in the last column. As can be seen, all three algorithms perform reasonably well, with CNN performs slightly better.

During our experiment, we also notice that RCN model runs significantly slower when doing inference.[11] With that being said, since the reference code is unoptimized and only serves as an example, this result should not be surprising. However, we still regard inference performance as one of the main weaknesses of RCN. As mentioned in section 4.2, RCN inference is done by doing loopy max-product belief propagation on the complex factor graph. This is an iterative method and may not even converge. Thus it is considered more difficult to compute than the simple forward propagation in CNN.

Method / Train Size	10×26	20×26	50×26	7800×26
RCN	70.06	79.63	85.10	
CNN	60.19	70.45	79.01	90.41
SVM	57.92	69.38	75.54	89.85
KNN	45.54	52.38	59.77	85.46

Table 2: Accuracy (in percentage)

6 Conclusion

References

[1] D. George, W. Lehrach, K. Kansky, M. Lázaro-Gredilla, C. Laan, B. Marthi, X. Lou, Z. Meng, Y. Liu, H. Wang, A. Lavin, and D. S. Phoenix, "A generative vision model that trains

[8] Loopy belief propagation is slow. This is the number of candidates generated using approximation before the belief propagation.

[9] We referenced the parameters in Pytorch example code, to make sure we get a reasonable result to compare with.

[10] `https://github.com/Rui-L/RCN_CNN_SVM_KNN-EMNIST`

[11] This is also why RCN is not applied to the full dataset.

with high data efficiency and breaks text-based captchas," *Science*, 2017. [Online]. Available: http://science.sciencemag.org/content/early/2017/10/25/science.aag2612

[2] J. S. Yedidia, W. T. Freeman, and Y. Weiss, "Understanding belief propagation and its generalizations," *Exploring artificial intelligence in the new millennium*, vol. 8, pp. 236–239, 2003.

[3] H. Bristow and S. Lucey, "Why do linear svms trained on HOG features perform so well?" *CoRR*, vol. abs/1406.2419, 2014. [Online]. Available: http://arxiv.org/abs/1406.2419

[4] G. Cohen, S. Afshar, J. Tapson, and A. van Schaik, "Emnist: an extension of mnist to handwritten letters," *arXiv preprint arXiv:1702.05373*, 2017.

[5] C. Sutton, A. McCallum *et al.*, "An introduction to conditional random fields," *Foundations and Trends® in Machine Learning*, vol. 4, no. 4, pp. 267–373, 2012.