Facial Expression Recognition Classification

Shala Chen
20698485
SYDE 675 Pattern Recognition

Abstract

This projects explored several classification methods on a human facial expression dataset. Part of the dataset was used as training data to build the classification model, the rest of the dataset was used as testing data to evaluate the classification performance. Methods included Minimum Euclidean Distance classifier, K Nearest Neighbors classifier, Support Vector Machine classifier, Neural Network classifier and Convolutional Neural Network classifier. The accuracy rate was used as a method of performance check.

Introduction

Facial recognition system had been used for identity verification in many ways. It was getting more and more precise and well combined with other application such as Facebook and iPhone camera while also popular as identification and commercial marketing tools. Although those systems were so smart about recognizing which person was your friend, I had not seen many examples of facial expression recognition systems. I wondered if it was possible to build a system that it would be able to tell if a person was happy or not, or what feeling was he or she expressing right now.

In this project, I used a facial expression dataset to train various classifiers with several different methods including Minimum Euclidean Distance(MED), K-Nearest Neighbor(KNN), Support Vector Machine(SVM), Neural Network(NN) and Convolutional Neural Network(CNN). These classification methods all had different characteristics and would be applied and compared their performance with each other.

Database description

I used the "Challenges in Representation Learning: Facial Expression Recognition Challenge dataset" from Kaggle website as my training and testing data. It contained 35887 images with various facial expressions including angry, disgust, fear, happy, sad, surprise and neutral. The images were all gray-scale in a dimension of 48x48 pixels. Following were some samples of the images. The models in the images were from different ages, genders, and races, some of them only had a face from chin to eyes, and some of them had specific gestures like holding face with a hand or leaning to an angle. Their backgrounds were not the same either, some were just black or white, while others may have actual scenes on the background.

Figure 1: Images in the data set

Due to the limitation of computing devices, I started with only 2 facial expression categories angry and happy, which had a total of 4953 angry facial expressions and 8989 happy facial expressions. I selected 20% of each category as testing data and the rest 80% as training data. By the end of this process, I had a training dataset with a total of 11153 images and a testing dataset with a total of 2789 images, as well as 2 separate files with their correct classes. Each row in the dataset represented one image and there were total 11153 rows of data. Every column represented row ordered pixel values of an image and there were 2304 (48 X 48 pixels) columns in total. The following figure was an example of a partial dataset.

11153x2304 double

	1	2	3	4	5	6	7	8	9	10	11	12
1	70	80	82	72	58	58	60	63	54	58	60	48
2	151	150	147	155	148	133	111	140	170	174	182	154
3	30	24	21	23	25	25	49	67	84	103	120	125
4	123	125	124	142	209	226	234	236	231	232	235	223
5	8	9	14	21	26	32	37	46	52	62	72	70
6	236	230	225	226	228	209	199	193	196	211	199	198
7	255	82	0	3	0	0	0	0	0	0	3	0
8	74	66	82	82	85	68	79	103	113	112	116	136
9	206	208	209	212	214	216	216	219	221	222	222	222
10	57	62	88	106	134	136	136	135	135	146	163	171
11	117	116	113	99	78	53	37	34	33	32	31	28

Figure 2: Data example for training data set

Methods

Minimum Euclidean Distance

The Minimum Euclidean Distance here was one form of the Mikowski metric

$$d_k(x,y) = [\sum_{i=1}^{n} |x_i - y_i|^k]^{1/k}$$

For $k = 2$, we get the l2 norm or say Euclidean distance.

$$d_k(x,y) = [\sum_{i=1}^{n} |x_i - y_i|^2]^{1/2}$$

For data point in matrix form, the Euclidean distance could also be expressed as

$$d_E(X, X_a) = [(X - X_a)^T (X - X_a)]^{\frac{1}{2}}$$

The Minimum Euclidean Distance classifier simply calculated the mean values of all features from the training data set and generated a classification boundary. When we were using this model to classify a testing data point, the model would simply calculate Euclidean distances from this data point to the means of 2 classes. If the distance between this data point and feature mean value of class A is shorter than to the mean value of class B, then we would say, this data point belongs to class A, otherwise, this data point belongs to class B instead.

Implementation

To get the classification boundary, I calculated centers of 2 classes using training sets. Using these two mean values, I compared every data points with both of them and took the one with minimum distance as this image's class prediction. The training and testing process were really fast. The accuracy of the whole testing dataset was 0.5809.

Discussion

An accuracy at 0.5809 indicated that there were only 1620 out of 2789 images correctly classified, it was not much better than random guesses. I think it's reasonable because a minimum euclidean distance classifier only considered average values of all the features instead of considering feature distributions at all. The feature variance and covariance were all ignored, every column feature was considered equally important, so as long as the image was slightly different, like a same face but in a different background might cause a very large Euclidean distance and thus being classified in to different group.

K Nearest Neighbor

K nearest neighbor classifier as a non-parametric method, the number of parameters grew with the amount of training data, thus it was more flexible. When using K nearest neighbor classifiers, for every testing data point, the model calculated distances from this point to all the training data points, then defined a set of K nearest neighbors. Within this nearest neighbor set, it counted how many member of each class and used their majority votes as the final prediction by returning empirical fractions for each class as a probability, then optionally take highest probability as class prediction for this data point.

$$P(y = c|x, D, k) = \frac{1}{k} \sum_i \in N_k(x, D) II(y_i = c)$$

Implementation

There were various distances could be used to get nearest neighbors such as Euclidean distance, weighted Euclidean distance for axis scaling purpose or Mahalanobis distance for specific class shapes. In this project, I used Euclidean distance to find the nearest neighbors.

I could not decide which K value I should use so I iterated all data several times to build classifiers with different Ks. In my first attempt, I intended to use a smaller training set to find the best K value and then apply it to the whole data set. So I used a training dataset with only 2000 images with 1000 images from each class to train classifiers with K values varied from 1 to 81. However, the accuracy just kept increasing as I increased K value. It could not be right because if the K values were big, the decision boundary would be too smooth and generous with outliers. I believed this was because there were not enough data samples to train the classifiers. In the end, there were 2304 columns of features that we need to utilize.

So in my second attempt, I used all 11153 samples to train the classifier and used K values from 1 to 81. The highest accuracy I got was 0.6884 when $K = 11$ according to the following figure.

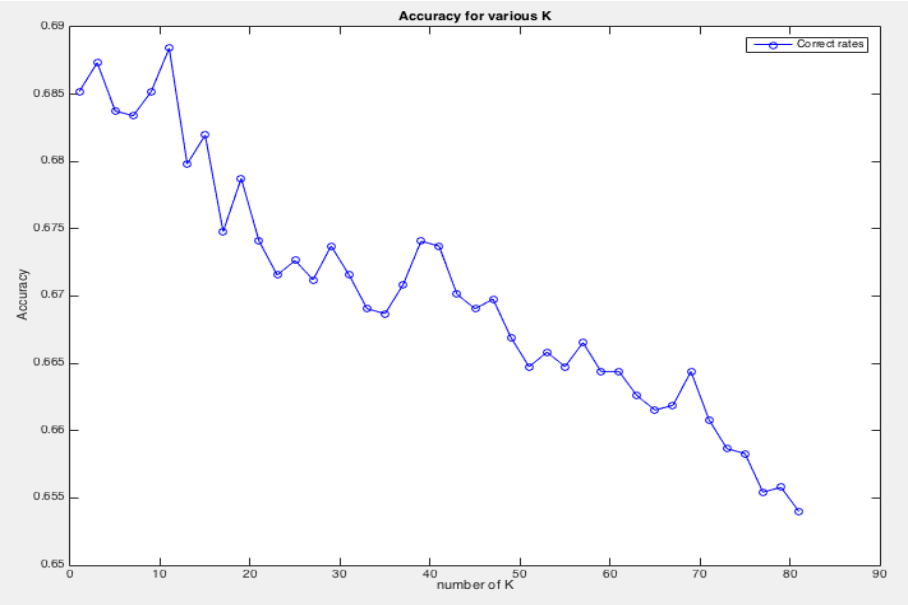


Figure 3: KNN classifier accuracy for different K value

Discussion

As we could see from above graph, when K kept increasing, the accuracy would decrease because there were too many neighbors being counted and some were too far away that their vote was actually not contributing to the classification result, instead, since their vote were equally evaluated, those far neighbors were creating noise rather than contributions. Since there were so many features that one data point was represented in a very high dimension, a smaller K value such as 3 sometimes had a better accuracy than the large K like 81.

In order to build a K nearest neighbor classifier, the model would simply memorize all the training data point and be able to correctly classify them when they showed up again. That's why we had better to use new images as testing data set, or the accuracy would not be fair.

There was one more thing need to be noticed that a KNN classifier relies on majority vote from neighbor points, thus the result would be biased towards the class with more training data points. Like in this dataset, there were more happy images rather than angry images, so when we were randomly selecting neighbors, there was a better chance that the neighbor was in a happy class rather than angry class.

Support Vector Machine

The goal of support vector machine classifiers were to separate different classes with a hyperplane induced from training samples. This hyperplane was like a decision boundary, that when there were new testing data points, it would just find which side of the boundary did it fell into, then pick that side as the class prediction.

When separating classes, the model tended to find a boundary that the distances from the closest vector to the hyperplace was maximized. Those vectors which determined the decision boundary were support vectors. Following was the equation to find a hyperplane.

$$w^T x + b = 0$$

Training a support vector machine classifier was like creating a gap as large as possible between different classes. The optimal situation was to get a large margin, which meaned this gap was wide. And each test data will be classified base on which side of the gap it fell into.

However, when the classes were not linearly separable, we had to allow some error in the training data using soft-margin SVM. In the soft-margin SVM, there could be some data samples fell into the gap and the optimization problem changed to minimizing

$$v(w, b, \xi) = \frac{1}{2}||w||^2 + c\sum_{i=1}^{n} \xi_i$$

where c was the soft margin parameter, subject to

$$z_i(w^T x_i + b) \geq 1 - \xi_i \forall_i, and \xi_i > 0 \forall_i$$

Those were situations when classes could be linearly separated. In many cases, the data points were more complicated that they were not linearly separable, that was when we need to use non-linear SVM. The vectors were transformed into higher dimension feature space using a non-linear mapping. However, calculating non-linear mapping could be very inefficient, to make the calculation easier, we could get the result of the dot product of two non-linear mappings instead of calculating them explicitly. Simply use kernel trick here and let

$$K(x_i, x_j) = < \Phi(x_i), \Phi(y_j) >$$

Then keep going with $K(x_i, x_j$ to select support vectors and creating the classifiers.

Implementation

With such a large data set consisting 2304 features, I decided to use Gaussian Radial Basis Function as my kernel function for the non-linear mapping process where $K(x_i, x_j)$ was defined as

$$K(x_i, x_j) = \exp(-\frac{||x_i - x_j||^2}{2\sigma^2})$$

Now that I had the functions I could use, first of all, I needed to find which data points were the support vectors. I had to solve

$$\max_{\alpha}(\sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1,j=1}^{n} \alpha_i \alpha_j z_i z_j K(x_i, x_j))$$

where

$$\sum_{i=1}^{n} \alpha_i z_i = 0, \alpha_i \geq 0$$

Now I could determine the support vectors by finding all the data points for which $\alpha_i \geq 0$ and labeling them as x_{sv}. I created the hyperplane using

$$w = \sum_{i=1}^{n} \alpha_i z_i x_i$$

and

$$b = z_{sv} - < w, x_{sv} >$$

To test new samples, simply let

$$y = w^T x + b = \sum_i \alpha_i K(x_{sv}, x) + b_i$$

If $y \geq 0$ then it belonged to class 1, else it belonged to class 2.

The only parameter I could change was the σ in the RBF function, so I tried σ values from 10 to 100 incremental by 5 and found the best $\sigma = 30$ on 2000 images and got the following result. The final accuracy was 0.7150 according to the calculation.

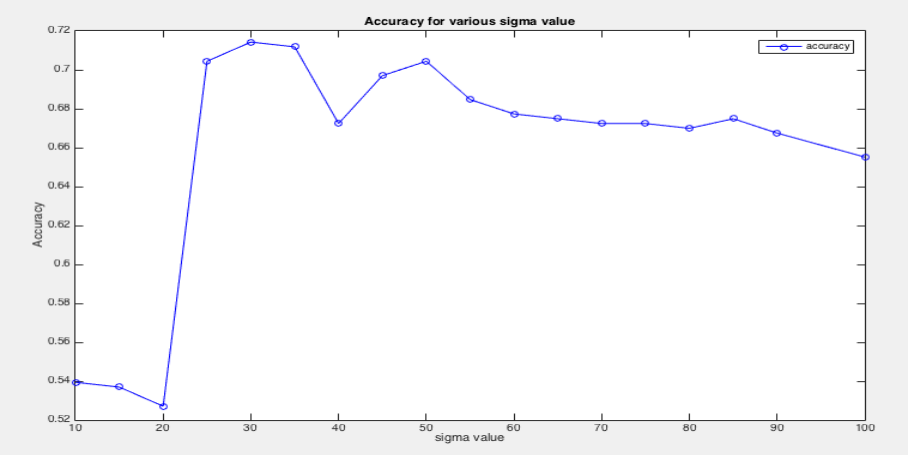


Figure 4: RBF-SVM classifier accuracy for different σ value using 2000 images

Then I tried σ values from 15 to 50 incremental by 5 to build the SVM classifier using all 11153 images. I didn't start with 10 because it seemed that 10 was too small to achieve a final converge within max iterations. The best accuracy of SVM classifiers using all images was 0.7763 when $\sigma = 20$

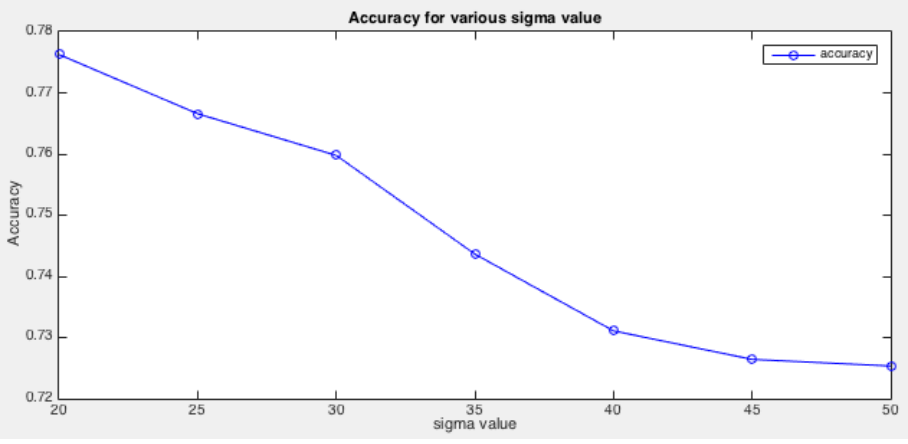


Figure 5: RBF-SVM classifier accuracy for different σ value using 11153 images

Discussion

We could find that the optimal σ for a large training dataset was smaller than for a small training dataset. It was reasonable because a smaller sigma could create a less smooth boundary like the case below

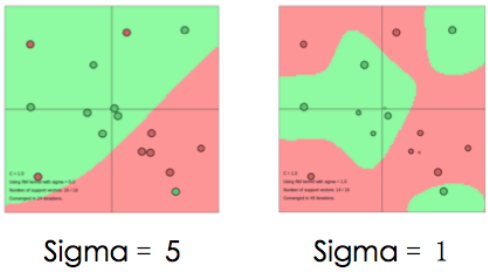


Figure 6: RBF-SVM classifier decision boundaries for different σ value

A classifier with more data points in the space would have a better probability to get a wavy decision boundary to better separate different classes.

The RBF kernel SVM was powerful because it was able to use the non-linear mapping to create a hyperplane as decision boundary in a higher dimension. This made almost all kinds of datasets linearly separable.

There was one thing to notice. Although RBF kernel SVM had a better performance than K Nearest Neighbor classifier, it took a much longer time to complete the calculation. Every iteration took more than 30 minutes to train and test the data. The higher accuracy came with a cost.

Neural Network

Neural Network was formed with several layers of weak learner which was formed with neurons. Those neurons were logistic or sigmoid units and some of them were hidden in the middle. Given several layers containing a lot of units, a neural network could represent any function.

The first layer was an input layer, each unit in this input layer was responsible for one feature from the training set. The last layer was an output layer, each output unit computed a weighted sum of all previous layers then passed it through a threshold or non-linear function to generate predictions.

Other than those layers, the hidden layers in the middle were more flexible that if there were too few units, the network would have too few parameters, thus being unable to learn complex functions. However, if there were too many units, the network would be over-parameterized. The hidden layers were able to extract more complex features and solve complex problems. The units were connected either fully or partially. The units within one layer usually shared an activation function. There were several activation functions could be used in hidden layers, such as sigmoid function

$$F_S(x)\frac{1}{1+e^{-x}}$$

or Rectified Linear Units (ReLU) function

$$F_R(x) = max\{0, z\}$$

A simple class of decision function with a three layer neural network could be represented as

$$g_k(x) = Z_k = f(\sum_{j=1}^{n_H} W_{kj} f(\sum_{i=1}^{d} w_{ji} x_i + w_{j0}) + w_0)$$

This discriminant function didn't specify layers and units used, generally, using more layers can reduce the number of units and be more generalizable.

We could derive gradient decent rules to train multilayer networks of sigmoid units and use backpropagation to reassign weights. At the beginning, the weights were just small random numbers, using forward propagation, we could compute the first network output, then compare the output with actual classes using loss functions, then use backpropagation to adjust each weight.

If there were too many hidden nodes, overfitting might happen during training process. We could use regularization to reduce overfitting and encourage network to use all the inputs instead of only using the strongest signals. Besides, we could also use dropout method to randomly ignore certain units during training process and do not update them for some rounds in gradient descent.

Implementation

To use neural network classifiers, first of all, we need to determine the network topology including how many hidden layers we should use and how many hidden nodes in each layer.

Using similar iterations from above, I iterated 2000 images with 1 to 20 layers and it seemed 12 hidden layers could reach the best performance, and the accuracy plot was like the following figure

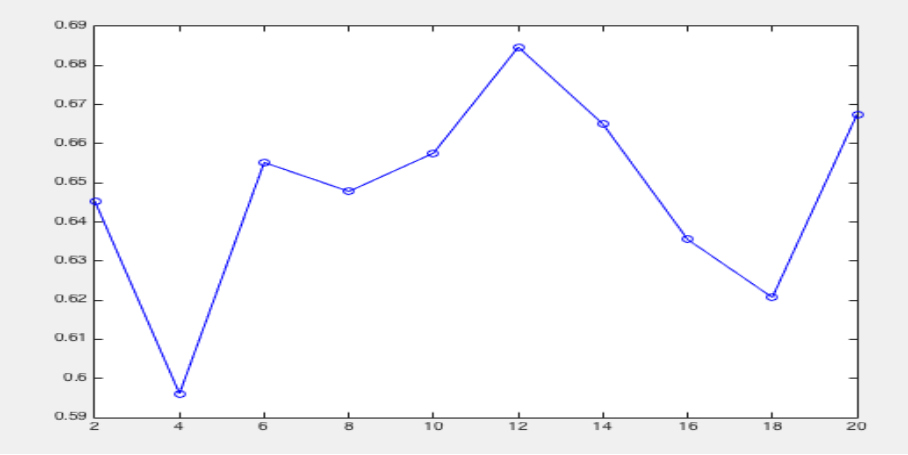


Figure 7: Neural Network classifier accuracy for 1-20 hidden layers using 2000 images

Then I applied the same iteration on all 11153 images and found that it was still optimal to have 12 hidden layers.

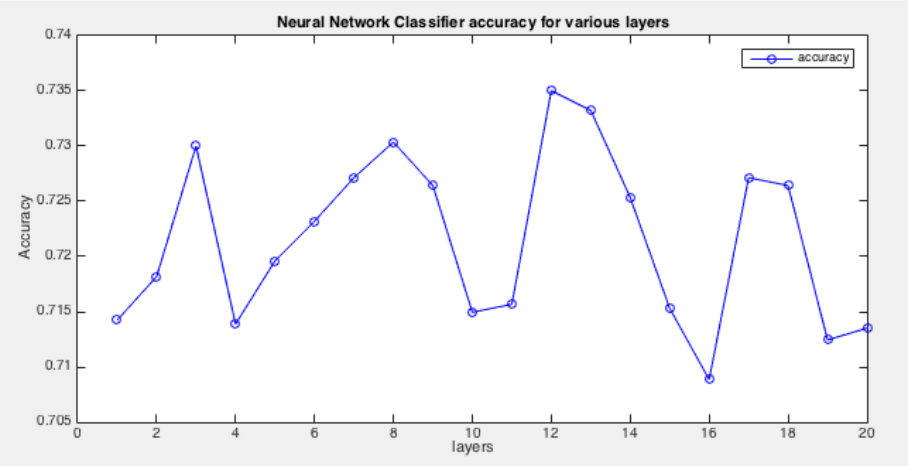


Figure 8: Neural Network classifier accuracy for 1-20 hidden layers using 11153 images

The highest accuracy was 0.7355 when there were 12 hidden layers and the confusion matrix was attached below.

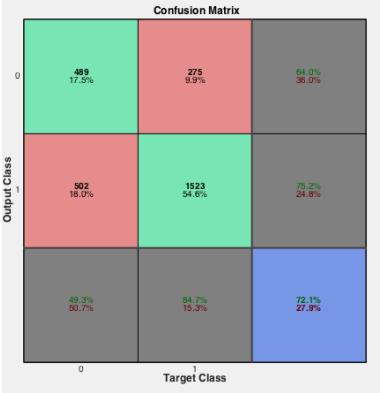


Figure 9: Confusion Matrix when trained with 11135 images

Discussion

Since the neural network classifiers assigned small random weights to all units at the beginning so the final prediction was slightly different from time to time.

For both large and small training dataset, the accuracies were the highest when there were 12 hidden layers and the accuracy was slightly higher with more training data points.

During this training process, I also noticed that the whole neural network learning process was a closed-loop system. It was hard to make any assumption or surely decide how many units or layers that we should use. For most of time, all I could do was guessing a range and try all of the possibilities.

Convolutional Neural Network

Convolutonal neural network classifiers usually had great performances on images classification. Different from regular neural network classifiers, convolutional neural network classifiers had one more step about feature extractions. When we saw a image of dog, we were able to classify it immediately based on identifiable features such as 4 legs, a tail or shape of head and ears. In a similar way, convolutional neural network models were able to identify the classes by extracting effective features such as edges and curves[3]. To achieve this task, a image would be passed through a series of convolutional layers, non-linear transformation layers, pooling layers for downsampling, and fully connected layers to get a class predictions or class probability predictions.

The first layer was always a convolutional layer. When an image input entered the model, it would be scanned by a small window filter from the upper left corner to the right until the small window slided over the whole image and covered all the pixels. During this scanning process, the model computed element wise multiplications to form a low level feature map. If the filter value and scanned image parts were mapped in a similar way, then we were going to get a very high score at this spot in the feature map.

Then we could pass this low level feature map into the second convolutional layer, which would use another filter set to find more detailed features like combination of curves and edges. After several round, some units in the layer would be able to tell if there was face or arms in this image.

Now that we had this higher level feature map, we could attach it to several fully connected layer to get either predictions or percentage distributions of possible classes. Just like we discussed in the neural network classifiers, the model assigned small random weights to each unit as initial state, used forward propagation to get an output, applied loss function on the output, then used back propagation to update the weights and parameters. This full iteration was called an epoch, and we could adjust the value of epoch to gain different final outputs.

Implementation

I switched from Matlab to Python to implement convolutional neural network classifiers. I used TensorFlow which was an open source software library for numerical computation as backend and a high-level neural network API Keras to build this model. Following were some parameters I adjusted while building convolutional neural network models:

- **Number of epoch** As I explained above, one epoch includes a full iteration of forward propagation, loss function computation, back propagation and weight update. With each epoch, the loss would decrease, while the output would be closer to the actual class and the accuracy would improve. So after experimenting the model with 2000 image training sets, I found that the accuracy kept unchanged after 30 epochs, so when I was using the whole training set to train the classifier, I set the number of epoch to be 20.
- **Batch size** Batch size determined how many images would used to propagate through the convolutional neural network in one turn to update each gradient. A smaller batch size would require less memory especially when the memory was not large enough to fit all the data. The network also trains faster with a smaller batch of images. However, if the batch size was too small, the estimate of the gradient would be less accurate. With a training set containing 11153 images, I chose the batch size to be 256.

- **Number of filters** This determined how many filters we were going to use in the convolutional layer. It also represented the dimensionality of the output space. The training images were in a size of 48 x 48 pixels, so I wanted to try 64 filters for the first run and change it in the later convolutional layers if needed.
- **Kernel size** Kernel size was the same as the window size, or the size of each filters. This was usually determined in a list of 2 numbers, representing window's height and width. I considered that the training images and models' faces were small, so that the facial expressions could be very detailed, so I set the window size to be 5 by 5.

After determining what parameters we should use, now we needed to decide the actual model structures. Keras API allowed me to manipulate convolutional neural network layers very easily and left a lot of freedom with each specific layer. Based on the number of training data set and image size, here was my structure for the convolutional neural network classifier.

1. **ZeroPadding2D** The purpose of a zero padding layer was adding zeros to the border of the images on all sides. I added a zero padding layer before any convolution layer in order to preserve more information about original input and better extract lower level features in the following step.
2. **2D convolution layer** Considering that the images were only 48 by 48 pixels large, this 2D convolutional layer would use 64 filters with filter size 5 x 5 pixel to scan over the whole image and create a low level feature map.

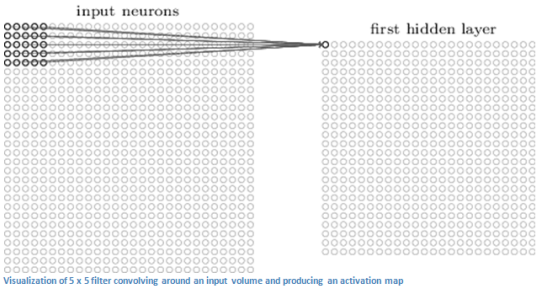


Figure 10: feature extraction in convolution layer

3. **Rectified Linear Units (ReLU) layer** A non-linear layer was applied right after the convolution layer. I used Rectified Linear Units layer in this model to introduce nonlinearity to the system since it had just been computing linear operations from the previous convolution layer. There were other non-linear functions that we could use, such as tanh function and sigmoid function. However, ReLU function worked much better because the training process was much faster without hugely sacrificing much accuracy [7]. It removed all the negative outputs values and kept the positive unchanged, which was simple and efficient.
4. **2D convolution layer** Another 2D convolution layer was applied to generate a higher level feature map based on the previous result. This convolution layer also consisted of 64 filters and windows of size 5 x 5.

5. **Rectified Linear Units (ReLU) layer** Same reason as above, another rectified linear unit layer was added to bring in non-linearity.

6. **2D max pooling layer** A pooling layer could also be referred as down sampling layer. It was like another window scanning over the feature map and extracting the most useful information within the windows. There were several methods to extract features using pooling layer, we could take the average number of the values in the windows, or take l2-norm value. I used a max pooling layer in this model, which only kept the largest number in the windows.

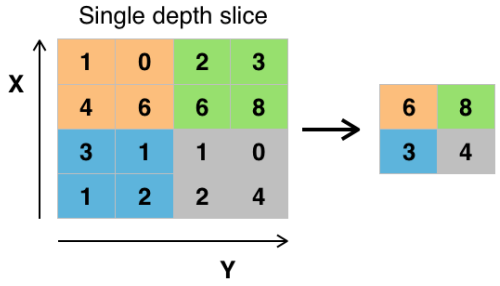


Figure 11: feature extraction in max pooling layer

There were three main reasons that we should use a max pooling layer.

The first one was that as long as we know there was a specific feature in the input, which was represented by the largest value, its exact location was no longer important, other than that, its relative location to other features were more important for future learning process. It would not affect the final result if we get rid of other unimportant values.

Secondly, eliminating other lower values could hugely decrease the data dimensions. If the window size was 2 by 2, and we only kept the largest value, then the output data would be only a quarter size compare to the input data.

Besides, using pooling layers could reduce overfitting in the model. An overfitting model fitted too well to the training set that it was not generalized enough for validation and testing processes, the direct consequence would be the accuracy was high on the training set, however, it was low on the testing set. A pooling layer only picked the largest value, and it was able to avoid overfitting[4].

In this model, I used a max pooling layer with stride size of 2 by 2. This layer would form a window of size 2 by 2 and keep the largest value within the window and form a new output with a quarter size.

7. **2D convolution layer** The same 2D convolution layer will be applied again to extract more detailed features.

8. **Rectified Linear Units (ReLU) layer** Another rectified linear unit layer was added to bring in non-linearity.

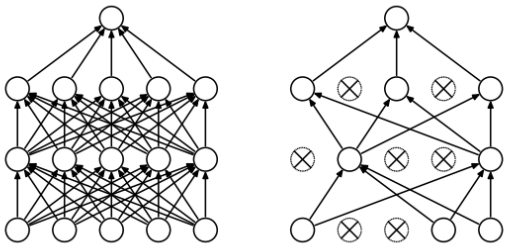
9. **2D convolution layer** Same function as above, a 2D convolution layer was applied with window size 5 by 5 for total of 64 filters.

10. **Rectified Linear Units (ReLU) layer** Another rectified linear unit layer.

11. **2D max pooling layer** The second 2D max pooling layer for extracting main features and reduce dimension.

12. **Flatten layer** A flatten layer transformed the input to a single vector output. It was a preparation for the following dense or fully connected layers.

13. **Dropout layer** After so many iterations of training, there might be overfitting existing in the model. The weights of each unit was so tuned to the training sample that when were given a new testing example, the performance would not be as good. This dropout layer simply changed random part of the weights to be 0 during forward propagation, in order to force the network to be redundant and avoid over fitting [6].



(a) Standard Neural Net (b) After applying dropout.

Figure 12: Ignore units in dropout layer

14. **Dense layer** Dense layer here worked the same as a regular fully connected neural network layer. In this dense layer, I set the units value to be 128 and then applied Rectified Linear Units (ReLU) function as activation function before generating output.

15. **Dense layer** In the second dense layer, there were total of 2 units because there were 2 classes in the training set. In this final layer, I used softmax function as activation function:

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} for j = 1, \ldots, K$$

    The output for softmax function was usually used to represent a probability distribution over different categories. It was also great to know that the softmax function was able to hightlight the largest value and suppress the small values.

16. **Compile** In the final compilation, I used cross entropy as loss function.

Now that the structure of the convolutional neural network was complete, I used the whole training data set to train the classifier, and the final accuracy for the training data set was 0.8997. For the testing set, the accuracy was 0.8203. The partial training process and result was listed below:

```
Epoch 13/20
11152/11152 [==============================] - 842s - loss: 0.3944 - acc: 0.8202
Epoch 14/20
11152/11152 [==============================] - 846s - loss: 0.3730 - acc: 0.8304
Epoch 15/20
11152/11152 [==============================] - 844s - loss: 0.3734 - acc: 0.8304
Epoch 16/20
11152/11152 [==============================] - 867s - loss: 0.3521 - acc: 0.8415
Epoch 17/20
11152/11152 [==============================] - 843s - loss: 0.3390 - acc: 0.8523
Epoch 18/20
11152/11152 [==============================] - 848s - loss: 0.3363 - acc: 0.8531
Epoch 19/20
11152/11152 [==============================] - 856s - loss: 0.3176 - acc: 0.8597
Epoch 20/20
11152/11152 [==============================] - 867s - loss: 0.3067 - acc: 0.8671
2788/2788 [==============================] - 92s
11152/11152 [==============================] - 372s
acc: 89.97%
```

Figure 13: Training process and result for convolutional neural network classifier

Discussion

The accuracies were in huge difference when using large training data set and small data set, so did the training time. 20 epochs on 11153 images data sample took more than 6 hours to train, which was very slow. Due to the feature extraction process in convolutional layer, the accuracy for the convolutional neural network classifier was higher than a regular neural network classifier. It transformed each image into feature points and used those features and feature combinations to find specific patterns in the testing image and assign class predictions.

Conclusion

In this project, I applied different classification methods on the human facial expression dataset. Those classifiers all had their own characteristics and advantages. Here were the accuracies of all of classifiers.

Table 1: Accuracies for different classifiers

Classifiers	Accuracies
Minimum Euclidean Distance	**0.5809**
K Nearest Neighbors	**0.6884**
Support Vector Machine	**0.7763**
Neural Network	**0.7355**
Convolutional Neural Network	**0.8203**

Minimum Euclidean distance, K nearest neighbors and support vector machine classifiers were all distance based classifiers. Minimum Euclidean distance classifier had the worst performance because it simply calculated average of all the features and compare vector distances from test sample to two class averages. K nearest neighbors was better because it utilized all the data points and used majority votes from the nearest neighbors. RBF support vector machine classifiers also considered all training points, it utilized only a few points as vector machine, but created a hyperplane that was able to separate different classes and had a good performance on image recognition.

Neural network classifier and convolutional neural network classifier used several layers and units to iterate data points forward and backward, calculating loss functions and updating unit weights. Other than that, the convolutional neural network was able to extract specific features from images and use those features and feature combinations to determine classes.

The training of RBF support vector machine model and convolutional neural network model was quite time consuming, I had to run the program over night in order to get a final result. The training process for K nearest neighbors also took some time, while the training for minimum Euclidean distance model and neural network model were much faster.

I personally prefer the convolutional neural network classifier for image recognitions, it was fascinating because this was somehow similar to how people recognize an object. the only limitation might be there wasn't a clear way that we could tell the performance of a specific structure without actually running it. There were still so many things to learn.

References

[1] "Challenges in Representation Learning: A report on three machine learning contests." I Goodfellow, D Erhan, PL Carrier, A Courville, M Mirza, B Hamner, W Cukierski, Y Tang, DH Lee, Y Zhou, C Ramaiah, F Feng, R Li, X Wang, D Athanasakis, J Shawe-Taylor, M Milakov, J Park, R Ionescu, M Popescu, C Grozea, J Bergstra, J Xie, L Romaszko, B Xu, Z Chuang, and Y. Bengio. arXiv 2013.

[2] Michael J. Lyons, Shigeru Akemastu, Miyuki Kamachi, Jiro Gyoba. Coding Facial Expressions with Gabor Wavelets, 3rd IEEE International Conference on Automatic Face and Gesture Recognition, pp. 200-205 (1998).

[3] Deshpande, A. (2017). A Beginner's Guide To Understanding Convolutional Neural Networks. Adeshpande3.github.io. Retrieved 21 April 2017, from https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/

[4] Deshpande, A. (2017). A Beginner's Guide To Understanding Convolutional Neural Networks Part 2. Adeshpande3.github.io. Retrieved 21 April 2017, from https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/

[5] Rosebrock, A. (2017). LeNet - Convolutional Neural Network in Python - PyImageSearch. PyImageSearch. Retrieved 21 April 2017, from http://www.pyimagesearch.com/2016/08/01/lenet-convolutional-neural-network-in-python/

[6] Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. Journal of Machine Learning Research, 15(1), 1929-1958.

[7] Nair, V., & Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In Proceedings of the 27th international conference on machine learning (ICML-10) (pp. 807-814).

[8] Zeiler, M. D., & Fergus, R. (2014, September). Visualizing and understanding convolutional networks. In European conference on computer vision (pp. 818-833). Springer International Publishing.