6.867 Final Project: Detecting Facial Expressions using CNNs and Facial Landmarks

Vickie Ye and Alexandr Wang

Abstract

In this project, we compared different methods for facial expression recognition using images from a Kaggle dataset released as a part of an ICML-2013 workshop on representation learning. We found that classification using features extracted manually from facial images using principal component analysis yielded on average 40% classification accuracy. Using features extracted by facial landmark detection, we received on average 52% classification accuracy. However, when we used a convolutional neural network, we received 65% classification accuracy.

1 Introduction

In this project, we compared classification using features manually extracted from facial images against classification using a convolutional neural net, which learns the significant features of images through convolutional and pooling neural layers. The two manual feature extraction methods we explored were principal component analysis (PCA) and facial landmark detection.

1.1 PCA

In facial recognition, PCA is used in generating a low-dimensional representation of face images as linear combinations of the first k eigenfaces of training data. Eigenfaces are the eigenvectors of the training data's covariance matrix; the first eigenfaces are the vectors along which the training data shows the highest variance. Thus, we can express a facial image vector of thousands of pixels in terms of the linearly independent basis of the first k eigenvectors.

1.2 Facial Landmark Detection

1.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) are artificial neural networks that work particularly well for image data. The structure of CNNs exploits strong local correlation in the inputs. This is done by enforcing local connectivity between neurons of adjacent layers. The inputs of a hidden unit at a particular layer n are some locally-connected subset of the units of the previous layer $n-1$, done in such a way that the input to layer n represent some tile of the units of layer $n-1$, where all the tiles overlap.

In addition, CNNs utilize shared parameterizations (weight vector and bias) across layers. By constraining the same types of weights across layers, essentially replicating units across layers, allows for features to be detected regardless to their position in the initial input, making the network much more robust to real world image data. Additionally, by constraining multiple weights to be the same reduces the parameters to be learnt, increasing learning efficiency.

2 Experimental Details

2.1 Datasets

For this project, we primarily used a dataset released by Kaggle as a part of an ICML-2013 workshop in representation learning.

2.2 PCA and Facial Landmarks

2.3 Convolutional Neural Network

To implement our convolutional neural network, we used Google's TensorFlow neural network library. Our model

follows the architecture described in [3], with a few differences in the top few layers. In particular, we use fully connected layers instead of locally connected layers. The model described in [3] was designed for ImageNet's object classification, but it translated well to our problem of classifying facial expressions.

2.3.1 Network Structure

We implemented a deep CNN with five layers, which are from front to back: `conv1`, `conv2`, `local3`, `local4`, and `softmax_linear`. This structure is illustrated in Figure 1.

The layers `conv1` and `conv2` are convolution layers, which divides the image into overlapping tiles and use those as inputs to the next layer. In detail, the expression is

$$y(i,j) = \sum_{di,dj} x(s_1 i + di, s_2 j + dj) f(di, dj)$$

where $y(\cdot,\cdot)$ is the output, di and dj are changes which range over the tile size, s_1 and s_2 are strides which measure how much you move between each convolution window across each dimension, $x(\cdot,\cdot)$ is the input, and $f(\cdot,\cdot)$ are filters, which are the weights for the layer. We didn't employ any regularization on these convolution layers, and we used window sizes of 5 by 5 pixels with a strides of 1.

As illustrated in Figure 1, after each convolution layer, we performed a `pool1` operation and a `norm1` operation. The `pool1` operation is max pooling, which will transform the input to take the maxima in tiles similarly to the convolution step. In detail,

$$y(i,j) = \max_{di,dj} x(s_1 i + di, s_2 j + dj)$$

with each term defined as before. Note that this is not a layer because there are no weights. The `norm1` operation is a local response normalization response, where each input is normalized by dividing it by the squared sum of inputs within a certain radius of it. This technique is described in more detail in [3], and serves as form of "brightness normalization" on the input.

The `local3` and `local4` are fully connected layers with rectified linear activation according to $f(x) = \max(x, 0)$.

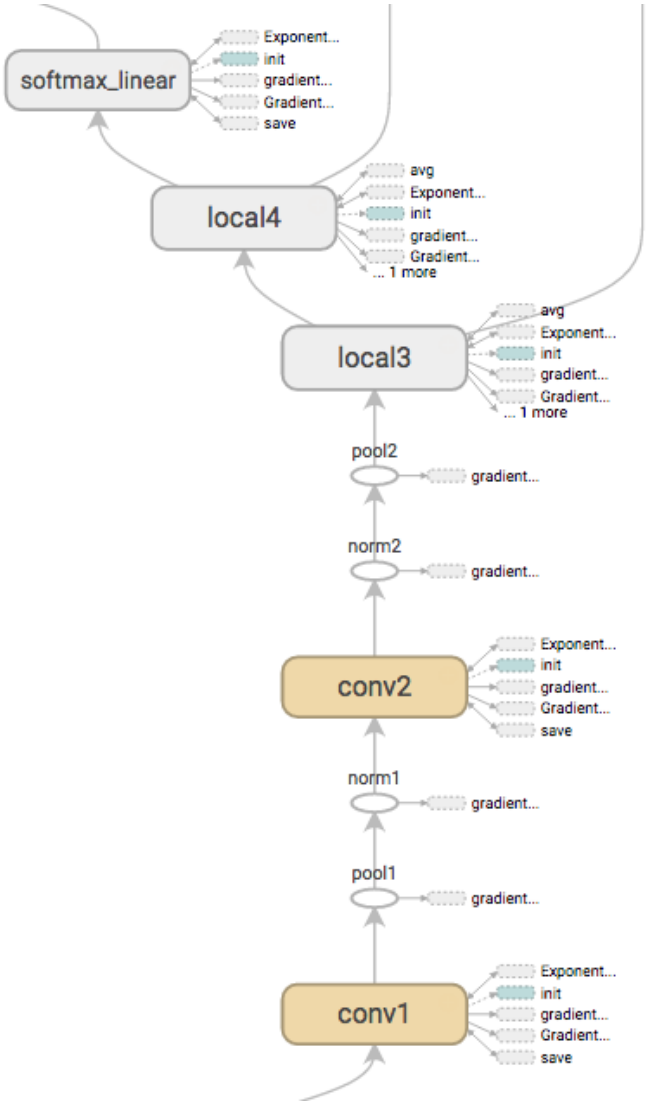

Figure 1: TensorFlow computation graph for the neural network in the CNN we implemented, illustrating each of the layers and computation done in between each layer.

We employ regularization on both of these errors by adding the L2-loss of weights on both layers to our total loss function.

Finally, the `softmax_linear` layer is linear transformation layer that produces logits according to a simple linear transformation based on learned weights and biases.

2.3.2 Training

The loss function we minimized was primarily the cross entropy between the multinomial distribution, formed by the normalized logits outputted by our CNN, and the probability distribution that is 1 for the correct class and 0 for all other classes. The expression for cross entropy between two probability distributions p, q is

$$H(p, q) = E_p[-\log q] = -\sum_x p(x) \log q(x)$$

which measures the divergence between the two discrete probability distributions. In addition, we added standard weight decay, i.e. L2-loss, for the weights of the `local3` and `local4` layers for weight regularization.

For training, we fed the model randomly distorted training examples (see subsection 2.3.3) in randomly shuffled batches of 128 samples each. For each of these examples, we would determine our loss based on our current inference. After each batch, we performed back propagation using gradient descent to learn the weights, and we called each time we finished a batch a step.

Since distorting inputs on disk is computationally expensive and bottlenecks the process, we spawn 16 threads which each asynchronously generated distorted inputs and add them to a queue which the training process pulls from.

One technique which improved model accuracy by 3% when implemented was using moving averages of the learned variables. During training, we calculate the moving averages of all learned variables, and during model evaluation, we substitute all learned parameters with the moving averages instead.

2.3.3 Minimizing Overfitting

Since the size of our training set was 28,709 labeled examples, and our neural net needed to go through many more than 28,000 examples to achieve convergence, we employed a few techniques to ensure that our convolutional neural net did not overfit to the training set.

First off, we implemented learning rate decay on our convolutional neural net, so that the learning rate decreased as it had been trained through more examples. We used exponential decay, so that that the learning rate decayed by a factor of 0.1 after training through $1,200,000$ examples, and we had it decay in a step change manner as is visible in Figure 2. We found that the step change learning rate decay worked better than a continuous exponential decay. This could be because maintaining a high learning rate initially could ensure that the CNN was trained towards a good local optimum in fewer steps, whereas a steadily decreasing learning rate could limit the range of the neural network.

In addition, we implemented distortion of the images while training to artificially increase the size of our training set, and make our convolutional neural net more robust to slightly distorted inputs. We processed our images in a few different ways. We would crop our 48 by 48 pixel images to a 40 by 40 pixel box, centrally for model evaluation and randomly for training. Then we approximately whiten the photos, scaling the pixel values linearly so that they pixel values have a mean of 0 and a standard deviation of 1, essentially normalizing them. This ensures consistent inputs to our neural network.

For training in particular, we would do a few more distortions to artificially increase our training set. We randomly flipped the image from left to right, randomly distorted the image brightness, and randomly distorted the image contrast for each image. These changes to randomly distort the images greatly improved the performance of the model, raising the correct classification rate from 45.4% to 53.8% when initially implemented.

3 Results and Analysis

3.1 Facial Landmarks

3.2 Convolutional Neural Network

We trained our model on Google Cloud Compute server with 4 CPUs and 15GB RAM for 20 hours using a training set of 28,709 labeled examples, and tested against a test set of 7,178 labeled examples. The model ran for 24,000 steps within those 20 hours, being fed a total of $3,072,000$ distorted training examples.

Our model achieved a correct classification rate of 65.2% on the 7-way classification problem, which is comparable to the 69% correct classification rate of the winning model on Kaggle. Our model correctly determines the correct label within the two highest logits with 82.3% accuracy, and determines the correct label within the three highest logits with 90.5% accuracy.

We can see how the total loss and the loss across the layers changed along with the learning rate in Figure 2. There are a couple interesting patterns. First, the weight decay loss on the `local3` and `local4` layers fell very quickly (by the 5000th step) and did not decrease past that, while still decreasing the cross entropy loss. The `local4` loss even steadily.

In addition, the total loss, and in particular the cross entropy loss, dropped dramatically once the learning rate dropped according to the exponential learning rate decay schedule. This makes sense, since it allows the neural net to much more finely optimize the weights on the space it settled in with the larger learning rate. Finally, at the extremely small learning rate of 0.001, the loss converged at around 0.5, and any changes in the model at this point did not increase the classification success on the test set.

We can also see the change in the parameters of the model, weights and biases, and how they changed over time in Figure 3.

3.3 Comparison between the two

References

[1] Lawrence, S.; Giles, L.; Tsoi, A. C.; Back, A. D. (1997) "Face Recognition: A Convolutional Neural-Network Approach" *Neural Networks, IEEE transactions on* 8 (1):98-113

[2] Matsugu, M.; Mori, K.; Mitari Y.; Kaneda Y. (2003) "Subject independent facial expression recognition with robust face detection using a convolutional neural network" *Neural Networks* 16 (5):555-559

[3] Krizhevsky, A.; Sutskever; Hinton, I.; Hinton, G. (2012) "ImageNet Classification with Deep Convolutional Neural Networks" *Advances in Neural Information Processing Systems* 25 (1):1097-1105

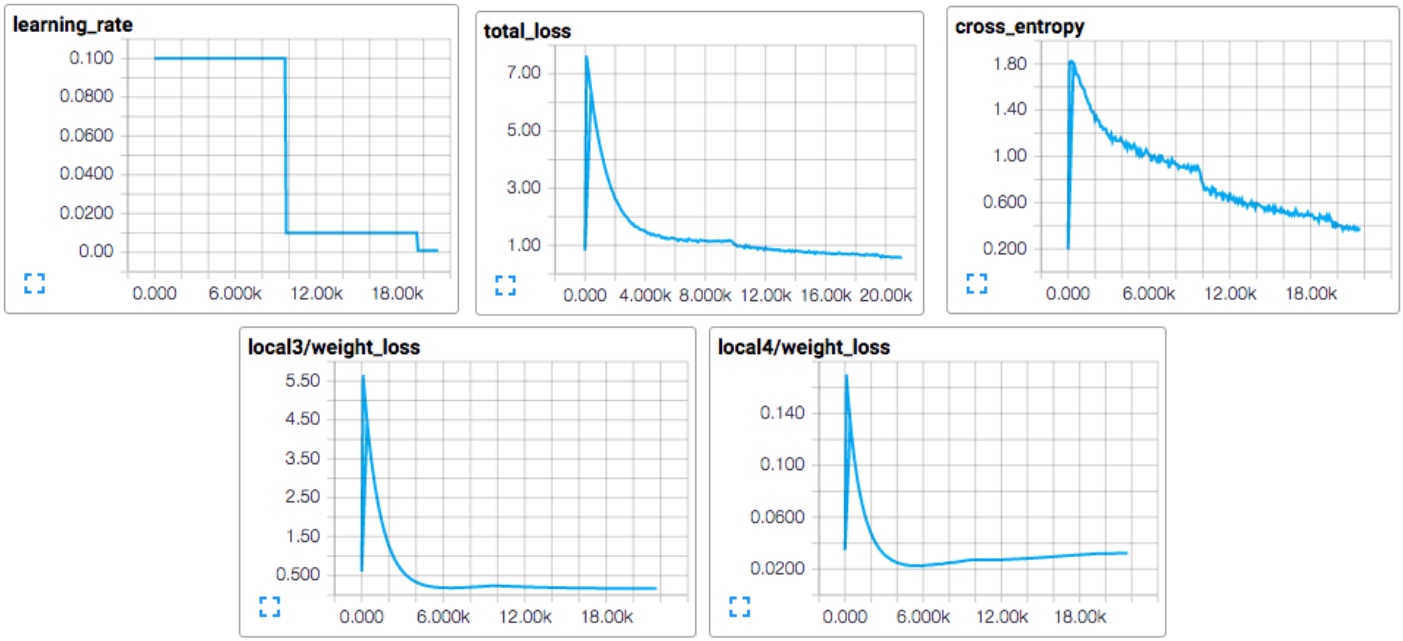

Figure 2: The total loss, cross entropy, and learning rate of the convolutional neural net over time. The x-axis is the step, and the y-axis is the value.

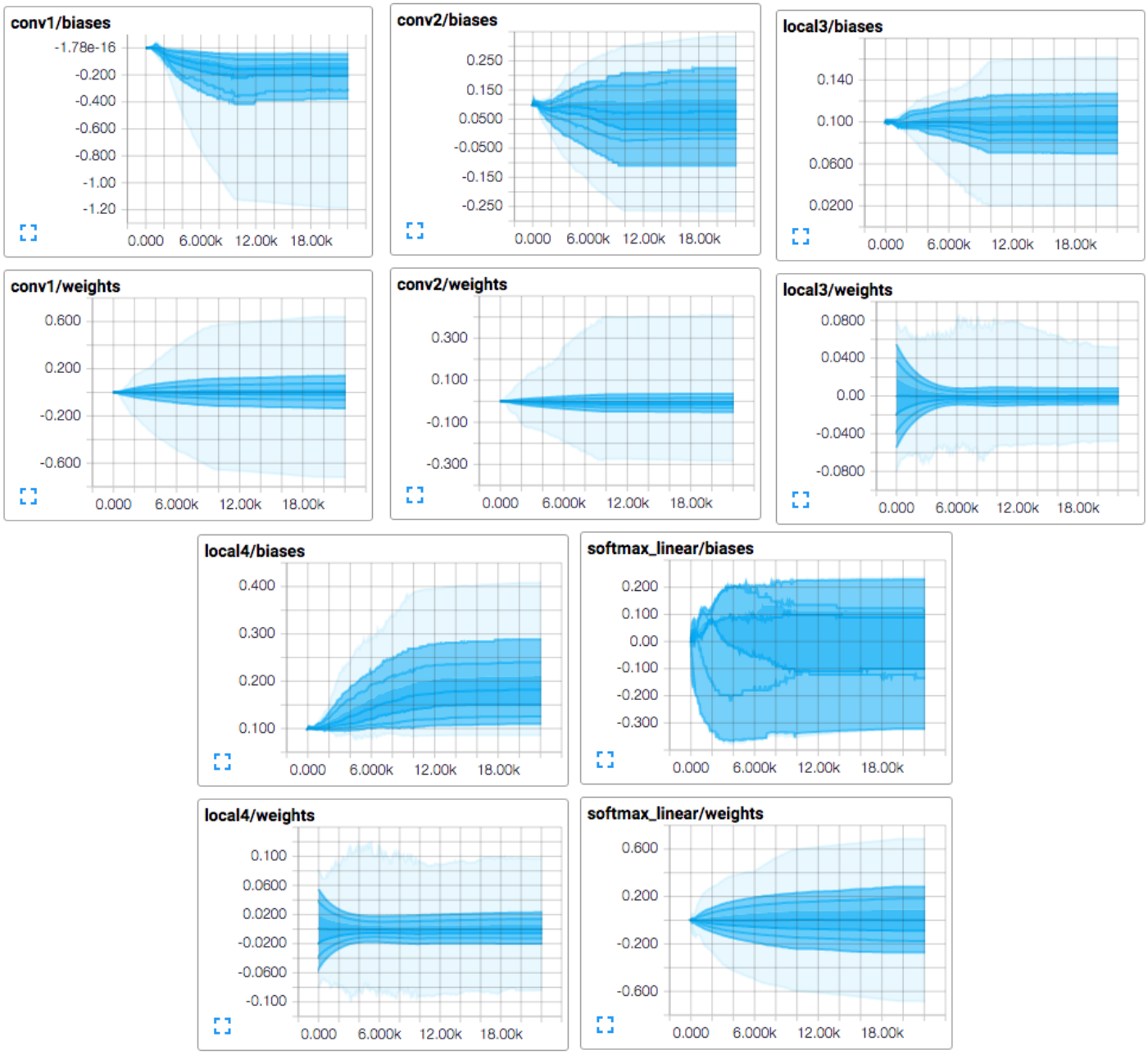

Figure 3: Histograms of the weights and biases of the layers of the CNN over time. The x-axis is the step, and the y-axis is the value.