Deep Feature Learning

James O'Reilly
james.oreilly@student.kuleuven.be

Introduction

In this assignment I discuss different methods for learning features from and classifying image data. I will first discuss principle component analysis, before briefly discussing stacked autoencoders, and then finally I will discuss convolutional neural networks using the AlexNet architecture as an example.

Principal Component Analysis

Principal component analysis is a dimensionality reduction methods which projects data from a higher dimension onto a lower dimension subspace such that the maximal amount of variability in the data is retained. The idea is that the low dimension representation will capture the structure of the data and the most salient features. A higher dimensional representation can then be reconstructed (imperfectly) by projecting from this lower dimensional subspace. I don't have space to full explain the mathematics behind PCA here, or the steps involved in the PCA algorithm, so I will include a link to a detailed review by Jolliffe.[1]

PCA for Random and Correlated Data

Here we compare the reduction of random and highly-correlated data. Intuitively, with randomly generated data, the distribution of eigenvalues should follow a Gaussian distribution, and so it is unlikely that the majority of variance in the dataset will be contained in only a small fraction of the dimensions. A 50×500 matrix of Gaussian random numbers was generated and PCA was used to try to reduce the dimensions. From this reduced representation, we tried to reconstruct the original dataset and measured the error of this reconstruction. The error is calculated as the RMSE between the original and reconstructed dataset. The percentage of variance captured in the lower dimensional representation was also calculated. Figure 1 shows their relationship.

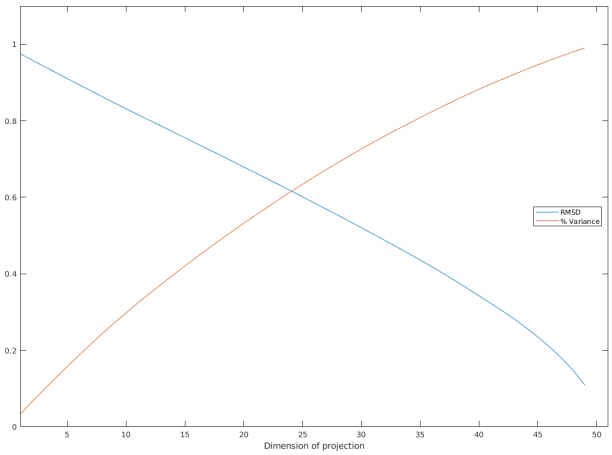

Figure 1: Caption

The same procedure was also performed on the p-component of the `choles_all` data, which is a 21×264 matrix. This data is highly correlated. Plots for the reconstruction error and the percentage of total variance contained in the lower dimensional representation are given in Figure 2. Note that the reconstruction error only

marginally improves after the dimensionality of the subspaces increases past 1. This is due to the fact that over 95% of the variance in the dataset is contained in this single dimension, as can be seen in the second plot.

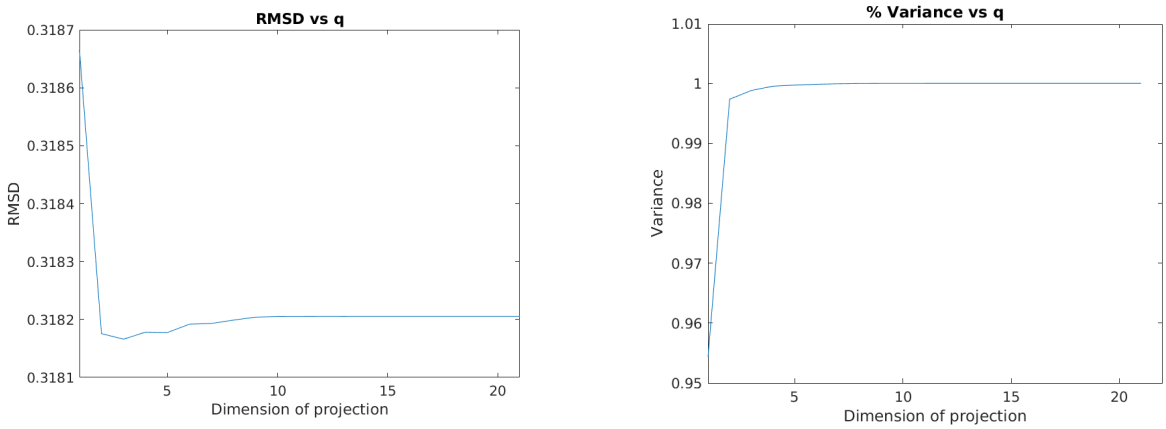

Figure 2: RMSD and variance for the `choles_all` vs dimensionality of subspace.

Principal Component Analysis on Hand-written Digits

Here we perform PCA on handwritten images of the digit '3' taken from the US Postal Service database. Displayed in Figure 3 is the mean '3', along with the reconstructed 3's from dimensionality reduction to one, two, three, and four dimensions.

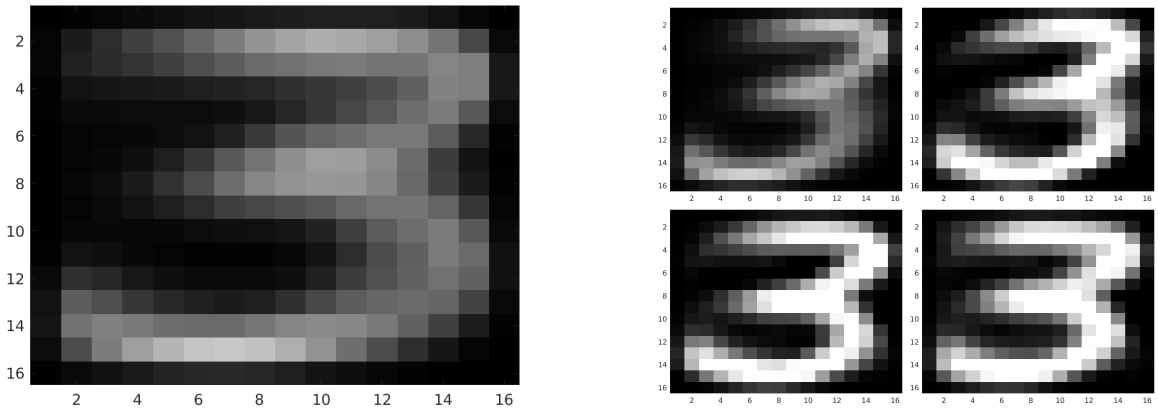

Figure 3: The mean '3' is given on the left. On the right are the reconstructions from one, two, three, and four dimensions (top left through bottom right).

The entire dataset was then compressed onto q principal components for $q = 1, \ldots, 50$ and the RMSE was plotted. The percentage of total variance was also plotted. The inverse relationship between these two values is clear in Figure 4.

The previous graph should illustrate an important fact – the squared reconstruction error induced by not using a certain principal component is proportional to its eigenvalue (eigenvalues are a measure of variance). That is why if the eigenvalues fall off quickly then projecting onto the first few components gives very small errors because the sum of the eigenvalues that we are not using is not very large.

Stacked Autoencoders

In this section we compare the performance of a stacked autoencoder to a normal mulltilayer neural network. The default stacked autoencoder (with which I will be comparing both the multilayer network and other stacked autoencoders) consists of two autoencoders and a softmax layer. The first autoencoder has 100 hidden units and is trained for 400 epochs. The second autoencoder has 50 hidden units and is trained for 100 epochs. The

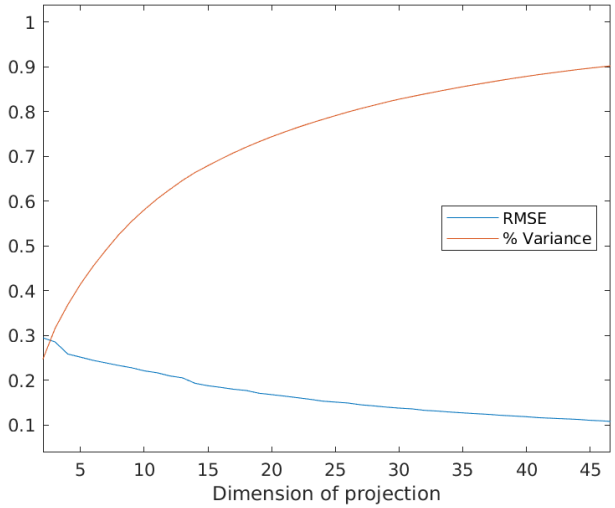

Figure 4: RMSE and % of total variance for the first 50 eigenvalues of the `threes` dataset. Note the inverse relationship between their slopes and values.

default stacked autoencoder has a classification accuracy of 85%. I will first try to optimise the hyperparameters of this stacked autoencoder (number of epochs, number of hidden units, number of layers).

Naturally, time is one major constraint, and so I first observed that the first autoencoder barely improves its performance after 250 epochs. Setting the number of epochs to 250 affords me some extra time which can be used elsewhere. I then observed that the second autoencoder was still improving at 100 epochs. I therefore increased the number of epochs to 200. These two changes increased the classification accuracy to 96 percent (without fine-tuning). Next I looked at changing the number of neurons in each layer. Changing the number of neurons in each layer affects the dimensionality of the low-dimension representation formed by the autoencoder. Increasing the number of hidden units in either the first of second layer had little impact on the classification accuracy. I then decreased the neurons in each layer to see if a lower-dimensional representation could still yield the same classification accuracy. Reducing the number of hidden units in the second autoencoder to 25 significantly impacted classification accuracy, reducing it to 77%. With this information, I was content with 100 hidden units in the first autencoder and 50 in the second.

Lastly, I needed to analyse the impact of additional layers on performance. Introducing a third autoencoder raises a number of questions, the primary question being how should I change the dimensionality of the other autoencoders in the network. I firstly tested adding a third autoencoder with 30 hidden units. Doing so reduced the classification accuracy. I then experimented with three autoencoders, with 150, 100, and 50 hidden units respectively. Doing this reduced the classification accuracy to less than 50%. Autoencoders with many hidden layers seldom work well because if the initialisation weights are too large, backpropogation finds a poor local minima. If the initialisation weights are too small, it takes a long time to optimise. If the initial weights are close to the solution however, then finetuning with backprop works well, however. Good initialisation weights can be learned using restricted Boltzmann machines.

The final architecture consists of two autoencoders and a softmax layer. The first autoencoder had 100 hidden units and was trained for 250 epochs. The second had 50 hidden units and was trained for 200 epochs. After fine-tuning, the network has a classification accuracy of 99.2%. The normal feedforward neural network, with two layers and the same number of neurons in each layer as the stacked autoencoder, had a classification accuracy of 95%.

Finetuning

The results for the stacked neural network can be improved by performing backpropagation on the whole multilayer network. This process is often referred to as fine-tuning. You fine-tune the network by retraining it on the training data in a supervised fashion. Why does this improve performance? Before finetuning, each of the autoencoders have been pre-trained individually: the weights for layer one and layer two and first learned and then frozen. In practice, initialising the weights in this manner has been shown to decrease training times and

start the fine-tuning stage much closer to good minima than random weight initialisation. The global fine-tuning uses backpropagation through the whole autoencoder to fine-tune the weights for optimal reconstruction.

Convolutional Neural Networks

In this section, a convolutional neural network (CNN) is trained on the Caltech 101 dataset for feature extraction. The network uses the AlexNet architecture, with five convolutional layers, each followed by a ReLu layer and a max-pooling layer.[2] Following the these layers are three fully-connected layers. A convolution is the simple application of a filter to an input that results in an activation. Repeated application of the same filter to an input results in a map of activations called a feature map, indicating the locations and strength of a detected feature in an input, such as an image.

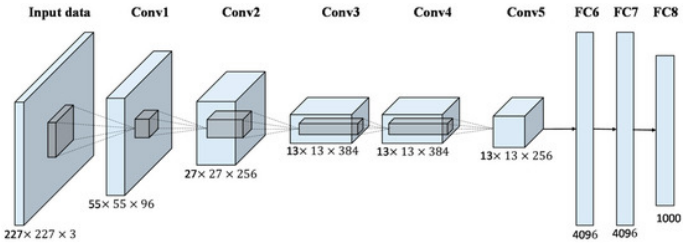

Figure 5: A simplified schema of the AlexNet architecture.

What do the weights in the first convolutional layer represent?

A visualisation of the weights in the first convolutional layer is given in Figure 6. This figure shows the 96 convolutional kernels learned by the network. Many of these kernels are frequency and orientation selective, selecting for edges in different orientations at different frequencies. Note also that some of the kernels select for edges with specific colours, or instead select for coloured blobs. Originally I was confused as to why this was only the case for some of the kernels. However, according to the paper on which this architecture was based, this network was trained with two different GPUs. The kernels on GPU1 are colour-agnostic, while the kernels on GPU2 are largely colour-specific. It is for this reason that only some of the kernels contain information about colour, and not due to the actual architecture of the network itself.

Inspect layers 1 to 5. If you know that a ReLU and a Cross Channel Normalization layer do not affect the dimension of the input, what is the dimension of the input at the start of layer 6 and why?

There is a simple formula for calculating the dimension of output from a convolutional or max pooling layer. The dimension is given by:

$$\text{output width} = \frac{W - F_w + 2P}{S_w} + 1$$

$$\text{output height} = \frac{H - F_h + 2P}{S_h} + 1$$

where W amd H are the width and height of the input, respectively. F is the size of the filter, P is the padding, and S is the stride size. The dimension of the original input into the network are $227 \times 227 \times 3$. The first convolution layer uses 96 kernels of size 11×11 with a stride of 4 and a padding of 0. As the inputs have the same width and height, we can use the same formula for both:

$$\frac{227 - 11 + 2(0)}{4} + 1 = 55$$

The output of the first convolution layer is therefore $55 \times 55 \times 96$. As both the ReLU layer and cross channel normalisation layer do not affect the dimension of the input, we next need to look at the max pooling layer. The same formula can be applied, giving

$$\frac{55 - 3 + 2(0)}{2} + 1 = 27$$

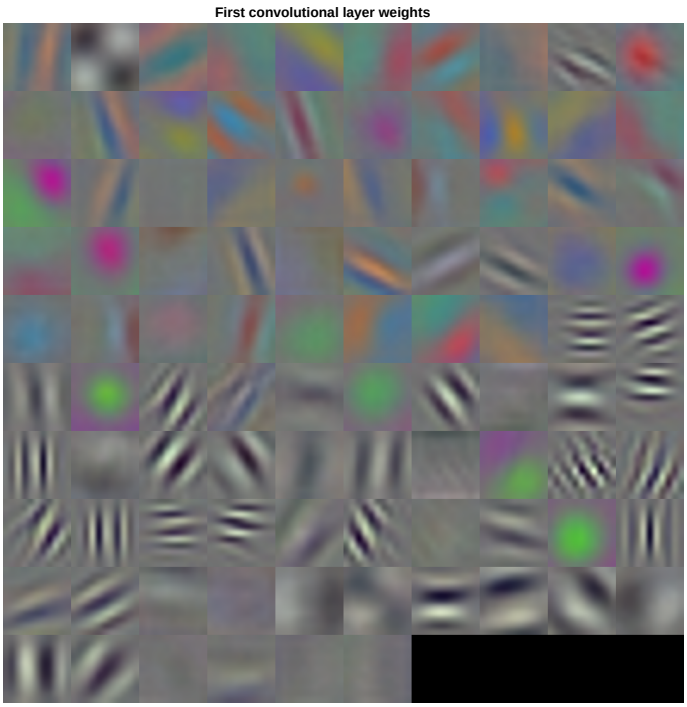

Figure 6: 96 convolutional kernels of size $11 \times 11 \times 3$ learned by the first convolutional layer on the $224 \times 224 \times 3$ input images.

The input to the second convolutional layer is therefore $27 \times 27 \times 96$.

What is the dimension of the inputs before the final classification part of the network (i.e. before the fully connected layers)? How does this compare with the initial dimension?

Applying the same formula for each of the convolution and max-pooling layers, we find the dimension of the inputs to the first fully-connected layer is $6 \times 6 \times 256$. The initial dimension was $227 \times 227 \times 3$. If we compute the number of actual input neurons in each case, there are 154587 initial input neurons, and 9216 input neurons to the first fully-connected layer. Through the convolution and max-pooling layers, the input size has been down-sampled by almost a factor of 15.

Briefly discuss the advantage of CNNs over fully connected networks for image classification.

The three primary advantages of CNNs for image classification come from local connectivity, parameter sharing, and translation invariance:

- **Local Connectivity:** Convolutional neural networks exploit spatially local correlation by enforcing a sparse local connectivity pattern between neurons of adjacent layers – each neuron is connected to only a small fraction of the input neurons. As each of the neurons are only locally connected, they can take advantage of local spatial coherence of images. This allows for a significant reduction in the number of operations needed to process an image. If the network was fully connected, there would likely be problems with both storage and computing the linear activations of hidden units.
- *Parameter Sharing:* This is the idea that units in the same convolution kernel share some parameters across layers. This further reduces the effective number of parameters, and therefore improves computational efficiency.
- **Translation Invariance:** Translation invariance (aka shift invariance) in a CNN refers to the fact that the CNN's output will be the same regardless of the position of the object in the image – this is useful if you just want to classify the image and don't care about the position. This is achieved in two ways. Firstly, due to parameter sharing, the weights of the filter are shared across patches of the image, and so the weights learnt will be invariant to position. Secondly, by performing max pooling we approximate translation invariance since subsequent layers of the CNN don't care about the specific position in the patch that the max value was in.

References

[1] Jolliffe, I. T., and Cadima, J. Principal component analysis: a review and recent developments. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 374*, 2065 (2016), 20150202.

[2] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (2012), pp. 1097–1105.