Autoencoders

8

8.1 Learning Representations

In this and the next chapter we turn our attention to unsupervised deep learning, also known as learning distributed representations or representation learning. But first we need to fill in a blank we had from Chap. 3. There we discussed PCA as a form of learning distributed representations, and formulated the problem as finding $Z = XQ$, where all features have been decorrelated. Here we will calculate the matrix Q. We will need to have a *covariance matrix* of X. The covariance matrix of a given matrix shows the entries of the original matrix. The covariance of two random variables X and Y is defined as $COV(X, Y) := \mathbb{E}((X - \mathbb{E}(X))(Y - \mathbb{E}(Y)))$ and show how two random variables change together. Remember that with a bit of hand waving everything relating to data can be thought of as a random variable. Also, with a bit more of hand waving, for a random variable X we may think of $\mathbb{E}(X) = MEAN(X)$.[1] This will only hold if the distribution of X is uniform, but it can be helpful from a practical perspective even when it is not, especially since in machine learning we will probably have some optimization somewhere so we can be a bit sloppy.

The attentive reader may notice that $\mathbb{E}(X)$ was actually a vector, while $MEAN(X)$ is a single value, but we will use something called *broadcasting* to make it right again. Broadcasting a value v into an n-dimensional vector $\mathbf{v}$ means simply to put the same v in every component of $\mathbf{v}$, or simply:

$$broadcast(v, n) = \underbrace{(v, v, v, \ldots, v)}_{n} \tag{8.1}$$

[1] The expected value is actually the weighted sum, which can be calculated from a frequency table. If 3 out of five students got the grade '5', and the other two got a grade '3', $\mathbb{E}(X) = 0.6 \cdot 5 + 0.4 \cdot 3$.

S. Skansi, *Introduction to Deep Learning*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-319-73004-2_8

We will denote the covariance matrix of the matrix X as $\Xi(X)$. This is not a standard notation, but (unlike the standard notation C or Σ) this notation will avoid confusion, since we are using the standard notations in a different sense in this book. To address the covariance matrix more formally, if we have a column vector $\mathbf{X} = (X_1, X_2, \ldots, X_d)^\top$ populated with random variables, the covariance matrix Ξ_X (which can also be denoted as Ξ_{ij}) can be defined as $\Xi_{ij} = COV(X_i, X_j) = \mathbb{E}((X_i - \mathbb{E}(X_i))(X_j - \mathbb{E}(X_j)))$, or if we write the whole $d \times d$ matrix:

$$\Xi_{\mathbf{X}} = \begin{bmatrix} \mathbb{E}((X_1 - \mathbb{E}(X_1))(X_1 - \mathbb{E}(X_1))) & \cdots & \mathbb{E}((X_1 - \mathbb{E}(X_1))(X_d - \mathbb{E}(X_d))) \\ \mathbb{E}((X_2 - \mathbb{E}(X_2))(X_1 - \mathbb{E}(X_1))) & \cdots & \mathbb{E}((X_2 - \mathbb{E}(X_2))(X_d - \mathbb{E}(X_d))) \\ \vdots & \ddots & \vdots \\ \mathbb{E}((X_d - \mathbb{E}(X_d))(X_1 - \mathbb{E}(X_1))) & \cdots & \mathbb{E}((X_d - \mathbb{E}(X_d))(X_d - \mathbb{E}(X_d))) \end{bmatrix} \tag{8.2}$$

It should now be clear that the covariance matrix actually measures 'self'-covariance, i.e. covariance between its own elements. Let us see what properties does a matrix $\Xi(X)$ have. First, it must be symmetric, since the covariance of X with Y is the same as the covariance of Y with X. $\Xi(X)$ is also a *positive-definite matrix*, which means that the scalar $v^\top Xz$ is positive for every non-zero vector v.

Let us turn to a slightly different topic, *eigenvectors*. Eigenvectors of a $d \times d$ matrix A are vectors whose *direction* does not change (but the length does) when they are multiplied by A. It can be proved that there are exactly d of them. How to find the eigenvectors is the hard part, and there are number of approaches, and one of the more popular ones is gradient descent. Since all numerical libraries can find eigenvectors for us, we will not go into details.

So the eigenvectors when multiplied by a matrix A do not change direction, only the length. It is common practice to normalize the eigenvectors and denote them by $\mathbf{v}_i$. This change of length is called the *eigenvalue*, usually denoted by λ_i. This actually gives rise to a fundamental property of eigenvectors and eigenvalues of a matrix, namely $A\mathbf{v}_i = \lambda_i \mathbf{v}_i$

Once we have the $\mathbf{v}$s and λs, we start by arranging the lambdas in descending order:

$$\lambda_1 > \lambda_2 > \ldots > \lambda_d$$

This also creates an arrangement in the corresponding eigenvectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_d$ (note that each of them is of the form $\mathbf{v}_i = (v_i^{(1)}, v_i^{(2)}, \ldots, v_i^{(d)})$, $1 \leq i \leq d$) since there is a one to one correspondence between them and the eigenvalues, so we can simply 'copy' the order of the eigenvalues on the eigenvectors. We create a $d \times d$ matrix with the eigenvectors as columns which are sorted with ordering of the the corresponding eigenvalues (in the last step we are simply renaming the entries to

follow the usual matric entry naming conventions):

$$V = (\mathbf{v}_1^\top, \mathbf{v}_2^\top, \ldots, \mathbf{v}_d^\top) = \begin{bmatrix} v_1^{(1)} & v_2^{(1)} & \cdots & v_d^{(1)} \\ v_1^{(2)} & v_2^{(2)} & \cdots & v_d^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ v_1^{(d)} & v_2^{(d)} & \cdots & v_d^{(d)} \end{bmatrix} = \begin{bmatrix} v_{11} & v_{12} & \cdots & v_{1d} \\ v_{21} & v_{22} & \cdots & v_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ v_{d1} & v_{d2} & \cdots & v_{dd} \end{bmatrix}$$

We now create a blank matrix of zeros (size $d \times d$) and put the lambdas in descending order on the diagonal. We call this matrix Λ:

$$V = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_d \end{bmatrix}$$

With this, we turn to the *eigendecomposition of a matrix*. We need to have a symmetric matrix A and then its eigendecomposition is:

$$A = V \Lambda V^{-1} \tag{8.3}$$

The only condition is that all eigenvectors $\mathbf{v}_i$ are linearly independent. Since Ξ is a symmetrical matrix with linearly independent eigenvectors, we can use the eigendecomposition to get the following equations which hold for any covariance matrix Ξ:

$$\Xi = V \Lambda V^{-1} \tag{8.4}$$

$$\Xi V = V \Lambda \tag{8.5}$$

Since V is orthonormal,[2] we also have $V^\top V = I$. Now we are ready to return to $Z = XQ$. Let us take a look at the transformed data Z. We can express the covariance of Z as the covariance of X multiplied by Q:

$$\Xi_Z = \frac{1}{d}((Z - MEAN(Z))^\top (Z - MEAN(Z))) = \tag{8.6}$$

$$= \frac{1}{d}((XQ - MEAN(X)Q)^\top (XQ - MEAN(X)Q)) = \tag{8.7}$$

$$= \frac{1}{d} Q^\top (X - MEAN(X))^\top (X - MEAN(X)) Q = \tag{8.8}$$

$$= Q^\top \Xi_X Q \tag{8.9}$$

[2] We omit the proof but it can be found in any linear algebra textbook, such as e.g. [1].

We now have to choose a matrix Q so that we get what we want (correlation zero and features ordered according to variance). We simply chose $Q := V$. Then we have:

$$\Xi_Z = V^\top \Xi_X V = V^\top V \Lambda = \Lambda \tag{8.10}$$

Let us see what we have achieved. All elements except the diagonal elements of Ξ_Z are zero, which means that the only correlation left in Z is along the diagonal. This is the covariance of a variable with itself, which is actually the variance we have encountered earlier, and the matrix is ordered in descending variance ($VAR(X_i) = COV(X_i, X_i) = \lambda_i$). This is everything we wanted. Note that we have done PCA for the 2D case with matrices but the same ideas hold for tensors. More on the principal component analysis can be found in [2].

So we have seen how we can create a different representation of the same data such that the features it is described with have a covariance of zero, and are sorted by variance. In doing so we have created a distributed representation of the data, since a column named 'height' does not exist anymore, and we have synthetic columns. The point here is that we can build various distributed representations, but we have to know what constraint we want the final data to obey. If we want this constraint to be left unspecified and we want to specify it not directly but by feeding examples, then we will have to employ a more general approach. This is the approach that leads us to autoencoders, which offer a surprising generality across many tasks.

8.2 Different Autoencoder Architectures

An autoencoder is a three-layered feed-forward neural network. They have one peculiarity: the targets $\mathbf{t}$ are actually the same values as inputs $\mathbf{x}$, which means that the task of the autoencoder is simply to recreate the inputs. So autoencoders are a form of unsupervised learning. This entails that the output layer has to have the same number of neurons as the input layer. This is all that is needed for a feed-forward neural network to be called an *autoencoder*. We can call this version the 'plain vanilla autoencoder'. There is a problem right away for plain vanilla autoencoders. If there are at least as many neurons in the hidden layer layer as there are in the input and output layer, the autoencoder is in danger of learning the identity function. This leads to a constraint, namely that there have to be less neurons in the hidden layer than in the input and output layers. We can call autoencoders which satisfy this property *simple autoencoders*. The outputs of the hidden layer of a fully trained autoencoder constitute a distributed representation, similar to PCA, and, as with PCA, this representation can be fed to a logistic regression or a simple feed-forward neural network as input and it will produce much better results than the regular representation.

But we can take another path, which is called *sparse autoencoders*. Let us say we constrain the number of neurons on the hidden layers to be at most double the number of neurons in the input layer, but we add a heavy dropout of e.g. 0.7. Then, we will have for each iteration less hidden neurons than input neurons, but at the same

time we will produce a large hidden layer vector. This large hidden layer vector is a (very large) distributed representation. What is happening here intuitively speaking is that simple autoencoders make a compact distributed representation, which is a different representation of the input. This makes it more easy for a simple neural network to digest it and process it, resulting in higher accuracy. Sparse autoencoders digest the inputs in the same way, but in addition, they learn redundancies and offer a more 'dilluted' and bigger vector, which is even simpler to process well. Recall how the hyperplane works in multiple dimensions and this will make sense. There is a different way to define sparse autoencoders, via a sparsity rate, which forces the activations below a certain threshold to be considered zero, it is similar to our approach.

We can also make the autoencoder's job harder, by inserting some noise into the input. This is done by creating a copy of the input with inserted random numbers at a fixed amount, e.g. on randomly chosen 10% of the input. The targets are a copy of the inputs without noise. These autoencoders are called *denoising autoencoders*. If we add explicit regularization, we obtain a flavour of autoencoders known as *contractive autoencoders*. Figure 8.1 offers an illustration of the various types of autoencoders. There are many other types of autoencoders, but they are more complex and fall outside the scope of this book. We point the interested reader to [3].

All of the autoencoders are used to preprocess data for a simple feed-forward neural network. This means that we have to get the preprocessed data from the autoencoder. This data is not the output of the whole autoencoder, but the output of the middle (hidden) layer, which is the layer that does the donkey work.

Let us address a technical issue. We have seen but not formally introduced the concept of a *latent variable*. A latent variable is a variable which lies in the background and is correlated with one or many 'visible' variables. We have seen an example in Chap. 3 when we addressed PCA in an informal manner, and we had synthetic properties behind 'height' and 'weight'. These are a prime example of a latent variable. When we hypothesize a latent variable (or create it), we postulate we have a probability distribution to define it. Note that it is a philosophical question whether we *discover* or *define* latent variables, but it is clear that we want our latent variables (the defined ones) to follow as closely as possible the latent variables in nature (the ones that we measure or discover). A distributed representation is a probability dis-

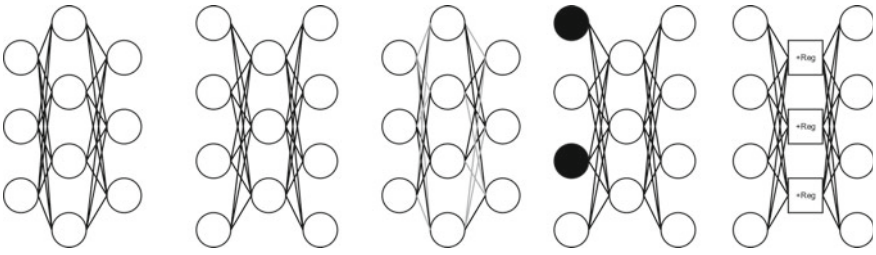

Fig. 8.1 Plain vanilla autoencoder, simple autoencoder, sparse autoencoder, denoising autoencoder, contractive autoencoder

tribution of latent variables which hopefully are the objective latent variables and learning will conclude when they are very similar. This means that we have to have a way of measuring similarities between probability distributions. This is usually done via the Kullback-Leibler divergence, which is defined as:

$$\mathbb{KL}(P, Q) := \sum_{n=1}^{N} P(n) \log \frac{P(n)}{Q(n)} \tag{8.11}$$

where P and Q are two probability distributions. Notice that $\mathbb{KL}(P, Q)$ is not symmetric (it will change if you change the P and Q). Traditionally, the Kullback-Liebler divergence is denoted as D_{KL}, but the notation we used is more consistent with the other notation in the book. There are a number of sources which provide more detail, but we will refer the reader to [3]. Autoencoders are a relatively old idea, and they were first proposed by Dana H. Ballard in 1987 [4]. Yann LeCun [5] also considered similar structures independently from Ballard. A good overview of the many types of autoencoders and their functionality can be found in [6] as an introduction to the stacked denoising autoencoders which we will reproduce in the next section.

8.3 Stacking Autoencoders

If autoencoders seem like LEGO bricks, you have the right intuition, and in fact they may be stacked together, and then they are called *stacked autoencoders*. But keep in mind that the real result of the autoencoder is not in the output layer, but the activations in the middle layer, which are then taken and used as inputs in a regular neural network. This means that to stack them we need not simply stick one autoencoder after the other, but actually combine their middle layers as shown in Fig. 8.2. Imagine that we have two simple autoencoders of size (13, 4, 13) and

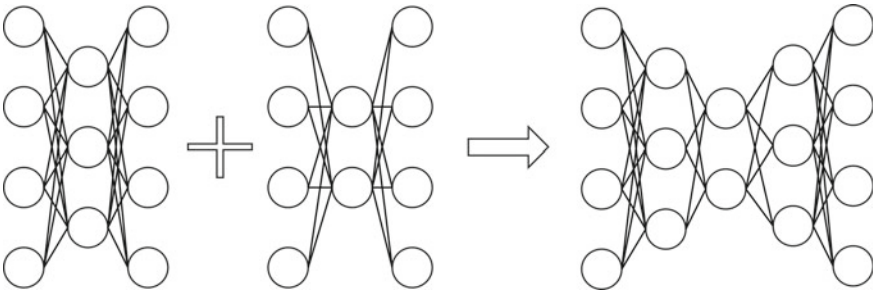

Fig. 8.2 Stacking a (4, 3, 4) and a (4, 2, 4) autoencoder resulting in a (4, 3, 2, 3, 4) stacked autoencoder

(13, 7, 13). Notice that if they want to process the same data they have to have the same input (and output) size. Only the middle layer or autoencoder architecture may vary. For simple autoencoders, they are stacked by creating a 13, 7, 4, 7, 13 stacked autoencoder. If you think back on what the autoencoder does, it makes sense to create a natural bottleneck. For other architectures, it may make sense to make a different arrangement. The real result of the stacked autoencoder is again the distributed representation built by the middle layer. We will be stacking denoising autoencoders following the approach of [6] and we present a modification of the code available at `https://blog.keras.io/building-autoencoders-in-keras.html`. The first part of the code, as always, consists of import statements:

```
from keras.layers import Input, Dense
from keras.models import Model
from keras.datasets import mnist
import numpy as np
(x_train, _), (x_test, _) = mnist.load_data()
```

The last line of code loads the MNIST dataset from the Keras repositories. You could do this by hand, but Keras has a built-in function that lets you load MNIST into Numpy[3] arrays. Note that the Keras function returns two pairs, one consists of train samples and train labels (both as Numpy arrays of 60000 rows), and the second consisting of test samples and test labels (again, Numpy arrays, but this time of 10000 rows). Since we do not need labels, we load them in the _ *anonymous* variable, which is basically a trash can, but we need it since the function needs to return two pairs and if we do not provide the necessary variables, the system will crash. So we accept the values and dump them in the variable _. The next part of the code preprocesses the MNIST data. We break it down in steps:

```
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
noise_rate = 0.05
```

This part of the code turns the original values ranging from 0 to 255 to values between 0 and 1, and declares their Numpy types as *float32* (decimal number with a precision of 32). It also introduces a noise rate parameter, which we will be needing shortly.

```
x_train_noisy = x_train + noise_rate * np.random.normal
(loc=0.0, scale=1.0, size=x_train.shape)
x_test_noisy = x_test + noise_rate * np.random.normal
(loc=0.0, scale=1.0, size=x_test.shape)
x_train_noisy = np.clip(x_train_noisy, 0.0, 1.0)
x_test_noisy = np.clip(x_test_noisy, 0.0, 1.0)
```

This part of the code introduces the noise into a copy of the data. Note that the `np.random.normal(loc=0.0, scale=1.0, size=x_train.shape)`

[3] Numpy is the Python library for handling arrays and fast numerical computations.

introduces a new array, of the size of the `x_train` array populated with a Gaussian random variable with `loc=0.0` (which is actually the mean), and a `scale=1.0` (which is the standard deviation). This is then multiplies with the noise rate and added to the data. The next two rows actually make sure that all the data is bound between 0 and 1 even after the addition. We can now reshape our arrays which are currently (60000, 28, 28) and (10000, 28, 28) into (60000, 784) and (10000, 784) respectively. We have touched upon this idea when we have first introduced MNIST, and now we can see the code in action:

```
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))
x_train_noisy = x_train_noisy.reshape((len(x_train_noisy), np.prod(x_train_noisy.shape[1:])))
x_test_noisy = x_test_noisy.reshape((len(x_test_noisy), np.prod(x_test_noisy.shape[1:])))
assert x_train_noisy.shape[1] == x_test_noisy.shape[1]
```

The first four rows reshape the four arrays we have, and the final row is a test to see whether the sizes of the noisy train and test vectors are the same. Since we are using autoencoders, this has to be the case. If they are somehow not the same, the whole program will crash here. It might seem strange to want to crash the program on purpose, but in this way we actually gain control, since we know where it has crashed, and by using as many tests as we can, we can quickly debug even very complex codes. This ends the preprocessing part of the code, and we continue to build the actual autoencoder:

```
inputs = Input(shape=(x_train_noisy.shape[1],))
encode1 = Dense(128, activation='relu')(inputs)
encode2 = Dense(64, activation='tanh')(encode1)
encode3 = Dense(32, activation='relu')(encode2)
decode3 = Dense(64, activation='relu')(encode3)
decode2 = Dense(128, activation='sigmoid')(decode3)
decode1 = Dense(x_train_noisy.shape[1], activation='relu')(decode2)
```

This offers a different view from what we are used to, since now we manually connect the layers (you can see the layer sizes, 128, 64, 32, 64, 128). We have added different activations just to show their names, but you can freely experiment with different combinations. What is important here to notice is that the input size and the output size are both equal to `x_train_noisy.shape[1]`. Once we have the layers specified, we continue to build the model (feel free to experiment with different optimizers[4] and error functions[5]):

```
autoencoder = Model(inputs, decode1)
autoencoder.compile(optimizer='sgd', loss='mean_squared_error',metrics=['accuracy'])
autoencoder.fit(x_train,x_train,epochs=5,batch_size=256,shuffle=True)
```

[4]Try'adam'.
[5]Try'binary_crossentropy'.

You should also increase the number of epochs once you get the code to work. Finally we get to the last part of the autoencoder code when we evaluate, predict and pull out the weight of the deepest middle layer. Note that when we print all the weight matrices, the right weight matrix (the result of the stacked autoencoder) is the first one where the dimensions start to increase (in our case (32, 64)):

```
metrics = autoencoder.evaluate(x_test_noisy, x_test, verbose=1)
print()
print("%s:%.2f%%" % (autoencoder.metrics_names[1], metrics[1]*100))
print()
results = autoencoder.predict(x_test)
all_AE_weights_shapes = [x.shape for x in autoencoder.get_weights()]
print(all_AE_weights_shapes)
ww=len(all_AE_weights_shapes)
deeply_encoded_MNIST_weight_matrix = autoencoder.get_weights()[int((ww/2))]
print(deeply_encoded_MNIST_weight_matrix.shape)
autoencoder.save_weights("all_AE_weights.h5")
```

The resulting weight matrix is stored in the variable `deeply_encoded_MNIST_weight_matrix`, which contains the trained weights for the middlemost layer of the stacked autoencoder, and this should afterwards be fed to a fully connected neural network together with the labels (the ones we dumped). This weight matrix is a distributed representation of the original dataset. A copy of all weights is also saved for later use in a H5 file. We have also added a variable `results` to make predictions with the autoencoder, but this is mainly used for assessing autoencoder quality, and not for actual predictions.

8.4 Recreating the Cat Paper

In this section, we recreate the idea presented in the famous 'cat paper', with the official title *Building High-level Features Using Large Scale Unsupervised Learning* [7]. We will present a simplification to better delineate the subtleties of this amazing paper. This paper became famous since the authors made a neural network which was capable of learning to recognize cats just by watching YouTube videos. But what does that mean? Let us take a step back. The 'watching' means simply that the authors sampled frames from 10 million YouTube videos, and took a number of 200 by 200 images in RGB. Now, the tricky part: what does it mean to 'recognize a cat'? Surely it could mean that they build a classifier which was trained on images of cats and then it classified cats. But the authors did not do this. They gave the network an unlabelled dataset, and then tested it against images of cats from ImageNet (negative samples were just random images not containing cats). The network was trained by learning to reconstruct inputs (it means that the number of output neurons is the same

as the number of input neurons), which makes it an autoencoder. Result neurons are found in the middle part of the autoencoder. The network had a number of result neurons (let us say there are 4 of them for simplicity), and they noticed that the activations of those neurons formed a pattern (activations are sigmoid so they range from 0 to 1). If the network was classifying something similar to what it has seen (cats), it formed a pattern, e.g. neuron 1 was 0.1, neuron 2 was 0.2, neuron 3 was 0.5 and neuron 4 was 0.2. If it got something it did not know about, neuron 1 would get 0.9, and the others 0. In this way, an implicit label generation was discovered.

But the cat paper presented another cool result. They asked the network what was in the videos, and the network drew the face of a cat (as the tech media formulated it). But what does that mean? It means that they took the best performing 'cat finder' neuron, in our case neuron 3, and found the top 5 images it recognized as cats. Suppose the cat finder neuron had activations of 0.94, 0.96, 0.97, 0.95 and 0.99 for them. They then combined and modified this image (with numerical optimization, similar to gradient descent) to find a new image such that given neuron gets the activation 1. Such image was a drawing of a cat face. It may seem like science fiction, but if you think about it, it is not that unusual. They picked the best cat recognizer neuron, and then selected top 5 images it was most confident of. It is easy to imagine that these were the clearest pictures of cat faces. It then combined them, added a little contrast, and there you have it—an image which produced the activation of 1 in that neuron. And it was an image of a cat different from any other image in the dataset. The neural network was set loose to watch YouTube videos of cats (without knowing it was looking at cats), and once prompted to answer what it was looking at, the network drew a picture of a cat.

We scaled down a bit, but the actual architecture used was immense: 16000 computer cores (your laptop has 2 or 4), and the network was trained over three days. The autoencoder had over 1 billion trainable parameters, which is still only a fraction of the number of synapses in the human visual cortex. The input images were a 200 by 200 by 3 tensors for training, and for testing 32 by 32 by 3. The authors used a receptive field of 18 by 18 similar to the convolutional networks, but the weights were not shared across the image but each 'tile' of the field had its own weights. The number of feature maps used was 8. After this, there was a pooling layer using L2 pooling. L2 pooling takes a region (e.g. 2 by 2) in the same way as max-pooling, but instead of outputting the max of the inputs, it squares all inputs, adds them, and then takes the square root of it and presents this as the output.

The overall autoencoder has three parts, all of them are of the same architecture. A part takes the input, applies the receptive field (no shared weights), and then applies L2 pooling, and finally a transformation known as local contrast normalization. After this part is finished, there are two more exactly the same. The whole network is trained with asynchronous SGD. This means that there are many SGDs working at once over different parts, and have a central weights repository. At the beginning of each phase, every SGD asks the repository for the update on weights, optimizes them a bit, and

then sends them back to the repository so that other instances running asynchronous SGD can use them. The minibatch size used was 100. We omit the rest of the details, and refer the reader to the original paper.

References

1. S. Axler, *Linear Algebra Done Right* (Springer, New York, 2015)
2. R. Vidal, Y. Ma, S. Sastry, *Generalized Principal Component Analysis* (Springer, London, 2016)
3. I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning* (MIT Press, Cambridge, 2016)
4. D.H. Ballard, Modular learning in neural networks, in *AAAI-87 Proceedings* (AAAI, 1987), pp. 279–284
5. Y. LeCun, *Modeles connexionnistes de l'apprentissage (Connectionist Learning Models)* (Université P. et M. Curie (Paris 6), 1987)
6. P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, P.-A. Manzagol, Stacked denoising autoencoders: learning useful representations in a deep network with a local denoising criterion. J. Mach. Learn. Res. **11**, 3371–3408 (2010)
7. Q.V. Le, M.A. Ranzato, R. Monga, M. Devin, K. Chen, G.S. Corrado, J. Dean, A.Y. Ng, Building high-level features using large scale unsupervised learning, in *Proceedings of the 29th International Conference on Machine Learning. ICML* (2012)