Visualizing Neural Networks 7

7.1 Introduction

A neural network is a method that transforms input data into a feature space through a highly nonlinear function. When a neural network is trained to classify input patterns, it learns a transformation function from input space to the feature space such that patterns from different classes become linearly separable. Then, it trains a linear classifier on this feature space in order to classify input patterns. The beauty of neural networks is that they simultaneously learn a feature transformation function as well as a linear classifier.

Another method is to design the feature transformation function by hand and train a linear or nonlinear classifiers to differentiate patterns in this space. Feature transformation functions such as histogram of oriented gradients and local binary pattern histograms are two of commonly used feature transformation functions. Understanding the underlying process of these functions is more trivial than a transformation function represented by a neural network.

For example, in the case of histogram of oriented gradients, if there are many strong vertical edges in an image we know that the bin related to vertical edges is going to be significantly bigger than other bins in the histogram. If a linear classifier is trained on top of these histograms and if the magnitude of weight of linear classifier related to the vertical bin is high, we can imply that vertical edges have a great impact on the classification score.

As it turns out from the above example, figuring out that how a pattern is classified using a linear classifier trained on top of histogram of oriented gradients is doable. Also, if an interpretable nonlinear classifier such as decision trees or random forest is trained on the histogram, it is still possible to explain how a pattern is classified using these methods.

© Springer International Publishing AG 2017

H. Habibi Aghdam and E. Jahani Heravi, *Guide to Convolutional Neural Networks*, DOI 10.1007/978-3-319-57550-6_7

The problem with deep neural networks is that it is hard or even impossible to inspect weights of neural networks and understand how the feature transformation function works. In other words, it is not trivial to know how a pattern with many strong vertical edges will be transformed into the feature space. Also, in contrast to histogram of oriented gradients where each axis in the feature spaces has an easy-to-understand meaning for human, axes of feature spaces represented by a neural network are not easily interpretable.

For these reasons, diagnosing neural networks and understanding the underlying process of a neural network are not possible. Visualization is a way to make sense of complex models such as neural networks. In Chap. 5, we showed a few data-oriented techniques for understanding the feature transformation and classification process of neural networks. In this chapter, we will briefly review these techniques again and introduce gradient-based visualization techniques.

7.2 Data-Oriented Techniques

In general, data-oriented visualization methods work by feeding images to a network and collecting information from desired neurons.

7.2.1 Tracking Activation

In this method, N images are fed into the network and the activation (i.e., output of neuron after applying the activation function) of a specific neuron on each of these images is stored in an array. This way, we will obtain an array containing N real numbers in which each real number shows the activation of a specific neuron. Then, $K \ll N$ images with the highest activations are selected (Girshick et al. 2014). This method shows that what information about objects in the receptive field of the neuron increases the activation of the neuron. In Chap. 5, we visualized the classification network trained on the GTSRB dataset using this method.

7.2.2 Covering Mask

Assume that image $\mathbf{x}$ is correctly classified by a neural network with a probability close to 1.0. In order to understand which parts of the image have a greater impact on the score, we can run a multi-scale scanning window approach. In scale s and at location (m, n) on $\mathbf{x}$, $x(m, n)$ and all pixels in its neighborhood are set to zero. The size of neighborhood depends on s. This is equivalent zeroing the inputs to the network. In other words, information in this particular part of the image is missing. If the classification score highly depends on the information centered at (m, n), the score must be dropped significantly by zeroing the pixels in this region. If the above procedure is repeated for different scales and on all the locations in the image, we

will end up with a map for each scale where the value of map will be close to 1 if zeroing its analogous region does not have any effect on the score. In contrast, the value of map will be close to zero if zeroing its corresponding region has a great impact on score. This method is previously used in Chap. 5 on the classification network trained on GTSRB dataset. One problem with this method is that it could be very time consuming to apply the above method on many samples for each class to figure out which regions are important in the final classification score.

7.2.3 Embedding

Embedding is another technique which provides important information about feature space. Basically, given a set of feature vectors $\mathscr{X} = \{\Phi(\mathbf{x}_1), \Phi(\mathbf{x}_2), \ldots, \Phi(\mathbf{x}_N)\}$, where $\Phi : \mathbb{R}^{H \times W \times 3} \rightarrow \mathbb{R}^d$ is the feature transformation function, the goal of embedding is to find the mapping $\Psi : \mathbb{R}^d \rightarrow \mathbb{R}^{\hat{d}}$ to project the d-dimensional feature vector into a $\hat{d}$-dimensional space. Usually, $\hat{d}$ is set to 2 or 3 since inspecting vectors visually in this spaces can be easily done using scatter plots.

There are different methods for finding the mapping Ψ. However, there is a specific mapping which is particularly used for mapping into two-dimensional space in the field of neural network. This mapping is called *t-distributed stochastic neighbor embedding* (t-SNE). It is a structure preserving mapping meaning that it tries to preserve the structure of neighbors in the $\hat{d}$-dimensional space as similar as possible to the structure of neighbors in d-dimensional space. This is an important property since it shows that how separable are patterns from different classes in the original feature space.

Denoting the feature transformation function up to layer L in a network by $\Phi_L(\mathbf{x})$, we collect the set $\mathscr{X}_L = \{\Phi_L(\mathbf{x}_1), \Phi_L(\mathbf{x}_2), \ldots, \Phi_L(\mathbf{x}_N)\}$ by feeding many images from different classes to the network and collecting $\Phi_L(\mathbf{x}_N)$ for each image. Then, the t-SNE algorithm is applied on $\mathscr{X}_L$ in order to find a mapping into the two-dimensional space. The mapped points can be plotted using scatter plots. This technique was used for analyzing networks in Chaps. 5 and 6.

7.3 Gradient-Based Techniques

Gradient-based methods explain neural networks in terms of their gradient with respect to the input image $\mathbf{x}$ (Simonyan et al. 2013). Depending on how the gradients are interpreted, a neural network can be studied from different perspectives.[1]

[1] Implementations of the methods in this chapter are available at *github.com/pcnn/*.

7.3.1 Activation Maximization

Denoting the classification score of $\mathbf{x}$ on class c with $S_c(\mathbf{x})$, we can find an input $\hat{\mathbf{x}}$ by maximizing the following objective function:

$$S_c(\hat{\mathbf{x}}) - \lambda \|\hat{\mathbf{x}}\|_2^2, \tag{7.1}$$

where λ is the regularization parameter defined by user. In other words, we are looking for an input image $\hat{\mathbf{x}}$ that maximizes the classification score on class c and it is always within n-sphere defined by the second term in the above function. This loss can be implemented using a Python layer in the Caffe library. Specifically, the layer accepts a parameter indicating the class of interest. Then, it will return the score of class of interest during forward pass. In addition, in the backward pass derivative of all classes except the class of interest will be set to zero. Obviously, any change in the inputs of layer other than class of interest does not change the output. Consequently, derivative of the loss with respect to these inputs will be equal to zero. In contrast, derivative of loss with respect to class of interest will be equal to 1 since it just passes the value from class of interest to the output. One can think of this loss as a multiplexer which directs inputs according to its address.

The derivative of the second term in the objective function with respect to classification scores is always zero. However, derivative of the second term with respect to input x_i is equal to $2\lambda x_i$. In order to formulate the above objective function as a minimization problem, we can simply multiply the function with -1. In that case, derivative of the first term with respect to the class of interest will be equal to -1. Putting all this together, the Python layer for the above loss function can be defined as follows:

```
class score_loss(caffe.Layer):                                                                  1
    def setup(self, bottom, top):                                                               2
        params = eval(self.param_str)                                                           3
        self.class_ind = params['class_ind']                                                    4
        self.decay_lambda = params['decay_lambda' ] if params.has_key('decay_lambda') else 0    5
                                                                                                6
    def reshape(self, bottom, top):                                                             7
        top[0].reshape(bottom[0].data.shape[0], 1)                                              8
                                                                                                9
    def forward(self, bottom, top):                                                             10
        top[0].data[...] = 0                                                                    11
        top[0].data[:, 0] = bottom[0].data[:, self.class_ind]                                   12
                                                                                                13
    def backward(self, top, propagate_down, bottom):                                            14
        bottom[0].diff[...] = np.zeros(bottom[0].data.shape)                                    15
        bottom[0].diff[:, self.class_ind] = -1                                                  16
                                                                                                17
        if len(bottom) == 2 and self.decay_lambda > 0:                                          18
            bottom[1].diff[...] = self.decay_lambda * bottom[1].data[...]                       19
```

After designing the loss layer, it has to be connected to the *trained* network. The following Python script shows how to do this.

```
def create_net(save_to, class_ind):
    L = caffe.layers
    P = caffe.params
    net = caffe.NetSpec()
    net.data = L.Input(shape=[{'dim':[1,3,48,48]}])
    net.tran = L.Convolution(net.data,
                             num_output=3,
                             group=3,
                             kernel_size=1,
                             weight_filler={'type':'constant',
                                            'value':1},
                             bias_filler={'type':'constant',
                                          'value':0},
                             param=[{'decay_mult':1},{'decay_mult':0}],
                             propagate_down=True)
    net.conv1, net.act1, net.pool1 = conv_act_pool(net.tran, 7, 100, act='ReLU')
    net.conv2, net.act2, net.pool2 = conv_act_pool(net.pool1, 4, 150, act='ReLU', group=2)
    net.conv3, net.act3, net.pool3 = conv_act_pool(net.pool2, 4, 250, act='ReLU', group=2)
    net.fc1, net.fc_act, net.drop1 = fc_act_drop(net.pool3, 300, act='ReLU')
    net.f3_classifier = fc(net.drop1, 43)
    net.loss = L.Python(net.f3_classifier, net.data, module='py_loss', layer='score_loss',
       param_str="{'class_ind':%d, 'decay_lambda':5}" %class_ind)
    with open(save_to, 'w') as fs:
        s_proto = 'force_backward:true\n' + str(net.to_proto())
        fs.write(s_proto)
        fs.flush()
        print s_proto
```

Recall from Chap. 4 that the Python file has to be placed next to the network definition file. We also set *force_backward* to *true* in order to force Caffe to always perform the backward pass down to the data layer. Finally, the image $\hat{\mathbf{x}}$ can be found by running the following momentum-based gradient descend algorithm.

```
caffe.set_mode_gpu()
root = '/home/pc/'
net_name = 'ircv1'
save_to = root + 'cnn_{}.prototxt'.format(net_name)
class_ind = 1
create_net(save_to, class_ind)

net = caffe.Net(save_to, caffe.TEST)
net.copy_from('/home/pc/cnn.caffemodel')

im_mean = read_mean_file('/home/pc/gtsr_mean_48x48.binaryproto')
im_res = read_mean_file('/home/pc/gtsr_mean_48x48.binaryproto')
im_res = im_res[np.newaxis,...]/255.

alpha = 0.0001
momentum = 0.9
momentum_vec = 0

for i in xrange(4000):
    net.blobs['data'].data[...] = im_res[np.newaxis, ...]
    net.forward()
    net.backward()
    momentum_vec = momentum * momentum_vec + alpha * net.blobs['data'].diff
    im_res = im_res - momentum_vec
    im_res = np.clip(im_res, -1, 1)


fig1 = plt.figure(1, figsize=(6, 6), facecolor='w')
plt.clf()
res = np.transpose(im_res[0].copy()*255+im_mean, [1, 2, 0])[:,:,[2,1,0]]
res = np.divide(res - res.min(), res.max()-res.min())
plt.imshow(res)
```

Lines 1–9 create a network with the Python layer connected to this network and loads weights of the trained network into the memory. Line 11 loads the mean image into memory. The variable in this line will be used for applying the backward transformation on the result for illustration purposes. Lines 12 and 13 initialize the optimization algorithm by setting it to the mean image.

Lines 15–17 configure the optimization algorithm. Lines 19–25 perform the momentum-based gradient descend algorithm. Line 18 executes the forward pass and the next line performs the backward pass and computes derivative of loss function with respect to the input data. Finally, the commands after the loop show the obtained image. Figure 7.1 illustrates the result of running the above script on each of classes, separately.

It turns out that classification score of each class mainly depends on pictograph inside of each sign. Furthermore, shape of each sign has impact on the classification score as well. Finally, we observe that the network does a great job in eliminating the background of traffic sign.

It is worth mentioning that the optimization is directly done on the classification scores rather than output of softmax function. The reason is that maximizing the output of softmax may not necessarily maximize the score of class of interest. Instead, it may try to reduce the score of other classes.

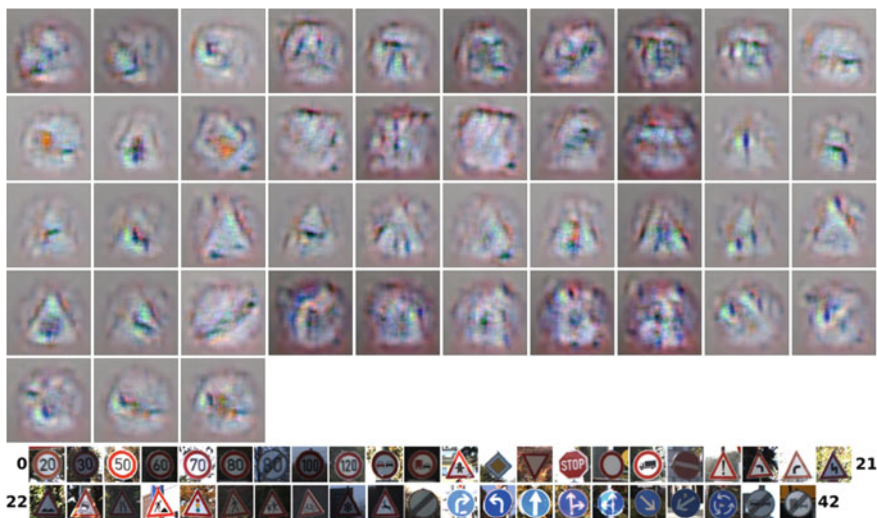

Fig. 7.1 Visualizing classes of traffic signs by maximizing the classification score on each class. The *top-left* image corresponds to class 0. The class labels increase from *left* to *right* and *top* to *bottom*

7.3.2 Activation Saliency

Another way for visualizing neural networks is to asses how sensitive is a classification score with respect to every pixel on the input image. This is equivalent to computing gradient of the classification score with respect to the input image. Formally, given the image $\mathbf{x} \in \mathbb{R}^{H \times W \times 3}$ belonging to class c, we can compute:

$$\nabla \mathbf{x}_{mnk} = \frac{\delta S_c(\mathbf{x})}{\mathbf{x}_{mnk}}, \qquad m = 0, \ldots, H, n = 0, \ldots, W, k = 0, 1, 2. \tag{7.2}$$

In this equation, $\nabla \mathbf{x}_{mnk} \mathbb{R}^{H \times W \times 3}$ stores the gradient of classification score with respect to every pixel in $\mathbf{x}$. If $\mathbf{x}$ is a grayscale image the output will only have one channel. Then, the output can be illustrated by mapping each gradient to a color. In the case that $\mathbf{x}$ is a color image, maximum of $\nabla \mathbf{x}$ is computed across channels.

$$\nabla \mathbf{x}'_{mn} = \max_{k=0,1,2} \nabla \mathbf{x}_{mnk}. \tag{7.3}$$

Then, $\nabla \mathbf{x}'_{mn}$ is illustrated by mapping each element in this matrix to a color. This roughly shows saliency of each pixel in $\mathbf{x}$. Figure 7.2 visualizes the class saliency of a random sample from each class.

In general, we see that the pictograph region in each image has a great effect on the classification score. Besides, in a few cases, we also observe that background pixels have impact on the classification score. However, this might not be generalized to all images in the same class. In order to understand expected saliency of pixel, we can compute $\nabla \mathbf{x}'$ for many samples from the same class and compute their average. Figure 7.3 shows expected class saliency obtained by computing the average of class saliency of 100 samples coming from the same class.

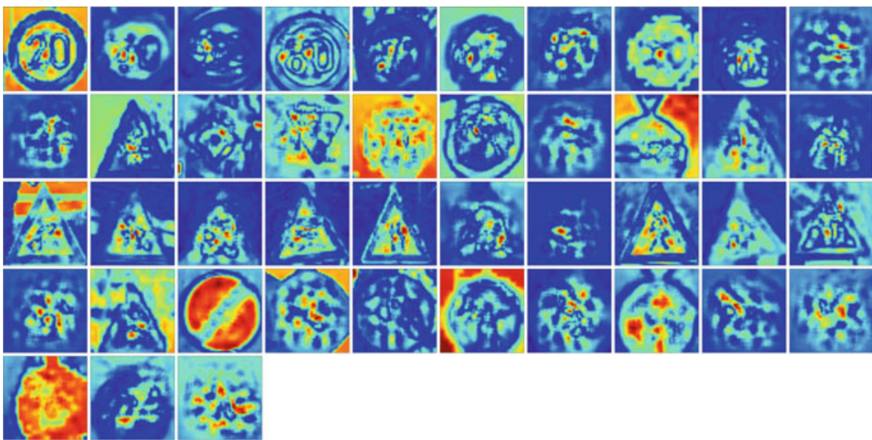

Fig. 7.2 Visualizing class saliency using a random sample from each class. The order of images is similar Fig. 7.1

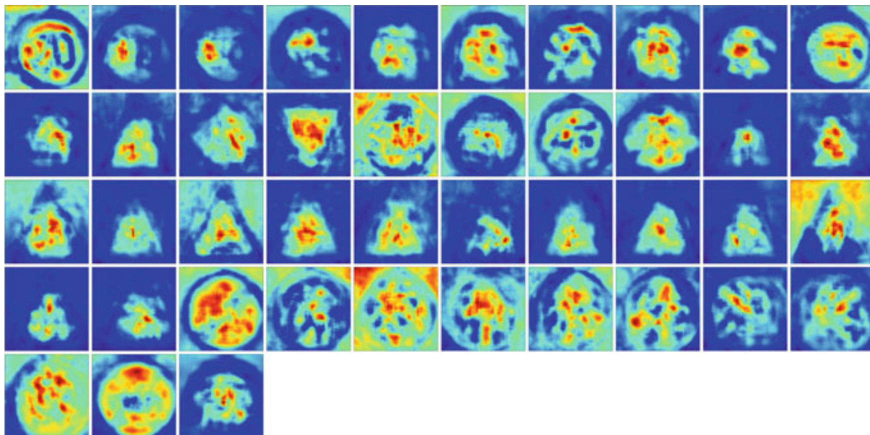

Fig. 7.3 Visualizing expected class saliency using 100 samples from each class. The order of images is similar to Fig. 7.1

The expected saliency reveals that the classification score mainly depends on pictograph region. In other words, slight changes in this region may dramatically change the classification score which in turn may alter the class of image.

7.4 Inverting Representation

Inverting a neural network (Mahendran and Vedaldi 2015) is a way to roughly know what information is retained by a specific layer in a neural network. Denoting the representation produced by L^{th} layer in a ConvNet for the input image $\mathbf{x}$ with $\Phi(\mathbf{x})_L$, inverting a ConvNet can be done by minimizing

$$\hat{\mathbf{x}} = \underset{\mathbf{x}' \in \mathbb{R}^{H \times W \times 3}}{\arg\min} \|\Phi(\mathbf{x}') - \Phi(\mathbf{x})\|_2 + \lambda \|\mathbf{x}'\|_p^p, \tag{7.4}$$

where the first term computes the Euclidean distance between the representations of the source image $\mathbf{x}$ and reconstructed image $\mathbf{x}'$ and the second term regularizes the cost by the *p-norm* of the reconstructed image.

If the regularizing term is omitted, it is possible to design a network using available layers in Caffe which accepts the representation of an image and tries to find the reconstructed image $\hat{\mathbf{x}}$. However, it is not possible to implement the above cost function including the second term using available layers in Caffe. For this reason, a Python layer has to be implemented for computing the loss and its gradient with respect to its bottoms. This layer could be implemented as follows:

```
class euc_loss(caffe.Layer):
    def setup(self, bottom, top):
        params = eval(self.param_str)
        self.decay_lambda = params['decay_lambda'] if params.has_key('decay_lambda') else 0
        self.p = params['p'] if params.has_key('p') else 2

    def reshape(self, bottom, top):
        top[0].reshape(bottom[0].data.shape[0], 1)

    def forward(self, bottom, top):

        if bottom[0].data.ndim == 4:
            top[0].data[:, 0] = np.sum(np.power(bottom[0].data-bottom[1].data,2), axis=(1,2,3))
        elif bottom[0].data.ndim == 2:
            top[0].data[:, 0] = np.sum(np.power(bottom[0].data - bottom[1].data, 2), axis=1)

        if len(bottom) == 3:
            top[0].data[:,0] += np.sum(np.power(bottom[2].data,2))

    def backward(self, top, propagate_down, bottom):
        bottom[0].diff[...] = bottom[0].data - bottom[1].data
        if len(bottom) == 3:
            bottom[2].diff[...] = self.decay_lambda *self.p* np.multiply(bottom[2].data[...], np.power(np.abs(bottom
            [2].data[...]),self.p-2))
```

Then, the above loss layer is connected to the network trained on the GTSRB dataset.

```
def create_net_ircv1_vis(save_to):
    L = caffe.layers
    P = caffe.params
    net = caffe.NetSpec()
    net.data = L.Input(shape=[{'dim':[1,3,48,48]}])
    net.rep = L.Input(shape=[{'dim': [1, 250, 6, 6]}])  #output shape of conv3

    net.tran = L.Convolution(net.data,
                        num_output=3,
                        group=3,
                        kernel_size=1,
                        weight_filler={'type':'constant',
                                'value':1},
                        bias_filler={'type':'constant',
                               'value':0},
                        param=[{'decay_mult':1},{'decay_mult':0}],
                        propagate_down=True)
    net.conv1, net.act1, net.pool1 = conv_act_pool(net.tran, 7, 100, act='ReLU')
    net.conv2, net.act2, net.pool2 = conv_act_pool(net.pool1, 4, 150, act='ReLU', group=2)
    net.conv3, net.act3, net.pool3 = conv_act_pool(net.pool2, 4, 250, act='ReLU', group=2)
    net.fc1, net.fc_act, net.drop1 = fc_act_drop(net.pool3, 300, act='ReLU')
    net.f3_classifier = fc(net.drop1, 43)
    net.loss = L.Python(net.act3, net.rep, net.data, module='py_loss', layer='euc_loss',
    param_str="{'decay_lambda':10,'p':6}")
```

The network accepts two inputs. The first input shows the reconstructed image and the second input indicates the representation of the source image. In the above network, our goal is to reconstruct the image using representation produced by the activation of the third convolution layer. The output shape of the third convolution layer is $250 \times 3 \times 3$. Hence, the shape of second input in the network is set to $1 \times 250 \times 6 \times 6$. Moreover, as it is proposed in Mahendran and Vedaldi (2015), we set the value of p

in the above network to 6. Having the network created, we can execute the following momentum-based gradient descend for finding $\hat{\mathbf{x}}$.

```
im_mean = read_mean_file('/home/pc/gtsr_mean_48x48.binaryproto')
im_mean = np.transpose(im_mean, [1, 2, 0])

im = cv2.imread('/home/pc/GTSRB/Training_CNN/00016/crop_00001_00029.ppm')
im = cv2.resize(im, (48,48))
im_net = (im.astype('float32')-im_mean)/255.
net.blobs['data'].data[...] = np.transpose(im_net, [2, 0, 1])[np.newaxis, ...]

net.forward()
rep = net.blobs['act3'].data.copy()


im_res = im*0
im_res = np.transpose(im_res, [2,0,1])

alpha = 0.000001
momentum = 0.9
momentum_vec = 0

for i in xrange(10000):
    net.blobs['data'].data[...] = im_res[np.newaxis, ...]
    net.blobs['rep'].data[...] = rep[...]

    net.forward()
    net.backward()

    momentum_vec = momentum * momentum_vec - alpha * net.blobs['data'].diff

    im_res = im_res  + momentum_vec
    im_res = np.clip(im_res, -1, 1)

plt.figure(1)
plt.clf()
res = np.transpose(im_res[0].copy(), [1, 2, 0])
res = np.clip(res*255 + im_mean, 0, 255)
res = np.divide(res - res.min(), res.max()-res.min())
plt.imshow(res[:,:, [2,1,0]])
plt.show()
```

In the above code, the source image is first fed to the network and the output of the third convolution layer is copied into memory. Then, the optimization is done in 10,000 iterations. At each iteration, the reconstructed image is entered to the network and the backward pass is computed down to the input layer. This way, gradient of the loss function is obtained with respect to the input. Finally, the reconstructed image is updated using the momentum gradient descend rule. Figure 7.4 shows the result of inverting the classification network from different layers. We see that the first convolution layer keeps photo-realistic information. For this reason, the reconstructed image is very similar to the source image. Starting from the second convolution layer, photo-realistic information starts to vanish and they are replaced with parts of image which is important to the layer. For example, the fully connected layer mainly depends on the specific part of pictograph on the sign and it ignores background information.

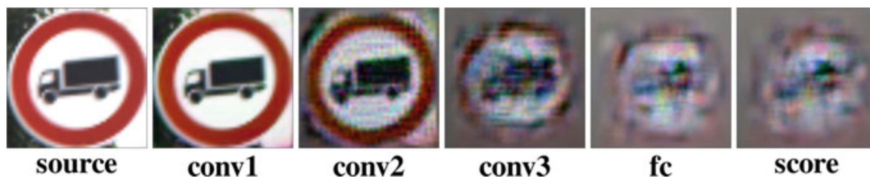

Fig. 7.4 Reconstructing a traffic sign using representation of different layers

7.5 Summary

Understanding behavior of neural networks is necessary in order to better analyze and diagnose them. Quantitative metrics such as classification accuracy and F1 score just give us numbers indicating how good is the classifier in our problem. They do not tell us how a neural network achieves this result. Visualization is a set of techniques that are commonly used for understanding structure of high-dimensional vectors.

In this chapter, we briefly reviewed data-driven techniques for visualization and showed that how to apply them on neural networks. Then, we focused on techniques that visualize neural networks by minimizing an objective function. Among them, we explained three different methods.

In the first method, we defined a loss function and found an image that maximizes the classification score of a particular class. In order to generate more interpretable images, the objective function was regularized using L_2 norm of the image. In the second method, gradient of a particular neuron was computed with respect to the input image and it is illustrated by computing its magnitude.

The third method formulated the visualizing problem as an image reconstruction problem. To be more specific, we explained a method that tries to find an image in which the representation of this image is very close to the representation of the original image. This technique usually tells us what information is usually discarded by a particular layer.

7.6 Exercises

7.1 Visualizing a ConvNet can be done by maximizing the softmax score of a specific class. However, this may not exactly generate an image that maximizes the classification score. Explain the reason taking into account the softmax score.

7.2 Try embed a feature extracted by neural network using local linear embedding method.

7.3 Use Isomap to embed features into two-dimensional space.

7.4 Assume an image of traffic signs belonging to class c which is correctly classified by the ConvNet. Instead of maximizing $S_c(\mathbf{x})$, try to minimize directly $S_c(\mathbf{x})$ such that $\mathbf{x}$ is no longer classified correctly by ConvNets but it is still easily recognizable for human.

References

Girshick R, Donahue J, Darrell T, Berkeley UC, Malik J (2014) Rich feature hierarchies for accurate object detection and semantic segmentation. doi:10.1109/CVPR.2014.81, arXiv:abs/1311.2524

Mahendran A, Vedaldi A (2015) Understanding deep image representations by inverting them. In: Computer vision and pattern recognition. IEEE, Boston, pp 5188–5196. doi:10.1109/CVPR.2015.7299155, arXiv:abs/1412.0035

Simonyan K, Vedaldi A, Zisserman A (2013) Deep inside convolutional networks: visualising image classification models and saliency maps, pp 1–8. arXiv:13126034