Specify Layers of Convolutional Neural Network

The first step of creating and training a new convolutional neural network (ConvNet) is to define the network layers. This topic explains the details of ConvNet layers, and the order they appear in a ConvNet.

The architecture of a ConvNet can vary depending on the types and numbers of layers included. The types and number of layers included depends on the particular application or data. For example, if you have categorical responses, you must have a softmax layer and a classification layer, whereas if your response is continuous, you must have a regression layer at the end of the network. A smaller network with only one or two convolutional layers might be sufficient to learn on a small number of gray scale image data. On the other hand, for more complex data with millions of colored images, you might need a more complicated network with multiple convolutional and fully connected layers.

You can define the layers of a convolutional neural network in MATLAB® in an array format, for example,

```
layers = [imageInputLayer([28 28 1])
          convolution2dLayer(5,20)
          reluLayer
          maxPooling2dLayer(2,'Stride',2)
          fullyConnectedLayer(10)
          softmaxLayer
          classificationLayer];
```

`layers` is an array of `Layer` objects. `layers` becomes an input for the training function `trainNetwork`.

Image Input Layer

Image input layer defines the size of the input images of a convolutional neural network and contains the raw pixel values of the images. You can add an input layer using the `imageInputLayer` function. Specify your image size in the `inputSize` argument. Size of an image corresponds to the height, weight, and the number of color channels of that image. For example, for a gray scale image, the number of channels would be 1, and for a color image it is 3.

In this layer, you can also specify the network to perform data augmentation, such as random flipping or cropping of the data, or data transformation by subtracting the mean of the image from the pixel values in the training set. The purpose of augmentation and normalization transforms is to reduce overfitting [2], which might occur with especially larger networks.

Convolutional Layer

Filters and Stride: A convolutional layer consists of neurons that connect to subregions of the input images or the outputs of the layer before it. A convolutional layer learns the features localized by these regions while scanning through an image. You can specify the size of these regions using the `filterSize` input argument in the call to `convolution2dLayer` function.

For each region, the `trainNetwork` function computes a dot product of the weights and the input, and then adds a bias term. A set of weights that are applied to a region in the image is called a *filter*. The filter moves along the input image vertically and horizontally, repeating the same computation for each region, i.e., convolving the input. The step size with which it moves is called a *stride*. You can specify this step size with the `Stride` name-value pair argument. These local regions that the neurons connect to might overlap depending on the `filterSize` and `'Stride'`.

The number of weights used for a filter is $h*w*c$, where h is the height, and w is the width of the filter size, and c is the number of channels in the input (for example, if the input is a color image, the number of color channels is 3). The number of filters determines the number of channels in the output of a convolutional layer. Specify the number of filters using the `numFilters` argument in the call to `convolution2dLayer`.

Feature Maps: As a filter moves along the input, it uses the same set of weights and bias for the convolution, forming a *feature map*. Hence, the number of feature maps a convolutional layer has is equal to the number of filters (number of channels). Each feature map has a different set of weights and a bias. So, the total number of parameters in a convolutional layer is $((h*w*c + 1)*Number\ of\ Filters)$, where 1 is for the bias.

Zero Padding: You can also apply zero padding to input image borders vertically and horizontally using the `'Padding'` name-value pair argument. Padding is basically adding rows or columns of zeros to the borders of an image input. It helps you control the output size of the layer it is added to.

Output Size: The output height and width of a convolutional layer is $(Input\ Size - Filter\ Size + 2*Padding)/Stride + 1$. This value must be an integer for the whole image to be fully covered. If the combination of these parameters does not lead the image to be fully covered, the software by default ignores the remaining part of the image along the right and bottom edge in the convolution.

Number of Neurons: The product of the output height and width gives the total number of neurons in a feature map, say $Map\ Size$. The total number of neurons (output size) in a convolutional layer, then, is $Map\ Size*Number\ of\ Filters$.

For example, suppose that the input image is a 28-by-28-by-3 color image. For a convolutional layer with 16 filters, and a filter size of 8-by-8, the number of weights per filter is 8*8*3 = 192, and the total number of parameters in the layer is (192+1) * 16 = 3088. Assuming stride is 4 in each direction and there is no zero padding, the total number of neurons in each feature map is 6-by-6 ((28 – 8+0)/4 + 1 = 6). Then, the total number of neurons in the layer is 6*6*16 = 256.

Learning Parameters: You can also adjust the learning rates and regularization parameters for this layer using the related name-value pair arguments while defining the convolutional layer. If you choose not to adjust them, `trainNetwork` uses the global training parameters defined by `trainingOptions` function. For details on global and layer training options, see Set Up Parameters and Train Convolutional Neural Network.

A convolutional neural network can consists of one or multiple convolutional layers. The number of convolutional layers depends on the amount and complexity of the data.

The results from the neurons of a convolutional layer usually pass through some form of nonlinearity. Neural Network Toolbox™ uses rectified linear units (ReLU) function for this purpose. Each convolutional layer can be followed by ReLU layer or a pooling layer.

ReLU Layer

A convolutional layer is usually followed by a nonlinear activation function. In MATLAB, it is a rectified linear unit (ReLU) function, specified by a ReLU layer. You can specify the ReLU layer using the `reluLayer` function. It performs a threshold operation to each element, where any input value less than zero is set to zero, i.e.,

$$f(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases}.$$

The ReLU layer does not change the size of its input.

Cross Channel Normalization (Local Response Normalization) Layer

This layer performs a channel-wise local response normalization. It usually follows the ReLU activation layer. You can specify this layer using the `crossChannelNormalizationLayer` function. This layer replaces each element with a normalized value it obtains using the elements from a certain number of neighboring channels (elements in the normalization window). That is, for each element x in the input, `trainNetwork` computes a normalized value x' using

$$x' = \frac{x}{\left(K + \frac{\alpha * ss}{windowChannelSize}\right)^{\beta}},$$

where K, α, and β are the hyperparameters in the normalization, and ss is the sum of squares of the elements in the normalization window [2]. You must specify the size of normalization window using the `windowChannelSize` argument in the call to `crossChannelNormalizationLayer` argument. You can also specify the hyperparameters using the `Alpha`, `Beta`, and `K` name-value pair arguments.

Note that, the previous normalization formula is slightly different than what is presented in [2]. You can obtain the equivalent formula by multiplying the `alpha` value by the `windowChannelSize`.

Max- and Average-Pooling Layers

Max- and average-pooling layers follow the convolutional layers for down-sampling, hence, reducing the number of connections to the following layers (usually a fully-connected layer). They do not perform any learning themselves, but reduce the number of parameters to be learned in the following layers. They also help reduce overfitting. You can specify these layers using the `maxPooling2dLayer` and `averagePooling2dLayer`.

Max-pooling layer returns the maximum values of rectangular regions of its input. The size of the rectangular regions are determined by the `poolSize` to `maxPoolingLayer`. For example, if `poolSize` is [2,3], the software returns the maximum value of regions of height 2 and width 3.

Similarly, the average-pooling layer outputs the average values of rectangular regions of its input. The size of the rectangular regions is determined by the `poolSize` in the call to `averagePoolingLayer`. For example, if `poolSize` is [2,3], the software returns the average value of regions of height 2 and width 3. The `maxPoolingLayer` and `averagepoolingLayer` functions scan through the input horizontally and vertically in step sizes you can specify using the `Stride` argument in the call to either function. If the `poolSize` is smaller than or equal to `Stride`, then the pooling regions do not overlap.

For nonoverlapping regions (`poolSize` and `Stride` are equal), if the input to the pooling layer is n-by-n, and the pooling region size is h-by-h, then the pooling layer down-samples the regions by h [6]. That is, the output of a max- or average-pooling layer for one channel of a convolutional layer is n/h-by-n/h. For overlapping regions, the output of a pooling layer is $(Input\ Size - Pool\ Size + 2*Padding)/Stride + 1$.

Dropout Layer

A dropout layer randomly sets the layer's input elements to zero with a given probability. You can specify the dropout layer using the `dropoutLayer` function.

Although the output of a dropout layer is equal to its input, this operation corresponds to temporarily dropping a randomly chosen unit and all of its connections from the network during training. So, for each new input element, `trainNetwork` randomly selects a subset of neurons, forming a different layer architecture. These architectures use common weights, but because the learning does not depend on specific neurons and connections, the dropout layer might help prevent overfitting [7], [2]. Similar to max- or average-pooling layers, no learning takes place in this layer.

Fully Connected Layer

The convolutional (and down-sampling) layers are followed by one or more fully connected layers. You can specify a fully connected layer using the `fullyConnectedLayer` function.

As the name suggests, all neurons in a fully connected layer connect to the neurons in the layer previous to it. This layer combines all of the features (local information) learned by the previous layers across the image to identify the larger patterns. For classification problems, the last fully connected layer combines them to classify the images. That is why, the `OutputSize` parameter in the last fully connected layer is equal to the number of classes in the target data. For regression problems, the output size must be equal to the number of response variables.

You can also adjust the learning rate and the regularization parameters for this layer using the related name-value pair arguments while defining the fully connected layer. If you choose not to adjust them, `trainNetwork` uses the global training parameters defined by `trainingOptions` function. For details on global and layer training options, see Set Up Parameters and Train Convolutional Neural Network.

Softmax and Classification Layers

For classification problems, a softmax layer and then a classification layer must follow the final fully connected layer. You can specify these layers using the `softmaxLayer` and `classificationLayer` functions, respectively.

The output unit activation function is the softmax function:

$$y_r(x) = \frac{\exp(a_r(x))}{\sum_{j=1}^{k} \exp(a_j(x))},$$

where $0 \le y_r \le 1$ and $\sum_{j=1}^{k} y_j = 1$.

The softmax function is the output unit activation function after the last fully connected layer for multi-class classification problems:

$$P(c_r|x,\theta) = \frac{P(x,\theta|c_r)P(c_r)}{\sum_{j=1}^{k} P(x,\theta|c_j)P(c_j)} = \frac{\exp(a_r(x,\theta))}{\sum_{j=1}^{k} \exp(a_j(x,\theta))},$$

where $0 \le P(c_r|x,\theta) \le 1$ and $\sum_{j=1}^{k} P(c_j|x,\theta) = 1$. Moreover, $a_r = \ln(P(x,\theta|c_r)P(c_r))$, $P(x,\theta|c_r)$ is the conditional probability of the sample given class r, and $P(c_r)$ is the class prior probability.

The softmax function is also known as the *normalized exponential* and can be considered the multi-class generalization of the logistic sigmoid function [8].

A classification output layer must follow the softmax layer. In the classification output layer, `trainNetwork` takes the values from the softmax function and assigns each input to one of the k mutually exclusive classes using the cross entropy function for a 1-of-k coding scheme [8]:

$$E(\theta) = -\sum_{i=1}^{n}\sum_{j=1}^{k} t_{ij} \ln y_j(x_i,\theta),$$

where t_{ij} is the indicator that the ith sample belongs to the jth class, θ is the parameter vector. $y_j(x_i,\theta)$ is the output for sample i, which in this case, is the value from the softmax function. That is, it is the probability that the network associates ith input with class j, $P(t_j = 1|x_i)$.

Regression Layer

You can also use ConvNets for regression problems, where the target (output) variable is continuous. In such cases, a regression output layer must follow the final fully connected layer. You can specify the regression layer using the `regressionLayer` function. The default activation function for the regression layer is the mean squared error:

$$MSE = E(\theta) = \sum_{i=1}^{n} \frac{(t_i - y_i)^2}{n},$$

where t_i is the target output, and y_i is the network's prediction for the response variable corresponding to observation i.

References

[1] Murphy, K. P. *Machine Learning: A Probabilistic Perspective*. Cambridge, Massachusetts: The MIT Press, 2012.

[2] Krizhevsky, A., I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks. " *Advances in Neural Information Processing Systems*. Vol 25, 2012.

[3] LeCun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W., Jackel, L.D., et al. "Handwritten Digit Recognition with a Back-propagation Network." In *Advances of Neural Information Processing Systems*, 1990.

[4] LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based Learning Applied to Document Recognition." *Proceedings of the IEEE.* Vol 86, pp. 2278–2324, 1998.

[5] Nair, V. and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In Proc. 27th International Conference on Machine Learning, 2010.

[6] Nagi, J., F. Ducatelle, G. A. Di Caro, D. Ciresan, U. Meier, A. Giusti, F. Nagi, J. Schmidhuber, L. M. Gambardella. "Max-Pooling Convolutional Neural Networks for Vision-based Hand Gesture Recognition". *IEEE International Conference on Signal and Image Processing Applications (ICSIPA2011)*, 2011.

[7] Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." *Journal of Machine Learning Research*. Vol. 15, pp. 1929-1958, 2014.

[8] Bishop, C. M. *Pattern Recognition and Machine Learning*. Springer, New York, NY, 2006.

See Also

`averagePooling2dLayer` | `classificationLayer` | `convolution2dLayer` | `crossChannelNormalizationLayer` | `dropoutLayer` | `fullyConnectedLayer` | `imageInputLayer` | `maxPooling2dLayer` | `reluLayer` | `softmaxLayer` | `trainingOptions` | `trainNetwork`

Related Topics

- Introduction to Convolutional Neural Networks

- Set Up Parameters and Train Convolutional Neural Network
- Resume Training from a Checkpoint Network
- Create Simple Deep Learning Network for Classification
- Pretrained Convolutional Neural Networks
- Deep Learning in MATLAB