2、What Is Deep Learning

specialize in 专门研究~

be around for 存在

hierarchical learning 层学习：where simple concepts are learned in the lower layers and more abstract patterns in the higher layers of the networkm, allows us to completely remove the hand-designed feature extracting process and treat CNNs as end-to-end learners. Each layer in the network uses the output of previous layes as “building blocks”to construct increasingly more abstract concepts. There layers are learned automatically – there is no hand-crafted feature engineering taking place in out network.

神经网络的历史：McCulloch and Pitts (binary classifier) -> Rosenblatt (Perceptron algorithm) -> Rumelhart (backpropagation algorithm) -> LeCun (Convolutional Neural Network)

Filter in lower levels of the network represent edges and corners, while higher level layers use the edges and corners to learn more abstract concepts useful for discriminating between image classes.

Histogram of Oriented Gradients (HOG)用于detecting objects in images.

CNN：instead of hand-defining a set of rules and algorithms to extract features from an image, there features are instead automatically learned from the training process.

the goal of machine learning: computers should be able to learn from experience of the problem they are trying to solve.

There is no consensus amongst experts on exactly what makes a neural network “deep”; however, we know that:

1. Deep learning algorithms learn in a hierarchical fashion and therefore stack multiple layers on top of each other to learn increasingly more abstract concepts.
2. A network should have >2 layers to be considered “deep”
3. A network with >10 layers is considered very deep

3、Image Fundamentals

The origin point (0,0) corresponds to the **upper-left** corner of the images. As we move down and to the right, both the x and y values increase, where we go x columns to the right and y rows down.

OpenCV and scikit-image represent RGB images as multi-dimensional NumPy arrays with shape (height, width, depth).

|  |
| --- |
| 1. **import** cv2 2. img = cv2.imread('example.jpg') 3. **print**(img.shape)  # 图像尺寸为1024\*768，而shape为(768,1024,3) 4. cv2.imshow("Image", img) 5. cv2.waitKey(0) 6. (b, g, r) = img[20, 100] |

aspect ratio: the ratio of the width to the height of the image.

Ignoring the aspect ratio can lead to images that look compressed and distorted. To prevent this behavior, we simple scale the width and height of an image by equal amounts when resizing an image.

Most neural networks and Convolutional Neural Networks applied to the task of image classification assume a fixed size input, meaning that the dimensions of all images you pass through the network must be the same. Common choices for width and height image sizes inputted to Convolutional Neural Networks include 32\*32, 64\*64, 224\*224, 227\*227, 256\*256, and 299\*299.

How are we supposed to preprocess these images?

1. simply ignore the aspect ratio and deal with the distortion
2. resizing the image along its shortest dimension and then taking the center crop.

4、Image Classification Basics

The goal of an image classification system is to take an input image and assign a label to it from our categories set.

In the context of image classification, our dataset is a collection of images. Each image is, therefore, a data point.

have to handle factors of variation: viewpoint variation, scale variation, deformation, occlusion variation, illumination, background clutter (背景杂波), intra-class variation.

canonical adj. 权威的

always consider the scope of your image classifier.

Instead of trying to construct a rule-based system to describe what each category “looks like”, we can instead take a data driven approach by supplying examples of what each category looks like and then teach our algorithm to recognize the difference between the categories using these examples.

four steps to constructing a deep learning model:

1. Gather your dataset

the number of images for each category should be approximately uniform.

Class imbalance

1. Split your dataset: 1) A training set; 2) A testing set

the training set and testing set are independent of each other and do not overlap. 一般分割比例为：66.6%/33.3%, 75%/25%, 90%/10%.

But what if you have hyperparameters to tune? normally allocate roughly 10~20% of the training data for validation.

1. Train your network
2. Evalute: precision, recall, f-measure

Generalization: the ability for a network to generalize and correctly predict the class label of an image that does not exist as part of its training or testing data.

如果模型的准确率欠佳：consider the set of factors of variation mentioned above. Does your training dataset accurately reflect exmplaes of thes factors of variation? If not, you will need to gather more training data.

5、Datasets for Image Classifiction

CALTECH-101：popular benchmark dataset for object detection.

6、Configuring your development environment

consider using a Linux environment such as Ubuntu.

7、Your first Image classification

should always be cognizant of your dataset size before even starting to work with image classification algorithm.

preprocessing methods for booosting classification accuracy: mean subtraction, sampleing random patches, simply resizing the image to a fixed size.

* a basic image preprocessor

|  |
| --- |
| 1. """ 2. load an image from disk and resized it to a fixed size. 3. building an image preprocessor that resizes the image, 4. ignore the aspect ratio. 5. """ 7. **import** cv2 9. **class** SimplePreprocessor: 10. **def** \_\_init\_\_(self, width, height, inter=cv2.INTER\_AREA): 11. # inter: interpolation method used when resizing 12. self.width = width 13. self.height = height 14. self.inter = inter 16. **def** preprocess(self, image): 17. # resize the image to a fixed size, 18. # ignoring the aspect ratio 19. **return** cv2.resize(image, (self.width, self.height), interpolation=self.inter) |

* data loader:

|  |
| --- |
| 1. """ 2. load small image datasets from disk 3. """ 5. **import** numpy as np 6. **import** cv2 7. **import** os 9. **class** SimpleDatasetLoader: 10. **def** \_\_init\_\_(self, preprocessors=None): 11. self.preprocessors = preprocessors 12. # first need to resize an image to a fixed size, 13. # then perform some sort of scaling, 14. # followed by converting the image array to a format suitable for Keras 15. **if** self.preprocessors **is** None: 16. self.preprocessors = [] 18. **def** load(self, imagePaths, verbose=-1): 19. # imagePaths: specifying the file paths to the images in our dataset residing on disk 20. # verbose: used to print updates to a console 21. data = [] 22. labels = [] 23. **for** (i, imagePath) **in** enumerate(imagePaths): 24. image = cv2.imread(imagePath) 25. # 目录结构: /dataset\_name/class/image.jpg 26. label = imagePath.split(os.path.sep)[-2] 27. **if** self.preprocessors **is** **not** None: 28. # loop over the preprocessors and apply each to the image 29. **for** p **in** self.preprocessors: 30. image = p.preprocess(image) 31. data.append(image) 32. labels.append(label) 34. # handle printing updates to our console 35. **if** verbose > 0 **and** i > 0 **and** (i + 1) % verbose == 0: 36. **print**("[INFO] processed {}/{}".format(i + 1, len(imagePaths))) 37. **return** np.array(data), np.array(labels) |

对于分类图片，使用hierarchical directory structure保存，如all images inside the dog subdirectory are examples of dog. Directory structure为：/dataset\_name/class/image.jpg.

**k-NN**：directly relies on the distance between feature vectors, classifies unknown data points by finding the most common class among the k closest examples. Each data points in the k closest data points casts a vote, and the category with the highest number of votes wins.

First, we need to select a distance metric or similarity function:

Euclidean distance (L2 distance):

Manhattan/city block (L1 distance):

Many machine learning algorithms assume that the class labels are encoded as integers.

|  |
| --- |
| 1. **from** sklearn.preprocessing **import** LabelEncoder 2. le = LabelEncoder() 3. labels = le.fit\_transform(labels) |

The k-NN algorithm is unable to learn any discrimination patterns between these species. This is one of the primary drawbacks of the k-NN algorithm.

The k-NN algorithm is more suited for low-dimensional feature spaces. Distances in high-dimensional feature spaces are often unintuitive. Most importantly, it gives us a baseline that we can use to compare neural networks and Convolutional Neural Networks to as we progress through the rest of the book.

Drawbacks of k-NN：

1. it does not actually learn anything – if the algorithm makes a mistake, it has no way to correct and improve itself for later classification.
2. without specialized data structures, the k-NN algorithm scales linearly with the number of data points, making it not only practically challenging to use in high dimensions, but theoretically questionable in terms of its usage.

8、Parameterized learning

parametric model: a learning model that summarizes data with a set of parameters of fixed size. No matter how much data you throw at the parametetric model, it will not change its mind about how many parameters it needs.

Parameterization involves defining a problem in terms of four key components:

1. data: includes both the data points and their associated class labels.
2. a scoring function: accpets our data as an input and maps the data to class labels.
3. a loss function: quantifies how well our predicted class labels agree with our ground-truth labels.
4. weights and biases: that we will actually be optimizing. Based on the output of our scoring function and loss function, we will be tweaking and fiddling with the values of the weights and biases to increase classification accuracy.

a simple linear mapping:

其中，有K个unique categories，W的维度为，的维度为，b的维度为.

two primary advantages to utilizing parameterized learning:

1. Once we are done training our model, we can discard the input data and keep only the weight W and the bias vector b.
2. Classifying new test data is fast.

discuss two important concepts: 1) loss function; 2) optimization methods.

How we go about updating the parameters of weight matrix W or bias vector b is an optimization problem.

A loss function can be used to quantify how well our scoring function is doing at classifying input data points.

abbreviate our scoring function as s:

Which implies that we can obtain the predicted score of the j-th class via the i-th data point:

We can put it all together, obtaining the hinge loss function (multi-class SVM loss):

The hinge loss function is summing across all incorrect classes () and comparing the output of our scoring function s returned for the j-th class label (the incorrect class) and the class (the correct class). A given is classified correctly when the loss .

To derive the loss across our entire training set, we simply take the mean over each individual :

对于Image #1，Dog，Cat，Panda对应的score分别为4.26, 1.33, -1.01，其hinge loss为：

squared hinge loss:

Softmax classifiers give you probabilities for each class label while hinge loss gives you the margin.

rank-5 accuracy: check to see if the ground-truth label is in the top-5 predicted labels returned by a network for a given input image.

our loss function should minimize the negative log likelihood of the correct class:

其中，

所以，interpret these scores as unnormalized log probabilities for each class label, 即cross-entropy loss is:

9、Optimization methods and regularization

optimization algorithm: iteratively evaluate your parameters, compute your loss, then take a small step in the direction that will minimize your loss.

Compute the gradient W across all dimensions using the following equation:

In >1 dimensions, our gradient becomes a vector of partial derivatives. 该求梯度方法存在的问题为：1) it is an approximation to the gradient; 2) it is painfully slow.

What gradient descent is? Attempting to optimize our parameters for low loss and high classification accuracy via an iterative process of taking a step in the direction that minimizes loss.

Optimization algorithm may not be guaranteed to arrive at even a local minimum in a reasonable amount of time, but it often finds a very low value of the function quickly enough to be useful. – Goodfellow

bias trick: a method of combining our weight matrix W and bias vector b into a single parameter. To combine both the bias and weigh matrix, we add an extra dimension to our input data X that holds a constant 1 – this is our bias dimension. Allowing us to learn only a single matrix of weights.

Good initialization is critical to training a neural network in a reasonable amount of time, so random initialization along with sinple heuristics win out in the vast majority of circumstances.

Vanilla gradient descent only performs a weight update once for every epoch.

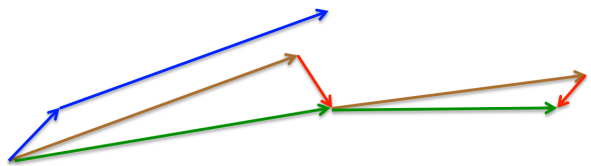
Stochastic Gradient Descent performs a weight update for every batch of training data, implying there are multiple weight updates per epoch. This approach leads to a faster, more stable convergence.

Typical batch sizes include 32, 64, 128, and 256.

Momentum: a method used to accelerate SGD, enabling it to learn faster by focusing on dimensions whose gradient point in the same direction.

The momentum term is commonly set to 0.9. Another common practice is to set to 0.5 until learning stabilizes and then increase it to 0.9.

Nesterov acceleration can be conceptualized as a corrective update to the momentum which lets us obtain an approximate idea of where our parameters will be after the update. Under Nesterov acceleration we would first make a big jump in the direction of our previous gradient (brown vector), measure the gradient, and then make a correction.



Rule of thumb: whenever using SGD, also apply momentum. In most cases, you can set it to 0.9. Although Karpathy suggests starting at 0.5 and increasing it to larger values as your epochs increase. SGD os easier to work with large datasets when using momentum, while, smaller datasets tend to enjoy the benefits of Nestetov acceleration.

Various types of regularization techniques: L1 regularization, L2 regularization (weight decay), Elastic Net, dropout, data augmentation, early stopping.

Regularization helps us control our model capacity.

A regularization penalty, a function that operates on our weight matrix, commonly written as a function, .

加入正则化后，损失函数变为：

Both the learning rate and the regularization term are the hyperparameters that you will spending the most time tuning.

Standard weight update rule:

Taking into account regularization, the weight uodate rule becomes:

Which regularization method you should use?

Treating this choice as a hyperparameter you need to optimize over and perform experiments to determine if regularization should be applied, and if so which method of regularization, and what the proper value of is.

Regularization can provide a boost in our testing accuracy and reduce overfitting, provided we can tune the hyperparameters right.

10、Neural network fundamentals

directed graph：有向图

Recommand starting with a ReLU to obtain a baseline accuracy, tune my network and optimizer parameters (architecture, learning rate, regularization strength, etc), and note the accuracy. Once reasonably satisfied with the acuracy, swap in an ELU.

Feedforward neural network: a connection between nodes is only allowed from nodes in layer to nodes in layer . There ar no backward or inter-layer connections allowed.

Perceptron training procedure:

1. Initialize our weight vector with small random values.
2. Until Perceptron converges:
   1. Loop over each feature vector and true class label in our training set D;
   2. Take and pass it through the network, calculating the output value: ;
   3. Update the weights : for all features .

The Perceptron training process is allowed to proceed until all training samples are classified correctly or a preset number of epochs is reached. it will never be able to correctly model the XOR function with a single layer Perceptron.

|  |
| --- |
| 1. **class** Perceptron: 2. **def** \_\_init\_\_(self, N, alpha=0.1): 3. # N: the number of columns in our input feature vectors 4. self.W = np.random.randn(N + 1) / np.sqrt(N) 5. self.alpha = alpha 7. **def** step(self, x): 8. **return** 1 **if** x > 0 **else** 0 10. **def** fit(self, X, y, epochs=10): 11. X = np.c\_[X, np.ones((X.shape[0]))] 12. **for** epoch **in** np.arange(0, epochs): 13. **for** (x, target) **in** zip(X, y): 14. p = self.step(np.dot(x, self.W)) 15. **if** p != target: 16. error = p - target 17. self.W += -self.alpha \* error \* x 19. **def** predict(self, X, addBias=True): 20. X = np.atleast\_2d(X) 21. **if** addBias: 22. X = np.c\_[X, np.ones((X.shape[0]))] 23. **return** self.step(np.dot(X, self.W)) |

The backpropagation algorithm phases:

1. The forward pass where our inputs are passed through the network and output predictions obtained.
2. The backward pass where we compute the gradient of the loss function at the final layer of the network and use this gradient to recursively apply the chain rule to update the weights in our network.

|  |
| --- |
| 1. **class** NeuralNetwork: 2. **def** \_\_init\_\_(self, layers, alpha=0.1): 3. # layers: list of integers, represent the actual architecture of the feedforward network 4. self.W = [] 5. self.layers = layers 6. self.alpha = alpha 8. **for** i **in** np.arange(0, len(layers) - 2): 9. w = np.random.randn(layers[i] + 1, layers[i+1] + 1) 10. self.W.append(w / np.sqrt(layers[i])) 11. w = np.random.randn(layers[-2] + 1, layers[-1]) 12. self.W.append(w / np.sqrt(layers[-2])) 14. **def** \_\_repr\_\_(self): 15. **return** "NeuralNetwork: {}".format("-".join(str(l) **for** l **in** self.layers)) 17. **def** sigmoid(self, x): 18. **return** 1.0 / (1 + np.exp(-x)) 20. **def** sigmoid\_deriv(self, x): 21. **return** x \* (1 - x) 23. **def** fit(self, X, y, epochs=1000, displayUpdate=100): 24. X = np.c\_[X, np.ones((X.shape[0]))] 25. **for** epoch **in** np.arange(0, epochs): 26. **for** (x, target) **in** zip(X, y): 27. self.fit\_partial(x, target) 28. **if** epoch == 0 **or** (epoch + 1) % displayUpdate == 0: 29. loss = self.calculate\_loss(X, y) 30. **print**("[INFO] epoch={}, loss={:.7f}".format(epoch + 1, loss)) 32. **def** fit\_partial(self, x, y): 33. # x: an individual data point from our design matrix 34. # y: the corresponding class label 35. A = [np.atleast\_2d(x)] 37. # feedforward 38. **for** layer **in** np.arange(0, len(self.W)): 39. net = A[layer].dot(self.W[layer]) 40. out = self.sigmoid(net) 41. A.append(out) 43. # backpropagation 44. error = A[-1] - y 45. D = [error \* self.sigmoid\_deriv(A[-1])] 46. **for** layer **in** np.arange(len(A) - 2, 0, -1): 47. delta = D[-1].dot(self.W[layer].T) 48. delta = delta \* self.sigmoid\_deriv(A[layer]) 49. D.append(delta) 51. D = D[::-1] 52. **for** layer **in** np.arange(0, len(self.W)): 53. self.W[layer] += -self.alpha \* A[layer].T.dot(D[layer]) 55. **def** predict(self, X, addBias=True): 56. p = np.atleast\_2d(X) 57. **if** addBias: 58. p = np.c\_[p, np.ones((p.shape[0]))] 59. **for** layer **in** np.arange(0, len(self.W)): 60. p = self.sigmoid(np.dot(p, self.W[layer])) 61. **return** p 63. **def** calculate\_loss(self, X, targets): 64. targets = np.atleast\_2d(targets) 65. predictions = self.predict(X, addBias=False) 66. loss = 0.5 \* np.sum((predictions - targets) \*\* 2) 67. **return** loss |

Multi-layer networks with nonlinear activation functions trained via backpropagation are so important – they enable us to learn patterns in datasets that are otherwise nonlinearly separable.

Basic feedforward networks with strictly fully-connected layers are not suitable for challenging image datasets.

four main ingredients in a neural network: a dataset, a model/architecture, a loss function, an optimization method.

How we initialize our weight matrices and bias vectors?

1. Constant initialization: 假设某一层有64个input，32个output，初始化为.

It is near impossible for us to break the symmetry of activations.

1. Uniform and normal distributions

|  |
| --- |
| 1. W1 = np.random.uniform(low=-0.05, high=0.05, size=(64, 32)) 2. W2 = np.random.normal(0.0, 0.5, size=(64, 32)) |

1. LeCun Uniform and Normal
2. Glorot/Xavier Uniform and Normal
3. He et al./Kaiming/MSRA Uniform and Normal

11、Convolutional Neural Network

CNNs give us two key benefits:

1. local invariance: allow us to classify an image as containing a particular object regardless of where in the image the object appears.
2. compositionality: each filter composes a local patch of lower-level features into a higher-level representation, this composition allows our network to learn more rich features deeper in the network.

In terms of deep learning, an (image) convolution is an element-wise multiplication of two matrices followed by a sum.

Convolution over a two-dimensional input image I and two-dimensional kernel K is defined as:

We can think of an image as big matrix and a kernel or convolutional matrix as a tiny matrix that is used for blurring, sharpening, edge detection, and other processing functions. Essentially, this tiny kernel sits on top of the big image and slides from left-to-right and top-to-buttom, applying a mathematical operation at each (x,y)-coordinate of the original image.

Our pixel coordinates must be integers! This reasoning is exactly why we use odd kernel sizes: to always ensure there is a valid (x,y)-coordinate at the center of the kernel.

The process of convolution:

1. Select an (x,y)-coordinate from the original image;
2. Place the center of the kernel at this (x,y)-coordinate;
3. Take the element-wise multiplication of the input image region and the kernel, then sum up the values of these multiplication operations into a single value. The sum of these multiplications is called the kernel output.
4. Use the same (x,y)-coordinates from Step#1, but this time, store the kernel output at the same (x,y)-location as the output image.

In most cases, we want our output image to have the same dimensions as our input image. To ensure the dimensions are the same, we apply padding.

Padding methods: replicate padding, zero padding.

Replicate padding is more commonly used when aesthetics are concerned while zero padding is best for efficiency.

|  |
| --- |
| 1. **from** skimage.exposure **import** rescale\_intensity 3. **def** convolve(image, K): 4. (iH, iW) = image.shape[:2] 5. (kH, kW) = K.shape[:2] 7. pad = (kW - 1) // 2 8. image = cv2.copyMakeBorder(image, pad, pad, pad, pad, cv2.BORDER\_REPLICATE) 9. output = np.zeros((iH, iW), dtype='float') 11. **for** y **in** np.arange(pad, iH + pad): 12. **for** x **in** np.arange(pad, iW + pad): 13. # extracts the Region of Interest from the image 14. roi = image[y-pad:y+pad+1, x-pad:x+pad+1] 15. k = (roi \* K).sum() 16. output[y-pad, x-pad] = k 18. # bring our output image back into the range [0, 255] 19. output = rescale\_intensity(output, in\_range=(0, 255)) 20. output = (output \* 255).astype('uint8') 21. **return** output 23. sharpen = np.array(([0, -1, 0], 24. [-1, 5, -1], 25. [0, -1, 0]), dtype='int') 27. image = cv2.imread(args['image']) 28. gray = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY) 29. convolveOutput = convolve(gray, sharpen) |

Neurons in subsequent layers will only be connected to a small region of the layer before it – we call this local connectivity which enables us to save a huge amount of parameters in our network.

Simple text diagrams describe of CNN: INPUT -> CONV -> POOL -> FC -> SOFTMAX.

Assuming that an activation always follows a convolution.

For inputs to the CNN, the depth is the numbe of channels in the image. For volumne deeper in the network, the depth will be the number of filters applied in the previous layer.

Considering the forward-pass of a CNN:

1. K kernels wating to applied to the image;
2. Eah kernel is convolved with the input volume;
3. The output of each convolution operation produces a 2D output, called an activation map.
4. Stack our K activation maps along the depth dimension of our array to form the final output volume.

receptive field 感受野

The network learns filters that activate when they see a specific type of feature at a given spatial location in the input volume.

When utilizing CNNs, we choose to connect each neuron to only a local region of the input volume – we call the size of this local region the receptive field of the neuron.

Assuming the volume is now 16\*16\*94. If we assume a receptive field of size 3\*3, then everu neuron in the CONV layer will have a total of 3\*3\*94=846 connections to the input volume.

Three parameters that control the size of an output volume: depth, stride, zero-padding size.

Without zero-padding, the spatial dimensions of the input volume would decrease too quickly, and we would not be able to train deep networks.

The number of filters K, which controls the depth of the output volume.

Compute the size of output volume as a function of the input volume size:

其中，W: the input volume size. F: the receptive field size. S: the stride. P: the amount of zero-padding. If it is not an integer, then the strides are set incorrectly.

The primary function of the POOL layer is to progressively reduce the spatial size of the input volume. Doing this allows us to reduce the amount of parameters and computations in the network – pooling also helps us control overfitting.

Max pooling is typically done in the middle of the CNN architecture to reduce spatial size, whereas average pooling is normally used as the final layer of the network. F一般为2或3，S一般为1或2。

It is becoming increasingly more common to not use POOL layers in the middle of the network architecture and only use average pooling at the end of the network if FC layers are to be avoided.

It’s common to use one or two FC layers prior to applying the softmax classifier.

Batch Normalization layer: used to normalize the activations of a given input volume before passing it into the next layer in the network.

We compute the and over mini-batch , where:

,

We set equal to a small positive value such as 1e-7 to avoid taking the square root of zeo.

使用BN的好处：

1. have approximately zero mean and unit variance.
2. extremely effective at reducing the number of epochs it takes to train.
3. lower final loss and a more stable loss curve.

Placing the BN after the nonlinear activation yields higher accuracy and lower loss in nearly all situations.

Dropout is actually a form of regularization that aims to help prevent overfitting by increasing testing accuracy. Dropout layers, with probability p, randomly disconnect inputs from the preceding layer to the next layer in the network architecture.

It is most common to place dropout layers with p=0.5 in-between FC layers of an architecture where the final FC layer is assumed to be our softmax classifier.

Most common CNN architecture:

INPUT -> [[CONV => RELU => BN?]\*N => POOL? => DO?]\*M => [FC => RELU => BN? => DO?]\*K => FC

其中，, , .

Stacking multiple CONV layers before applying a POOL layer allows the CONV layers to develop more complex features before the destructive pooling operation is performed.

12、Training your first CN

- Create our image-to-array peprocessor

|  |
| --- |
| 1. **from** keras.preprocessing.image **import** img\_to\_array 3. **class** ImageToArrayPreprocessor: 4. **def** \_\_init\_\_(self, dataFormat=None): 5. self.dataFormat = dataFormat 7. **def** preprocess(self, image): 8. **return** img\_to\_array(image, data\_format=self.dataFormat) 9. sp = SimplePreprocessor(32, 32) 10. iap = ImageToArrayPreprocessor() 11. sdl = SimpleDatasetLoader(preprocessors=[sp, iap]) 12. (data, labels) = sdl.load(imagePaths, verbose=500) |

The benefit of defining a class to handle this type of image preprocessing rather than simply calling img\_to\_array on every single image is that we can now chain preprocessors together as we load datasets from disk.

|  |
| --- |
| 1. **from** keras.models **import** Sequential 2. **from** keras.layers **import** Conv2D 3. **from** keras.layers **import** Activation 4. **from** keras.layers **import** Flatten 5. **from** keras.layers **import** Dense 6. **from** keras **import** backend as K 8. # prefer to define network architecture inside a class 9. # to keep the code organized 10. **class** ShallowNet: 11. @staticmethod 12. **def** build(width, height, depth, classes): 13. model = Sequential() 14. inputShape = (height, width, depth) 15. **if** K.image\_data\_format() == 'channels\_first': 16. inputShape = (depth, height, width) 18. model.add(Conv2D(32, (3, 3), padding='same', input\_shape=inputShape)) 19. model.add(Activation('relu')) 20. model.add(Flatten()) 21. model.add(Dense(classes)) 22. model.add(Activation('softmax')) 23. **return** model |

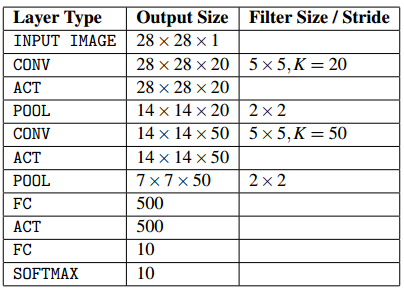
13、Saving and loading your models

Model serialization: a way to save our model to disk after training is complete and then simply load it from disk when we want to classify new images.

Always take special care to ensure your testing images were preprocessed in the same way as your training images.

14、LeNet：Recognizing handwritten digits

LeNet architecture: INPUT => CONV => TANH => POOL => CONV => TANH => POOL => FC => TANH => FC.



15、MiniGGNet: Going deeper with CNNs

VGGNet: uses 3\*3 kernels throughout the entire architecture, stacking multiple CONV=>RELU layer sets before applying a POOL operation.

Doing this allows the network to learn more rich features from the CONV layers prior to downsampling the spatial input size via the POOL operation.

CONV=>RELU=>BN: allows our network to learn more rich features, a common practice when training deeper CNNs.

It is common to increase the number of filters as the spatial input size decreases deeper in the network.

Typically you will see dropout with p=0.5 applied in between FC layers.

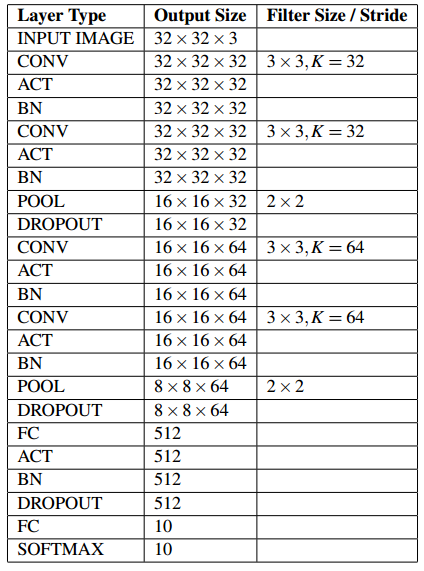
Decaying the learning rate is helpful in reducing overfitting and obtaining high classification accuracy – the smaller the learning rate is, the smaller the weight updates will be. A common setting for decay is to divide the initial learning rate by the total number of epochs.

为了查看BN是否对模型的预测准确率有效，可在有BN和无BN时分别运行模型。

Batch normalization can lead to a faster, more stable convergence with higher accuracy.

However, the advantages will come at the expense of training time – batch normalization will require more “wall time”to train the network.

A table summary of the MiniVGGNet architecture



16、Learning rate schedulers

Two primary types of learning rate schedulers:

1. Learning rate schedulers that decrease gradually based on the epoch number.
2. Learning rate schedulers that drop based on specific epoch.

The rule of thumb of time-based schedule: decay = alpha\_init / epochs.

|  |
| --- |
| 1. opt = SGD(lr=0.01, decay=0.01/40, momentum=0.9, nesterov=True) |

Step-based decay: systematically drop the learning rate after specific epochs during training.

Two options for applying step decay:

1. Define an equation that models the piecewise drop in learning rate we wish to achieve.
2. After some number of epochs at a given learning rate, ctrl+c to stop the scripts, adjust our learning rate, and continue training.

When applying step decay, we often drop our learning rate by either half or an order of magnitude after every fixed number of epochs.

LearningRateScheduler class: allow us to define a custom learning rate function and then have it automatically applied during the training process. This function should take the epoch number as an argument and then compute our desired learning rate based on a function that we define.

其中，: initial learning rate, : the factor value controlling the rate in which the learning rate drops, : the drop every epochs value, : the current epoch. The larger our factor F is, the slower the learning rate will decay.

|  |
| --- |
| 1. **def** step\_decay(epoch): 2. initAlpha = 0.01 3. factor = 0.5 4. dropEvery = 5 5. alpha = initAlpha \* (factor \*\* np.floor((1 + epoch) / dropEvery)) 6. **return** float(alpha) 8. **print**("[INFO] compiling network...") 9. callbacks = [LearningRateScheduler(step\_decay)] 10. opt = SGD(lr=0.01, decay=0.01/40, momentum=0.9, nesterov=True) 11. model = MiniVGGNet.build(width=32, height=32, depth=3, classes=10) 12. model.compile(loss='categorical\_crossentropy', optimizer=opt, metrics=['accuracy']) 14. **print**("[INFO] training network...") 15. H = model.fit(trainX, trainY, validation\_data=(testX, testY), batch\_size=64, epochs=40, verbose=1, callbacks=callbacks) |

Typically your first experiment would not use any type of decay or learning rate scheduling so you can obtain a baseline accuracy and loss/accuracy curve.

然后，尝试使用time-based schedule和step-based decay.

Nothing beats actually ruuning the experiments yourself.

17、Spotting underfitting and overfitting

Our goal when training a machine learning model:

1. Reduce the training loss as much as possible.
2. While ensuring the gap between the training and testing loss is reasonably small.

An increase in validation loss over a series of consecutive epochs is a heavy indicator of overfitting. Simply need to mind the gap between the training and validation loss. As long as the gap does not increase dramatically, we know there is an acceptable level of overfitting.

How do we combat overfitting?

1. Reduce the complexity of the model, opting for a more shallow network with less layers and neurons.
2. Apply regularization mothods, such as weight decay, dropout, data augmentation, etc.

当出现overfitting时，从以下各方面评估模型：

1. Are you applying any regularization techniques?
2. Is your learning rate too high?
3. Is your network too deep?

How we can create a real-time training monitor for Keras.

TrainingMonitor callback: be called at the end of every epoch when training a network with Keras.

|  |
| --- |
| 1. **from** keras.callbacks **import** BaseLogger 2. **import** matplotlib.pyplot as plt 3. **import** numpy as np 4. **import** json 5. **import** os  8. **class** TrainingMonitor(BaseLogger): 9. **def** \_\_init\_\_(self, figPath, jsonPath=None, startAt=0): 10. super(TrainingMonitor, self).\_\_init\_\_() 11. self.figPath = figPath 12. self.jsonPath = jsonPath 13. self.startAt = startAt 15. **def** on\_train\_begin(self, logs={}): 16. # initialize the history dictionary 17. self.H = {} 18. **if** self.jsonPath **is** **not** None: 19. **if** os.path.exists(self.jsonPath): 20. # loading the training history 21. self.H = json.load(open(self.jsonPath).read()) 22. **if** self.startAt > 0: 23. **for** k **in** self.H.keys(): 24. self.H[k] = self.H[k][:self.startAt] 26. # Each list is updated at the end of every epochs, 27. # thus enabling us to plot an updated loss and accuracy curve 28. # as soon as the epoch completes. 29. **def** on\_epoch\_end(self, epoch, logs={}): 30. # logs: contains the training and validation loss + accuracy for the current epoch 31. **for** (k, v) **in** logs.items(): 32. l = self.H.get(k, []) 33. l.append(v) 34. self.H[k] = l 36. **if** self.jsonPath **is** **not** None: 37. f = open(self.jsonPath, "w") 38. f.write(json.dumps(self.H)) 39. f.close() 41. **if** len(self.H['loss']) > 1: 42. N = np.arange(0, len(self.H['loss'])) 43. plt.style.use('ggplot') 44. plt.figure() 45. plt.plot(N, self.H['loss'], label='train\_loss') 46. plt.plot(N, self.H['val\_loss'], label='val\_loss') 47. plt.plot(N, self.H['acc'], label='train\_acc') 48. plt.plot(N, self.H['val\_acc'], label='val\_acc') 49. plt.title('Training Loss and Accuracy [Epoch {}]'.format(len(self.H['loss']))) 50. plt.xlabel('Epoch #') 51. plt.ylabel('Loss/Accuracy') 52. plt.legend() 53. plt.savefig(self.figPath) 54. plt.close() |

Wait until you see clear signs of overfitting, then kill the process.

In nearly all situations you should first attempt applying stronger regularization than reducing the size of your network – the exception being if you are attempting to train a massively deep network on a tiny dataset.

18、Checkpointing models

A good application of checkpointing is to serialize your network to disk each time there is an improvement during training.

|  |
| --- |
| 1. fname = os.path.sep.join([args['weights'], 'weights-{epoch:03d}-{val\_loss:.4f}.hdf5']) 2. checkpoint = ModelCheckpoint(fname,monitor='val\_loss',mode='min',save\_best\_only=True, verbose=1) 3. callbacks = [checkpoint] 5. **print**("[INFO] training network...") 6. H = model.fit(trainX, trainY, validation\_data=(testX, testY), 7. batch\_size=64, epochs=40, callbacks=callbacks, verbose=1) |

若只想保存一个最优的结果，ModelCheckpoint的fname不能有template variables.