Determinant

The absolute value of the determinant can be thought of as a measure of how much multiplication by the matrix expands or contracts space. If the determinant is 0, then space is contracted completely along at least one dimension, causing it to lose all of its volume. If the determinant is 1, then the transformation preserves volume.

Marginal Probability

Marginal Probability: Sometimes we know the probability distribution over a set of variables and we want to know the probability distribution over just a subset of them. The probability distribution over the subset is known as the marginal probability distribution

$$\forall x \in \mathrm{x}, P(\mathrm{x} = x) = \sum_{y} P(\mathrm{x} = x, \mathrm{y} = y).$$

$$p(x) = \int p(x, y) dy.$$

Covariance

The covariance gives some sense of how much two values are linearly related to each other, as well as the scale of these variables. $\mathrm{Cov}(f(x), g(y)) = \mathbb{E}\left[(f(x) - \mathbb{E}\left[f(x)\right])\left(g(y) - \mathbb{E}\left[g(y)\right]\right)\right]$

Covariance Matrix

The covariance matrix of a random vector x ∈ Rn is an n × n matrix, such that

$$\mathrm{Cov}(\mathrm{x})_{i,j} = \mathrm{Cov}(\mathrm{x}_i, \mathrm{x}_j)$$

The diagonal elements of the covariance give the variance:

$$\mathrm{Cov}(\mathrm{x}_i, \mathrm{x}_i) = \mathrm{Var}(\mathrm{x}_i)$$

The following properties are all useful enough that you may wish to memorize them:

$$\sigma(x) = \frac{\exp(x)}{\exp(x) + \exp(0)}$$

$$\frac{d}{dx}\sigma(x) = \sigma(x)(1 - \sigma(x))$$

$$1 - \sigma(x) = \sigma(-x)$$

$$\log \sigma(x) = -\zeta(-x)$$

$$\frac{d}{dx}\zeta(x) = \sigma(x)$$

$$\forall x \in (0, 1),\ \sigma^{-1}(x) = \log\left(\frac{x}{1 - x}\right)$$

$$\forall x > 0,\ \zeta^{-1}(x) = \log\left(\exp(x) - 1\right)$$

$$\zeta(x) = \int_{-\infty}^{x} \sigma(y)dy$$

$$\zeta(x) - \zeta(-x) = x$$

Bayes' Rule:

$$P(\mathrm{x} \mid \mathrm{y}) = \frac{P(\mathrm{x})P(\mathrm{y} \mid \mathrm{x})}{P(\mathrm{y})}.$$

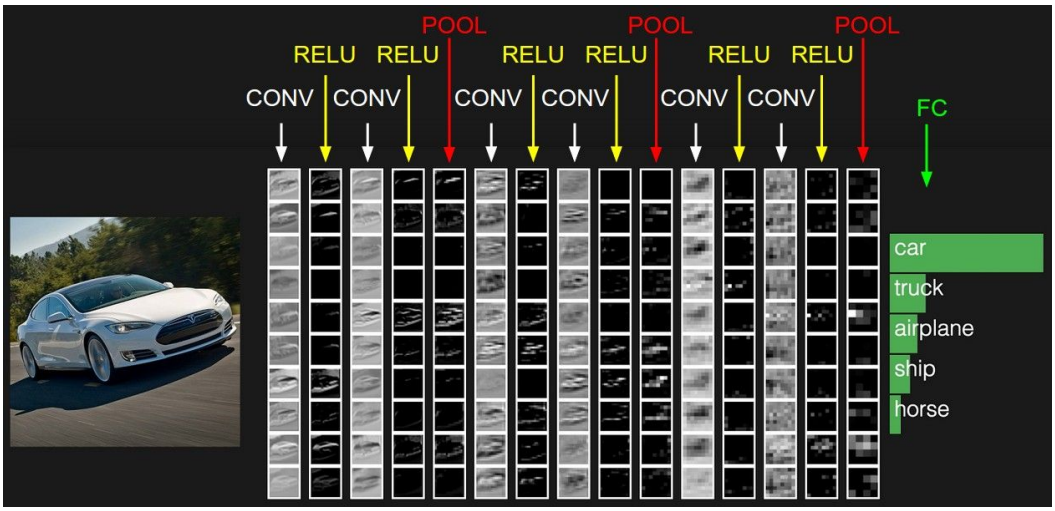
Note that while $P(\mathrm{y})$ appears in the formula, it is usually feasible to compute $P(\mathrm{y}) = \sum_x P(\mathrm{y} \mid x)P(x)$, so we do not need to begin with knowledge of $P(\mathrm{y})$.

A simplified ConvNet and its structure:

Example Architecture: Overview. We will go into more details below, but a simple ConvNet for CIFAR-10 classification could have the architecture [INPUT - CONV - RELU - POOL - FC]. In more detail:

- INPUT [32x32x3] will hold the raw pixel values of the image, in this case an image of width 32, height 32, and with three color channels R,G,B.
- CONV layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. This may result in volume such as [32x32x12] if we decided to use 12 filters.

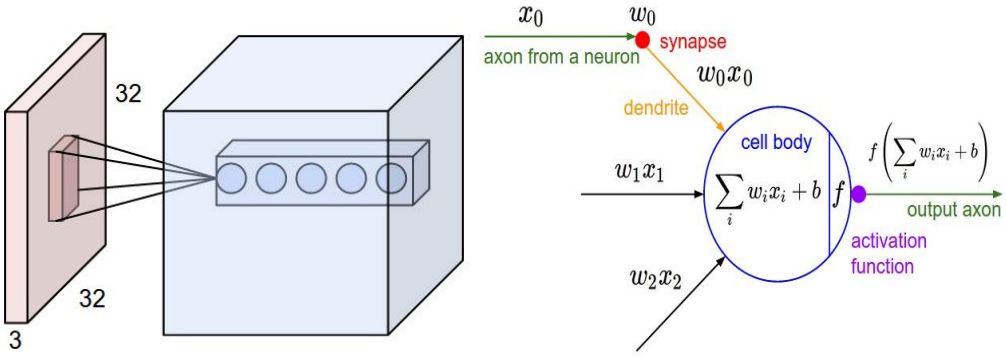
- RELU layer will apply an elementwise activation function, such as the max(0,x)max(0,x) thresholding at zero. This leaves the size of the volume unchanged ([32x32x12]).
- POOL layer will perform a downsampling operation along the spatial dimensions (width, height), resulting in volume such as [16x16x12].
- FC (i.e. fully-connected) layer will compute the class scores, resulting in volume of size [1x1x10], where each of the 10 numbers correspond to a class score, such as among the 10 categories of CIFAR-10. As with ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume.



The activations of an example ConvNet architecture. The initial volume stores the raw image pixels (left) and the last volume stores the class scores (right). Each volume of activations along the processing path is shown as a column. Since it's difficult to visualize 3D volumes, we lay out each volume's slices in rows. The last layer volume holds the scores for each class, but here we only visualize the sorted top 5 scores, and print the labels of each one.

Convolutional Neural Network vs. Neural Network:

Left: An example input volume in red (e.g. a 32x32x3 CIFAR-10 image), and an example volume of neurons in the first Convolutional layer. Each neuron in the convolutional layer is connected only to a local region in the input volume spatially, but to the full depth (i.e. all color channels). Note, there are multiple neurons (5 in this example) along the depth, all looking at the same region in the input - see discussion of depth columns in text below.
Right: The neurons from the Neural Network chapter remain unchanged: They still compute a dot product of their weights with the input followed by a non-linearity, but their connectivity is now restricted to be local spatially.

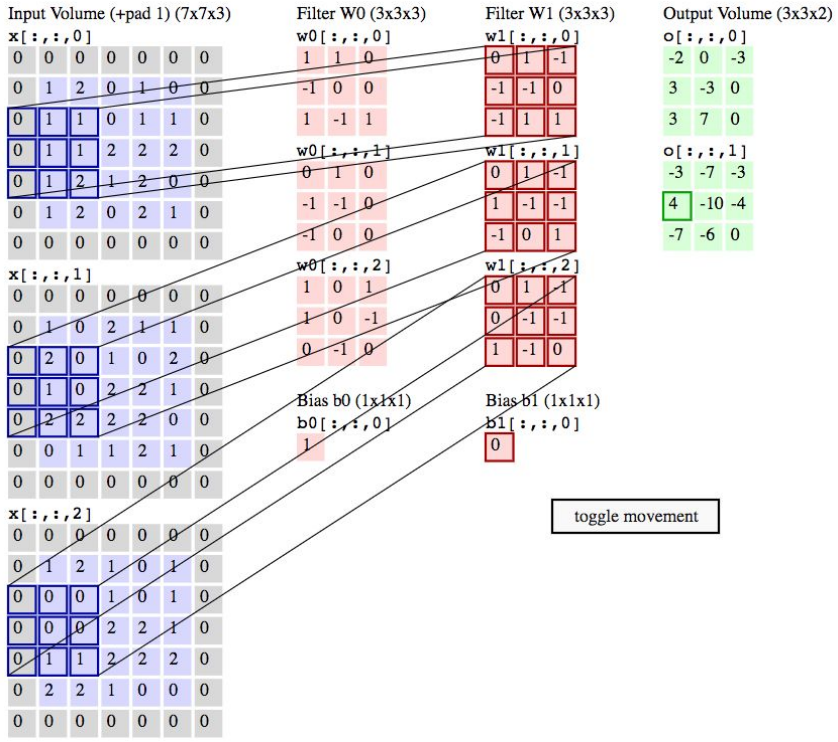


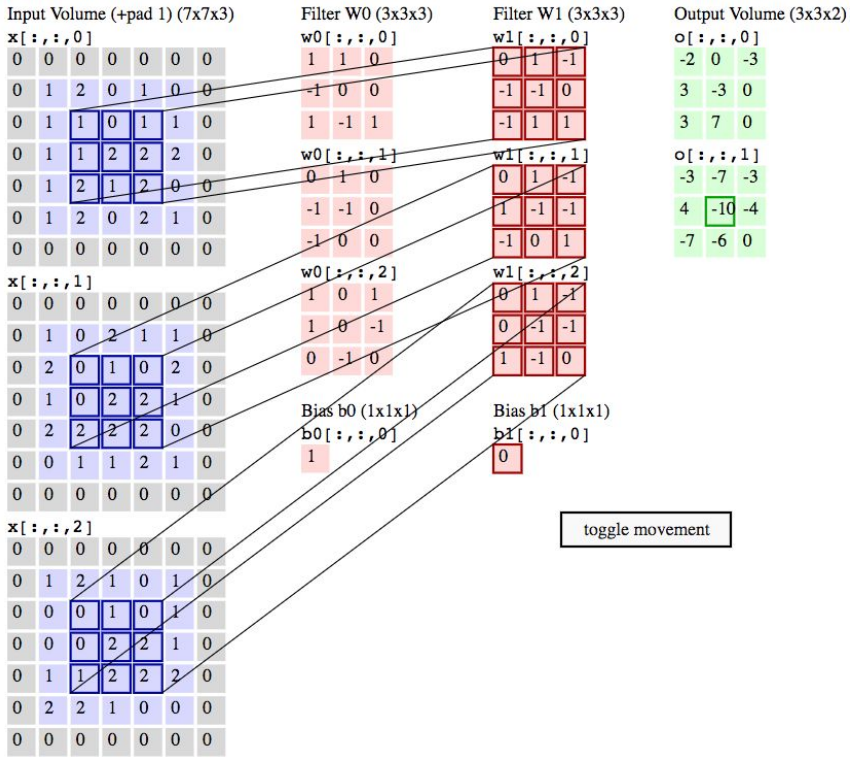
Parameter Sharing:

We can dramatically reduce the number of parameters by making one reasonable assumption: That if one feature is useful to compute at some spatial position (x,y), then it should also be useful to compute at a different position (x2,y2). In other words, denoting a single 2-dimensional slice of depth as a depth slice (e.g. a volume of size [55x55x96] has 96 depth slices, each of size [55x55]), we are going to constrain the neurons in each depth slice to use the same weights and bias. With this parameter sharing scheme, the first Conv Layer in our example would now have only 96 unique set of weights (one for each depth slice), for a total of 96*11*11*3 = 34,848 unique weights, or 34,944 parameters (+96 biases). Alternatively, all 55*55 neurons in each depth slice will now be using the same parameters. In practice during backpropagation, every neuron in the volume will compute the gradient for its weights, but these gradients will be added up across each depth slice and only update a single set of weights per slice.

If detecting a horizontal edge is important at some location in the image, it should intuitively be useful at some other location as well due to the translationally-invariant structure of images. There is therefore no need to relearn to detect a horizontal edge at every one of the 55*55 distinct locations in the Conv layer output volume.

Convolutional Neural Network Demo:





Implemented as Matrix Multiplication:

- The local regions in the input image are stretched out into columns in an operation commonly called im2col. For example, if the input is [227x227x3] and it is to be convolved with 11x11x3 filters at stride 4, then we would take [11x11x3] blocks of pixels in the input and stretch each block into a column vector of size 11*11*3 = 363. Iterating this process in the input at stride of 4 gives (227-11)/4+1 = 55 locations along both width and height, leading to an output matrix X_col of im2col of size [363 x 3025], where every column is a stretched out receptive field and there are 55*55 = 3025 of them in total. Note that since the receptive fields overlap, every number in the input volume may be duplicated in multiple distinct columns.

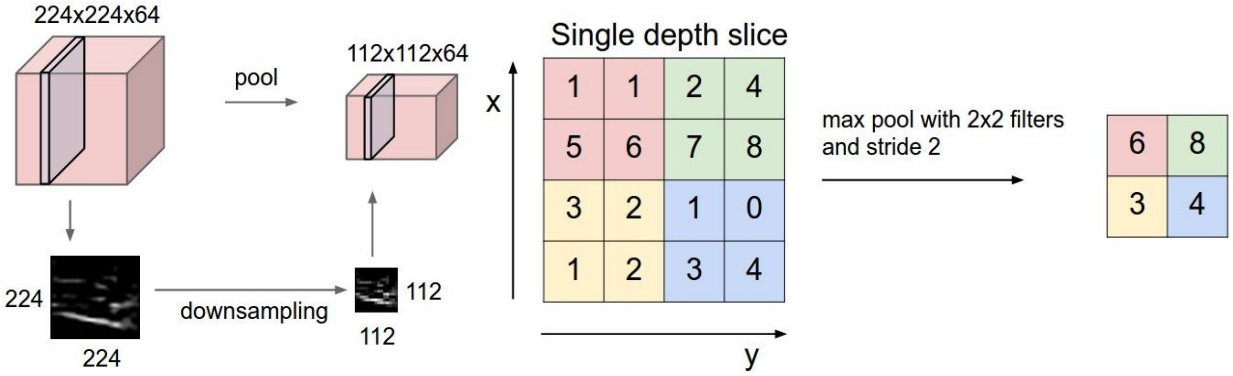
- The weights of the CONV layer are similarly stretched out into rows. For example, if there are 96 filters of size [11x11x3] this would give a matrix W_row of size [96 x 363].

- The result of a convolution is now equivalent to performing one large matrix multiply np.dot(W_row, X_col), which evaluates the dot product between every filter and every receptive field location. In our example, the output of this operation would be [96 x 3025], giving the output of the dot product of each filter at each location.

- The result must finally be reshaped back to its proper output dimension [55x55x96].

Pooling Layer:

It is common to periodically insert a Pooling layer in-between successive Conv layers in a ConvNet architecture. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting.



Layer Sizing Patterns

Until now we've omitted mentions of common hyperparameters used in each of the layers in a ConvNet. We will first state the common rules of thumb for sizing the architectures and then follow the rules with a discussion of the notation:

The **input layer** (that contains the image) should be divisible by 2 many times. Common numbers include 32 (e.g. CIFAR-10), 64, 96 (e.g. STL-10), or 224 (e.g. common ImageNet ConvNets), 384, and 512.

The **conv layers** should be using small filters (e.g. 3x3 or at most 5x5), using a stride of $S=1$, and crucially, padding the input volume with zeros in such way that the conv layer does not alter the spatial dimensions of the input. That is, when $F=3$, then using $P=1$ will retain the original size of the input. When $F=5$, $P=2$. For a general F, it can be seen that $P=(F-1)/2$ preserves the input size. If you must use bigger filter sizes (such as 7x7 or so), it is only common to see this on the very first conv layer that is looking at the input image.

The **pool layers** are in charge of downsampling the spatial dimensions of the input. The most common setting is to use max-pooling with 2x2 receptive fields (i.e. $F=2$), and with a stride of 2 (i.e. $S=2$). Note that this discards exactly 75% of the activations in an input volume (due to downsampling by 2 in both width and height). Another slightly less common setting is to use 3x3 receptive fields with a stride of 2, but this makes. It is very uncommon to see receptive field sizes for max pooling that are larger than 3 because the pooling is then too lossy and aggressive. This usually leads to worse performance.

Reducing sizing headaches. The scheme presented above is pleasing because all the CONV layers preserve the spatial size of their input, while the POOL layers alone are in charge of down-sampling the volumes spatially. In an alternative scheme where we use strides greater than 1 or don't zero-pad the input in CONV layers, we would have to very carefully keep track of the input volumes throughout the CNN architecture and make sure that all strides and filters "work out", and that the ConvNet architecture is nicely and symmetrically wired.

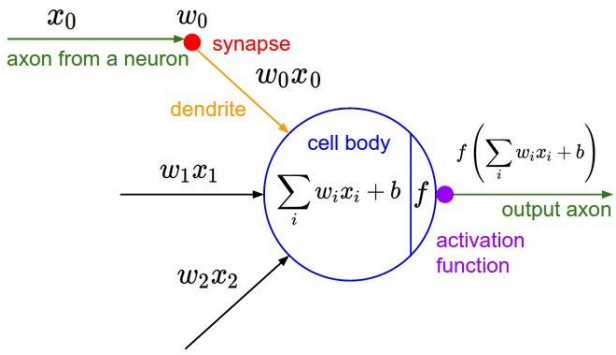
Why use stride of 1 in CONV? Smaller strides work better in practice. Additionally, as already mentioned stride 1 allows us to leave all spatial down-sampling to the POOL layers, with the CONV layers only transforming the input volume depth-wise.

Why use padding? In addition to the aforementioned benefit of keeping the spatial sizes constant after CONV, doing this actually improves performance. If the CONV layers were to not zero-pad the inputs and only perform valid convolutions, then the size of the volumes would reduce by a small amount after each CONV, and the information at the borders would be "washed away" too quickly.

Compromising based on memory constraints. In some cases (especially early in the ConvNet architectures), the amount of memory can build up very quickly with the rules of thumb presented above. For example, filtering a 224x224x3 image with three 3x3 CONV layers with 64 filters each and padding 1 would create three activation volumes of size [224x224x64]. This amounts to a total of about 10 million activations, or 72MB of memory (per image, for both activations and gradients). Since GPUs are often bottlenecked by memory, it may be necessary to compromise. In practice, people prefer to make the compromise at only the first CONV layer of the network. For example, one compromise might be to use a first CONV layer with filter sizes of 7x7 and stride of 2 (as seen in a ZF net). As another example, an AlexNet uses filter sizes of 11x11 and stride of 4.

An example code for forward-propagating a single neuron:

```
class Neuron(object):
  # ...
  def forward(self, inputs):
    """ assume inputs and weights are 1-D numpy arrays and bias is a number """
    cell_body_sum = np.sum(inputs * self.weights) + self.bias
    firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum)) # sigmoid activation function
    return firing_rate
```



What neuron type should I use:

Use the ReLU non-linearity, be careful with your learning rates and possibly monitor the fraction of “dead” units in a network. If this concerns you, give Leaky ReLU or Maxout a try. Never use sigmoid. Try tanh, but expect it to work worse than ReLU/Maxout.

Neural Network With At Least One Hidden Layer Are Universal Approximator:

It turns out that Neural Networks with at least one hidden layer are *universal approximators*. That is, it can be shown that given any continuous function f(x) and some $\epsilon>0$, there exists a Neural Network g(x) with one hidden layer (with a reasonable choice of non-linearity, e.g. sigmoid) such that $\forall x, |f(x)-g(x)|<\epsilon$. In other words, the neural network can approximate any continuous function.

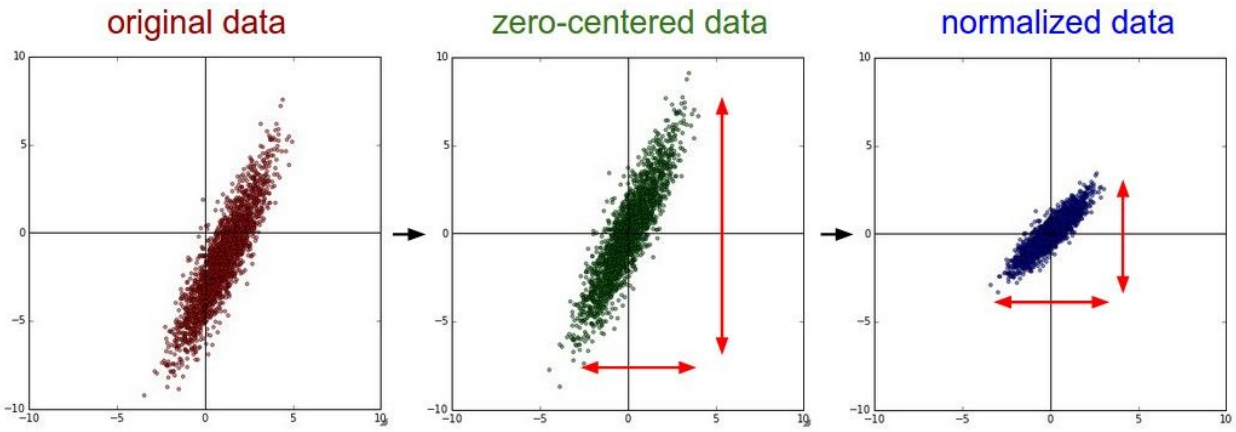
Smaller Neural Network is Preferred? WRONG.

However, this is incorrect - there are many other preferred ways to prevent overfitting in Neural Networks that we will discuss later (such as L2 regularization, dropout, input noise). In practice, it is always better to use these methods to control overfitting instead of the number of neurons.
The takeaway is that you should not be using smaller networks because you are afraid of overfitting. Instead, you should use as big of a neural network as your computational budget allows, and use other regularization techniques to control overfitting.

Data Preprocessing:

- **Mean Subtraction**: is the most common form of preprocessing. It involves subtracting the mean across every individual *feature* in the data, and has the geometric interpretation of centering the cloud of data around the origin along every dimension. In numpy, this operation would be implemented as: `X -= np.mean(X, axis = 0)`. With images specifically, for convenience it can be common to subtract a single value from all pixels (e.g. `X -= np.mean(X)`), or to do so separately across the three color channels.
- **Normalization**: refers to normalizing the data dimensions so that they are of approximately the same scale. There are two common ways of achieving this normalization. One is to divide each dimension by its standard deviation, once it has been zero-centered: (`X /= np.std(X, axis = 0)`). Another form of this preprocessing normalizes each dimension so that the min and max along the dimension is -1 and 1 respectively. It only makes sense to apply this

preprocessing if you have a reason to believe that different input features have different scales (or units), but they should be of approximately equal importance to the learning algorithm. In case of images, the relative scales of pixels are already approximately equal (and in range from 0 to 255), so it is not strictly necessary to perform this additional preprocessing step.



- **PCA and Whitening** is another form of preprocessing. In this process, the data is first centered as described above. Then, we can compute the covariance matrix that tells us about the correlation structure in the data:

```
# Assume input data matrix X of size [N x D]
X -= np.mean(X, axis = 0) # zero-center the data (important)
cov = np.dot(X.T, X) / X.shape[0] # get the data covariance matrix
```

We then compute the SVD vectorization of the data covariance matrix:

```
U,S,V = np.linalg.svd(cov)
```

where the columns of `U` are the eigenvectors and `S` is a 1-D array of the singular values. To decorrelate the data, we project the original (but zero-centered) data into the eigenbasis:

```
Xrot = np.dot(X, U) # decorrelate the data
```

Notice that the columns of `U` are a set of orthonormal vectors (norm of 1, and orthogonal to each other), so they can be regarded as basis vectors. The projection therefore corresponds to a rotation of the data in `X` so that the new axes are the eigenvectors. If we were to compute the covariance matrix of `Xrot`, we would see that it is now diagonal. A nice property of `np.linalg.svd` is that in its returned value `U`, the eigenvector columns are sorted by their eigenvalues. We can use this to reduce the dimensionality of the data by only using the top few eigenvectors, and discarding the dimensions along which the data has no

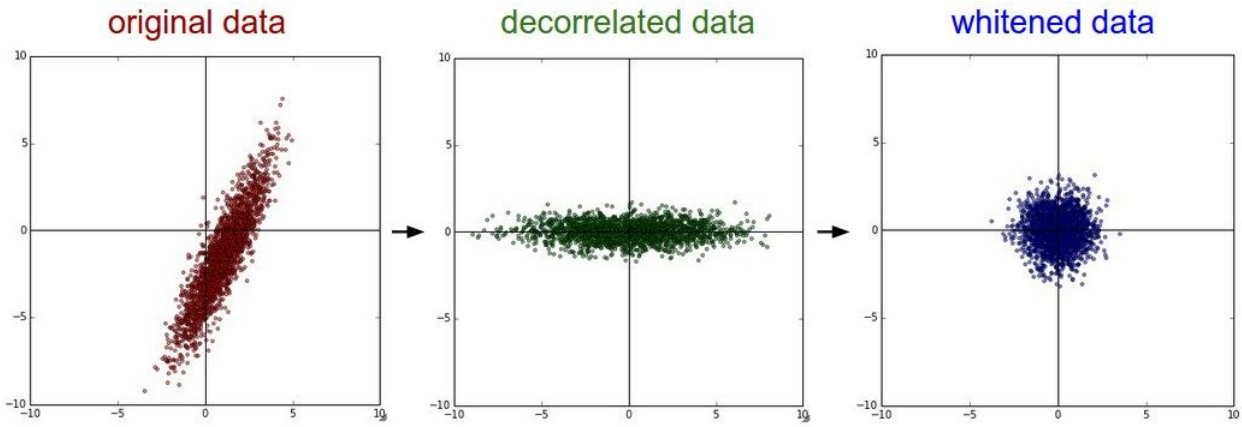
variance. This is also sometimes referred to as Principal Component Analysis (PCA) dimensionality reduction:

```
Xrot_reduced = np.dot(X, U[:,:100]) # Xrot_reduced becomes [N x 100]
```

The whitening operation takes the data in the eigenbasis and divides every dimension by the eigenvalue to normalize the scale. The geometric interpretation of this transformation is that if the input data is a multivariable gaussian, then the whitened data will be a gaussian with zero mean and identity covariance matrix. This step would take the form:

```
# whiten the data:
# divide by the eigenvalues (which are square roots of the singular value
Xwhite = Xrot / np.sqrt(S + 1e-5)
```

***Warning**: Exaggerating noise.* Note that we're adding 1e-5 (or a small constant) to prevent division by zero. One weakness of this transformation is that it can greatly exaggerate the noise in the data, since it stretches all dimensions (including the irrelevant dimensions of tiny variance that are mostly noise) to be of equal size in the input. This can in practice be mitigated by stronger smoothing (i.e. increasing 1e-5 to be a larger number).



PCA / Whitening. **Left**: Original toy, 2-dimensional input data. **Middle**: After performing PCA. The data is centered at zero and then rotated into the eigenbasis of the data covariance matrix. This decorrelates the data (the covariance matrix becomes diagonal). **Right**: Each dimension is additionally scaled by the eigenvalues, transforming the data covariance matrix into the identity matrix. Geometrically, this corresponds to stretching and squeezing the data into an isotropic gaussian blob.

- **In practice.** We mention PCA/Whitening in these notes for completeness, but these transformations are not used with Convolutional Networks. However, it is

very important to zero-center the data, and it is common to see normalization of every pixel as well.

- **Common pitfall**. An important point to make about the preprocessing is that any preprocessing statistics (e.g. the data mean) must only be computed on the training data, and then applied to the validation / test data. E.g. computing the mean and subtracting it from every image across the entire dataset and then splitting the data into train/val/test splits would be a mistake. Instead, the mean must be computed only over the training data and then subtracted equally from all splits (train/val/test).