Lecture 5 - http://cs231n.github.io/convolutional-networks/

- Difference between CNN and NN is, CNN assumes that input is image format
  - The layers of CNN have neurons arranged in 3 dimensions - width, height, depth
- Images of size 300*300*3 means 300 wide, 300 high, 3 color channels (R, G, B)
  - This also means there are 300*300*3 weights
  - For the input layer, width and height should be image size (this also means each image should be the same size), depth will be 3 which means (Red, Green, Blue channels)
- The nice feature of zero padding is that it will allow us to control the spatial size of the output volumes
- We can compute the spatial size of the output volume as a function of the **input volume size** (***W***), the receptive field size of the Conv Layer neurons (***F***), the **stride** with which they are applied (***S***), and the **amount of zero padding** used (***P***) on the border. You can convince yourself that the correct formula for calculating how many neurons "fit" is given by $(W−F+2P)/S+1$. For example for a 7x7 input and a 3x3 filter with stride 1 and pad 0 we would get a 5x5 output. With stride 2 we would get a 3x3 output.
  - In general, **set zero padding to be P = (F-1)/2 when the stride is S = 1** ensures that the input volume and output volume will have the same size spatially
  - Sizing the ConvNets appropriately so that all the dimensions "work out" can be a real headache (I agree!!), which the use of zero-padding and some design guidelines will significantly alleviate.

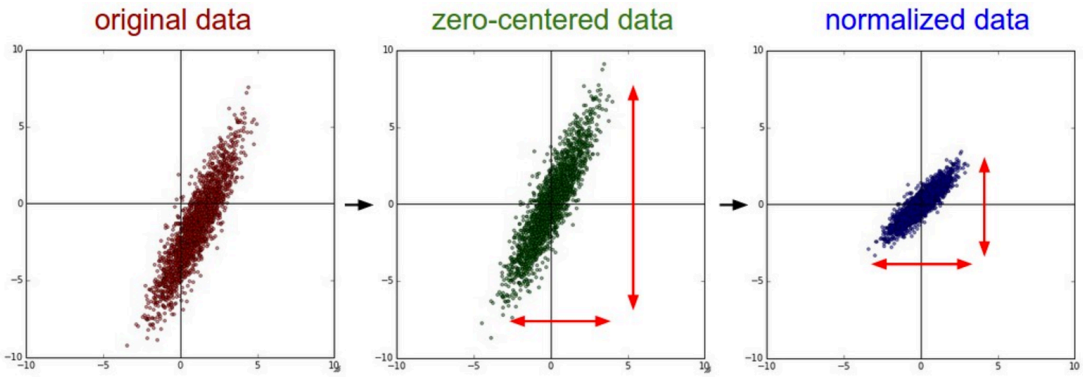
Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.

- Getting rid of pooling - Discarding pooling layers has also been found to be important in training good generative models, such as variational autoencoders (VAEs) or generative adversarial networks (GANs). It seems likely that future architectures will feature very few to no pooling layers.

Details about Activation Function: http://cs231n.github.io/neural-networks-1/
Data Preprocessing for CNN: http://cs231n.github.io/neural-networks-2/

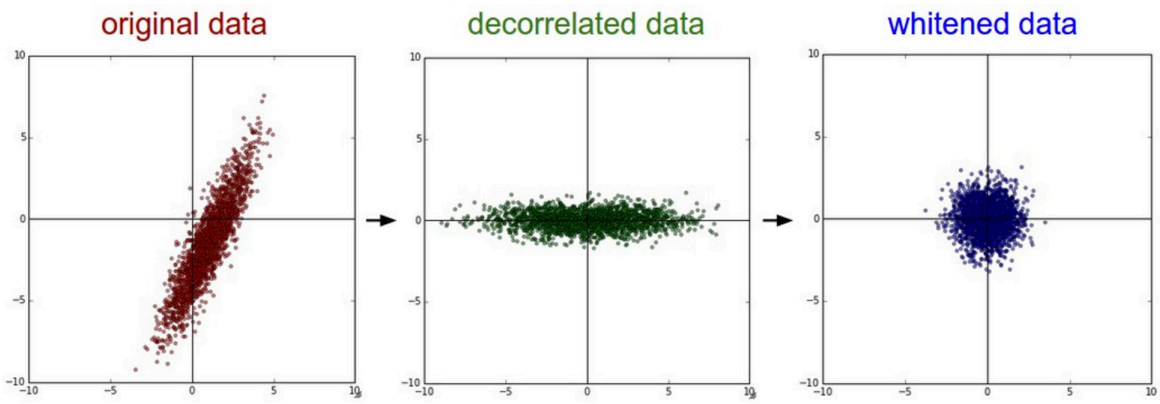
- Mean subtraction: subtract a value from all pixels
- Normalization
  - One approach is to divide each dimension by its standard deviation, once it has been zero-centered
  - Another method is to normalize each dimension to [-1, 1] range. This method can be used when different input has different scale. But **for 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.



Common data preprocessing pipeline. **Left**: Original toy, 2-dimensional input data. **Middle**: The data is zero-centered by subtracting the mean in each dimension. The data cloud is now centered around the origin. **Right**: Each dimension is additionally scaled by its standard deviation. The red lines indicate the extent of the data - they are of unequal length in the middle, but of equal length on the right.

  - PCA and Whitening method
    - Use PCA to find x dimensions that are most variant, convert the original dataset of [N*D] to one of size [N*x]
    - Whitening: if the input data is a multivariable gaussian, then the whitened data will be a gaussian with zero mean and identity covariance matrix
      - *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.

- Details about **Weight Initialization, Regularization and Loss Functions**
  - dropout is a regularization method, like L1, L2

Summary

In summary:

- The recommended preprocessing is to center the data to have mean of zero, and normalize its scale to [-1, 1] along each feature
- Initialize the weights by drawing them from a gaussian distribution with standard deviation of $\sqrt{2/n}$, where n is the number of inputs to the neuron. E.g. in numpy: `w = np.random.randn(n) * sqrt(2.0/n)`.
- Use L2 regularization and dropout (the inverted version)
- Use batch normalization
- We discussed different tasks you might want to perform in practice, and the most common loss functions for each task

Details about Optimization Methods: http://cs231n.github.io/neural-networks-3/

Summary

To train a Neural Network:

- Gradient check your implementation with a small batch of data and be aware of the pitfalls.
- As a sanity check, make sure your initial loss is reasonable, and that you can achieve 100% training accuracy on a very small portion of the data
- During training, monitor the loss, the training/validation accuracy, and if you're feeling fancier, the magnitude of updates in relation to parameter values (it should be ~1e-3), and when dealing with ConvNets, the first-layer weights.
- The two recommended updates to use are either SGD+Nesterov Momentum or Adam.
- Decay your learning rate over the period of the training. For example, halve the learning rate after a fixed number of epochs, or whenever the validation accuracy tops off.
- Search for good hyperparameters with random search (not grid search). Stage your search from coarse (wide hyperparameter ranges, training only for 1-5 epochs), to fine (narrower rangers, training for many more epochs)
- Form model ensembles for extra performance