most of the information is still preserved. **Right:** Visualization of the "white" representation, where the variance along every one of the 144 dimensions is squashed to equal length. Here, the whitened 144 numbers are rotated back to image pixel basis by multiplying by U.transpose()[:144,:]. The lower frequencies (which accounted for most variance) are now negligible, while the higher frequencies (which account for relatively little variance originally) become exaggerated.

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).

Weight Initialization

We have seen how to construct a Neural Network architecture, and how to preprocess the data. Before we can begin to train the network we have to initialize its parameters.

Pitfall: all zero initialization. Lets start with what we should not do. Note that we do not know what the final value of every weight should be in the trained network, but with proper data normalization it is reasonable to assume that approximately half of the weights will be positive and half of them will be negative. A reasonable-sounding idea then might be to set all the initial weights to zero, which we expect to be the “best guess” in expectation. This turns out to be a mistake, because if every neuron in the network computes the same output, then they will also all compute the same gradients during backpropagation and undergo the exact same parameter updates. In other words, there is no source of asymmetry between neurons if their weights are initialized to be the same.

Small random numbers. Therefore, we still want the weights to be very close to zero, but as we have argued above, not identically zero. As a solution, it is common to initialize the weights of the neurons to small numbers and refer to doing so as *symmetry breaking*. The idea is that the neurons are all random and unique in the beginning, so they will compute distinct updates and integrate themselves as diverse parts of the full network. The implementation for one weight matrix might look like `W = 0.01* np.random.randn(D,H)`, where `randn` samples from a zero mean, unit standard deviation gaussian. With this formulation, every neuron’s weight vector is initialized as a random vector sampled from a multi-dimensional gaussian, so the neurons point

in random direction in the input space. It is also possible to use small numbers drawn from a uniform distribution, but this seems to have relatively little impact on the final performance in practice.

Warning: It's not necessarily the case that smaller numbers will work strictly better. For example, a Neural Network layer that has very small weights will during backpropagation compute very small gradients on its data (since this gradient is proportional to the value of the weights). This could greatly diminish the "gradient signal" flowing backward through a network, and could become a concern for deep networks.

Calibrating the variances with 1/sqrt(n). One problem with the above suggestion is that the distribution of the outputs from a randomly initialized neuron has a variance that grows with the number of inputs. It turns out that we can normalize the variance of each neuron's output to 1 by scaling its weight vector by the square root of its *fan-in* (i.e. its number of inputs). That is, the recommended heuristic is to initialize each neuron's weight vector as: `w = np.random.randn(n) / sqrt(n)`, where `n` is the number of its inputs. This ensures that all neurons in the network initially have approximately the same output distribution and empirically improves the rate of convergence.

The sketch of the derivation is as follows: Consider the inner product $s = \sum_i^n w_i x_i$ between the weights w and input x, which gives the raw activation of a neuron before the non-linearity. We can examine the variance of s:

$$\begin{aligned}
\text{Var}(s) &= \text{Var}(\sum_i^n w_i x_i) \\
&= \sum_i^n \text{Var}(w_i x_i) \\
&= \sum_i^n [E(w_i)]^2 \text{Var}(x_i) + E[(x_i)]^2 \text{Var}(w_i) + \text{Var}(x_i)\text{Var}(w_i) \\
&= \sum_i^n \text{Var}(x_i)\text{Var}(w_i) \\
&= (n\text{Var}(w))\,\text{Var}(x)
\end{aligned}$$

where in the first 2 steps we have used properties of variance. In third step we assumed zero mean inputs and weights, so $E[x_i] = E[w_i] = 0$. Note that this is not generally the case: For

example ReLU units will have a positive mean. In the last step we assumed that all w_i, x_i are identically distributed. From this derivation we can see that if we want s to have the same variance as all of its inputs x, then during initialization we should make sure that the variance of every weight w is $1/n$. And since $\text{Var}(aX) = a^2\text{Var}(X)$ for a random variable X and a scalar a, this implies that we should draw from unit gaussian and then scale it by $a = \sqrt{1/n}$, to make its variance $1/n$. This gives the initialization `w = np.random.randn(n) / sqrt(n)`.

A similar analysis is carried out in Understanding the difficulty of training deep feedforward neural networks by Glorot et al. In this paper, the authors end up recommending an initialization of the form $\text{Var}(w) = 2/(n_{in} + n_{out})$ where n_{in}, n_{out} are the number of units in the previous layer and the next layer. This is based on a compromise and an equivalent analysis of the backpropagated gradients. A more recent paper on this topic, Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification by He et al., derives an initialization specifically for ReLU neurons, reaching the conclusion that the variance of neurons in the network should be $2.0/n$. This gives the initialization `w = np.random.randn(n) * sqrt(2.0/n)`, and is the current recommendation for use in practice in the specific case of neural networks with ReLU neurons.

Sparse initialization. Another way to address the uncalibrated variances problem is to set all weight matrices to zero, but to break symmetry every neuron is randomly connected (with weights sampled from a small gaussian as above) to a fixed number of neurons below it. A typical number of neurons to connect to may be as small as 10.

Initializing the biases. It is possible and common to initialize the biases to be zero, since the asymmetry breaking is provided by the small random numbers in the weights. For ReLU non-linearities, some people like to use small constant value such as 0.01 for all biases because this ensures that all ReLU units fire in the beginning and therefore obtain and propagate some gradient. However, it is not clear if this provides a consistent improvement (in fact some results seem to indicate that this performs worse) and it is more common to simply use 0 bias initialization.

In practice, the current recommendation is to use ReLU units and use the `w = np.random.randn(n) * sqrt(2.0/n)`, as discussed in He et al..

Batch Normalization. A recently developed technique by Ioffe and Szegedy called Batch Normalization alleviates a lot of headaches with properly initializing neural networks by explicitly forcing the activations throughout a network to take on a unit gaussian distribution at the beginning of the training. The core observation is that this is possible because normalization is a simple differentiable operation. In the implementation, applying this technique usually amounts to insert the BatchNorm layer immediately after fully connected layers (or convolutional layers, as we'll soon see), and before non-linearities. We do not expand on this technique here because it is

well described in the linked paper, but note that it has become a very common practice to use Batch Normalization in neural networks. In practice networks that use Batch Normalization are significantly more robust to bad initialization. Additionally, batch normalization can be interpreted as doing preprocessing at every layer of the network, but integrated into the network itself in a differentiable manner. Neat!

Regularization

There are several ways of controlling the capacity of Neural Networks to prevent overfitting:

L2 regularization is perhaps the most common form of regularization. It can be implemented by penalizing the squared magnitude of all parameters directly in the objective. That is, for every weight w in the network, we add the term $\frac{1}{2}\lambda w^2$ to the objective, where λ is the regularization strength. It is common to see the factor of $\frac{1}{2}$ in front because then the gradient of this term with respect to the parameter w is simply λw instead of $2\lambda w$. The L2 regularization has the intuitive interpretation of heavily penalizing peaky weight vectors and preferring diffuse weight vectors. As we discussed in the Linear Classification section, due to multiplicative interactions between weights and inputs this has the appealing property of encouraging the network to use all of its inputs a little rather than some of its inputs a lot. Lastly, notice that during gradient descent parameter update, using the L2 regularization ultimately means that every weight is decayed linearly: `W += -lambda * W` towards zero.

L1 regularization is another relatively common form of regularization, where for each weight w we add the term $\lambda \mid w \mid$ to the objective. It is possible to combine the L1 regularization with the L2 regularization: $\lambda_1 \mid w \mid + \lambda_2 w^2$ (this is called Elastic net regularization). The L1 regularization has the intriguing property that it leads the weight vectors to become sparse during optimization (i.e. very close to exactly zero). In other words, neurons with L1 regularization end up using only a sparse subset of their most important inputs and become nearly invariant to the "noisy" inputs. In comparison, final weight vectors from L2 regularization are usually diffuse, small numbers. In practice, if you are not concerned with explicit feature selection, L2 regularization can be expected to give superior performance over L1.

Max norm constraints. Another form of regularization is to enforce an absolute upper bound on the magnitude of the weight vector for every neuron and use projected gradient descent to enforce the constraint. In practice, this corresponds to performing the parameter update as normal, and then enforcing the constraint by clamping the weight vector $\vec{w}$ of every neuron to satisfy $\|\vec{w}\|_2 < c$. Typical values of c are on orders of 3 or 4. Some people report improvements when using this form of regularization. One of its appealing properties is that network cannot "explode" even when the learning rates are set too high because the updates are always bounded.