Batch Normalization and Weight Initialization

Batch Normalization

- Batch normalisation is a technique for improving the performance and stability of neural networks, and also makes more sophisticated architectures work in practice (like deep convolutional generative adversarial networks -DCGANs).

- Helps optimization (though it is not an optimization algorithm!)

- The idea is to normalise the inputs of each layer in such a way that they have a mean output activation of zero and standard deviation of one. This is analogous to how the inputs to networks are standardised.

- Batch Normalization is used in almost all CNN architectures.

Batch Normalization
Internal Covariate Shift

– Assume we have a flowers dataset and we want to build a binary classifier for roses: the output is 1 if the image is that of a rose and the output is 0 otherwise.

– Consider a subset of the training data that primarily has red rose buds as rose examples and wildflowers as non-rose examples.

Source: https://www.learnopencv.com/batch-normalization-in-deep-networks/

Batch Normalization
Internal Covariate Shift

– Consider another subset, that has fully blown roses of different colors as rose examples and other non-rose flowers in the picture as non-rose examples.

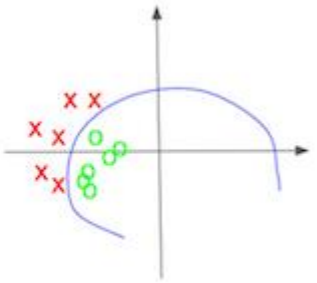
Source: https://www.learnopencv.com/batch-normalization-in-deep-networks/

Batch Normalization

Internal Covariate Shift

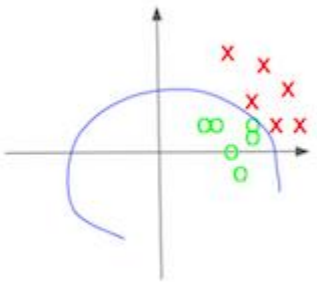
Rose
(y=1)

Not Rose
(y=0)



Rose
(y=1)

Not Rose
(y=0)



– The two subsets have very different distributions. The last column shows the distribution of the two classes in the feature space using red and green dots. The blue line show the decision boundary between the two classes.

Source: https://www.learnopencv.com/batch-normalization-in-deep-networks/

Batch Normalization
Internal Covariate Shift

- Intuitively, it makes sense that every mini-batch used in the training process should have the same distribution. In other words, a mini-batch should have images randomly selected from both subsets in each mini-batch.

- This difference in distribution is called the **covariate shift**. When the mini-batches have images uniformly sampled from the entire distribution, there is negligible covariate shift.

- However, when the mini-batches are sampled from only one of the two subsets shown in the previous slide, there is a significant covariate shift. This makes the training of the rose vs non-rose classifier very slow.

Source: https://www.learnopencv.com/batch-normalization-in-deep-networks/

Batch Normalization

Internal Covariate Shift

- An easy way to solve this problem for the input layer is to randomize the data before creating mini-batches.

- But, how do we solve this for the hidden layers? Just as it made intuitive sense to have a uniform distribution for the input layer, it is advantageous to have the same input distribution for each hidden unit over time while training.

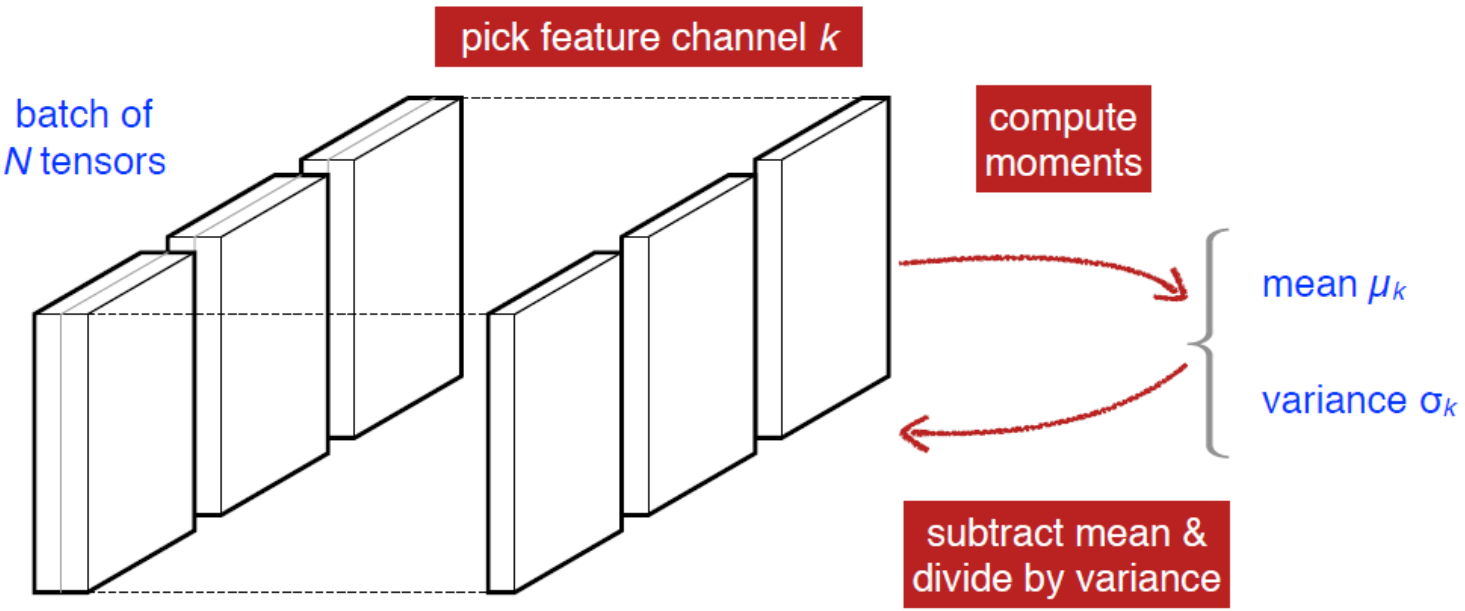
- But in a neural network, each hidden unit's input distribution changes every time there is a parameter update in the previous layer: **internal covariate shift**.

- This makes training slow and requires a very small learning rate and a good parameter initialization. This problem is solved by normalizing the layer's inputs over a mini-batch: **Batch Normalization**.

Source: https://www.learnopencv.com/batch-normalization-in-deep-networks/

Batch normalization

Condition features



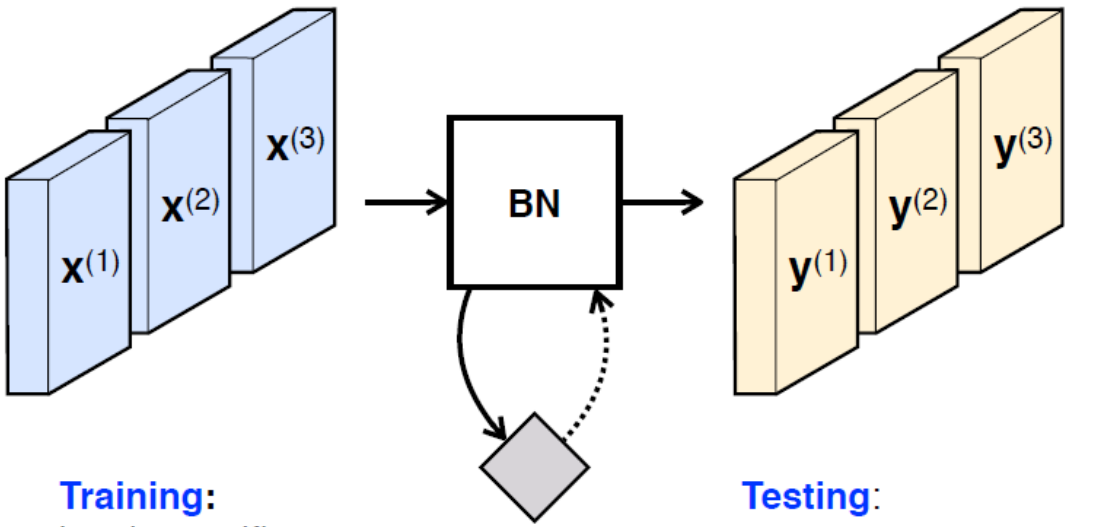
Standardize the response of each feature channel *within the batch*

- Average over spatial locations
- Also, average over multiple images in the batch (e.g. 16-256)

S. Ioffe and C. Szegedy. *Batch normalization: Accelerating deep network training by reducing internal covariate shift.* CoRR, 2015

Batch normalization

Training vs testing modes

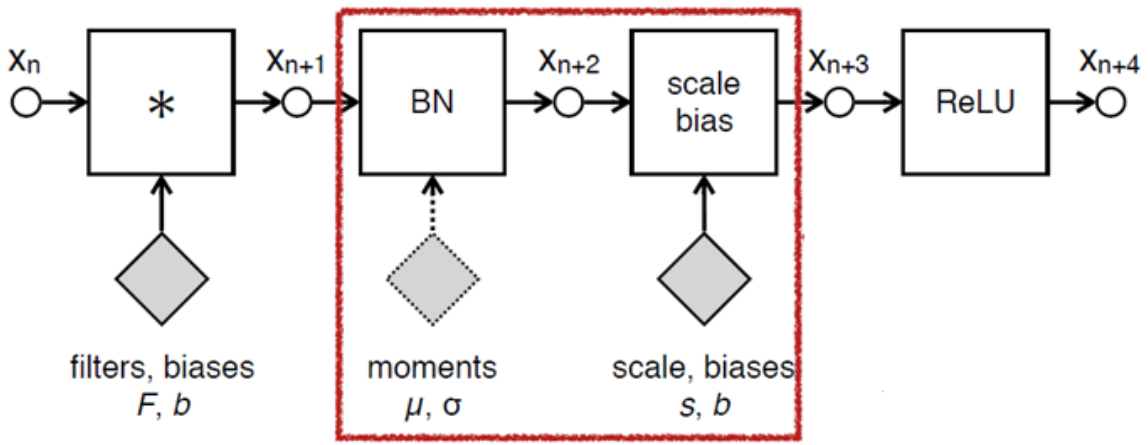


Moments (mean & variance)

- **Training**: compute anew for each batch
- **Testing**: fixed to their average values

Batch normalization

Utilization



Batch normalization is used after filtering, before ReLU

It is always followed by channel-specific scaling factor *s* and bias *b*

Noisy bias/variance estimation **replaces dropout regularization**

Note: Just normalizing a layer would limit the layer in terms of what it can represent. For example, if we normalize the inputs to a sigmoid function, then the output would be bound to the linear region only. So we also need to scale and apply bias to the normalized values otherwise.

Batch Normalization

Advantages Of Batch Normalization - I

- Reduces internal covariate shift.
- Reduces the dependence of gradients on the scale of the parameters or their initial values.
- Regularizes the model and reduces the need for dropout, photometric distortions, local response normalization and other regularization techniques.

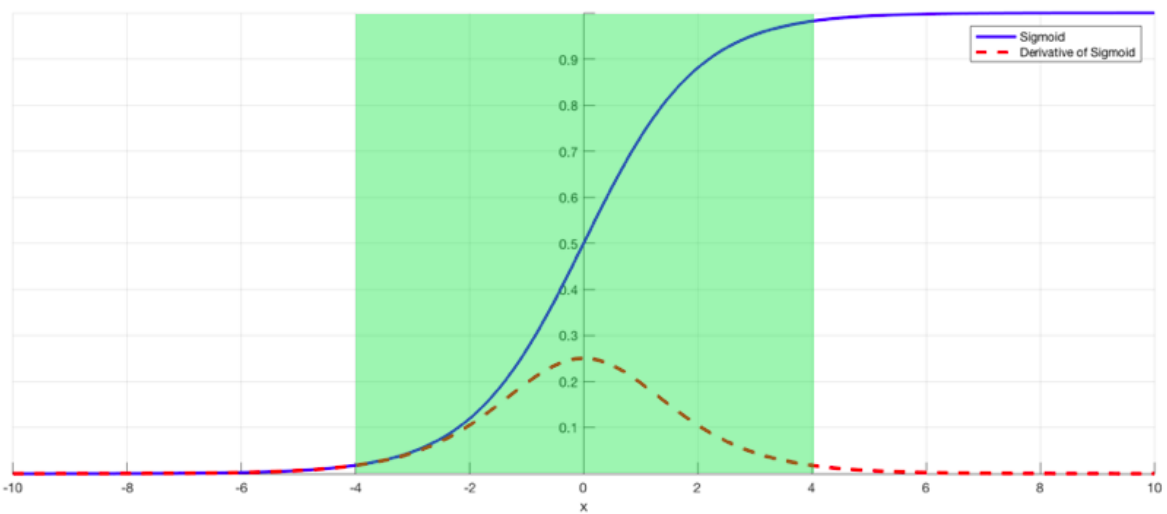
Source: https://gist.github.com/shagunsodhani/4441216a298df0fe6ab0

Batch Normalization

Advantages Of Batch Normalization - II

– Allows use of saturating nonlinearities and higher learning rates.

– Normalizes the input so |x| doesn't reach the outer edges of the sigmoid function and most of it falls in the green region, where the derivative isn't too small.



Source: https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484

Network Initialization

- If all weights are initialized to the same value (e.g. zero or one). Each hidden unit will get exactly the same signal.
- If all weights are initialized to 1, each unit gets signal equal to sum of inputs.
- If all weights are zeros, every hidden unit will get zero signal.
- No matter what was the input - if all weights are the same, all units in hidden layer will be the same too.
- So you should initialize weights randomly (with different values)

Xavier Initialization

- Now we know that we need to initialize our weights randomly, but:
  - If the weights in a network start **too small**, then the signal shrinks as it passes through each layer until it's too tiny to be useful.
  - If the weights in a network start **too large**, then the signal grows as it passes through each layer until it's too massive to be useful.

- Xavier initialization makes sure the weights are 'just right', keeping the signal in a reasonable range of values through many layers.

Glorot, X. and Bengio, Y., 2010, "Understanding the Difficulty of Training Deep Feedforward Neural Networks", In Proc. International Conference on Artificial Intelligence and Statistics (pp. 249-256).

Xavier Initialization

- Generally the input is normalized to have a zero mean and unit variance.

- One good way is to assign the weights from a Gaussian distribution to have zero mean and some finite variance.

- Let's consider a linear neuron:

$$y = w_1x_1 + w_2x_2 + \ldots + w_Nx_N + b$$

- With each passing layer, we want the variance to remain the same. This helps us keep the signal from exploding to a high value or vanishing to zero.

- In other words, the weights needs to be initialized in such a way that the variance remains the same for *x* and *y* . This initialization process is known as Xavier initialization.

Xavier Initialization

$$\text{Var}(y) = \text{Var}(w_1x_1 + w_2x_2 + ... + w_Nx_N + b)$$

– input data has unit variance, zero mean, and they are independent

$$\text{Var}(y) = N\ \text{Var}(w_i), \text{ for all } i \in 1,\cdots,N$$

Therefore, we initialize the weights from an IID normal so that:

$$w_i \sim N(0,1/N)$$

.

Xavier Initialization - Example

- Consider a fully connected multilayer perceptron with only 1 hidden unit and a softmax output layer.

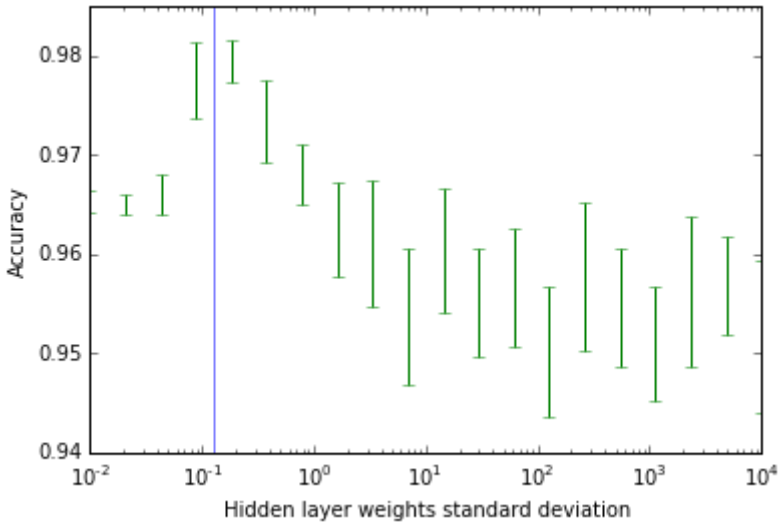
- For illustration purposes, we will initialize the input layer weights to random values and keep them fixed and only update the output layer weights.

- The dataset is the digits dataset provided by Scikit-learn: 1797 examples of 8x8 images (64 inputs to each neuron) showing the different digits between 0 and 9.

- The network has a hidden layer with 500 neurons.

- We consider several different variance initializations for the hidden layer weights and for each variance initialization we run the simulation 10 times and show the accuracy on a test set 33% of the data based on different standard deviations (square root of variance) used for initializing the weights:

Xavier Initialization



- The vertical blue line corresponds to picking an Xavier initialization, so that the variance of the hidden weights are 1/64.
- Each error bar is the result of running the simulation 10 times using the same variance for this hidden weight initialization.
- In this experiment we are keeping the initialization of the hidden layer weights fixed at random but only vary their variance.
- The Xavier initialization starts us off in a better region and we in effect have to do less work during optimization (fewer gradient descent iterations).

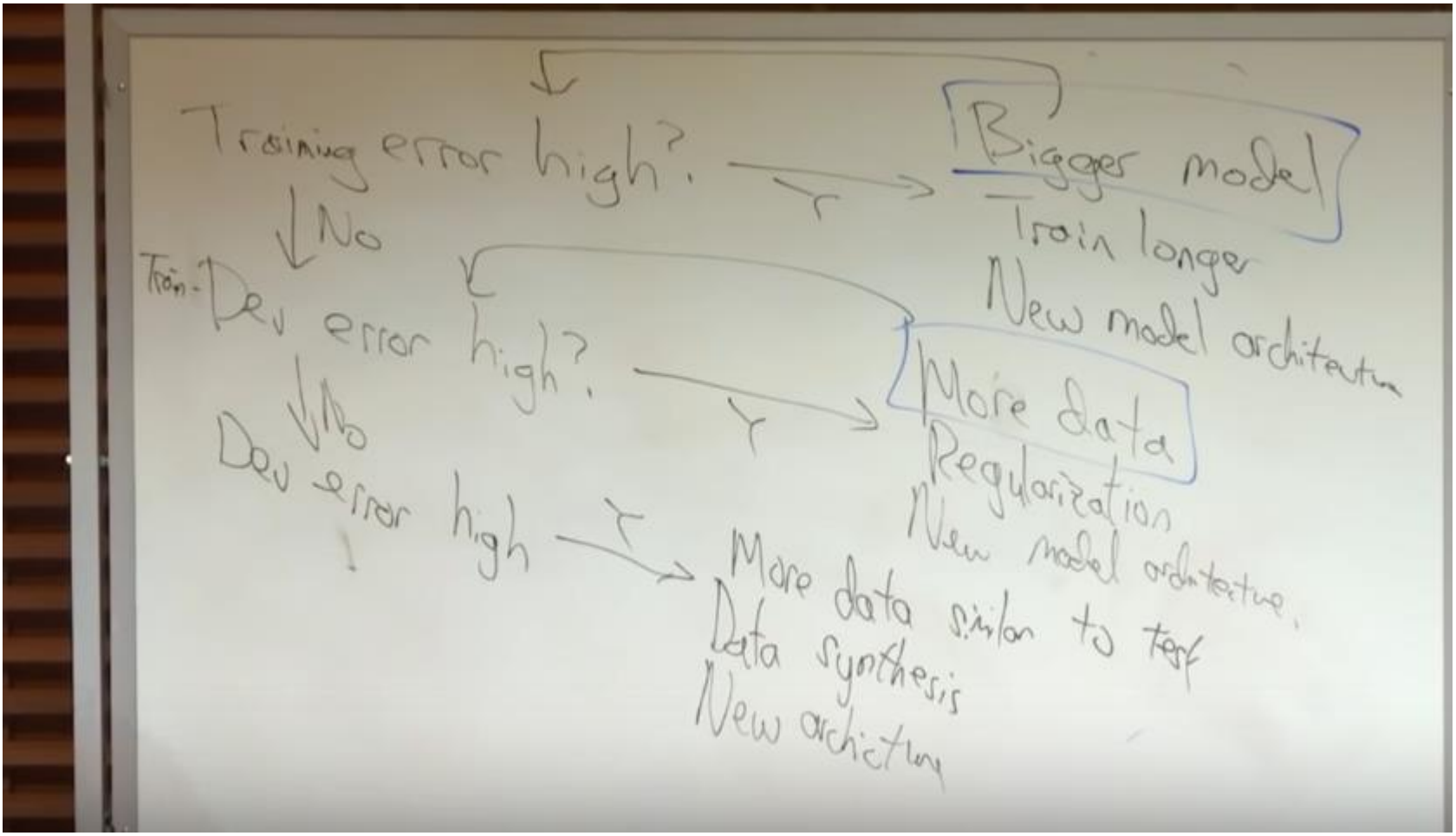
Kaiming Initialization

- Xavier initialization assumes that the activation function is linear
- Kaiming initialization takes activation function into account.

- For ReLU activation:

$$W \sim \mathcal{N}\left(0, \frac{2}{n^l}\right)$$

He, K., Zhang, X., Ren, S. and Sun, J., 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision* (pp. 1026-1034).



– Source: Andrew Ng

CNNs for other types of data

– Convolution networks are applicable to many types of data in array form

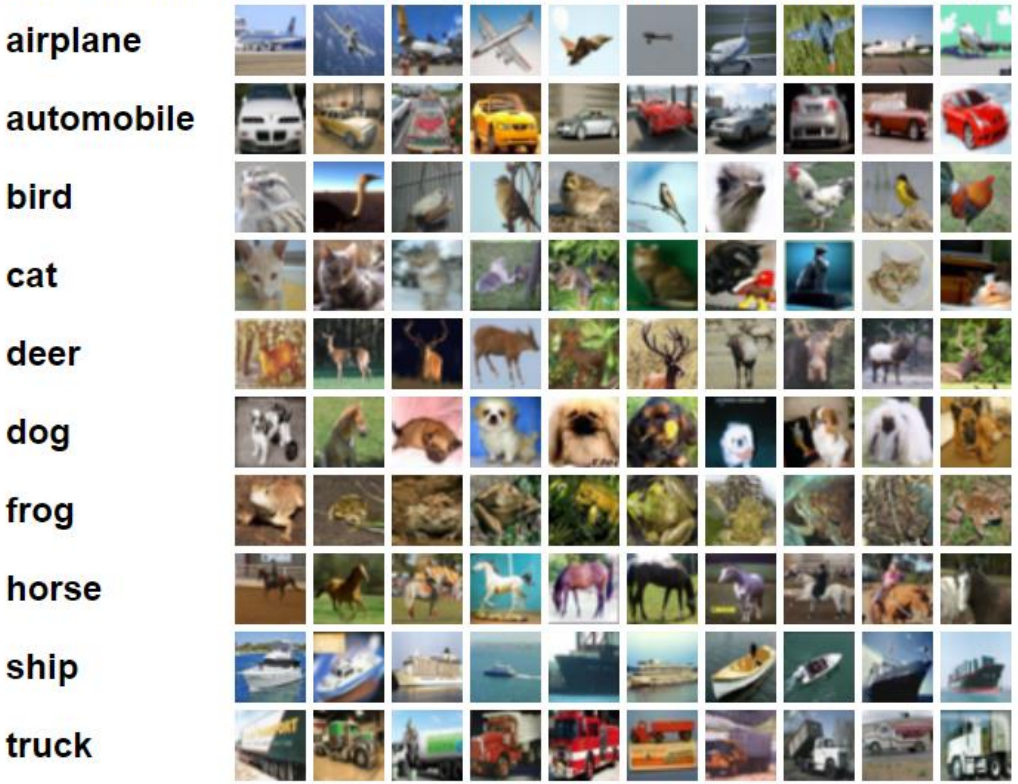
– Most natural data have local correlation, in images: strong chance that two neighboring pixels have the same color - leading to the idea of local processing

CNN Activations as Deep Features

CIFAR-10 Dataset

– The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes,
– There are 50000 training images and 10000 test images.
– The dataset is divided into five training batches and one test batch, each with 10000 images.

Here are the classes in the dataset, as well as 10 random images from each:



CIFAR-10 Dataset

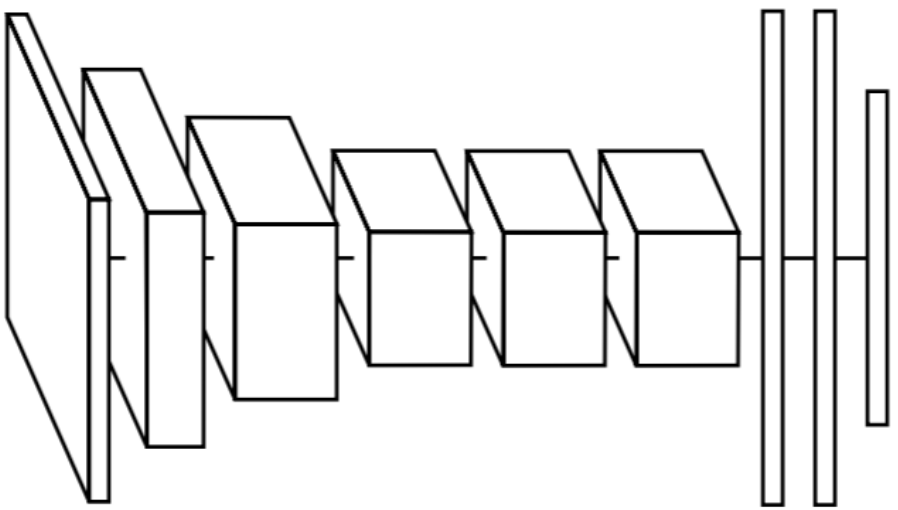
– ConvNetJS demo on CIFAR-10: https://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html

CNN Features

- Do features extracted from the CNN generalize other tasks and datasets?
  - Donahue et al. (2013), Chatfield et al. (2014), Razavian et al. (2014), Yosinski et al. (2014), etc.
- CNN activations as deep features
- Finetuning CNNs

CNN activations as deep features

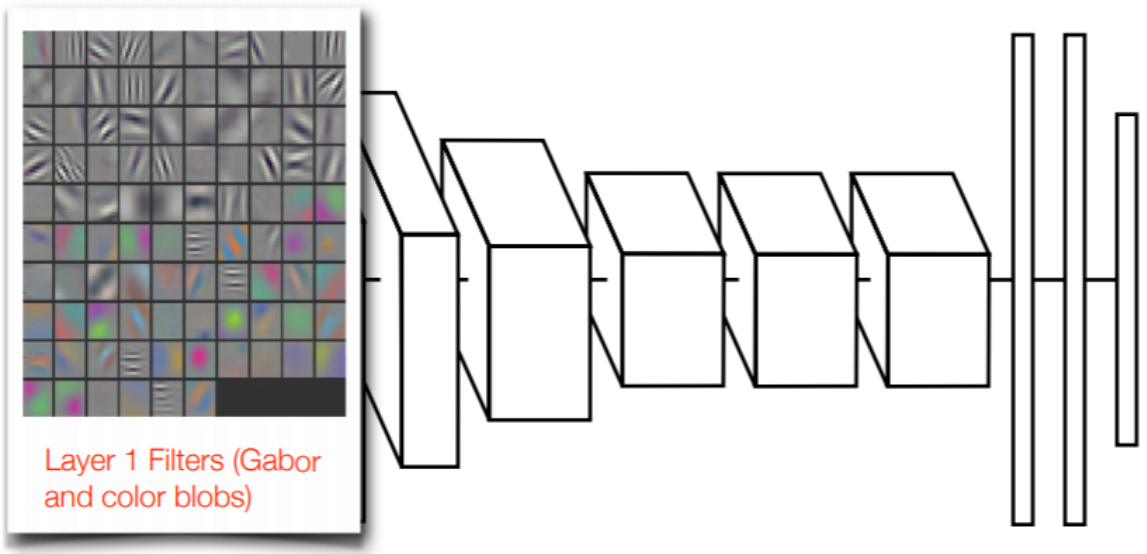
- CNNs discover effective representations. Why not to use them?



Slide credit: Jason Yosinsk

CNN activations as deep features

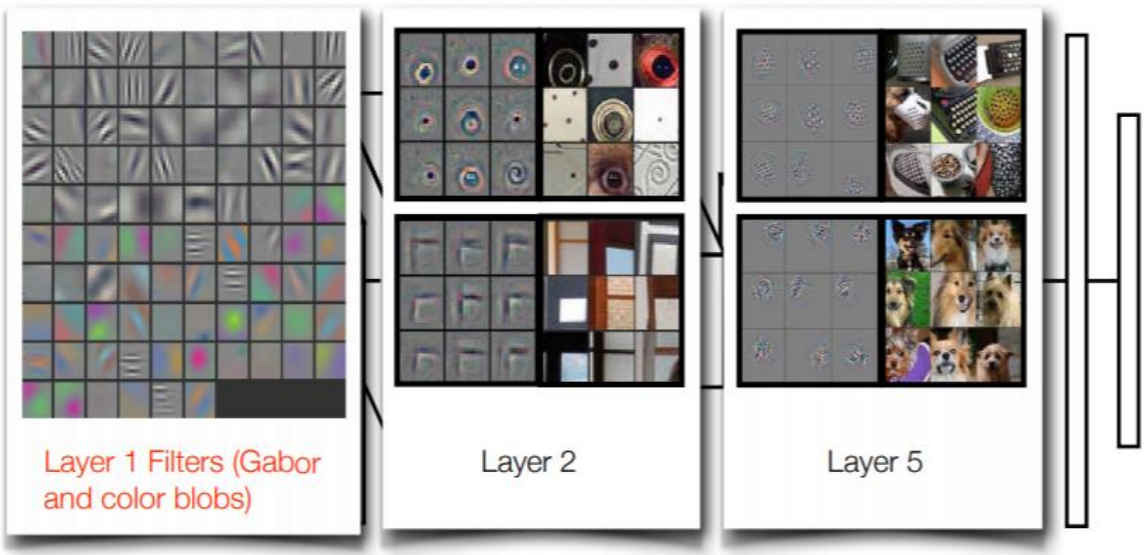
- CNNs discover effective representations. Why not to use them?



Slide credit: Jason Yosinski

CNN activations as deep features

- CNNs discover effective representations. Why not to use them?

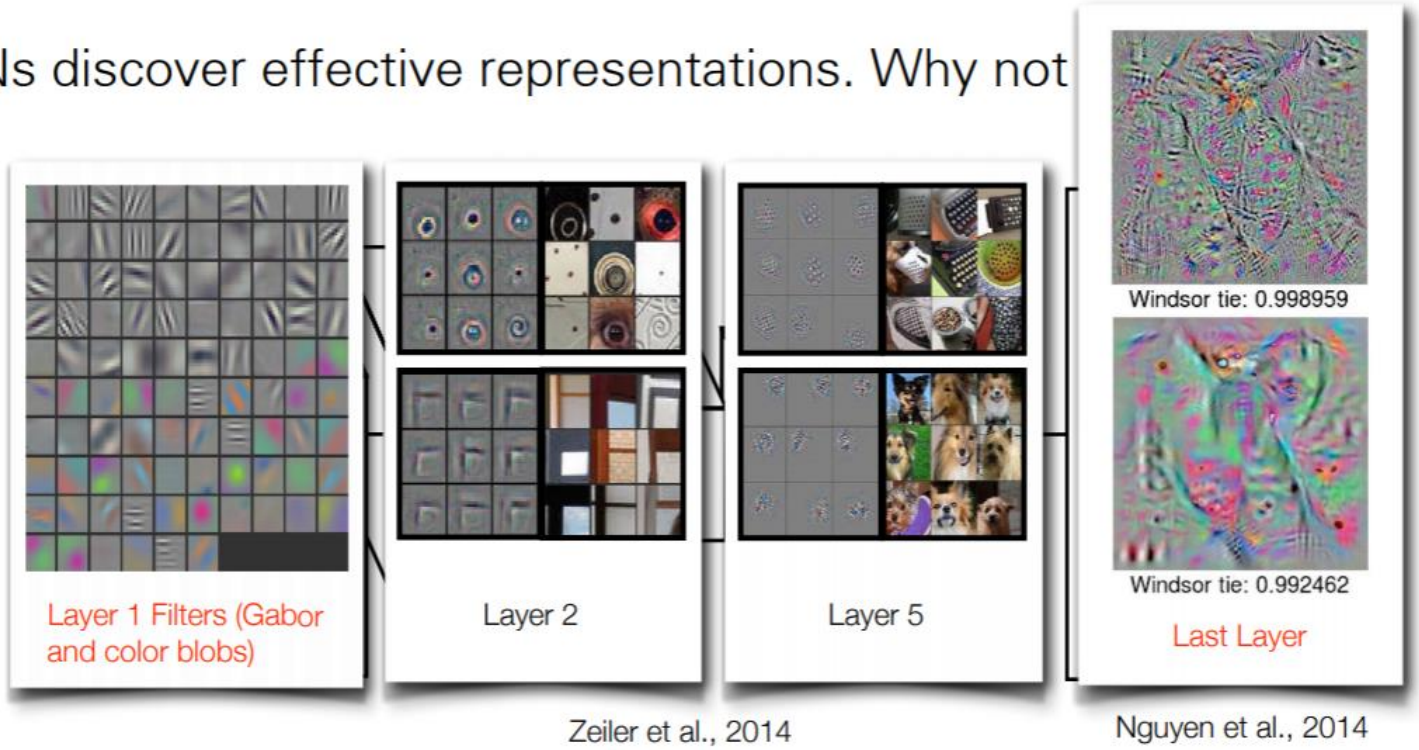


Zeiler et al., 2014

Slide credit: Jason Yosinski

CNN activations as deep features

- CNNs discover effective representations. Why not



Slide credit: Jason Yosinski

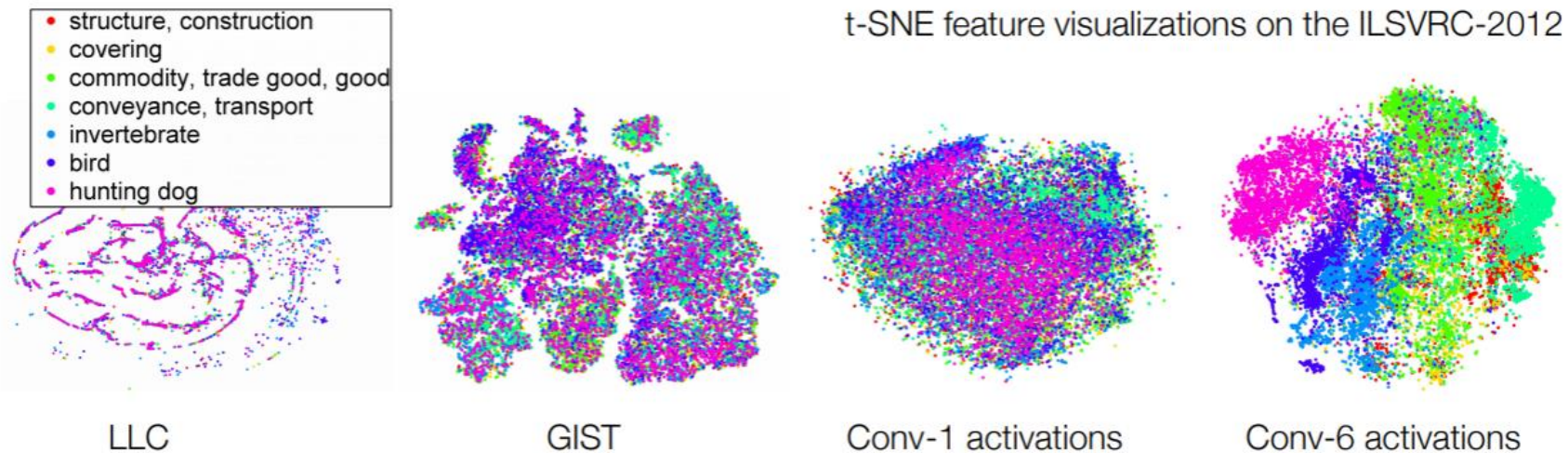
t-Distributed Stochastic Neighbor Embedding (t-SNE)

- t-SNE is a very popular dimensionality reduction technique that is usually used to map high dimensional data to 2 or 3 dimensions in order to visualize it.

- It does so by computing affinities between points, and trying to maintain these affinities in the new, low-dimensional space.

CNNs as deep features

- CNNs discover effective representations. Why not to use them?

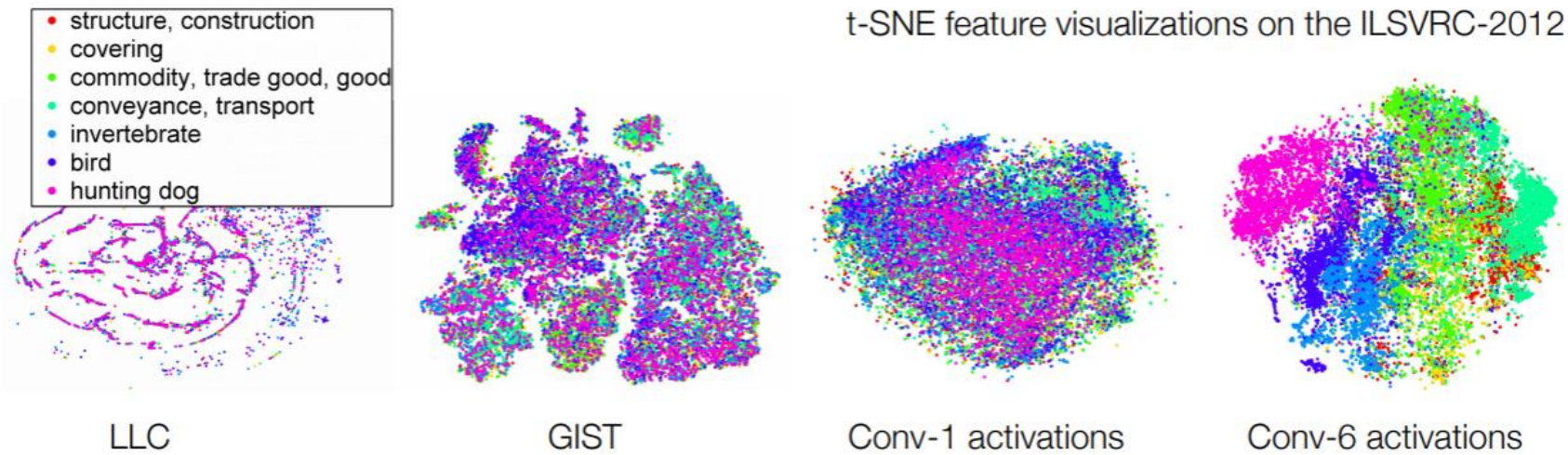


Donahue et al. **DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition**, 2014

Locality-constrained linear coding (LLC) algorithm follows the standard **descriptor coding + spatial pooling** strategy to extract features from the image

CNNs as deep features

- CNNs discover effective representations. Why not to use them?



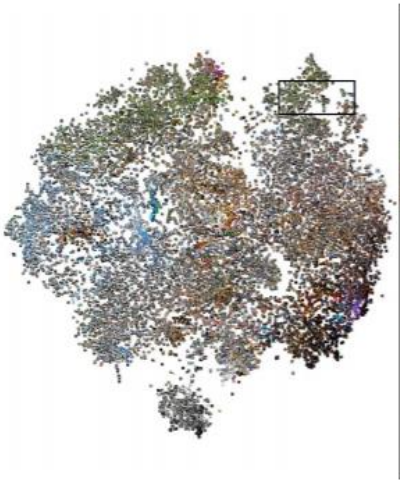
Donahue et al. **DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition**, 2014

GIST descriptor is computed by

- Convolve the image with 32 Gabor filters at 4 scales, 8 orientations, producing 32 feature maps of the same size of the input image.
- Divide each feature map into 16 regions (by a 4x4 grid), and then average the feature values within each region.
- Concatenate the 16 averaged values of all 32 feature maps, resulting in a 16x32=512 GIST descriptor.
- Intuitively, GIST summarizes the gradient information (scales and orientations) for different parts of an image, which provides a rough description (the gist) of the scene.

Transfer Learning with CNNs

- A CNN trained on a (large enough) dataset generalizes to other visual tasks



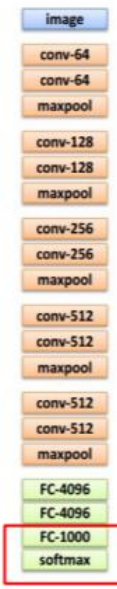
A. Joulin, L.J.P. van der Maaten, A. Jabri, and N. Vasilache **Learning visual features from Large Weakly supervised Data.** ECCV 2016

Slide credit: Joan Bruna

Transfer Learning with CNNs

- Keep layers 1-7 of our ImageNet-trained model fixed
- Train a new softmax classifier on top using the training images of the new dataset.

1. Train on Imagenet



2. Small dataset: feature extractor

Freeze these

Train this

3. Medium dataset: finetuning

more data = retrain more of the network (or all of it)

Freeze these

tip: use only ~1/10th of the original learning rate in finetuning top layer, and ~1/100th on intermediate layers

Train this