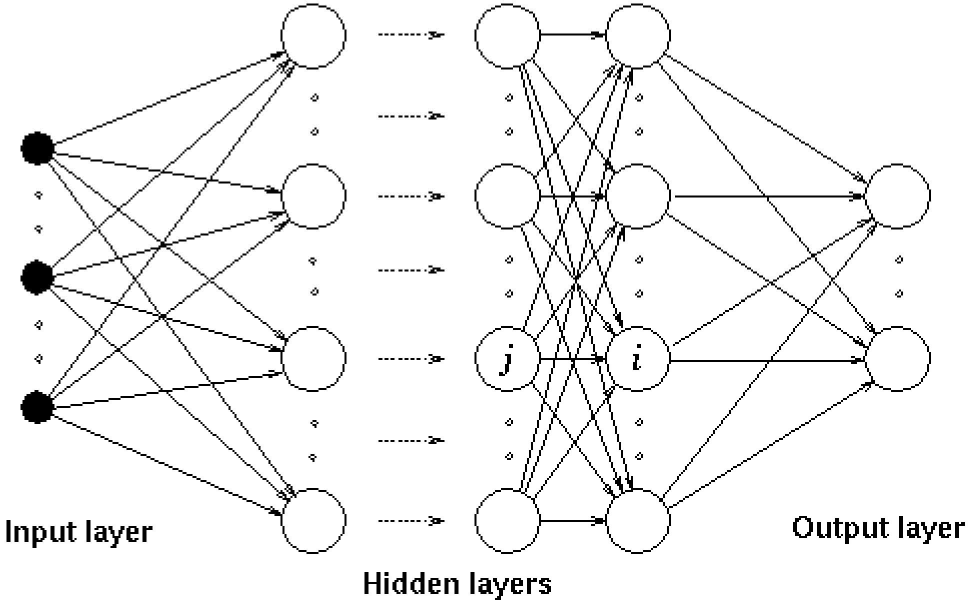
Lecture 12

Convolutional Neural Networks

Zoran B. Djordjević

Structure of Neural Networks

- So far we have shown a few small examples of neural networks and created an impression that we have as many input nodes as we have features in analyzed data objects and as many output nodes as number of classes (categories) we are trying to identify.



- We also said that we could have many hidden layers. The nature of those hidden layers was left unspecified. There are apparently many types of hidden layers.
- We will introduce several type of those on the example of Convolution Neural Network (CNN).

Structure of CNN

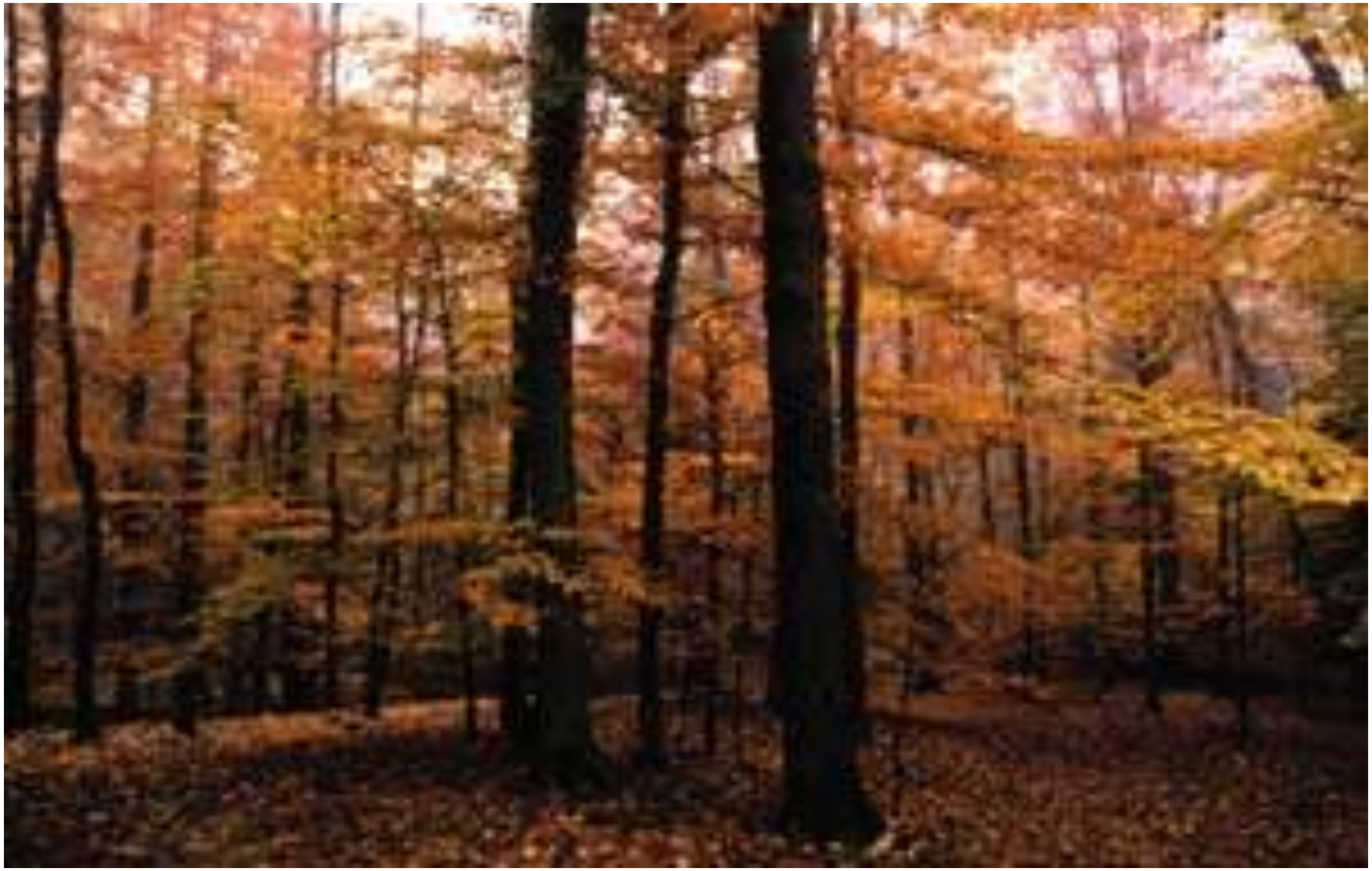
- “Ordinary” Neural Networks receive an input (a single vector), and transform it through a series of *hidden layers*. Each hidden layer is made up of a set of neurons, where each neuron could be fully connected to all neurons in the previous layer, and where neurons in a single layer function completely independently and do not share any connections. The last fully-connected layer is called the “output layer” and in classification settings it represents the class scores.
- Convolutional Neural Networks are very similar to ordinary Neural Networks: they are made up of neurons that have learnable weights and biases.
- Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other.
- CNNs still have a loss function (e.g. SVM/Softmax) on the last (fully-connected) layer and all the tips/tricks we developed for training regular Neural Networks still apply.
- So what does change? Conv Net architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network.

Fully Connected NN

- *Regular Neural Nets don’t scale well to full images*.
- CIFAR-10 and CIFAR-100 are labeled subsets of the 80 million tiny images dataset. They were collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton.
- CIFAR-10, images are only of size 32x32x3 (32 wide, 32 high, 3 color channels), so a single fully-connected neuron in a first hidden layer of a regular Neural Network would have 32*32*3 = 3072 weights. This amount still seems manageable, but clearly this fully-connected structure does not scale to larger images.
- For example, an image of more respectable size, e.g. 200x200x3, would lead to neurons that have 200*200*3 = 120,000 weights. Moreover, we would almost certainly want to have several such neurons, so the parameters would add up quickly!
- Clearly, this full connectivity is wasteful and the huge number of parameters would quickly lead to overfitting.
- Images actually do not contain that much information. Take a look at an image of forest on the next slide.

A view of a forest

- This photo has perhaps 300x200X3 = 60,000 pixels x3 RGB colors. That does not mean that it has 180,000 relevant features.
- We could extract tree trunks, branches, and leaves and would come up with a modest number of components constituting this image.

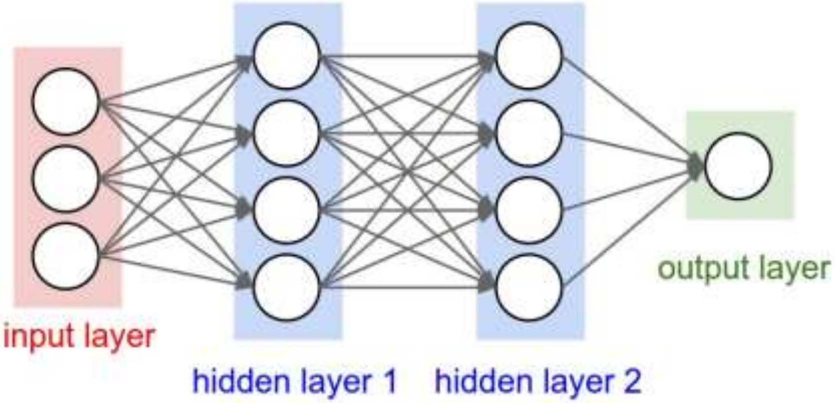


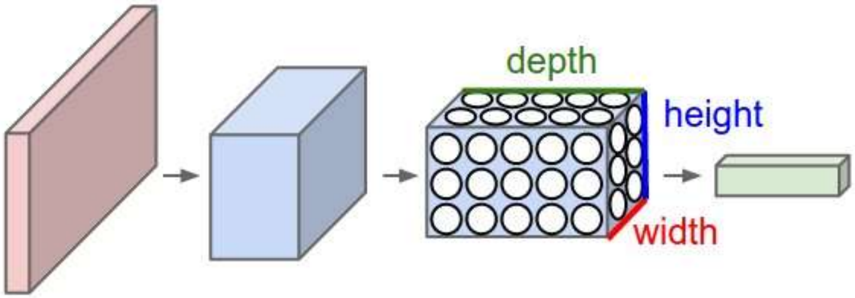
~ 30 trunks
~ 100 branches
~ 300 leaves

- Information content of this image is moderately small.

3d Volumes of Artificial Neurons

- Convolutional Neural Networks take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way. In particular, unlike a regular Neural Network, the layers of a ConvNet have neurons arranged in 3 dimensions: **width, height, depth**.
- Note that the word *depth* here refers to the third dimension of an activation volume, not to the depth of a full Neural Network, which can refer to the total number of layers in a network. Depth of CNN reflects the volume of images, which are 3 dimensional (32x32x3 - width, height, depth, respectively).
- In CNN, neurons in one layer will only be connected to a small region of the layer before it, instead of all of the neurons in a fully-connected manner. Moreover, the final output layer could have dimensions 1x1x10, because by the end of the Conv Net architecture we could reduce the full image into a single vector of class scores, arranged along the depth dimension.





Convolution

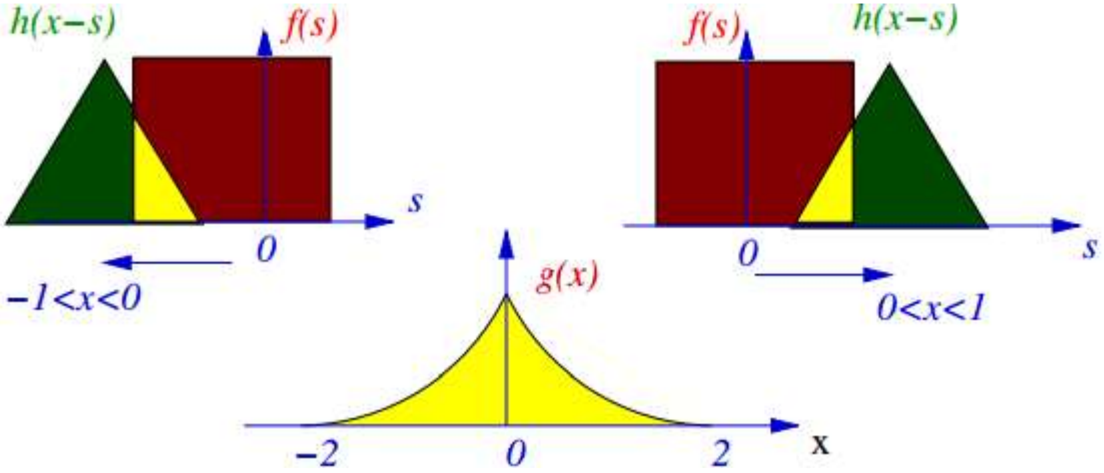
- The convolution defines a product on the linear space of integrable functions.
- Convolution is typically presented as some kind of special product between two functions, with a thick asterisk “*” or a dot in a circle “⨀” as the multiplication symbol rather that a simple “ ·”, dot, or “ “, absence of a dot, for regular product.

$$f * h = h * f$$

- Where the above usually means the following integral expression:

$$g(x) = f * h = \int_{-\infty}^{\infty} f(s)h(x-s)ds$$

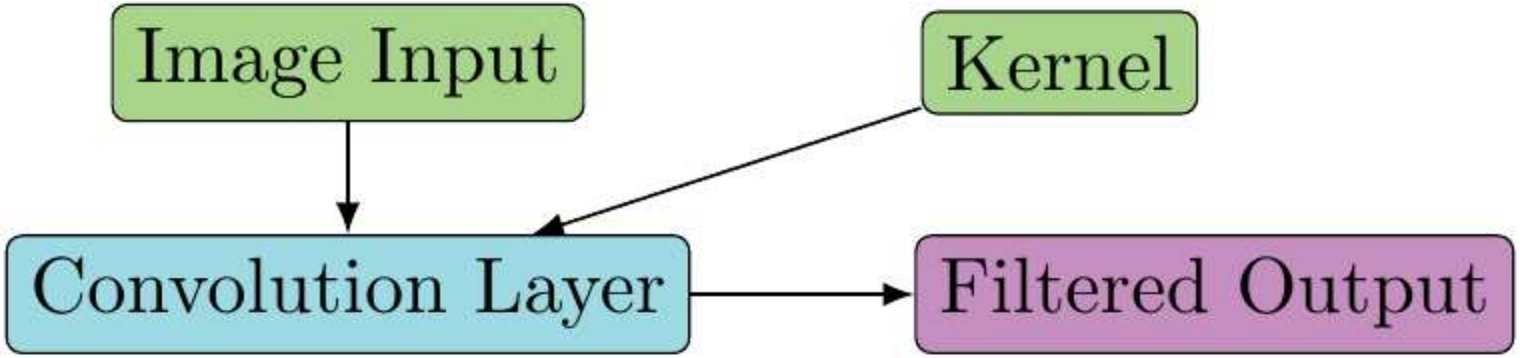
- You visualize the convolution by imagining two functions. Let us say $f(s)$ is stationary and $h(x - s)$ is rolling from left to right. The rolling overlap of two functions is the convolution. Image below illustrates the concept. Notice that convolution operation does not depend on the position of function f(s). Convolution introduces translational invariance.



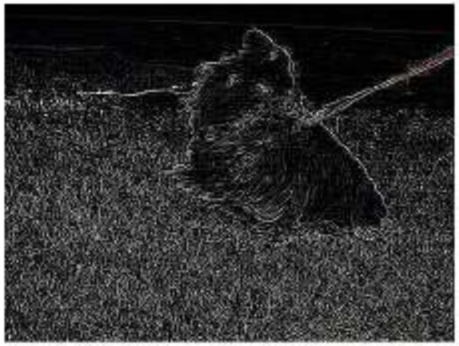
Edge Detection in Image Processing

$f(t) =$

$$g(t) = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$



$(f \star g)(t)$



- Operator g(t) is a discrete version of a Laplacian, the second order special derivative.

Feature Extraction using Convolutions

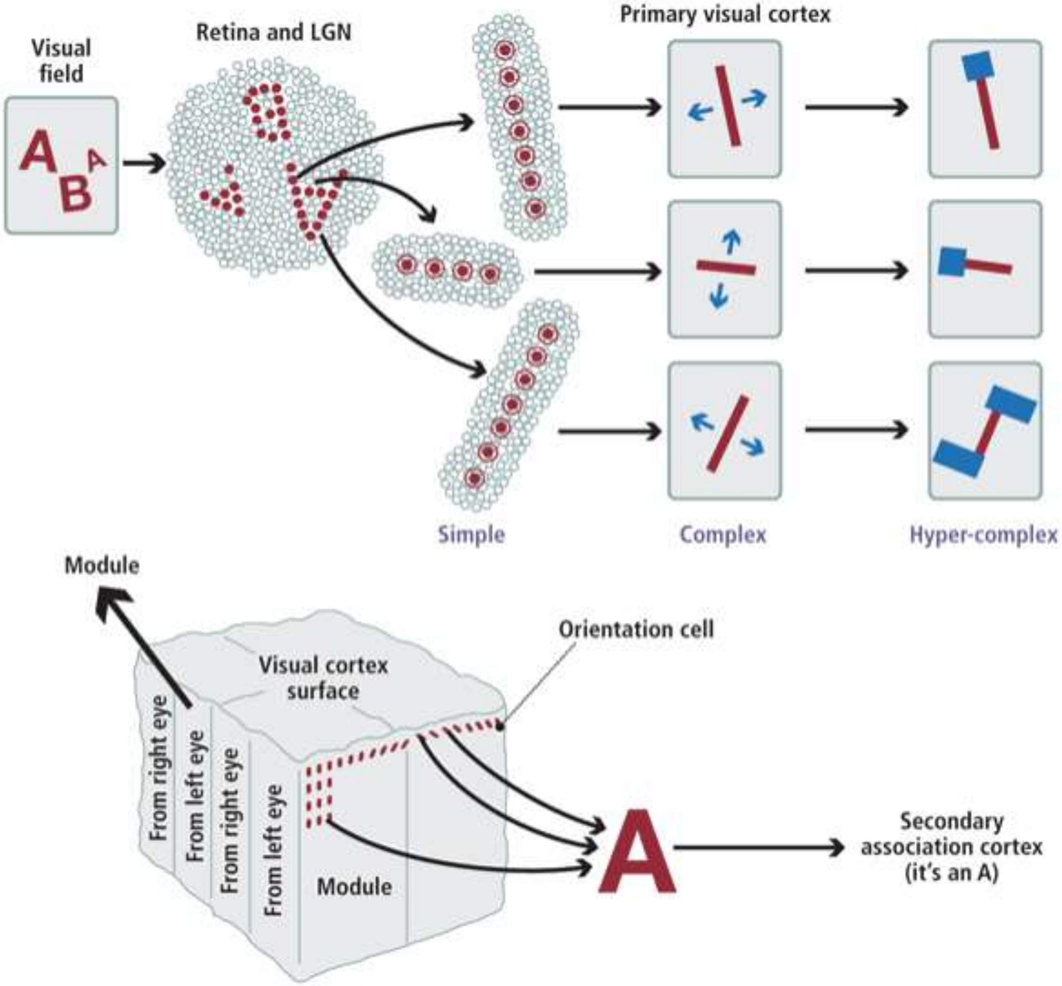
- One design choice that we so far assumed was that we are "fully connecting" all the hidden units to all the input units. On the relatively small images that we were working with (e.g., 28x28 images for the MNIST dataset), it was computationally feasible to learn features on the entire image. However, with larger images (e.g., 96x96 images) learning features that span the entire image (fully connected networks) is very computationally expensive--you would have about 10^4 input units, and assuming you want to learn 100 features, you would have on the order of 10^6 parameters to learn. The feedforward and backpropagation computations would also be about 10^2 times slower, compared to 28x28 images.
- One simple solution to this problem is to restrict the connections between the hidden units and the input units, allowing each hidden unit to connect to only a small subset of the input units. Specifically, each hidden unit will connect to only a small contiguous region of pixels in the input.
- For input modalities different than images, there is often also a natural way to select "contiguous groups" of input units to connect to a single hidden unit as well; for example, for audio, a hidden unit might be connected to only the input units corresponding to a certain time span of the input audio clip.
- This idea of having locally connected networks also draws inspiration from how the early visual system is wired up in biology. Specifically, neurons in the visual cortex have localized receptive fields (i.e., they respond only to stimuli in a certain location).

Visual Cortex Motivation

- CNNs take a biological inspiration from the visual cortex. The visual cortex has small regions of cells that are sensitive to specific alignment of the visual field. This idea was demonstrated by an experiment by Hubel and Wiesel in 1962. Two neuroscientists showed that some individual neuronal cells in the brain responded (are fired) only in the presence of edges of a certain orientation. For example, some neurons fired when exposed to vertical edges and some when shown horizontal or diagonal edges.
- Hubel and Wiesel found out that all of these neurons were organized in a columnar architecture and that together, they were able to produce visual perception.
- This idea of specialized components inside of a system having specific tasks (the neuronal cells in the visual cortex looking for specific visual characteristics) is the one that we use in computer vision, and is the basis behind CNNs.
- The neurons of primary visual cortex are exquisitely sensitive to several complex stimulus attributes, including orientation of a contour, direction of motion, size, and binocular disparity.
- In many cases, selectivity is absolute: cortical neurons will not respond at all unless the stimulus matches the neuron’s preferences for each of these attributes.

Visual Cortex

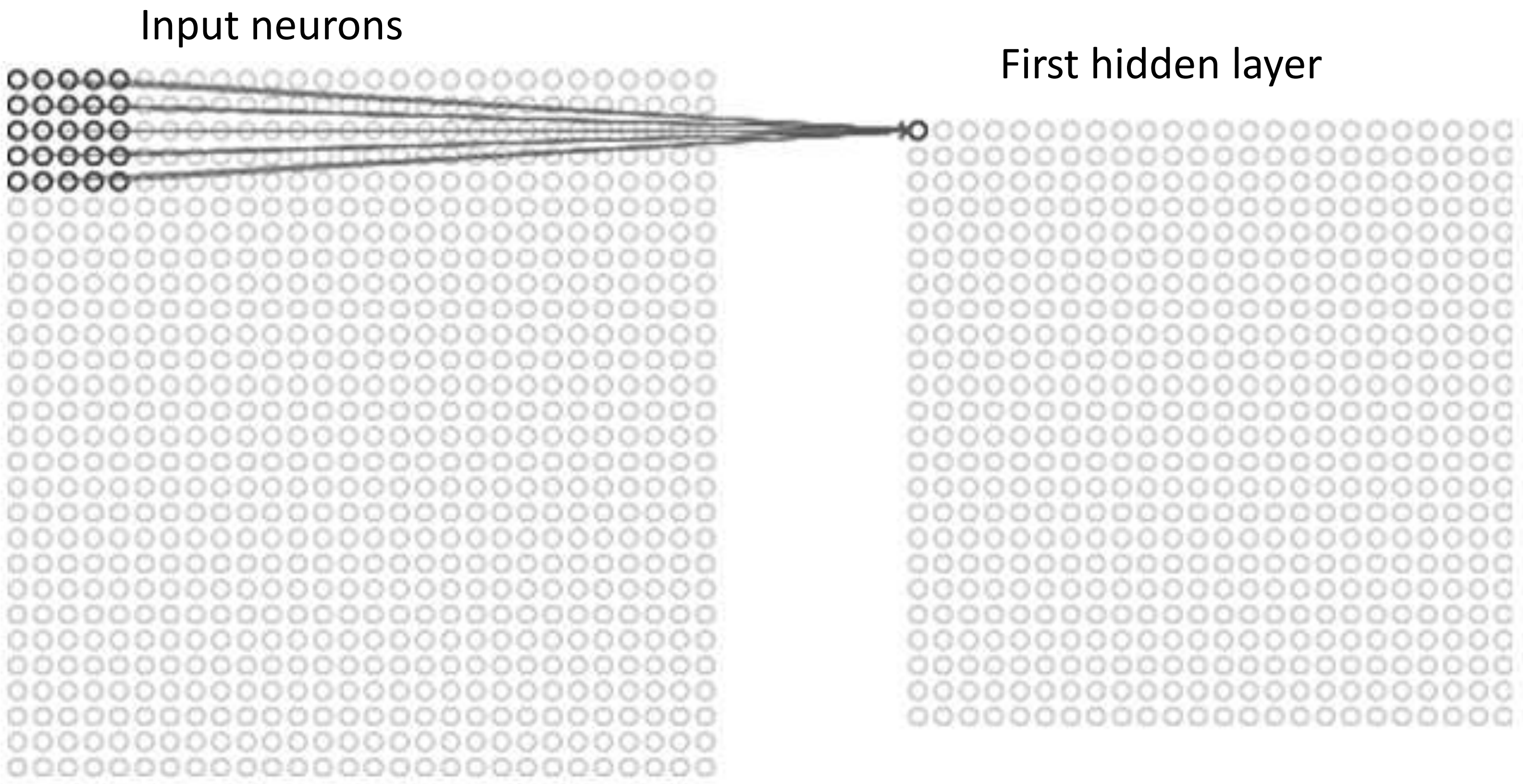
- In the Visual Cortex, simple cells become active when they are subjected to stimuli such as edges.
- Complex cells then combine the information of several simple cells and detect the position and orientation of a structure.
- Hyper-complex cells then detect endpoints and crossing lines from this position and orientation information, which is then used in the brain's secondary cortex for information association. (*Image courtesy of University of Colorado*)



Convolution Layer

- The first layer in a CNN is always a **Convolutional Layer**. The input could be a 32 x 32 x 3 array of pixel values. Now, the best way to explain a conv layer is to imagine a flashlight that is shining over the top left of the image. Let’s say that the light this flashlight shines covers a 5 x 5 area.
- Next, let’s imagine this flashlight sliding across all the areas of the input image. In machine learning terms, this flashlight is called a **filter** (or sometimes referred to as a **neuron** or a **kernel**) and the region that it is shining over is called the **receptive field**.
- This filter is also an array of numbers (the numbers are called **weights** or **parameters**).
- The depth of this filter has to be the same as the depth of the input, so the dimensions of this filter is typically 5 x 5 x 3.
- Look at the first position the filter is in for example. It could be the top left corner. As the filter is sliding, or **convolving**, around the input image, it is multiplying the values in the filter with the original pixel values of the image (performing **element wise matrix multiplications**).
- These multiplications are all summed up (there would be 75 multiplications in total).
- The result is a single number. This number is calculated when the filter is at the top left of the image. As we slide the filter over the image we repeat this process for every location on the input volume. Typically, we move the filter to the right by 1 unit, then right again by 1, and so on.
- Every unique location on the input volume produces a number. After sliding the filter over all the locations, we are left with is a 28 x 28 x 1 array of numbers, which we call an **activation map** or **feature map**. The reason you get a 28 x 28 array is that there are 28x28 = 784 different locations that a 5 x 5 filter can fit on a 32 x 32 input image.

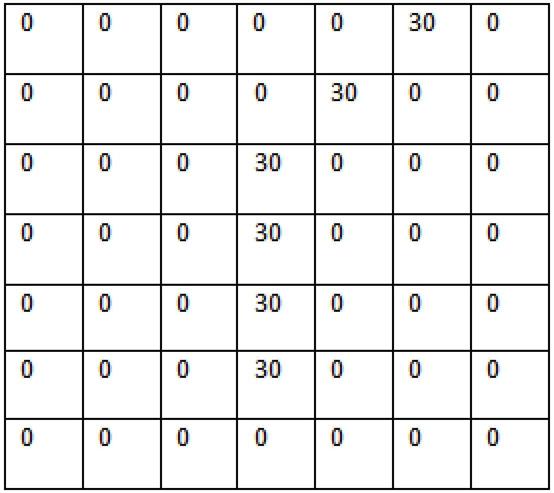
Convolution Operation



- Visualization of a 5x5 filter “convoluting” around an input volume (volume = image area x depth of 3 colors) and producing an activation map.

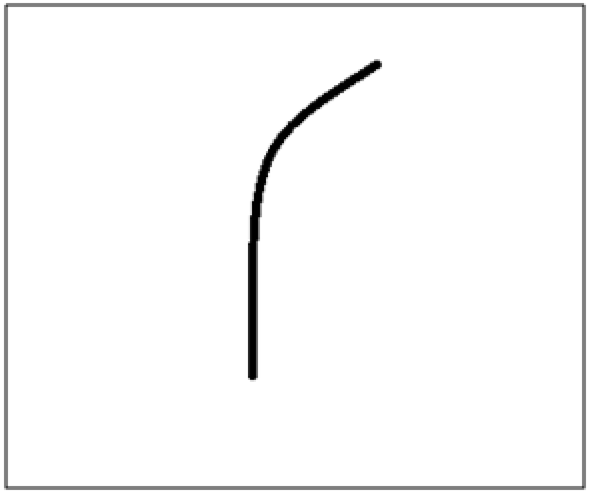
Feature Selection

- Each of these filters can be thought of as **feature identifiers**. When we say features, we think about things like straight edges, simple colors, and curves.
- Consider the simplest characteristics that all images have in common with each other. Let’s say our first filter is 7 x 7 x 3 and is a curve detector.
- For the moment we will ignore the fact that the filter is 3 units deep. We will only consider the top slice of the filter and the image, for simplicity.
- As a curve detector, the filter will have a pixel structure in which there will be higher numerical values along the area that is a shape of a curve.
- Filters that we’re talking about are matrices made of numbers



0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

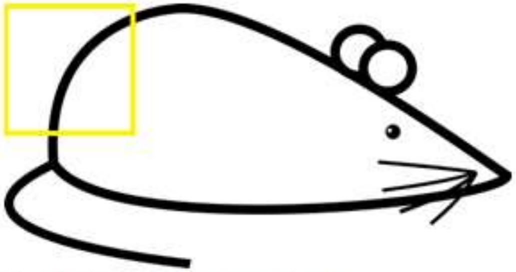


Visualization of a curve detector filter

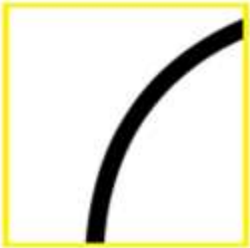
Mouse Parts

- Now let’s take an example of an image that we want to classify, and let’s put our filter at the top left corner.

Original image



Visualization of the filter on the image



Visualization of the receptive field

0	0	0	0	0	0	30
0	0	0	0	50	50	50
0	0	0	20	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0

Pixel representation of the receptive field

*

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

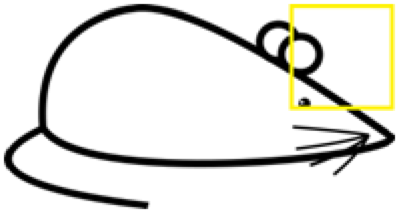
Pixel representation of filter

Multiplication and Summation = (50*30)+(50*30)+(50*30)+(20*30)+(50*30) = 6600 (A large number!)

- If in the input image, there is a shape that generally resembles the curve that this filter is representing, then all of the multiplications summed together will result in a large value

Mouse Parts

- What happens when we move our filter to the right.



Visualization of the filter on the image

0	0	0	0	0	0	0
0	40	0	0	0	0	0
40	0	40	0	0	0	0
40	20	0	0	0	0	0
0	50	0	0	0	0	0
0	0	50	0	0	0	0
25	25	0	50	0	0	0

Pixel representation of receptive field

*

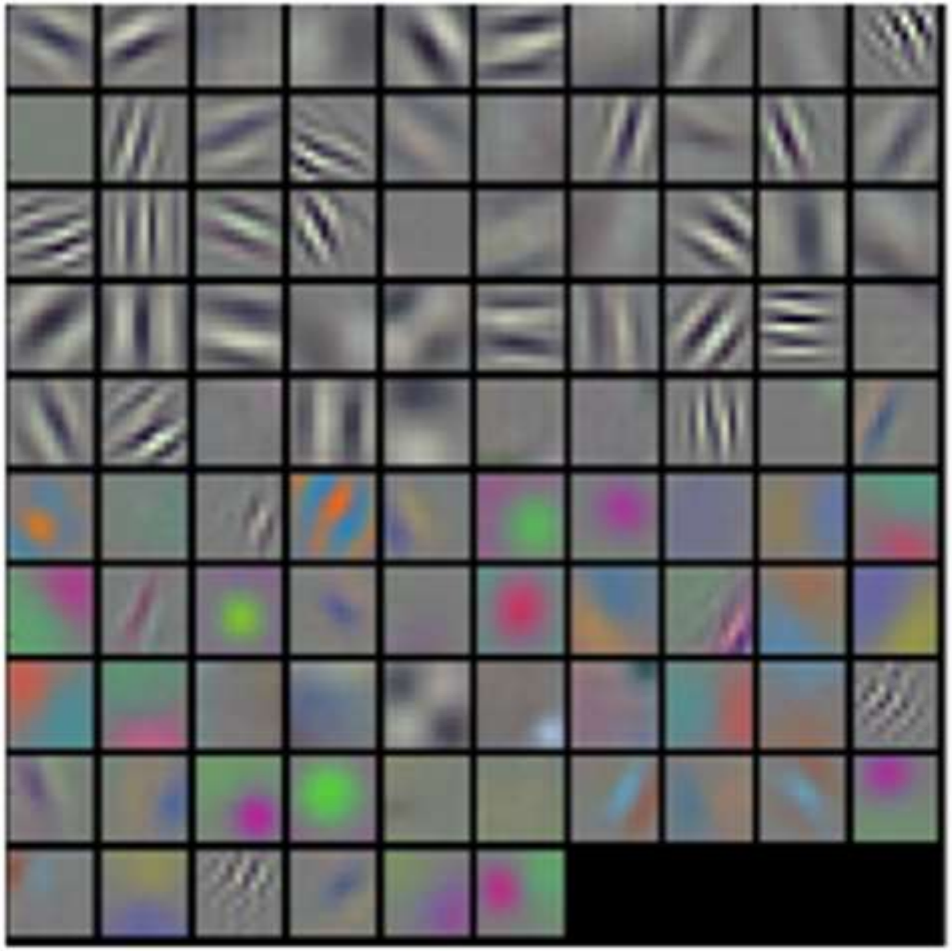
0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

Multiplication and Summation = 0

- The value is much lower! This because there is nothing in the image section that responded to the curve detector filter.
- The output of this conv layer is an activation map. So, in the simple case of a one filter convolution (and if that filter is a curve detector), the activation map will show the areas in which there at mostly likely to be curves in the picture. In above example, the top left value of our 28 x 28 x 1 activation map will be 6600. This high value means that it is likely that there is some sort of curve in the input volume that caused the filter to activate.
- The top right value in our activation map will be 0 because there is nothing in the input volume that caused the filter to activate (there is no curve in that region of the original image). This is just for one filter which detect lines that curve outward and to the right.
- We could have other filters for lines that curve to the left or for straight edges.
- The more filters, the greater the depth of the activation map, and the more information we have about the input volume.

General Convolution Filters (Kernels)



Example filters learned by Krizhevsky et al. Each of the 96 filters shown here is of size [11x11x3], and each one is shared by the 55*55 neurons in one depth slice. Notice that the parameter sharing assumption is relatively reasonable: 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.

Gabor Filters or Wavelets

- Complex exponentials multiplied by Gaussians, are referred to as Gabor filters or wavelets.

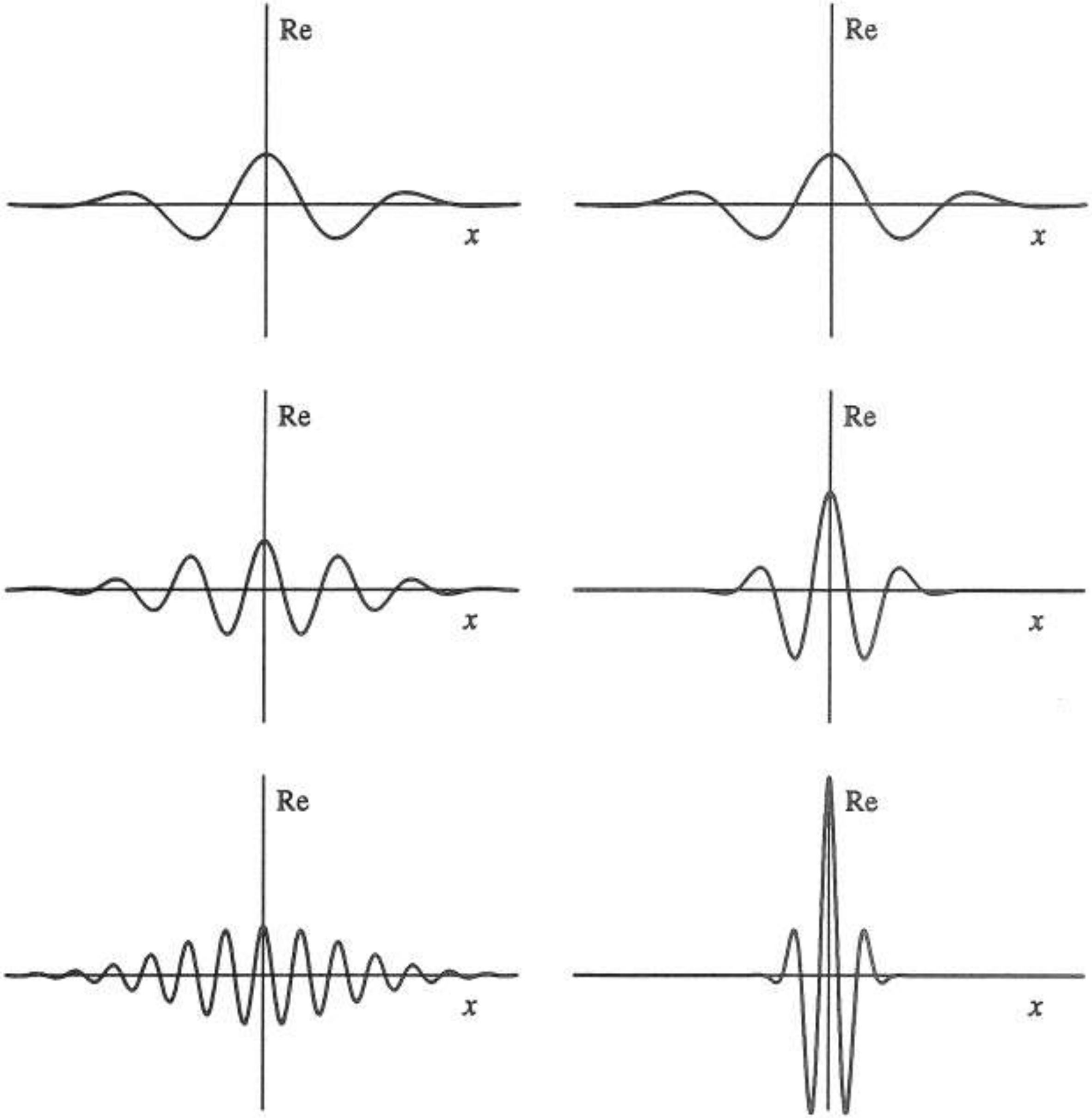
$$f(x) = e^{-(x-x_0)^2/a^2} e^{-ik_0(x-x_0)} \quad F(k) = e^{-(k-k_0)^2 a^2} e^{ix_0(k-k_0)}$$

- Gabor wavelets are localized at position (epoch) x_0, modulated by frequency k_0, and have a width or spread *a.*
- Fourier Transforms $F(k)$ of a Gabor filter has the exact same functional form, but with parameters interchanged or inverted.
- Because of the optimality of such wavelets, Gabor (1946) proposed using them as an expansion basis to represent signals in broadcast telecommunications as the elementary functions for encoding continuous-time information.
- Unfortunately, such functions are mutually non-orthogonal. It is difficult to obtain the actual coefficients in order to expand a given signal in this basis.
- The first constructive method for finding such Gabor coefficients was developed only in 1981 by the Dutch physicist Martin Bastiaans.

What are Wavelets

- Fourier analysis has its limitations, most notably, it is not well suited for transient, time–varying signals.
- We want the representations of the data to adapt to the nature of the information.
- Different groups of researchers in disparate fields developed techniques to decompose signals into pieces that could be localized in time and analyzed at different scales of resolution.
- Wavelet is a mathematical function used in digital signal processing for compressing and analyzing sound and images.
- The wavelet transform is a technique to deconstruct images or other signals into a superposition of basic functions (wavelets).
- Wavelet decomposition can be used as a compression mechanism. Rather than store the entire image we could store only wavelet coefficients.
- Very good quality image compression that takes up as little as one fourth of the space of a JPEG.
- In digital signal processing, the wavelet transform could be used in recovery of weak signals from noise, for example.
- It appears that Gabor wavelets or filters were instrumental in success of CNNs and are relevant for understanding or neural structures and processes in visual cortex and perhaps other parts of our brains. They are relevant in brains of cats as well.

Form of Gabor Wavelets in Real Plane



Generalization to Two Dimensions, Images

- Two-dimensional Gabor wavelets over the image domain (x; y) have the functional form:

$$f(x,y) = e^{-\left[(x-x_0)^2/\alpha^2+(y-y_0)^2/\beta^2\right]} e^{-i[u_0(x-x_0)+v_0(y-y_0)]}$$

- Or, in the Fourier domain:

$$F(u,v) = e^{-\left[(u-u_0)^2\alpha^2+(v-v_0)^2\beta^2\right]} e^{i[x_0(u-u_0)+y_0(v-v_0)]}$$

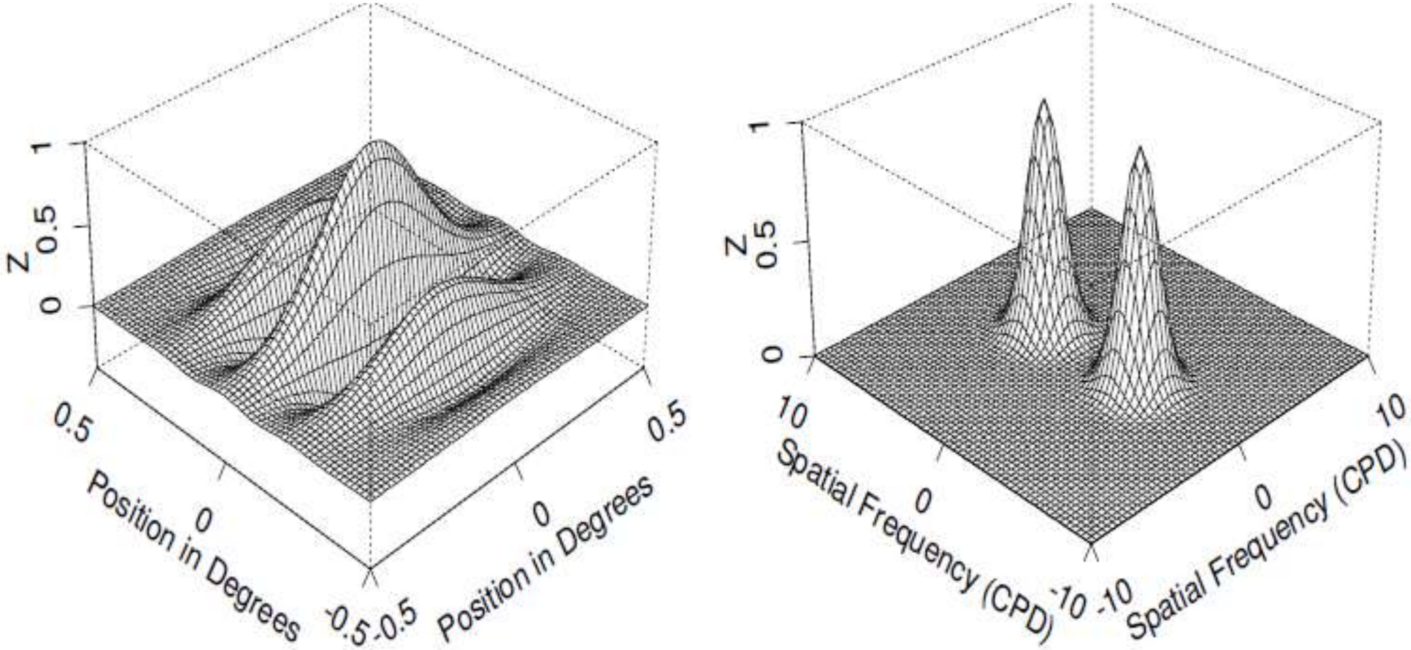


Figure 1: The real part of a 2-D Gabor wavelet, and its 2-D Fourier transform.

Characteristics of Vision, Need for Many Scales

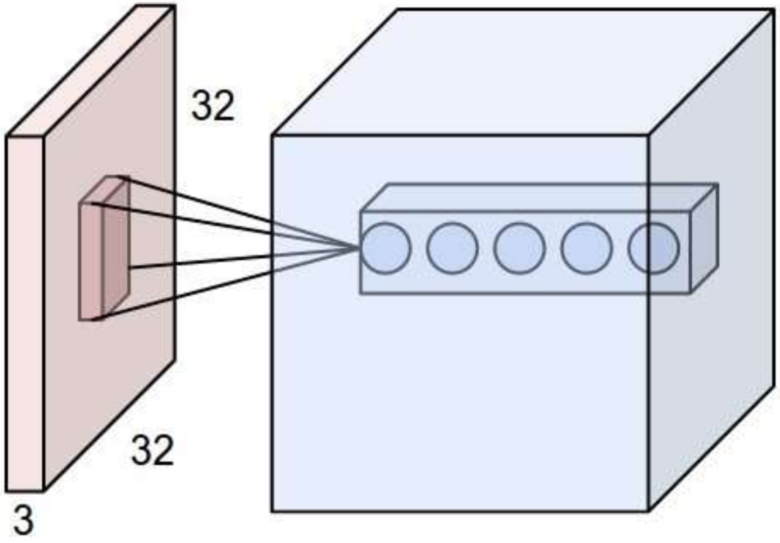
- Consider how our eyes look at the world. In the real world, you can observe a forest like the one shown in the photograph on the next page from many vantage points—in effect, at different scales of resolution.
  - From the window of a cross-country jet, for example, the forest appears to be a solid canopy of green.
  - From the window of an automobile on the ground, the canopy resolves into individual trees.
  - If you get out of the car and move closer, you begin to see branches and leaves.
  - If you then pull out a magnifying glass, you might find a drop of dew at the end of a leaf.
  - As you zoom in at smaller and smaller scales, you can find details that you didn't see before.
- Try to do that with a photograph, however, and you will be disappointed. Enlarge the photograph to get "closer" to a tree and all you'll have is a fuzzier tree; the branch, the leaf, the drop of dew are not to be found.
- Although our eyes can see the forest at many scales of resolution, the camera can show only one at a time.
- Computers do worse than cameras; in fact, their level of resolution is inferior. On a computer screen, the photograph becomes a collection of pixels that are much less sharp than the original.

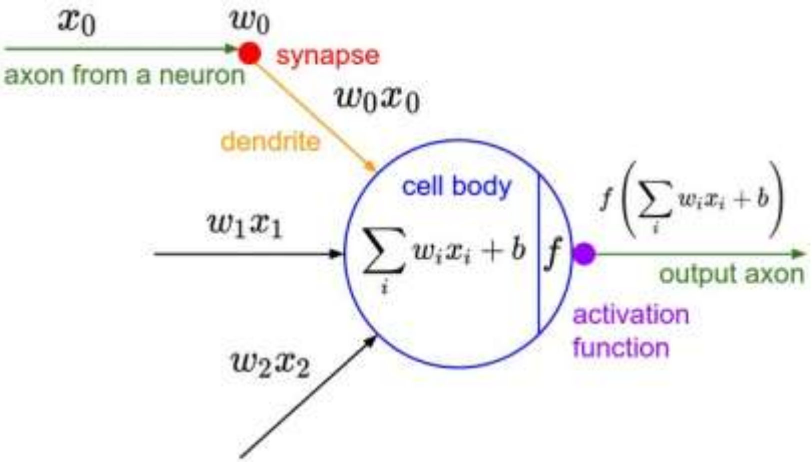
Local Connectivity

- When dealing with high-dimensional inputs such as images, as we saw above, it is impractical to connect neurons to all neurons in the previous volume. Instead, we will connect each neuron to only a local region of the input volume.
- The spatial extent of this connectivity is a hyper parameter called the **receptive field** of the neuron (the filter size). The extent of the connectivity along the depth axis is always equal to the depth of the input volume. It is important to emphasize again this asymmetry in how we treat the spatial dimensions (width and height) and the depth dimension: The connections are local in space (along width and height), but always full along the entire depth of the input volume.
- For example, suppose that the input volume has size [32x32x3], (e.g. an RGB image). If the receptive field (or the filter size) is 5x5, then each neuron in the Conv Layer will have weights to a [5x5x3] region in the input volume, for a total of 5*5*3 = 75 weights (and +1 bias parameter).
- Notice that the extent of the connectivity along the depth axis must be 3, since this is the depth of the input volume.
- Suppose an input volume had size [16x16x20]. Then using an example receptive field size of 3x3, every neuron in the Conv Layer would now have a total of 3*3*20 = 180 connections to the input volume. Notice that, again, the connectivity is local in space (e.g. 3x3), but full along the input depth (20).

Local Connectivity

- Consider an input volume on the left (e.g. a 32X32X3 image), and a 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 along the depth, all looking at the same region of the input .
- **Right:** The neurons 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.





Layers used in CNN

- Simple ConvNet is a sequence of layers, and every layer of a ConvNet transforms one volume of activations to another through a differentiable function. We use three main types of layers to build ConvNet architectures: **Convolutional Layer**, **Pooling Layer**, and **Fully-Connected Layer** (the same as seen in regular Neural Networks).
- We will stack these layers to form a full ConvNet **architecture**.
- *Example Architecture: Overview*. We will go into more details below, but a simple ConvNet for image classification could have the architecture [INPUT - CONV - RELU - POOL - FC]. I
- 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 down sampling 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.

Spatial Arrangement

- We have explained the connectivity of each neuron in the Conv Layer to the input volume, but we haven’t yet discussed how many neurons there are in the output volume or how they are arranged. Three hyper parameters control the size of the output volume: the **depth, stride** and **zero-padding**.
- First, the **depth** of the output volume is a hyper parameter: it corresponds to the number of filters we would like to use, each learning to look for something different in the input. For example, if the first Convolutional Layer takes as input the raw image, then different neurons along the depth dimension may activate in presence of various oriented edged, or blobs of color. We will refer to a set of neurons that are all looking at the same region of the input as a **depth column** (some people also prefer the term *fibre*).
- Second, we must specify the **stride** with which we slide the filter. When the stride is 1 then we move the filters one pixel at a time. When the stride is 2 (or uncommonly 3 or more, though this is rare in practice) then the filters jump 2 pixels at a time as we slide them around. This will produce smaller output volumes spatially.
- As we will soon see, sometimes it will be convenient to pad the input volume with zeros around the border. The size of this **zero-padding** is a hyperparameter. The nice feature of zero padding is that it will allow us to control the spatial size of the output volumes (most commonly as we’ll see soon we will use it to exactly preserve the spatial size of the input volume so the input and output width and height are the same).

Matrix Multiplication

- The convolution operation essentially performs dot products between the filters and local regions of the input. A common implementation pattern of the CONV layer is to take advantage of this fact and formulate the forward pass of a convolutional layer as one big matrix multiply as follows:
- 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].

Dilated Convolutions

- A recent development (e.g. see paper by Fisher Yu and Vladlen Koltun) is to introduce one more hyperparameter to the CONV layer called the *dilation*.
- WE have created an impression that CONV filters are contiguous. However, it's possible to have filters that have spaces between each cell, called dilation. As an example, in one dimension a filter w of size 3 would compute over input x the following: w[0]*x[0] + w[1]*x[1] + w[2]*x[2]. This is dilation of 0. For dilation 1 the filter would instead compute w[0]*x[0] + w[1]*x[2] + w[2]*x[4]; In other words there is a gap of 1 between the applications.
- This can be very useful in some settings to use in conjunction with 0-dilated filters because it allows you to merge spatial information across the inputs much more aggressively with fewer layers. For example, if you stack two 3x3 CONV layers on top of each other than you can convince yourself that the neurons on the 2nd layer are a function of a 5x5 patch of the input (we would say that the *effective receptive field* of these neurons is 5x5). If we use dilated convolutions then this effective receptive field would grow much quicker.

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. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, using the MAX operation. The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2 downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations. Every MAX operation would in this case be taking a max over 4 numbers (little 2x2 region in some depth slice). The depth dimension remains unchanged.
- Average pooling was often used historically but has recently fallen out of favor compared to the max pooling operation, which has been shown to work better in practice.

Logits

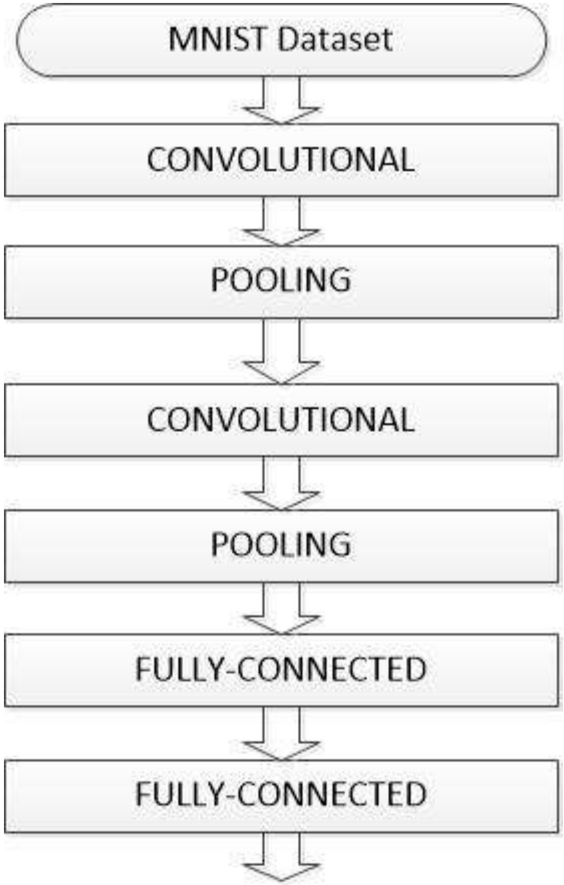
- Logit is a function that maps probabilities $([0,1])$ to the entire real axis $R(-\infty,\infty)$.
- Probability of 0.5 corresponds to a logit of 0.
- Negative logit corresponds to probabilities less that 0.5, positive to probabilities $> \ 0.5$.
- Logit is calculated as:

$$L = \ln(\frac{p}{1-p}) \, , p \ = \ \frac{1}{1+ \, e^{-L}}$$

- The logit function is the inverse of the sigmoidal "logistic" function or logistic transform used in mathematics, especially in statistics.
- When the function's variable represents a probability p, the logit function gives the log-odds, or the logarithm of the odds: $p/(1 \ - \ p)$.

MNIST Example

- We will use a simple architecture with several layers



Layer Definition

- As our graph indicates we will use two simple convolution layers and 2 fully connected layers. We define those by these two functions:

```
# Define a simple convolutional layer
def conv_layer(input, channels_in, channels_out):
   w = tf.Variable(tf.zeros([5, 5, channels_in, channels_out]))
   b = tf.Variable(tf.zeros([channels_out]))
   conv = tf.nn.conv2d(input, w, strides=[1, 1, 1, 1], padding="SAME")
   act = tf.nn.relu(conv + b)
   return act
```

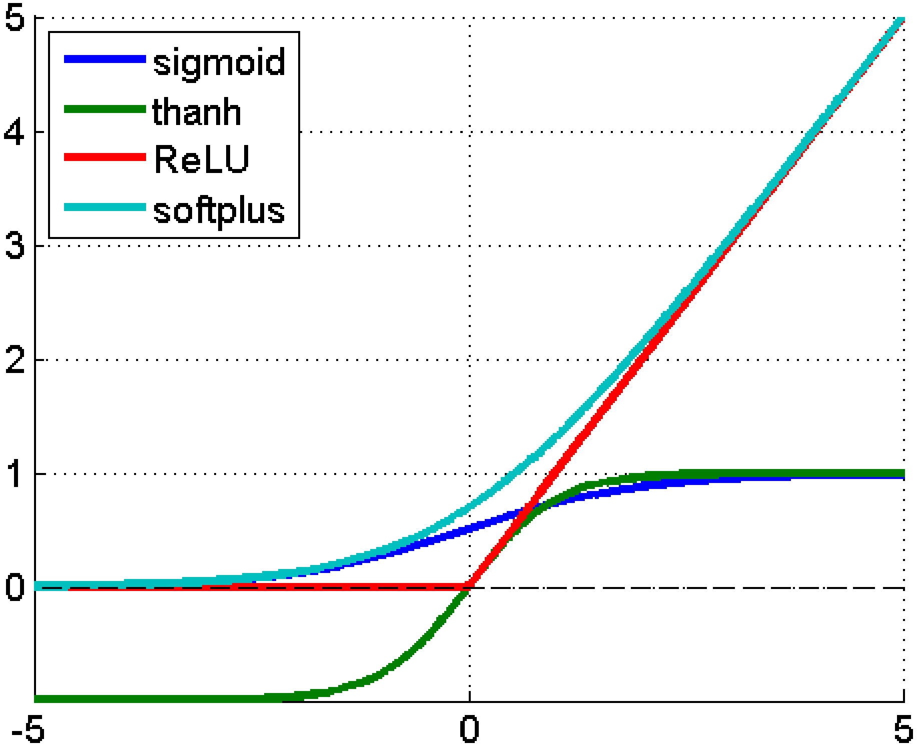
- Convolution filter is `[5x5]`. Variables `w` and `b` contain weights and biases. We will apply convolution with with those weights and biases and simple strides. Since the edges of our images are made of zeros, we will use `padding = "SAME"`

```
# And a fully connected layer
def fc_layer(input, channels_in, channels_out):
   w = tf.Variable(tf.zeros([channels_in, channels_out]))
   b = tf.Variable(tf.zeros([channels_out]))
   act = tf.nn.relu(tf.matmul(input, w) + b)
   return act
```

- `Relu` function is used in place of sigmoid and `matmul` of inputs with weights increment by biases is our regular input into non-linera layer

ReLU activation function

- We advertised sigmoid function so far however in many problems we need to use other non-linear functions. Rectifier Linear Unites are frequent choice:



- Why use ReLU. $sigmod, \sigma(x)$ has those very flat regions left and right of zero. If you find yourself with weights in those flat regions slopes of sigmoids are zeros and your gradient descent propagation will be very, very slow. That is not practical.

Feed-forward part of our Network

- Place holders take images and labels. Images are reshaped to look like 28x28 image

```
x = tf.placeholder(tf.float32, shape=[None, 784])
y = tf.placeholder(tf.float32, shape=[None, 10])
x_image = tf.reshape(x, [-1, 28, 28, 1])
```

- Create the network by stacking convolutional layers followed by pooling layers. We use max_pool-ing on regions of size 2X2.

```
conv1 = conv_layer(x_image, 1, 32)
pool1 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], 
padding="SAME")
conv2 = conv_layer(pooled, 32, 64)
pool2 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], 
padding="SAME")
```

- We flatten the image into long vectors

```
flattened = tf.reshape(pool2, [-1, 7 * 7 * 64])
fc1 = fc_layer(flattened, 7 * 7 * 64, 1024)
logits = fc_layer(fc1, 1024, 10)
```

Loss & Training

- Compute cross entropy as our loss function

```
cross_entropy = tf.reduce_mean(
tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=y))
```

- Use an AdamOptimizer to train the network

```
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
```

- Adam, an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments. The method is straightforward to implement, is computationally efficient, has little memory requirements, is invariant to diagonal rescaling of the gradients, and is well suited for problems that are large in terms of data and/or parameters. The method is also appropriate for non-stationary objectives and problems with very noisy and/or sparse gradients.
- This example will work with several other optimimizers.
- compute the accuracy so that it can be printed to the console

```
correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

Initialize All Variables, train the Model

- Initialize all the variables

```
sess.run(tf.global_variables_initializer())
```

- Train for 2000 steps. For each step we take a batch of MNIST data

```
for i in range(2000):
   batch = mnist.train.next_batch(100)
```

- Occasionally report accuracy

```
   if i % 500 == 0:
     [train_accuracy] = sess.run([accuracy], feed_dict={x: batch[0], y: 
batch[1]})
      print("step %d, training accuracy %g" % (i, train_accuracy))
```

- Of course this happens when we `session.run` the training step

```
      sess.run(train_step, feed_dict={x: batch[0], y_true: batch[1]})
```

- We also visualize the Graph using FileWriter

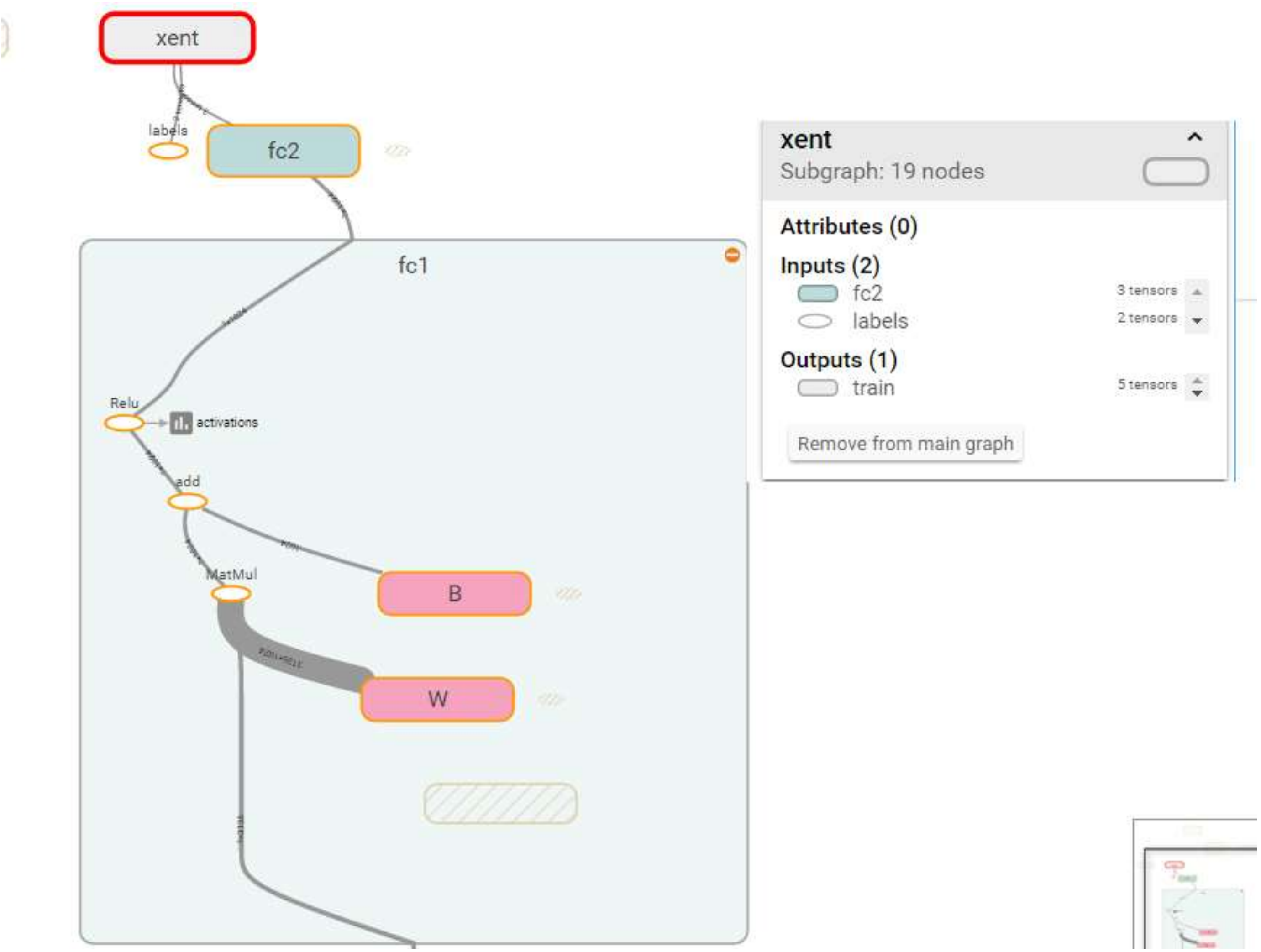
```
tf.summary.FileWriter (‘log/mnist/1’):
```

- Python class that writes data for TensorBoard. Finally we open TensorBoard

```
$ tensorboard --logdir /log/mnist/1
```

Trace Inputs

- You can select a variable or name scope and Trace inputs will tell you where all its data is coming from.



xent

Subgraph: 19 nodes

Attributes (0)

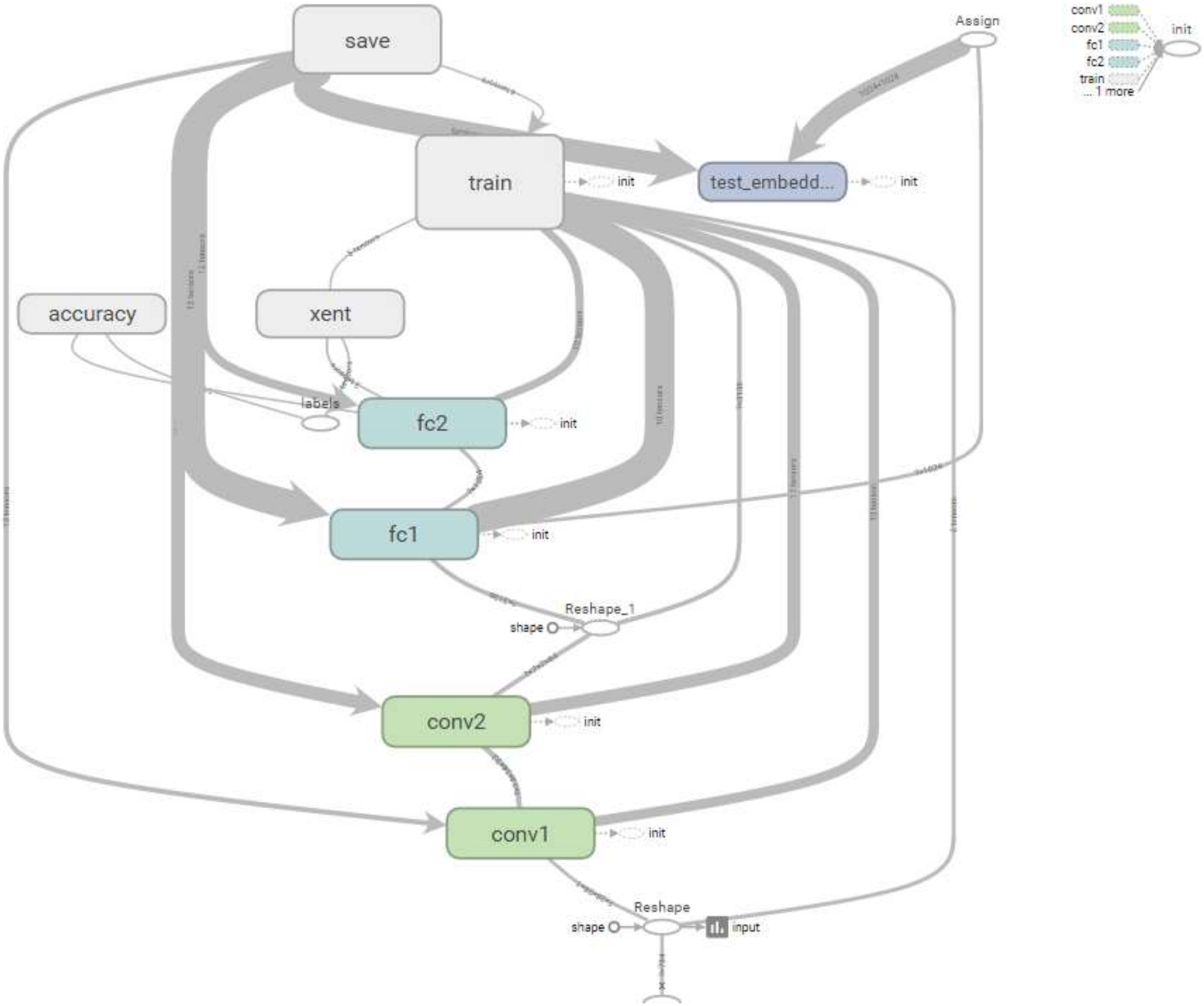
Inputs (2)

🟦	fc2	3 tensors
○	labels	2 tensors

Outputs (1)

▭	train	5 tensors

Main Graph

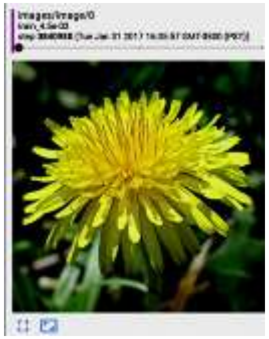

save
Assign
conv1
conv2
fc1
fc2
train
... 1 more
init
train
init
test_embedd...
init
accuracy
xent
labels
fc2
init
fc1
init
Reshape_1
shape
conv2
init
conv1
init
Reshape
shape
input

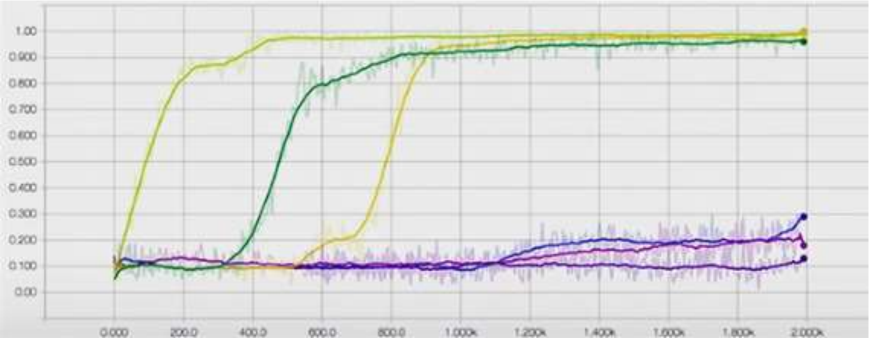
Summaries

summary (n):

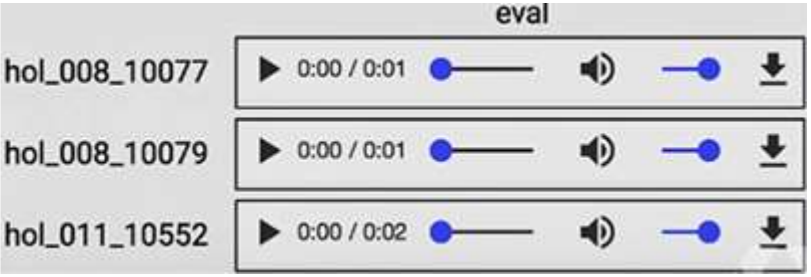
- Summaries are TensorFlow ops that take regular tensors and output protocol buffers containing "summarized" data
- Examples:

- **tf.summary.scalar** create nice graphs of data
- **tf.summary.image** use when you want to verify input images or generated images





- tf.summary.audio use for music or speech
- tf.summary.histograms summaries of statistical data

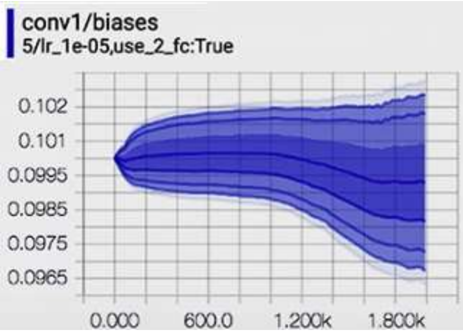


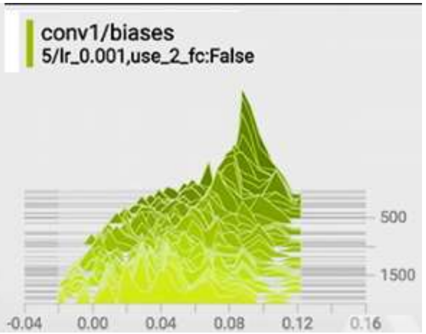
eval

hol_008_10077 ▶ 0:00 / 0:01

hol_008_10079 ▶ 0:00 / 0:01

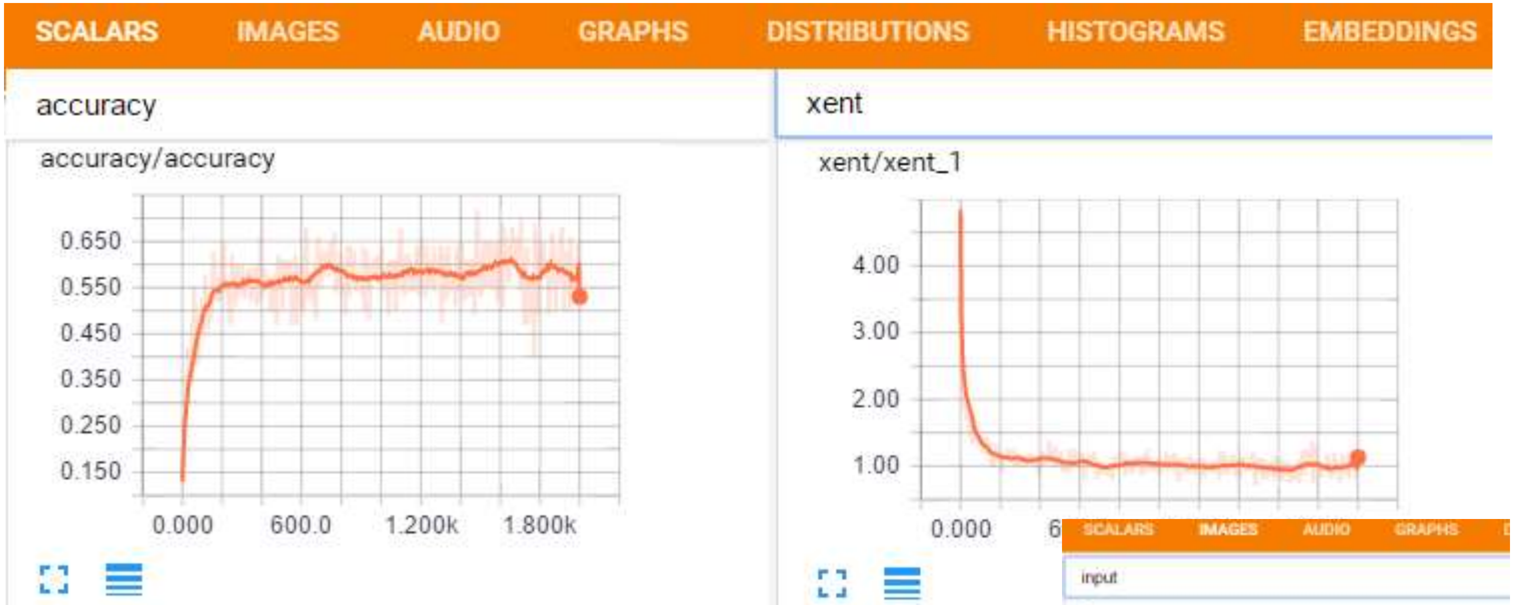
hol_011_10552 ▶ 0:00 / 0:02



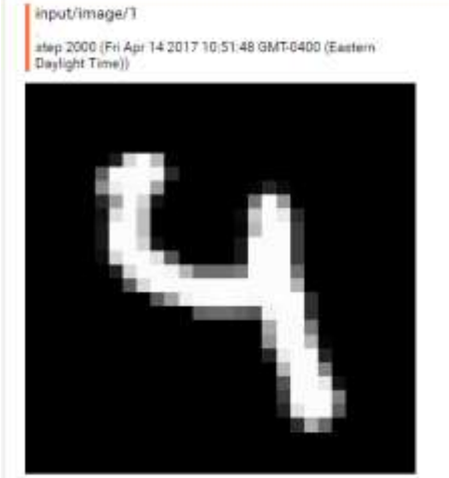


Collect Summaries

- `tf.summary.scalar('cross_entropy', xent)`
- `tf.summary.scalar('accuracy', accuracy)`

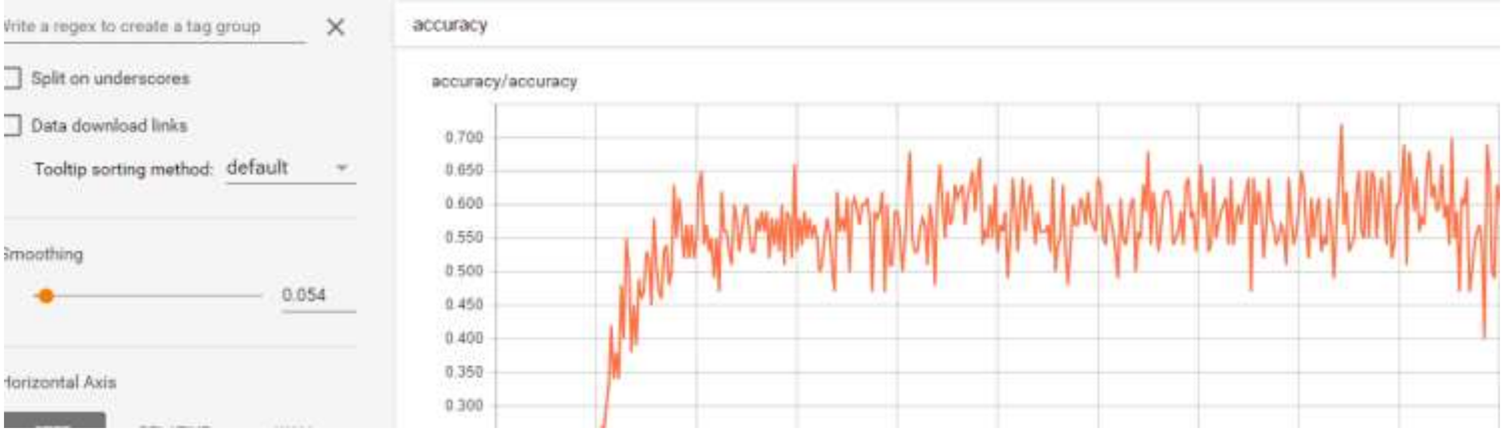


- `tf.summary.image('input', x_image, 3)`


SCALARS
IMAGES
AUDIO
GRAPHS
input
input/image/1
step 2000 (Fri Apr 14 2017 10:51:48 GMT-0400 (Eastern Daylight Time))

Smoothing Scalar Graphs

- When data are very noisy you do not see trends clearly.
- Smoothing sliding bar allows you to see trends better by choosing the length of averaging interval.



Write a regex to create a tag group ✕

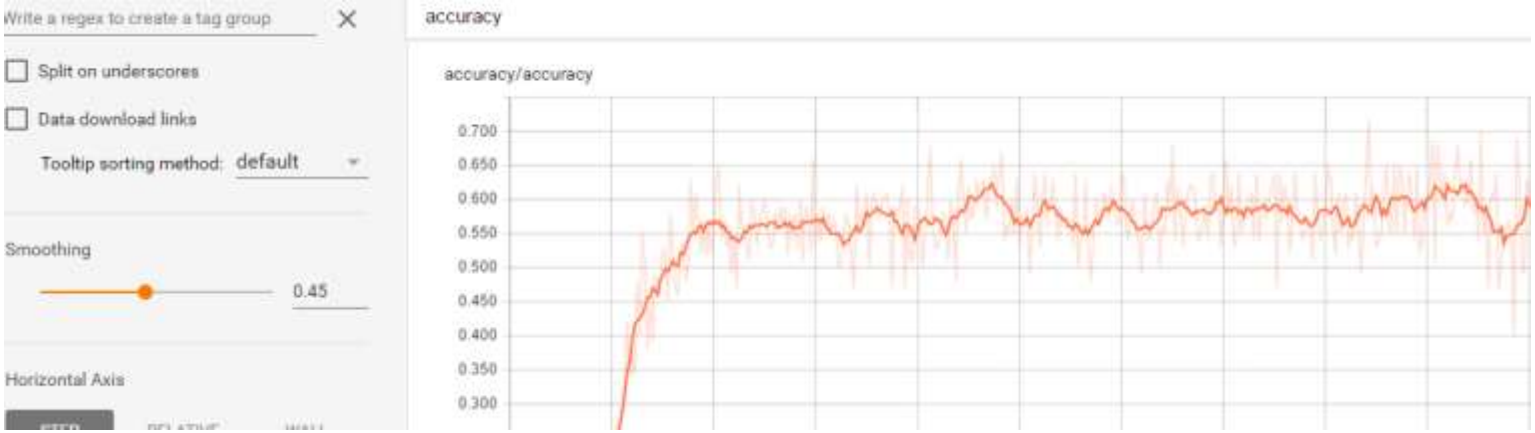
☐ Split on underscores

☐ Data download links

Tooltip sorting method: default

Smoothing 0.054

Horizontal Axis



Write a regex to create a tag group ✕

☐ Split on underscores

☐ Data download links

Tooltip sorting method: default

Smoothing 0.45

Horizontal Axis

Add Histogram Summaries

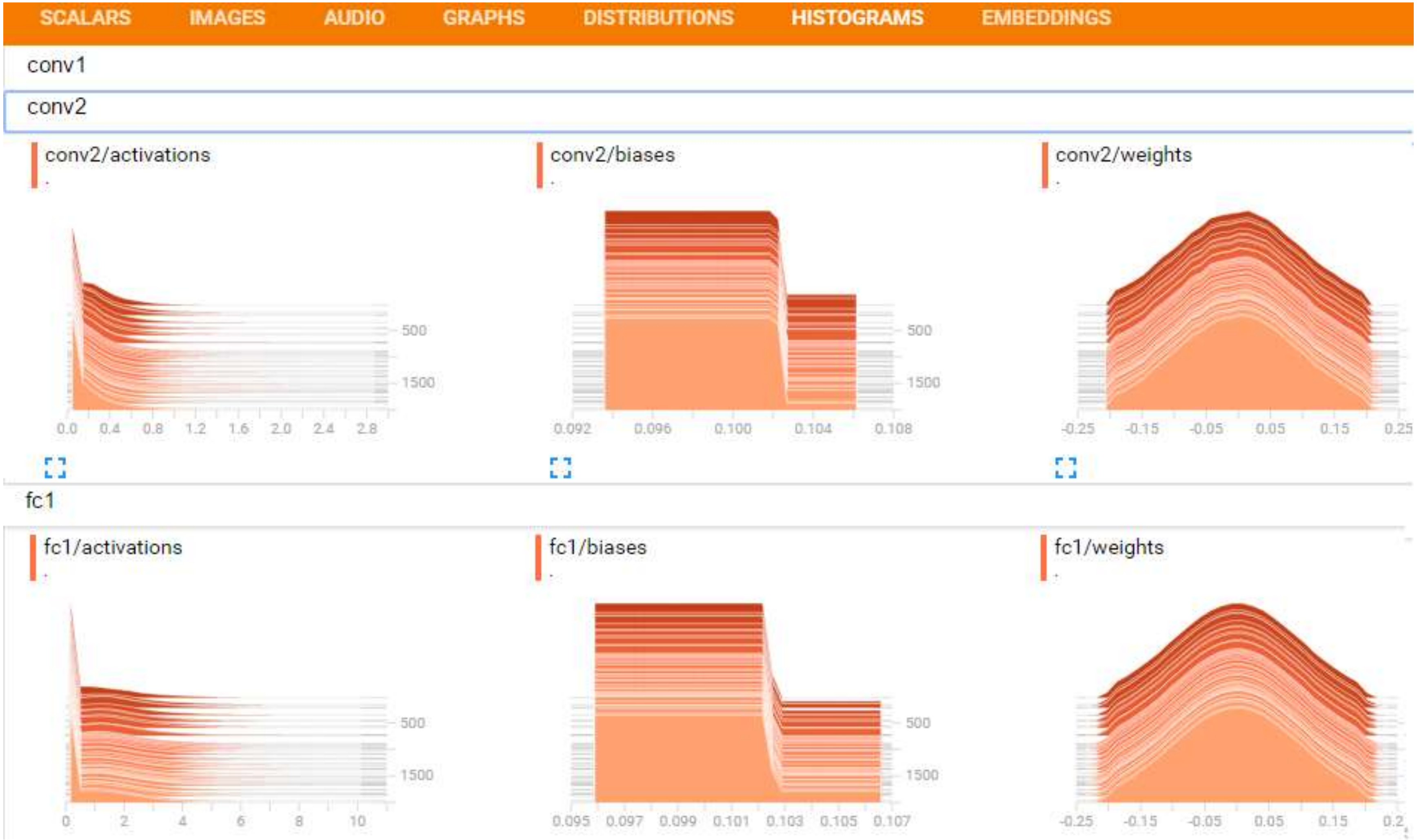
- In each layer we will add histograms showing us the weight, biases and the activations

```
def conv_layer(input, channels_in, channels_out, name="conv"):
   with tf.name_scope(name):
      w = tf.Variable(tf.zeros([5,5,channels_in, channels_out]), name="W")
      b = tf.Variable(tf.zeros([channels_out]), name="B")
      conv = tf.nn.conv2d(input, w, strides=[1, 1, 1, 1], padding="SAME")
      act = tf.nn.relu(conv + b)
      tf.summary.histogram("weights", w)
      tf.summary.histogram("biases", b)
      tf.summary.histogram("activations", act)
      return tf.nn.max_pool(act,ksize=[1,2,2,1],strides=[1,2,2,1],
padding="SAME")
```

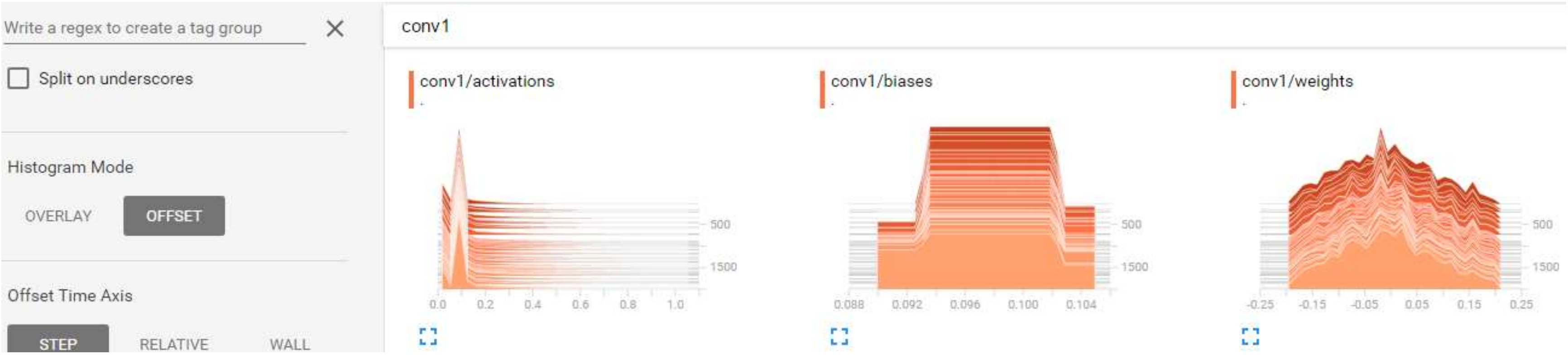
- Collecting and passing all summaries on the graph top the TensorBoard is done by command

```
merged_summary = tf.summary.merge_all()
```

Select HISTOGRAMS tab and expand



Histogram Mode: Offsets and Overlays



Write a regex to create a tag group ×

□ Split on underscores

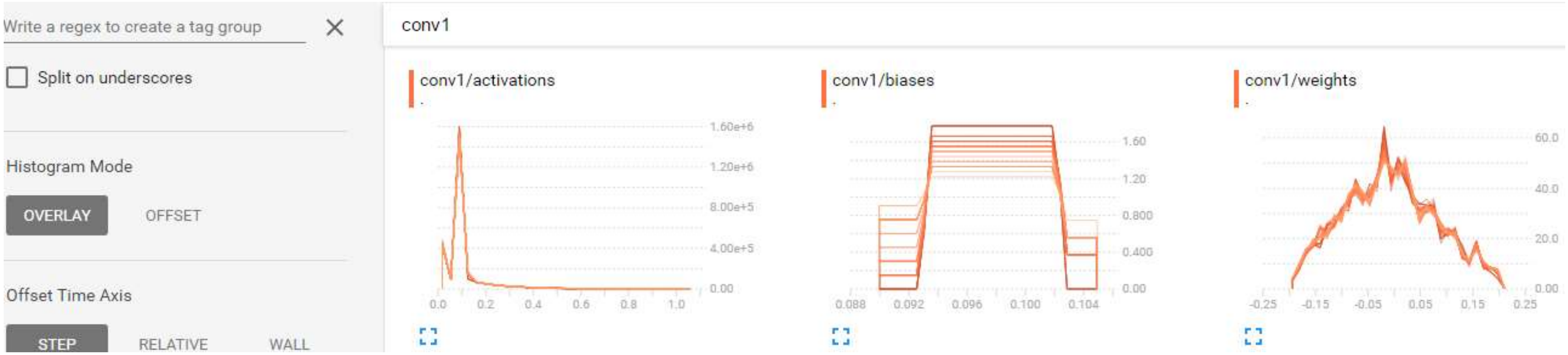
Histogram Mode

OVERLAY [✓ OFFSET]

Offset Time Axis

[✓ STEP] RELATIVE WALL

conv1



Write a regex to create a tag group ×

□ Split on underscores

Histogram Mode

[✓ OVERLAY] OFFSET

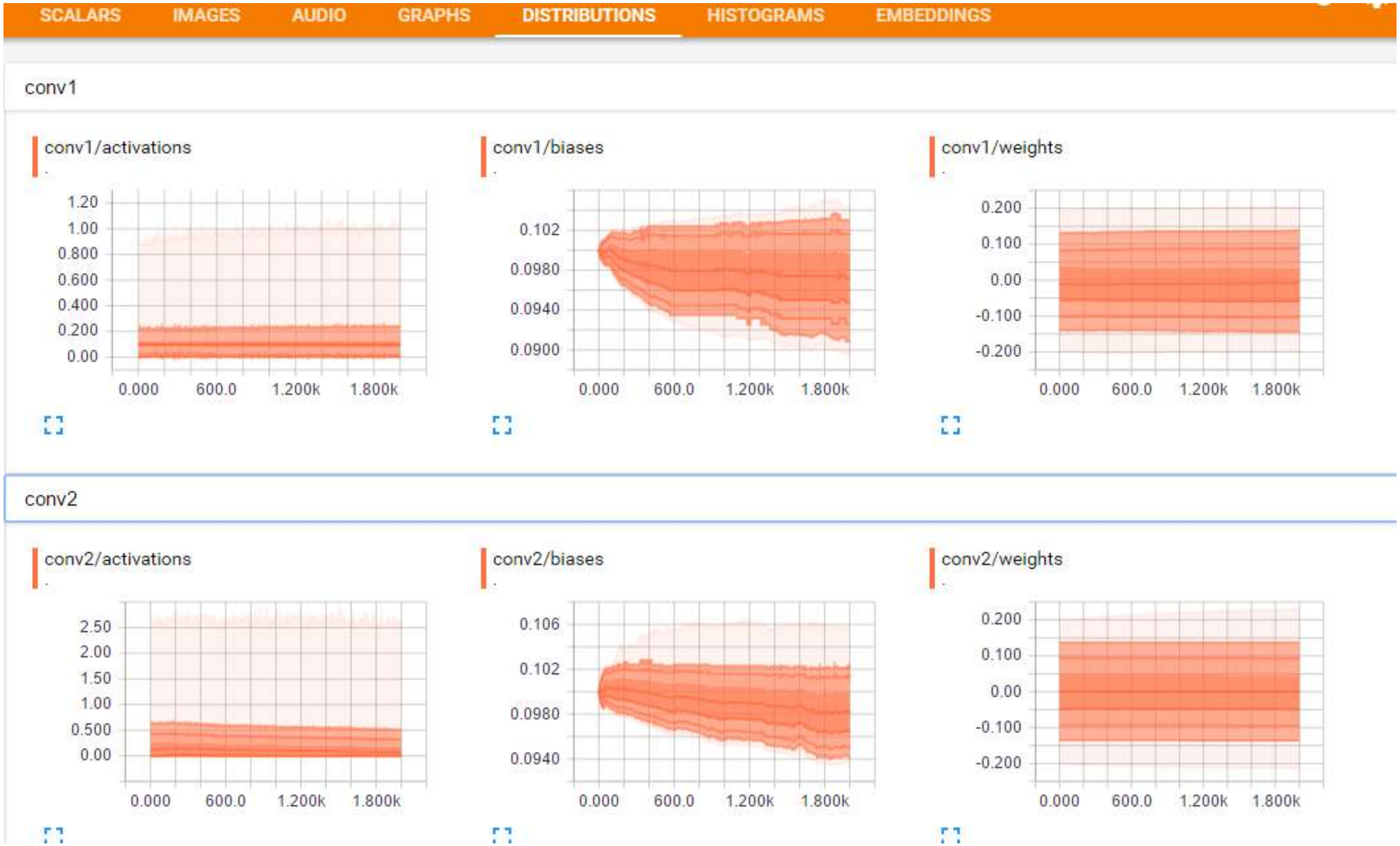
Offset Time Axis

[✓ STEP] RELATIVE WALL

conv1

Histograms as Distributions

- Data presented in Histograms can also we visualized as “distributions’. Just go to the Distributions tab and you will get slightly different view of data:



Can publish summary-s during the run

- We can push summaries out periodically, not only at the end of the run. For example our driver code could read:

```
merged_summary = tf.summary.merge_all()
writer = tf.summary.FileWriter("/log/mnist/3")
writer.add_graph(sess.graph)
for i in range(2001):
   batch = mnist.train.next_batch(100)
   if i % 500 == 0:
      s = sess.run(merged_summary, feed_dict={x: batch[0], y: batch[1]})
      writer.add_summary(s, i)
   sess.run(train_step, feed_dict={x: batch[0], y: batch[1]})
```

```
$ tensorboard --logdir /log/mnist/3
```

Experimenting with different models

- TensorFlow allows us to change our model and its parameters easily.
- For example we could readily try working with 1 rather than 2 convolution layers, or could vary the learning rate. TensorBoard would let us easily compare the results

```
# Try a few learning rates
for learning_rate in [1E-3, 1E-4, 1E-5]:
  # Try a model with fewer layers
  for use_two_fc in [True, False]:
     for use_two_conv in [True, False]:
     # Construct a hyperparameter string for each (e.g.: "lr_1E-3fc2conv2)
     hparam_str=make_hparam_string(learning_rate, use_two_fc, use_two_conv)
     writer = tf.summary.FileWriter("/log/" + hparam_str)
     # Actually run with the new settings
     mnist(learning_rate,use_two_fully_connected_layers,use_two_conv_layers,writer)
```

- Now, we will train the several model several with 3 learning rates and 4 different architectures. Every run will send its data (protobuffers with graph and summary data) to a different directory.
- We need to open TensorBoard in the parent directory, so it could see all of new runs

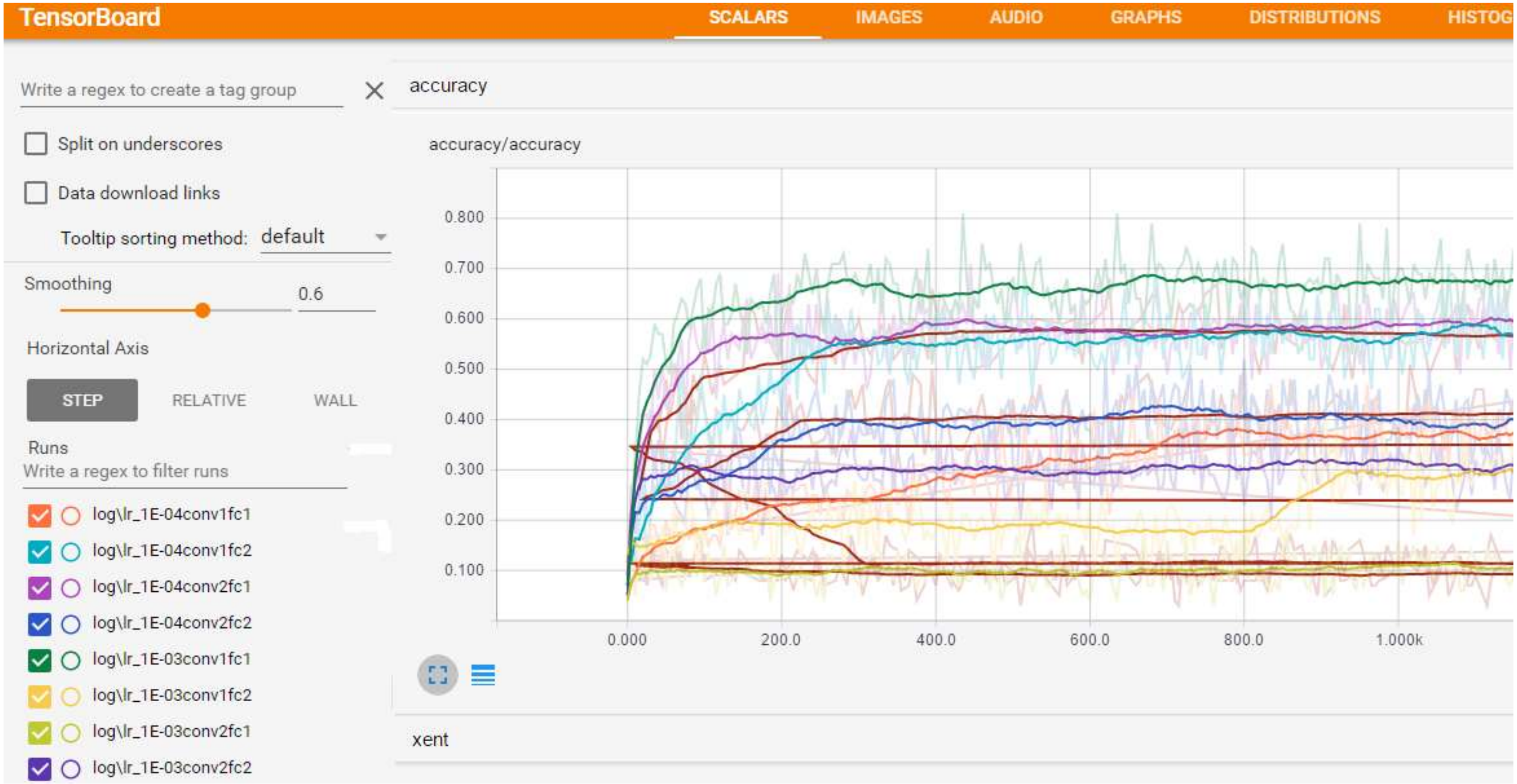
```
$ tensorboard --logdir=log
```

- If the parent directory does not work, pass comma separated list of run names and directories, like:

```
tensorboard --logdir=run01:/lr_1E-03conv1fc1, run02:/lr_1E-03conv1fc2,
run03:/lr_1E-03conv2fc1, run04:/lr_1E-03conv2fc2,….
```

Multiple Runs

- When you open TensorBoard you will see differently colored run names in the bottom left of the screen.
- Every summary now has data coming from different runs and you could make comparison.



☐ Split on underscores

☐ Data download links

Tooltip sorting method: default

Smoothing 0.6

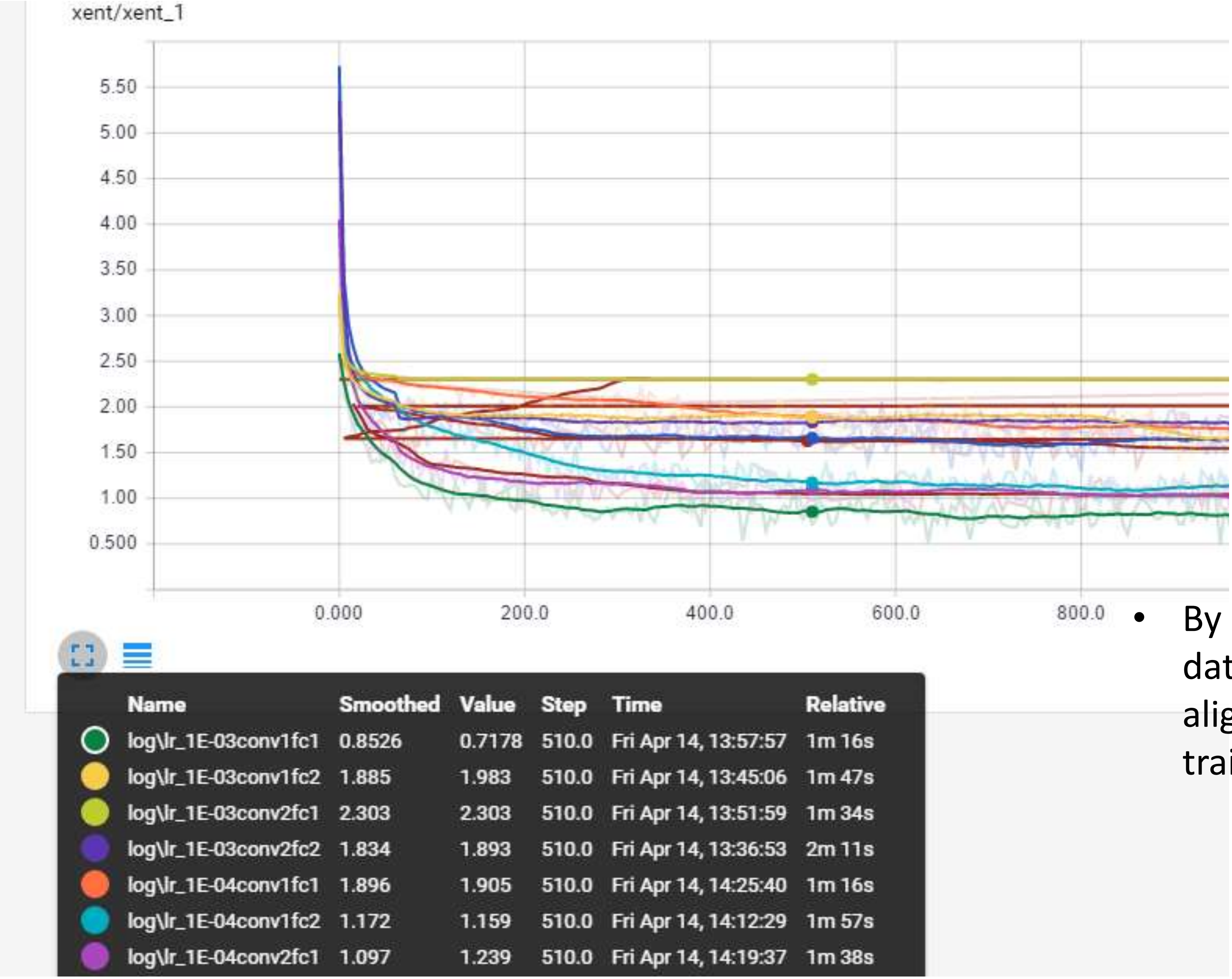
Horizontal Axis: STEP

Runs

Write a regex to filter runs

☑ ○ log\lr_1E-04conv1fc1
☑ ○ log\lr_1E-04conv1fc2
☑ ○ log\lr_1E-04conv2fc1
☑ ○ log\lr_1E-04conv2fc2
☑ ○ log\lr_1E-03conv1fc1
☑ ○ log\lr_1E-03conv1fc2
☑ ○ log\lr_1E-03conv2fc1
☑ ○ log\lr_1E-03conv2fc2

Slider Reads precise values at every run

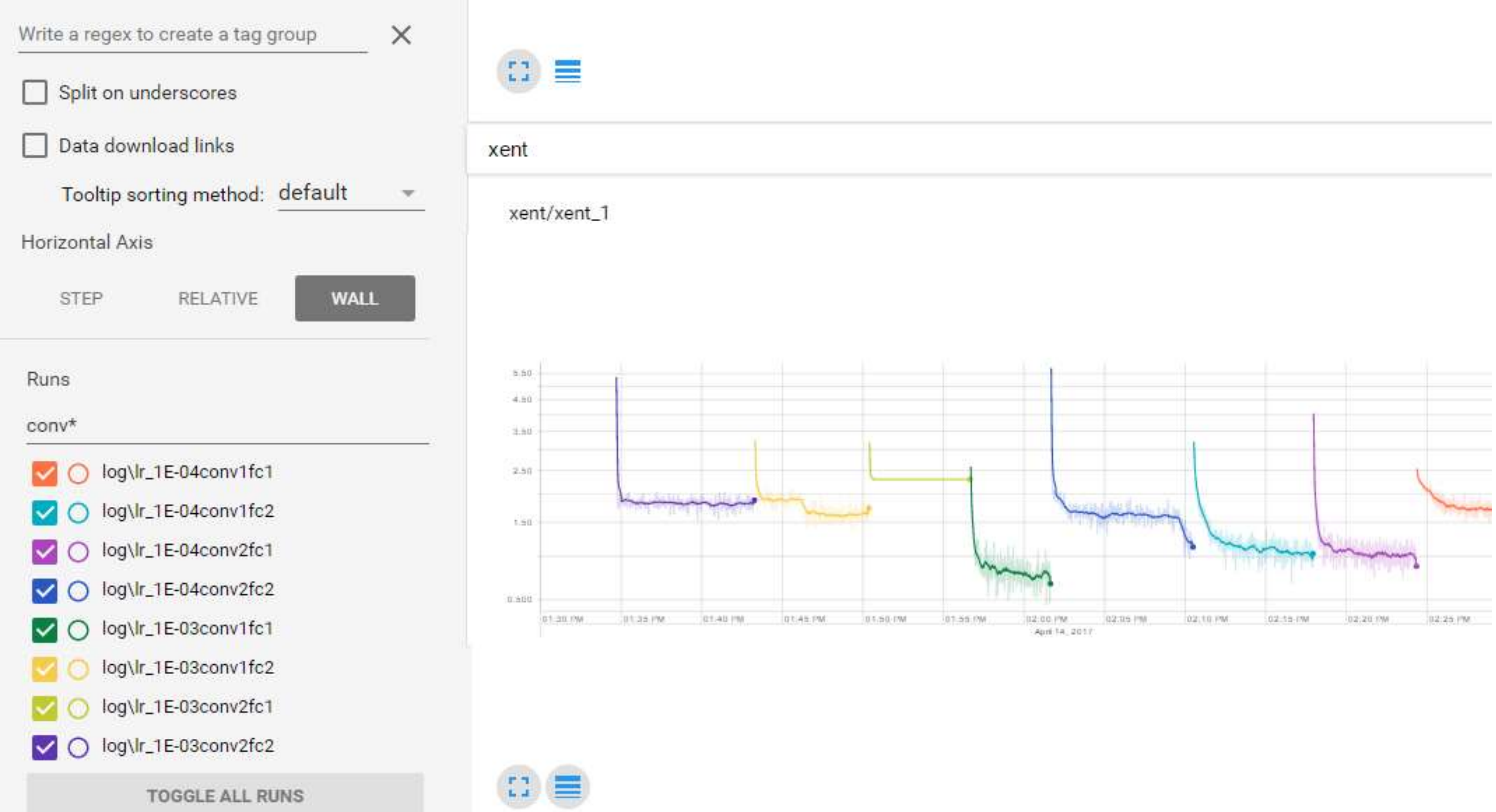


	Name	Smoothed	Value	Step	Time	Relative
○	log\lr_1E-03conv1fc1	0.8526	0.7178	510.0	Fri Apr 14, 13:57:57	1m 16s
●	log\lr_1E-03conv1fc2	1.885	1.983	510.0	Fri Apr 14, 13:45:06	1m 47s
●	log\lr_1E-03conv2fc1	2.303	2.303	510.0	Fri Apr 14, 13:51:59	1m 34s
●	log\lr_1E-03conv2fc2	1.834	1.893	510.0	Fri Apr 14, 13:36:53	2m 11s
●	log\lr_1E-04conv1fc1	1.896	1.905	510.0	Fri Apr 14, 14:25:40	1m 16s
●	log\lr_1E-04conv1fc2	1.172	1.159	510.0	Fri Apr 14, 14:12:29	1m 57s
●	log\lr_1E-04conv2fc1	1.097	1.239	510.0	Fri Apr 14, 14:19:37	1m 38s

- By default data are aligned on training steps

Data by the Wall Time

- You can select runs by the regular expression in the filed below Runs.
- You can see the by the real (wall) time



Write a regex to create a tag group

□ Split on underscores

□ Data download links

Tooltip sorting method: default

Horizontal Axis

STEP RELATIVE **WALL**

Runs

conv*

☑ ○ log\lr_1E-04conv1fc1

☑ ○ log\lr_1E-04conv1fc2

☑ ○ log\lr_1E-04conv2fc1

☑ ○ log\lr_1E-04conv2fc2

☑ ○ log\lr_1E-03conv1fc1

☑ ○ log\lr_1E-03conv1fc2

☑ ○ log\lr_1E-03conv2fc1

☑ ○ log\lr_1E-03conv2fc2

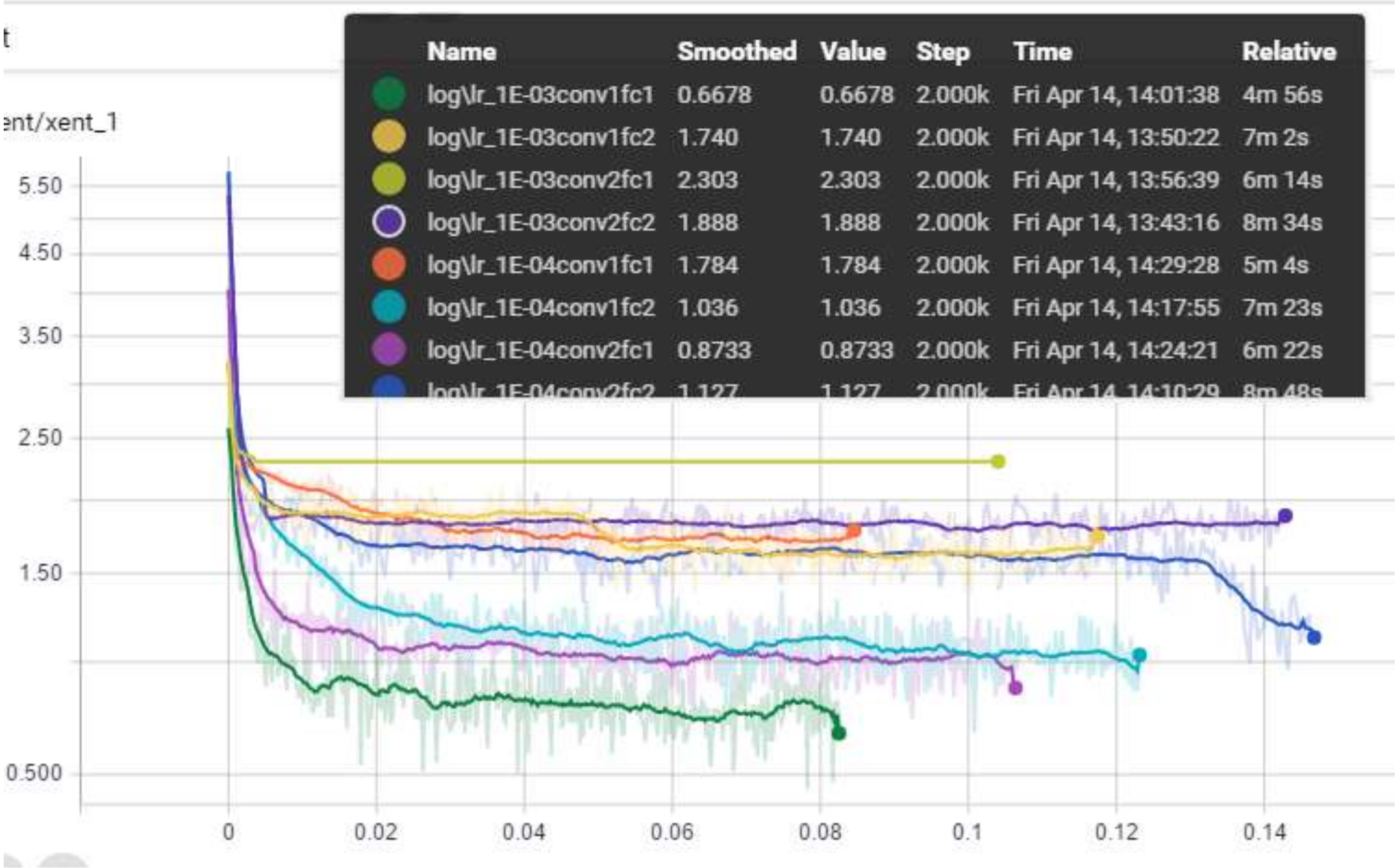
TOGGLE ALL RUNS

xent

xent/xent_1

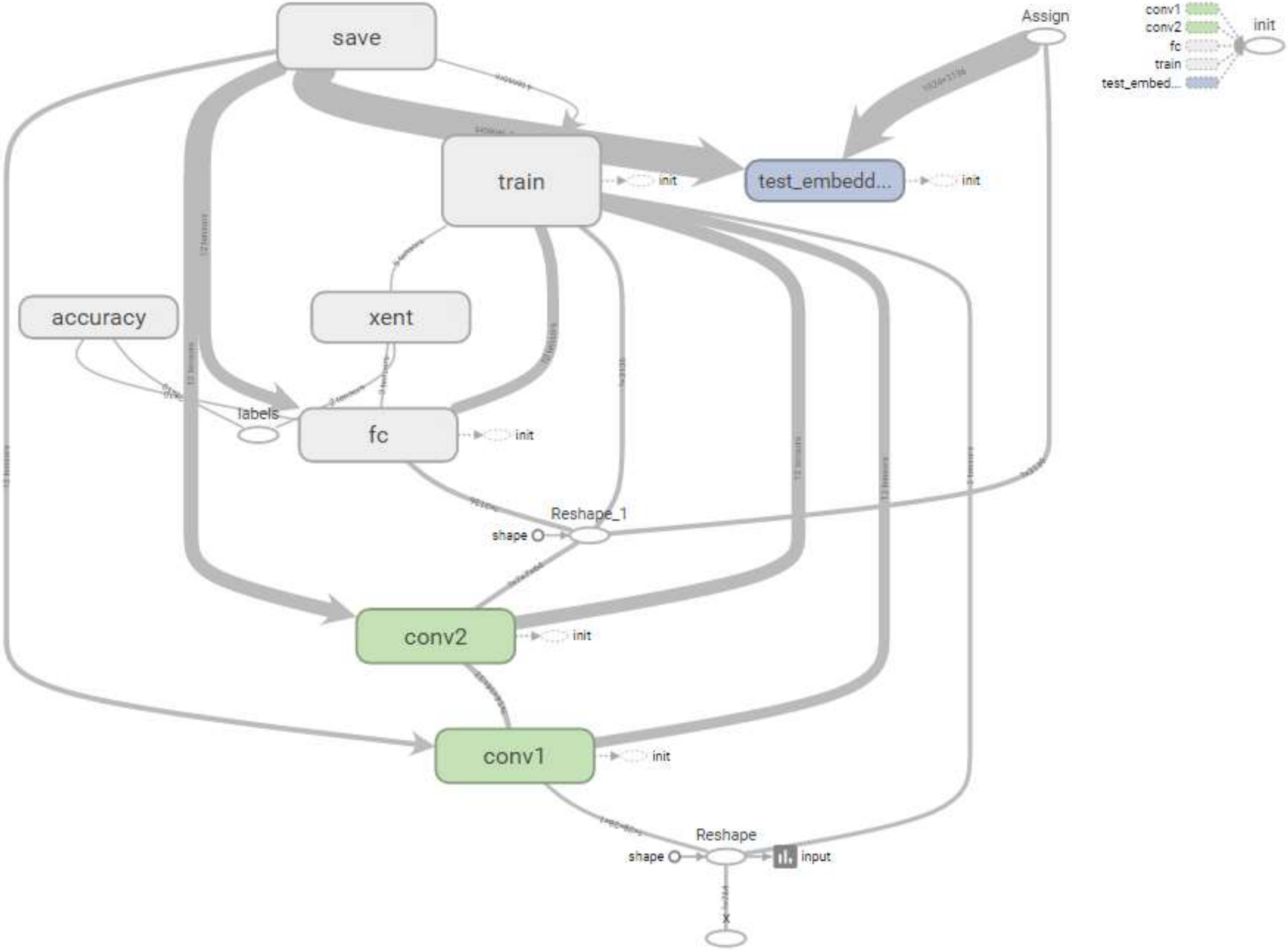
Horizontal Axis: RELATIVE

- If you select RELATIVE value for the horizontal axis you will see that some runs last longer than the others.



	Name	Smoothed	Value	Step	Time	Relative
●	log\lr_1E-03conv1fc1	0.6678	0.6678	2.000k	Fri Apr 14, 14:01:38	4m 56s
●	log\lr_1E-03conv1fc2	1.740	1.740	2.000k	Fri Apr 14, 13:50:22	7m 2s
●	log\lr_1E-03conv2fc1	2.303	2.303	2.000k	Fri Apr 14, 13:56:39	6m 14s
○	log\lr_1E-03conv2fc2	1.888	1.888	2.000k	Fri Apr 14, 13:43:16	8m 34s
●	log\lr_1E-04conv1fc1	1.784	1.784	2.000k	Fri Apr 14, 14:29:28	5m 4s
●	log\lr_1E-04conv1fc2	1.036	1.036	2.000k	Fri Apr 14, 14:17:55	7m 23s
●	log\lr_1E-04conv2fc1	0.8733	0.8733	2.000k	Fri Apr 14, 14:24:21	6m 22s
●	log\lr_1E-04conv2fc2	1.127	1.127	2.000k	Fri Apr 14, 14:10:29	8m 48s

Can Compare Main Graphs



Embedding Visualizer

- TensorBoard can take our many dimensional data and project them into three dimensions. For those of you who are more familiar with ML techniques, TB performs the PCA (Principal Component Analysis). It finds three most dominant eigen vectors in the space of our data and project all data points onto those three vectors. By doing that the dimension of the dataset is reduced from an arbitrarely large value to 3, which we can visualize.
- Code that accomplishes the creation of that visualizer follows.

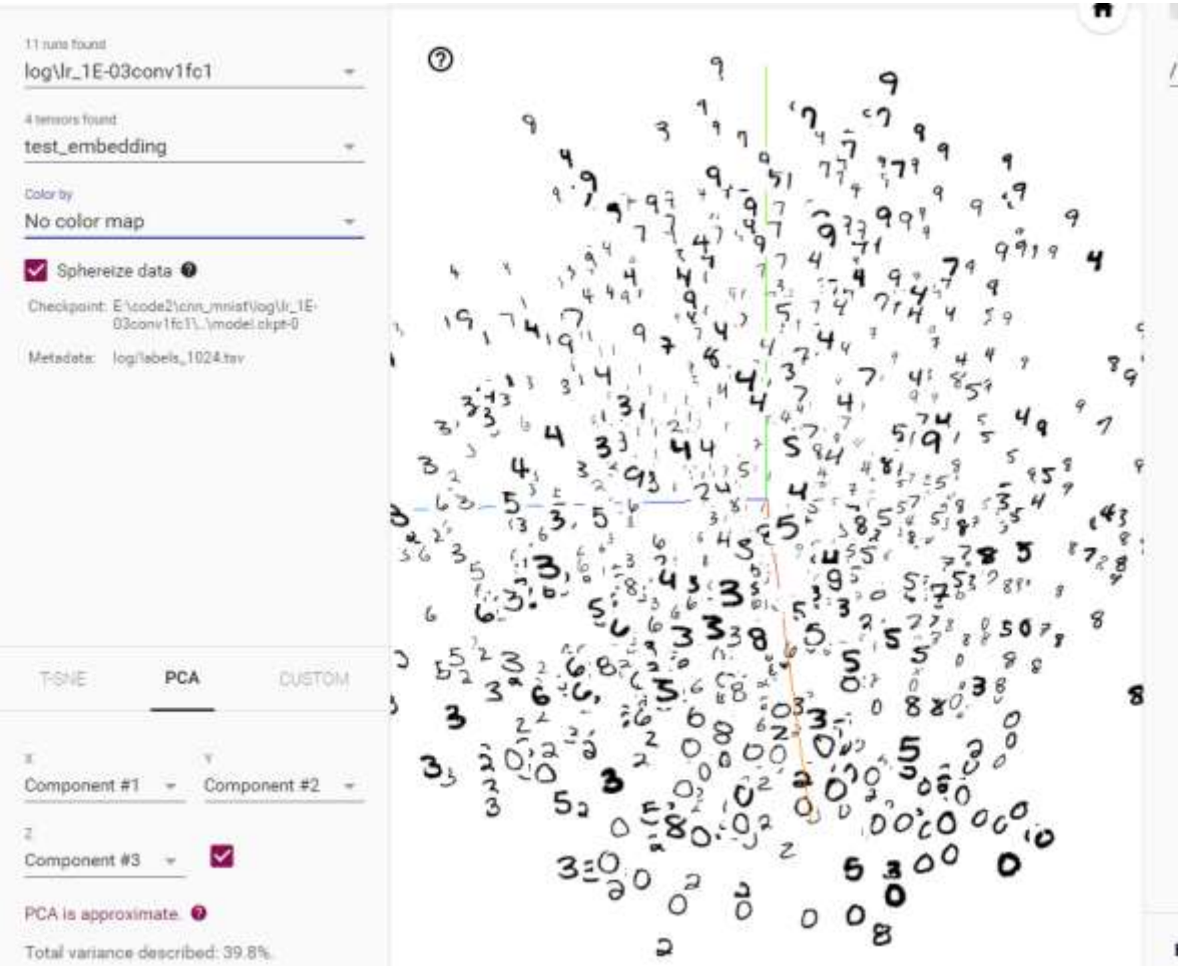
```
embedding = tf.Variable(tf.zeros([10000, embedding_size]), name="test_embedding")
assignment = embedding.assign(embedding_input)
config = tf.contrib.tensorboard.plugins.projector.ProjectorConfig()
embedding_config = config.embeddings.add()
embedding_config.tensor_name = embedding.name
embedding_config.sprite.image_path = os.path.join(LOG_DIR, 'sprite.png')
embedding_config.metadata_path = LOGDIR + 'labels_1024.tsv'
# Specify the width and height of a single thumbnail.
embedding_config.sprite.single_image_dim.extend([28, 28])
tf.contrib.tensorboard.plugins.projector.visualize_embeddings(writer, config)
for i in range(2001):
   batch = mnist.train.next_batch(100)
   if i % 5 == 0:
     [train_accuracy, s]=sess.run([accuracy,summ],feed_dict={x:batch[0],y: batch[1]})
     writer.add_summary(s, i)
   if i % 500 == 0:
      sess.run(assignment,feed_dict={x:mnist.test.images,y_true:mnist.test.labels})
      saver.save(sess, os.path.join(LOG_DIR, "model.ckpt"), i)
   sess.run(train_step, feed_dict={x: batch[0], y_true: batch[1]})
```

t-SNE

- **t-distributed stochastic neighbor embedding (t-SNE)** is a machine learning algorithm for dimensionality reduction developed by Geoffrey Hinton and Laurens van der Maaten.
- t-SNE is a nonlinear dimensionality reduction technique that is particularly well-suited for embedding high-dimensional data into a space of two or three dimensions, which can then be visualized in a scatter plot.
- Specifically, it models each high-dimensional object by a two- or three-dimensional point in such a way that similar objects are modeled by nearby points and dissimilar objects are modeled by distant points.
- https://www.youtube.com/watch?v=NEaUSP4YerM
- https://en.wikipedia.org/wiki/T-distributed_stochastic_neighbor_embedding

Embedding is Representation of Data in Space of Weights

- TensorFlow (TensorBoard) is tracing representing each initial data object to its logits, i.e. representation before the final layer and then performing PCA on those values. If you go to EMBEDDING tab, you will see projections of every data point:



11 runs found

log\lr_1E-03conv1fc1

4 tensors found

test_embedding

Color by

No color map

☑ Sphereize data

Checkpoint: E:\code2\cnn_mnist\log\lr_1E-03conv1fc1\..\model.ckpt-0

Metadata: log/labels_1024.tsv

X: Component #1

Y: Component #2

Z: Component #3 ☑

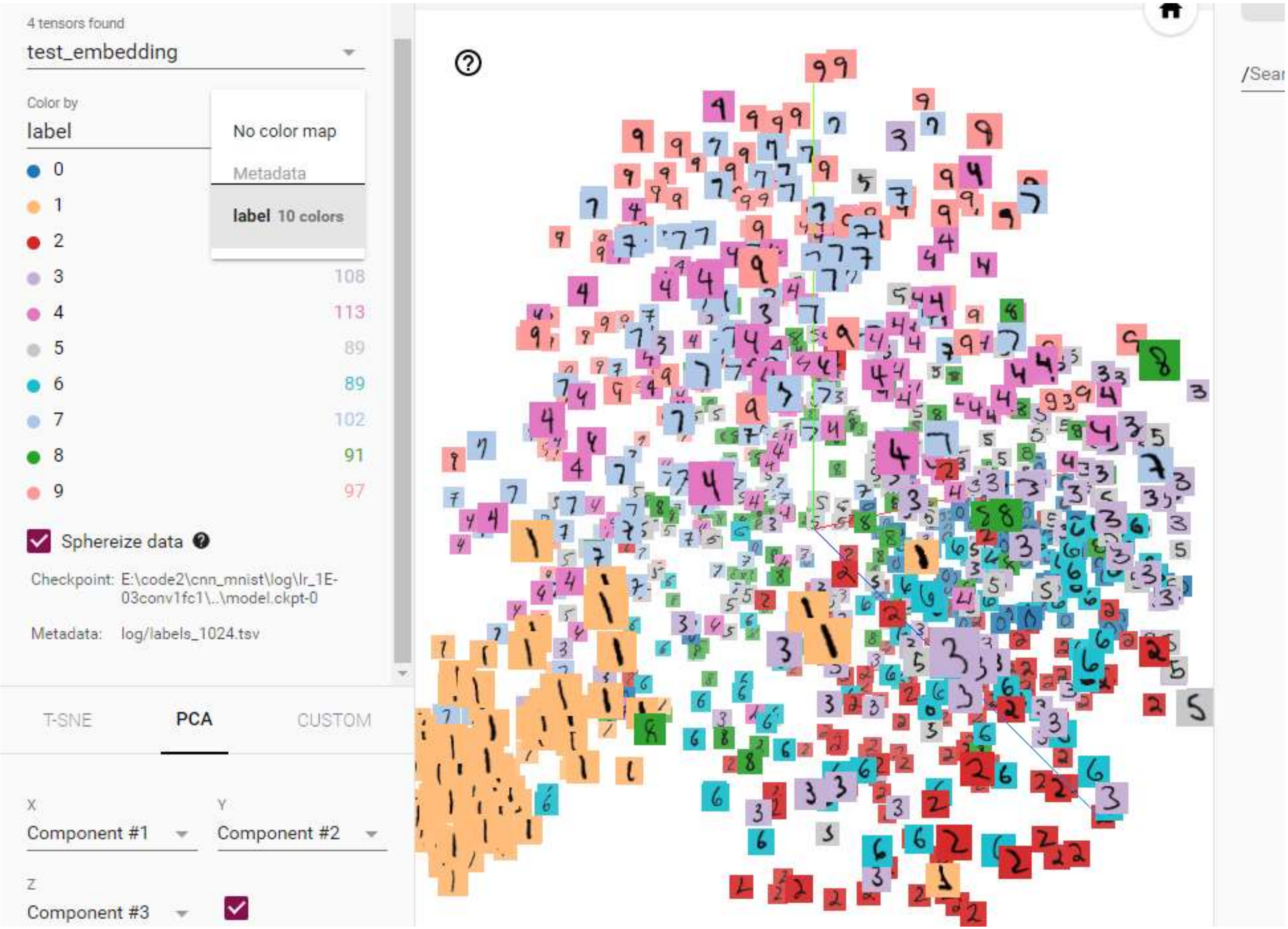
PCA is approximate.

Total variance described: 39.8%

- These are 1024 random points from MNIST. We passed them by a file:

`labels_1024.tsv`

Introduce Color Labels



4 tensors found

test_embedding

Color by

label

● 0

● 1

● 2

● 3 108

● 4 113

● 5 89

● 6 89

● 7 102

● 8 91

● 9 97

☑ Sphereize data

Checkpoint: E:\code2\cnn_mnist\log\lr_1E-03conv1fc1\..\model.ckpt-0

Metadata: log/labels_1024.tsv

T-SNE PCA CUSTOM

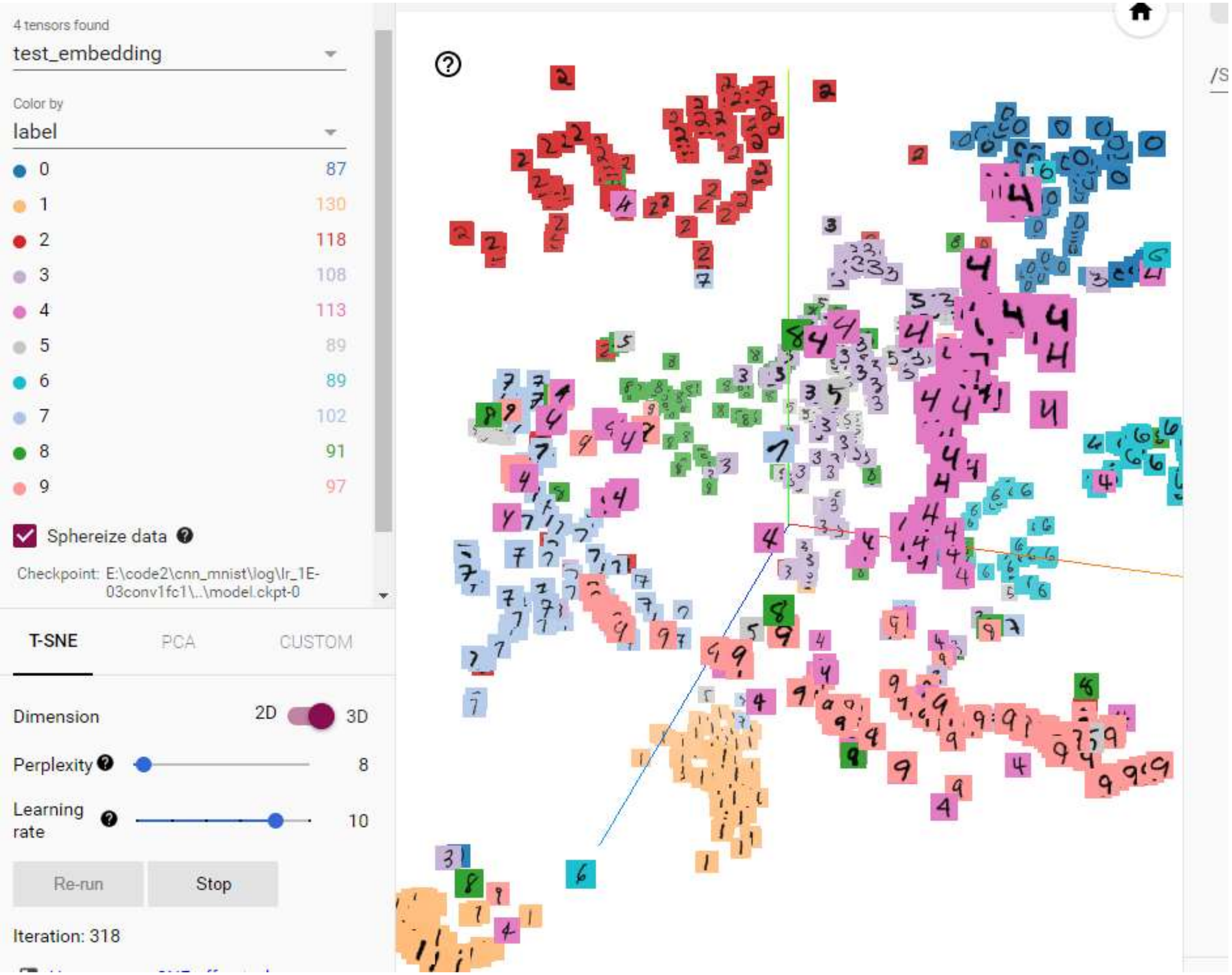
X Component #1

Y Component #2

Z Component #3 ☑

T-SNE Local Organization

- T-SNE feature organizes data by local similarity.



4 tensors found

test_embedding

Color by

label

● 0	87
● 1	130
● 2	118
● 3	108
● 4	113
● 5	89
● 6	89
● 7	102
● 8	91
● 9	97

☑ Sphereize data

Checkpoint: E:\code2\cnn_mnist\log\lr_1E-03conv1fc1\..\model.ckpt-0

Dimension 2D 3D

Perplexity 8

Learning rate 10

Iteration: 318