deepViz: Visualizing Convolutional Neural Networks for Image Classification

Daniel Bruckner Joshua Rosen Evan R. Sparks

UC Berkeley
{bruckner, joshrosen, sparks}@cs.berkeley.edu

Abstract

Deep convolutional neural networks have recently shown state of the art performance on image classification problems. However, their inner workings remain a mystery to machine learning experts, particularly when compared to better studied and less complex algorithms such as SVM and Logistic Regression. As a result, constructing and debugging effective convolutional neural networks is time-consuming and error-prone, as it often involves a substantial amount of trial and error. We introduce deepViz, a system designed to allow experts to understand their models and diagnose issues with the model structure, enabling more rapid iteration during the model construction process and faster convergence to a suitable model for the task at hand.

1 Introduction

Deep learning, or the use of deep (i.e., many-layered) convolutional neural networks for machine recognition and classification, is advancing the limits of performance in domains as varied as computer vision, speech, and text ([Zeiler and Fergus 2013], [Dean et al. 2012]). Improvements in both hardware and software performance have enabled the development of larger networks that have achieved record results ([Krizhevsky et al. 2012]). The promise of deep learning is to automate feature engineering, a task that otherwise requires application of both domain expertise and machine learning expertise. Neural networks offer a coherent framework to train multifaceted models that compose featurization and classification components in a unified pipeline.

But to architect a deep network is not trivial. There are three primary challenges. First, because convolutional networks compose many functional components—whose values as individuals and as a whole are not well understood ([Jarrett et al. 2009])—they are difficult to design. Second, each component of a network may have dozens of hyper-parameters associated with it, all of which must be tuned for peak performance. And finally, the complexity of neural networks has protected them from the rigorous formalism of other fields of machine learning, so practitioners can only rely on anecdotal results to guide design. An intimate community has focused on convolutional networks, so few know of good design heuristics. Moreover, the majority of this community is now employed by Google and Facebook. Given these challenges, how can a relative novice to field learn to obtain results on par with the field's leading experts?

We propose visualization as a means to make high-end deep learning approachable. Convolutional networks are not a black box: their internals (filters, namely) can be visualized in a natural way as bitmaps. What is more, because neural networks are composed of layers, intermediate representations of transformed data can be read out after each layer. When a network is applied to image processing, these intermediate forms are themselves images, and can be visualized as such. That is, it is possible to see what the model sees at different stages of computation. As shown by [Zeiler et al. 2011] and [Vondrick et al. 2013] visualization of these reconstructed intermediate states can be an aid to model tuning and debugging.

Our system, deepViz, is an interactive visualization system for deep learning. Given a trained convolutional model and an image corpus, deepViz offers displays and interactions that empower a user to explore their model and its relationship to the data.

deepViz targets an iterative workflow for development and refinement of convolutional networks. After a user chooses an architecture and parameters for a network and then trains it against some data set, they can use deepViz to observe snapshot visualizations of the model at different time steps during the training process. These snapshots give insight into the model and into properties of the training process itself (e.g., rates of convergence, and correlated structure across time and space) that allow the user to diagnose shortcomings and highlight strengths of the current model. These observations lead to revisions to the model architecture and further rounds of training, visualization, and evaluation.

The deepViz interface provides access to several classes of visualization. Users can view bitmap representations of filter banks in convolutional layers and of weight matrices in fully connected layers. For all types of layers, a user may select an image from the input data set and view the output that image induces on the layer. These featurized representations allow a deeper understanding of the function of particular layers and filters, and can be used to spot sources of confusion in the model.

The system also gives users views of the model's final output. The first of these is an interactive confusion matrix that plots the number of images by true class and predicted class, and allows the user to view example images in each category. Another visualization shows the results of a clustering algorithm run on the featurized output of the model for each image. This view is intended to highlight subclasses of images, for example, the class of airplanes may split into a cluster of images of planes in profile and another cluster of images shots head on (or from above or below). Understanding how the model partitions the input data (or fails to partition) is another aid to diagnosis.

All of the visualizations in deepViz are driven by a timeline system. As mentioned above, each display represents a snapshot of the model at a particular point in its training. An interactive slider input allows users to control which snapshot to show, or the view can be displayed as an animation spanning the entire training process. Animations give the user a sense of how components of the model evolve during training. Users can stop and start animation at any point with a button click.

The remainder of this paper is organized as follows. Section 2 discusses related work, and Section 3 gives a brief introduction to convolutional neural networks and how the are designed and trained. The deepViz system architecture is described in Section 4, and the key features, visualizations, and interactions of the system are presented in Section 5. Section 6 describes directions for future work and Section 7 concludes.

2 Related Work

For several decades, convolutional neural networks have been applied to problems in computer vision. In the nineties, such architectures achieved breakthrough performance in applications like handwriting detection and face recognition. LeNet [LeCun et al. 1998], a system for handwritten digit recognition, is among the first to achieve near-human accuracy. LeNet notably includes an interactive visualization system that displays featurized versions of input images as well as predictions. The visualizations allow direct and compelling demonstration of important properties of the system like invariance to translations and deformations of the input. While LeNet's visualizations provide evidence for the system's merits—and add to its merits, since an inspectable model is better than a black box—they do not serve as design aids to practitioners.

Increases in computing resources have allowed deeper and more complex convolutional to be trained. [Krizhevsky et al. 2012] achieved record results in the ImageNet Large Scale Visual Recognition Challenge [ImageNet 2013] with an eight layer network trained on two GPUs over twelve days. For our project, we have used open source code released by [Krizhevsky 2012] to train large models. [Donahue et al. 2013] introduced `decaf`, a system written in Python and developed here at Berkeley, to make convolutional network processing more approachable. `deepViz` uses `decaf` extensively for interactive image featurization and for offline statistical computations.

Much recent work explores the growing design space of convolutional networks. [Jarrett et al. 2009] evaluate architectural variations of different hand-designed networks on several data sets. Others, like [Yamins et al. 2013], use Bayesian methods to automatically search the parameter space of convolutional networks. `deepViz` intends to supplement such efforts by helping users develop heuristics to guide search and evaluation in this increasingly complex space.

Beyond neural networks, visualization has been used in computer vision more generally as a tool to aid in feature evaluation. [Vondrick et al. 2013] argue for the necessity of visual inspection of image features to understand models' failures. They use feature inversion algorithms, whereby an image is featurized and then recovered to a transformed but basically intelligible format, to give intuitive access to abstract feature representations.

Feature inversion has been applied to convolutional neural networks to obtain several interesting results. [Le et al. 2011] perform inverse optimization on a network trained by unsupervised learning to construct the optimal inputs for particular neurons. In particular, they find single deep neurons trained to respond to faces (both human and feline) and bodies. This year, [Zeiler and Fergus 2013] use a type of feature inversion called deconvolution to render re-weighted versions of input that highlight the areas, patterns, and textures of an image deemed most important by a particular part of the network. These re-weighted images are both accessible and informative, and the authors used insights from these images to refine their network design to achieve state-of-the-art performance. In the future, we plan to add forms of feature inversion to `deepViz`.

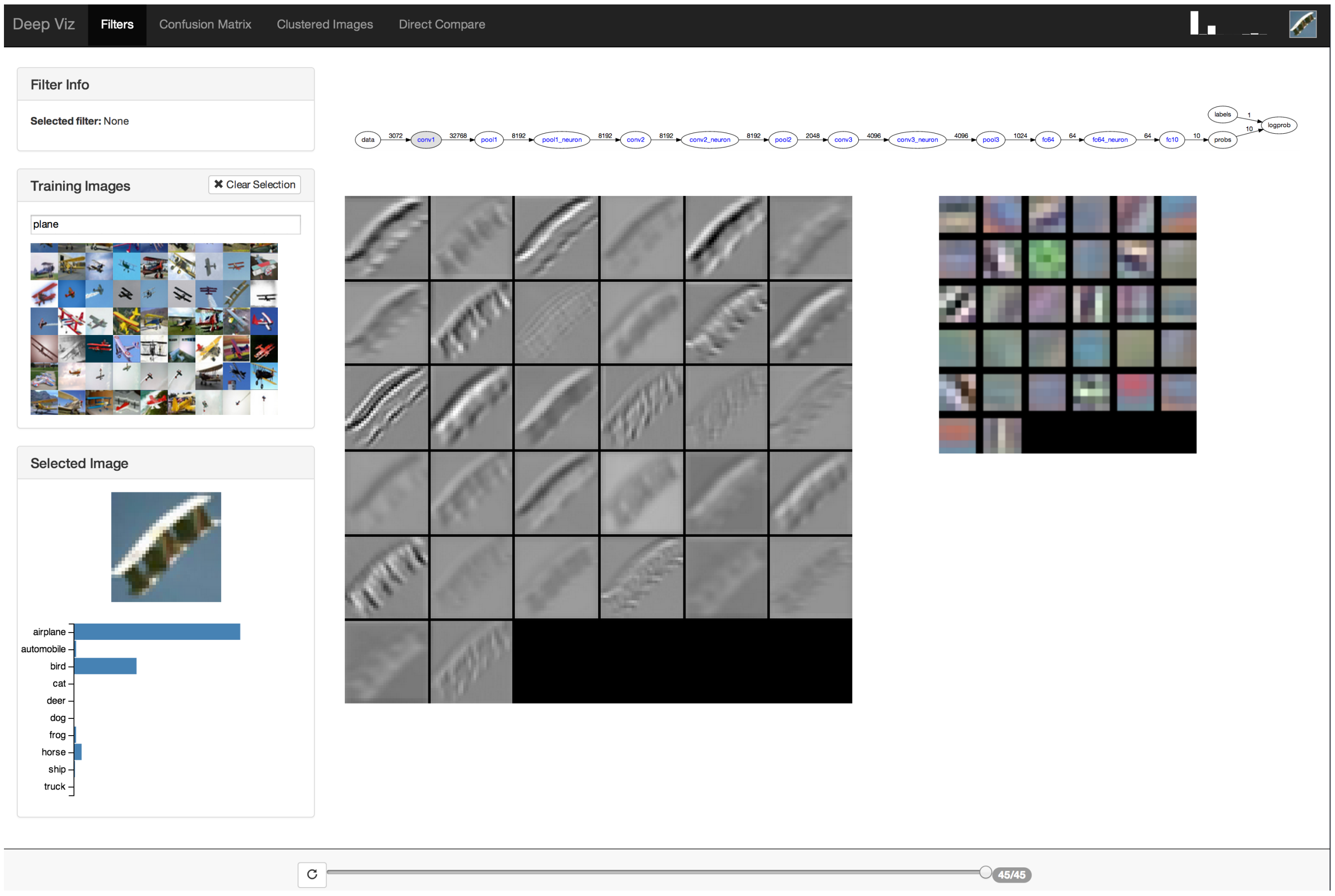


Figure 1: *The* `deepViz` *interface*

3 Background

We begin by describing the structure of deep convolutional neural networks and provide a rough overview of the design space that the architect of such a network must consider explore when constructing it. After establishing this architecture, we describe the conventional training process used to converge on an appropriate model for a particular problem. While these networks can be trained for unsupervised learning tasks, we focus on the supervised case since our target problem (image classification) falls into that family.

3.1 Convolutional Neural Networks

In general, an artificial neural network consists of a succession of layers of so-called *neurons*. A neuron computes a function on inputs from the preceding layer and passes the result, sometimes called the neuron's *activation*, to outputs in the succeeding layer. Within each layer, all neurons compute the same function, but individual neurons may have distinct sets of inputs and outputs and may assign different weights to their inputs. Different types of layers are defined by the number and pattern of connections between neurons. In a *fully connected* layer, the neurons receive input from every output in the preceding layer. In a *locally connected* layer, the neurons are indexed spatially, and each only takes input from nearby outputs. A *convolutional layer* is a type of locally connected layer where the weights that each neuron applies to its inputs are shared in a particular way. We describe convolution in greater detail below.

A neural network architecture for image classification combines a diversity of functions and connectivity structures using several layers. The first layers are convolutional and produce a featurized representation of an image. Afterwards, a non-linear transformation is often applied, followed by a linear classifier such as logistic regression or SVM. The output of the network (usually a vector of predicted probabilities) can be assessed relative to a true image label, and the result can be used by an optimization algorithm like gradient descent to train the network. The great appeal of neural networks is that training can be applied to the featurization layers as well as the classifier. This end-to-end training algorithm, called *back-propagation*, is the current state-of-the-art in image classification and other domains such as speech recognition [Dean et al. 2012]. Typically, training is an iterative process that involves multiple passes of the input data until the model converges.

3.1.1 Convolution

The convolution of an image is produced by applying a filter to image, and produces a new image. A filter is a $k \times k$ weight-matrix where k is an odd number (so that the matrix has unique center). Pixels in the convolved image are produced by placing the filter on top of the image, with its center aligned at the corresponding input pixel, and computing the dot product of the filter with the pixels below it. The convolution can be imagined as the result of moving a filter across the image that replaces each pixel with some function of its neighborhood. This process is illustrated in Figure 2.[Wikipedia 2013]

In the context of neural networks, a convolutional layer applies many filters to its input to generate a *feature map*, which is essentially a stack of convolved images, or equivalently, one convolved image with an arbitrary number of channels per pixel. In addition, convolutional layers are often bundled with several auxiliary layers that apply a fixed transformation to the convolved feature map. These auxiliary layers include **normalization** (of pixel values within a neighborhood), **pooling** (aggregation of small patches of pixels, for example, by averaging or taking the maximum pixel value), **down-sampling**, and the application of various **non-linear functions** to pixel values.

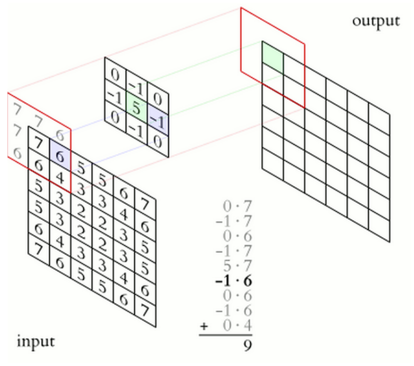


Figure 2: *Image Convolution.*

So far, our discussion of convolution has tended to the abstract, and the reader would be justified to ask "what's the point?" In fact, convolutions are capable of transforming images in many useful and concrete ways, like emphasizing edges and computing gradients of hue and value. Moreover, deep successions of convolutions have been shown to produce image encodings that are favorable for classification, namely due to invariance to translation and deformation [Bruna and Mallat 2012]. But exactly what is computed—and its usefulness for classification—depends on the filters used, and therefore success of a convolutional network crucially depends crucially on choosing good filters.

3.2 Design Space

Recent success in image classification has come from going deeper: using more filters in more layers. Back-propagation automates the training of the filter weights in these deep networks, and with larger data sets, like ImageNet [ImageNet 2013], deeper and richer models can be trained. But the ease of training deep networks belies the difficulty of their design.

While we have presented a basic overview of the workings of a convolutional neural network, we have glossed over several details of the network structure that represent design points that model's builder must consider.

Particular parameters that must be tuned include:

- Size of filters - Determining an appropriate size for convolutional filters is not a precise science. Too small and the features are in some sense "too coarse", too large and model complexity explodes with little benefit.
- Number of layers - Additional layers seems to improve model performance, but they increase model complexity, and too many layers may cause the signal-to-noise ratio during back-propagation to be too low for the first few layers to be trained into anything useful.
- Filters per layer - Again, models generally perform better with more filters, but at what point are diminishing returns outweighed by the increased model complexity and training time?
- Layer connectivity - Besides convolutional layers, what other types of layers should be used? Some top results have mixed fully-connected and locally-connected layers with convolutional ones to great effect [Krizhevsky et al. 2012].

- Initialization - Should we initialize our weights uniformly, randomly, or to some structure? Does it make a difference?
- Auxiliary layers - The choice of pooling and normalization function can have a significant impact on model accuracy, and each comes bundled with several numeric parameters. How do you tune them?
- Non-linear functions - Surprisingly, the choice of what non-linearity to apply after a convolution can have dramatic impact on training run-time performance. Indeed, [Krizhevsky et al. 2012] note that the use of the "relu" non-linearity instead of the sigmoid function makes a large difference in their models' performance.
- Optimization parameters - As with any ML model, learning parameters like step size and regularization must be tuned to maximize accuracy and convergence speed. Algorithms like AdaGrad [Duchi et al. 2011] are often used to manage some of these parameters, functional dependencies between parameters can make tuning difficult.

All of these parameters can have a dramatic impact on model performance and complexity. By offering visual tools to explore the effects of these design decisions, `deepViz` enables users to explore the design space without "shooting in the dark" or trying all permutations of these possible choices.

4 Architecture

Our system consists of several components that work together to drive the interactive visual display. We make use of several external libraries - in particular, `cudaconvnet`[Krizhevsky 2012] and `decaf`[Donahue et al. 2013] provide the basic framework for loading, training, and displaying components of convolutional neural networks. Python's [NumPy] and [scikit.learn] libraries provide tools for linear algebra, multidimensional array manipulation, and clustering. We make heavy use of [jQuery], [Bootsrap], [D3], and [Vega] to produce interactive data displays.

4.1 Model Training and Snapshotting

Before we can visualize a model, we must train it. One of our goals is to understand how models change over the course of their training process, so we capture snapshots of the state of the model at several points during training.

We train our models using `cudaconvnet`, a GPU-accelerated library for training CNNs. This tool provides an excellent platform for training convolutional models with fairly general architectures.

To collect the data required for our visualizations, we instrumented `cudaconvnet` to dump complete snapshots of the model to disk several times during training.

To ensure that our visualizations show the current state of the art in image classification networks, we trained our models according to [Krizhevsky 2012] using the CIFAR-10 dataset.

4.2 Model Queries and Visualization Generation

Our system's data model for convolutional neural networks consists of four basic components: checkpoints, layers, filters, and channels. Models have identical structure across checkpoints, although the learned weights may vary from checkpoint to checkpoint. Each layer has a different number of channels and filters associated with it.

Internally, our system allows the user to specify either a single point in this space or a set of points (often defined by a *range*). If a range is not specified, all points along a particular axis are returned by default. This means that if a user asks for 'checkpoint 1', then all layers, filters, and channels of that checkpoint are returned as a single object. If a user selects a single layer and checkpoint, then only a single image is returned by the back end. If several are selected, then the result is displayed as a collection of images.

For performance, each layer is rendered as a single image. Instead of dealing with thousands of tiny images, the browser only needs to deal with a few large ones. In our description of the front end, we will discuss how we enable interaction with individual components each of these images.

Optionally, a user may ask to *apply* a selected subset of the model to an input image. In this case, the query semantics are identical, except that a user must specify which image from the image corpus to apply the model to. Our system then (using `decaf`) applies the model to the input image, and retrieves the output activations of the image at each selected layer.

4.2.1 Image Corpus

We provide the user the ability to specify an image corpus that can be used to provide visual examples of model behavior, as above. This corpus is also used to calculate model statistics and to drive image-specific visualizations, like the Confusion Matrix. We have implemented a corpus layer that interacts with the CIFAR-10 dataset, but this architecture is extensible and it will be easy to add support for additional image corpora.

4.2.2 Rendering Visualizations

Regardless of the mode the user chooses, our system uses `decaf` to render visualizations. Internally, `decaf` takes the convolutional layers, normalizes them, and then visualizes them on a grid - typically displaying filters in rows with each channel shown in a separate column. This is an instance of visualizing small multiples [Tufte 1991], with position encoding pieces of the model structure.

One consideration that we have not incorporated yet is `decaf`'s filter image normalization strategy. While their normalization strategy is appropriate when displaying a single layer, it may cause some information loss when displaying layers over time. That is, we would rather see the layers *globally* normalized across all model checkpoints, as it would more accurately show model transitions over time. We leave this as future work.

We make use of `decaf`'s graph representation of the network to draw a graph that displays the structure of the network and the connectivity between its layers.

4.3 Statistics Engine

Several of our visualizations require model statistics that must be computed on the full training set. The models and corpora are quite large, so computing it's unrealistic to compute these in an online manner. Instead, we provide an engine to compute statistics in batches and save them to a database. This database is queried via the same web service that powers our filter visualizations.

Currently, the statistics engine calculates the following statistics over the corpus at each time step: the set of class probabilities by image, a confusion matrix indicating counts of predicted/actual class for all class pairs over all images, an index of images by the combination of their predicted/actual classes, and a set of

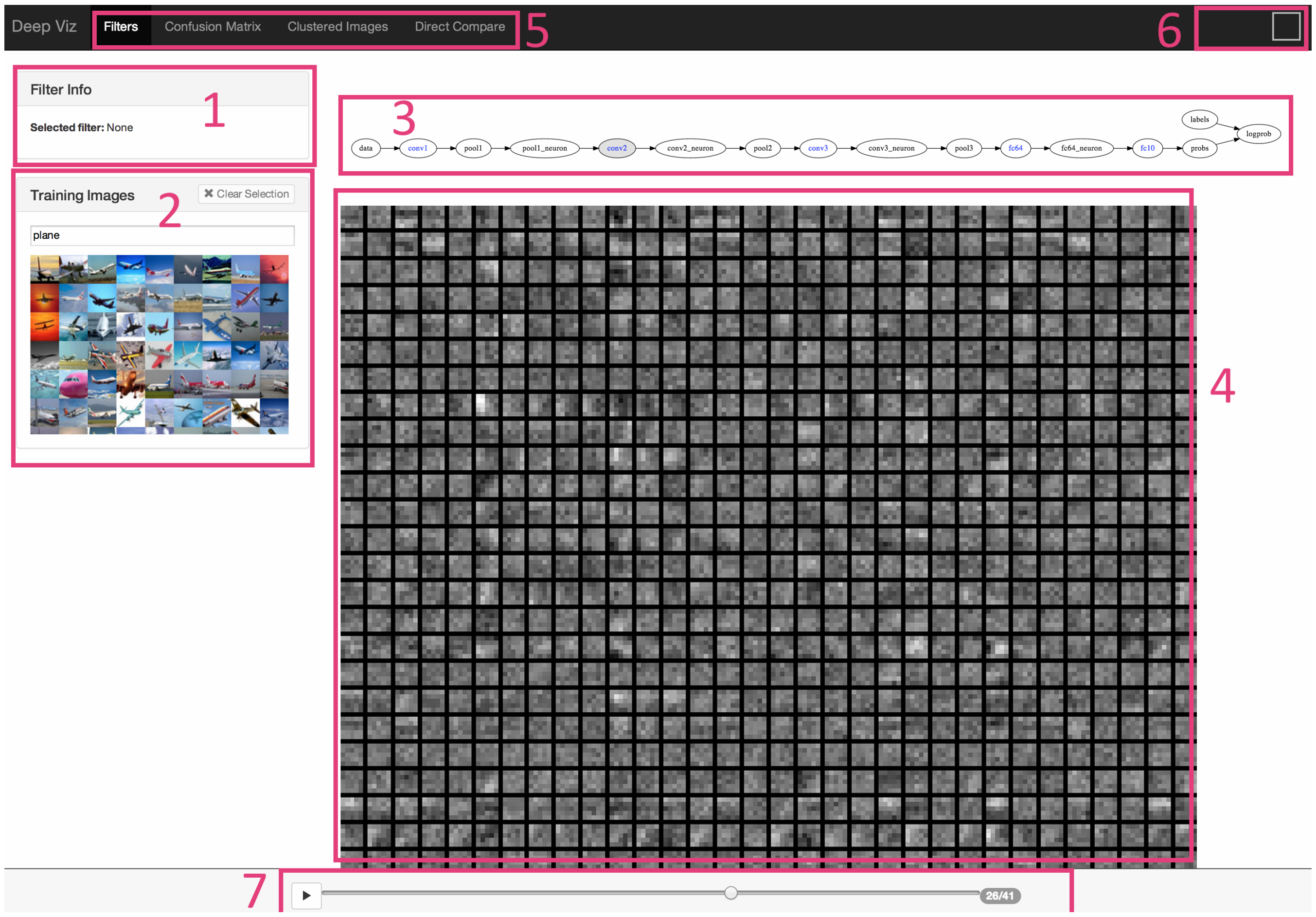


Figure 3: *`deepViz` main interface. 1) Filter details. 2) Image selector. 3) Network overview. 4) Filter visualization. 5) Visualization selector. 6) Selection helper. 7) Animation slider.*

clusters and the k-nearest neighbors of those clusters that are calculated from the last fully connected output layer of the image. Each of these statistics is used to drive one of the views described later.

To calculate these statistics, we use the `numpy` library and `scikit.learn`'s KMeans clustering function. The statistics for each checkpoint can be computed independently, so this process could be accelerated by using a parallel or distributed computing engine like Hadoop or Spark.

4.4 Web Application Back End

We provide access to the model query interface, visualization generation, and statistics engine via a RESTful interface which is backed by a Flask application.

Front end clients have the ability to request model state, as well as applications of model states to individual images, via custom URLs. Generally, the server's response is either a PNG image or a JSON object (if multiple images are requested). The client is responsible for handling and displaying these objects and enabling interaction. The server makes heavy use of caching to reduce latency when the same object is requested multiple times. We typically run the application in local mode, but do not see an issue with running it over the internet, and are considering adding the ability for users to upload their own models and image corpora.

4.5 Web Front End

Our web-based front end is composed of a collection of common web technologies: namely Javascript, HTML, CSS, jQuery, and Bootstrap. We also make use of d3 and Vega for certain visual components.

We made efforts to keep the web front-end very lightweight, but had to perform several optimizations in order to meet our design goals. When brushing across time steps, we needed to avoid flickering animation, since brief flashes of the page's background could contribute to change blindness and make it difficult for users to spot subtle differences in the figures.[Healey 2007] To achieve smooth, flicker-free animation, images are pre-fetched by the browser and positioned off-screen until they have loaded completely. With this approach, updating the canvas is seamless and doesn't require a round-trip to the web service.

5 Visualization Components

We present four main visualizations designed to help model builders understand their convolutional neural networks. While users may elect to use any of these visualizations, our primary display is a time-lapse view of model development for a particular filter. Within this view, the user can see how individual layers act on an input image by searching training corpus and choosing an image, which applies the model to that input image and displaying the image's activations at the selected filter layer.

All of our visualizations focus on helping users understand how the model changes over the course of training. That is, each view provides some notion of viewing and comparing models across time steps. We feel that this is an important and differentiating characteristic of our work which may enable new insights into the model training process. The additional visualizations serve as supporting information to help the user assess hypotheses about the cause of certain types of errors and understand the interaction between classes. Several of these visualization components are general enough to be used with other types of image classification systems. For example - the confusion matrix with small multiples of images would be a useful feature to have in any pipeline that involves multinomial classification.

All visualizations are displayed using a 3-layer model trained on the CIFAR-10[Krizhevsky et al.] dataset based on [Krizhevsky 2012]. The images displayed come from this training set.

5.1 Filters

Our system's main user interface is shown in Figure 3. At the bottom of the window is timeline control that supports pause, play, rewind, and *sliding* interactions. All of our views update in response to the current timeline value. These animated views allow users to see how the model evolves over time—in particular, especially at the early layers, we can easily observe how structure emerges. Indeed, the filter layers at this early stage end up looking much like Gabor filters [Movellan], which have used for years as featurizers for image classification algorithms. They can be interpreted as scale- and rotational-invariant edge detectors, an effect that we can see in Figure 1. It is important to note that the model converges on these filters *automatically* as part of the training process.

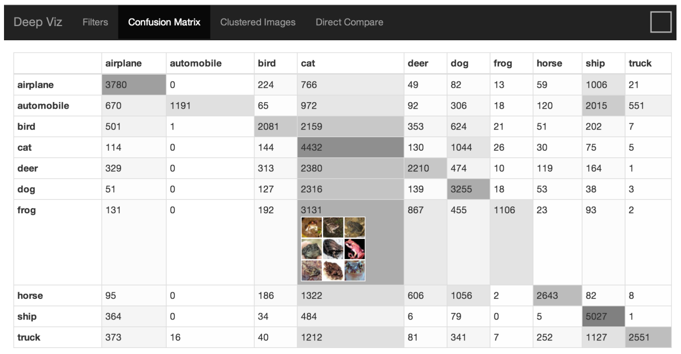
The top of the page displays a graphical representation of the neural network's layers. Nodes in this network may be selected with the mouse in order to choose which layer's filters are displayed in the main view.

Details about the currently displayed filters are provided in the upper-left corner of the display. Useful information like filter and channel position, as well as values of the current filter currently populate this display. In the future, this view might display information, such as the current filter's functional dependencies and the relative importance of its activation to each image class.

We provide a search interface for the image training corpus that allows users to search for and select images to pass through the network. In 1, we have searched for the word "plane" and selected a picture of an airplane. Our selected image is passed through the current model snapshot and we can visualize the output at each convolutional layer. Users can also click on images displayed in other views to select them. The sidebar displays a histogram of the model's predicted classes for the selected image. A more compact version of this histogram is displayed in the top-right corner of the page and is present in each of our four views. This mini-histogram allows users to quickly get a sense of the model's confidence in its predictions, which can be useful when using other views to inspect misclassified images.

5.2 Confusion Matrix

The confusion matrix view, shown in Figure 4, helps users to diagnose "hot-spots" of misclassification in their model. The matrix's rows corresponds to true image classes and its columns correpond to the model's predicted classes. In each cell, we display the number of images of true class x that (mis)classified as class y. A perfect classifier would produce a diagonal confusion matrix (with zeros everywhere but on the main diagonal).



	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	3780	0	224	766	49	82	13	59	1006	21
automobile	670	1191	65	972	92	306	18	120	2015	551
bird	501	1	2081	2159	353	624	21	51	202	7
cat	114	0	144	4432	130	1044	26	30	75	5
deer	329	0	313	2380	2210	474	10	119	164	1
dog	51	0	127	2316	139	3255	18	53	38	3
frog	131	0	192	3131	867	455	1106	23	93	2
horse	95	0	186	1322	606	1056	2	2643	82	8
ship	364	0	34	484	6	79	0	5	5027	1
truck	373	16	40	1212	81	341	7	252	1127	2551

Figure 4: *Confusion Matrix*

Cells are shaded according to their value, which can help draw attention to misclassifications, since off-diagonal cells that are dark indicate high rates of classification error.

When the user mouses over individual cell, the cells expand to show a sample of images that fall into that cell. These small multiples enable users to get an intuitive feel for which sorts of images tend to be misclassified. If the misclassified images share common visual structure, the user may choose to give special treatment to this structure in a future version of their model. For example, if dark pictures tend to be misclassified, the user might choose to normalize input images before feeding them into the network.

Like the main filter display, the confusion matrix view is linked to the timeline slider to show how the model evolves over time.

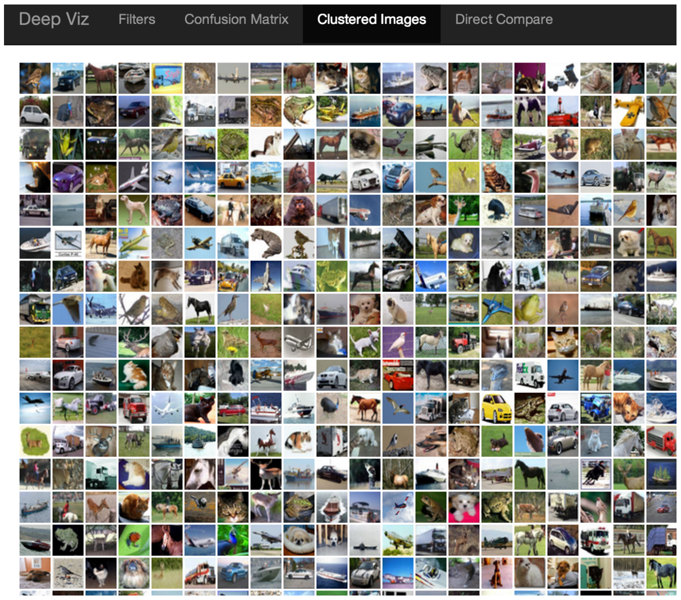


Figure 5: *Cluster View*

5.3 Clustered Images

To further aid in the diagnosis of classification errors, the clustered images view (shown in Figure 5) displays a set of sample images clustered by their similarity in the *last* layer of fully-connected output.

With this particular model, this is a 10-dimensional vector of neuron activations. We cluster these into 30 clusters using K-Means with a Euclidean distance metric. For each cluster, we display the top 20 closest images to the cluster center. If a user wants to understand what might be causing a certain group of misclassifications, they can inspect these clusters and see if any there is, for example, a sub-class of airplane images that looks more like a bird than an airplane. The user may then adjust the parameters of their model to better handle this case — for example, by increasing the resolution of their filters at an early layer.

Again, the time slider appears in this view to enable the user to see how these clusters evolve as the output of the fully connected layer changes at each model checkpoint.

In the current model, this view is not particularly interesting. We are unsure of the cause, but it may be because the scale of the FC10 activations is non-uniform across images. In future work, we may experiment with alternate distance metrics in the clustering algorithm (such as cosine similarity), or perform clustering on different layers, such as the FC64 layer.

To our knowledge, this is a novel approach for diagnosing misclassification issues in the context of convolutional neural networks.

5.4 Direct Comparison

In the direct comparison view, shown in Figure 6, we allow the user to select a set of layers and filters and compare them directly across multiple checkpoints. We make use of our backend's flexible query functionality to request images for several model checkpoints, layers, filters, and channels all at once. The results are displayed, again using the small multiples technique, with position encoding the model checkpoint in rows, and convolutional layer in columns.

This view enables the user to better understand the *global* structure of some subset of the model and directly compare snapshots of the model at two time steps without waiting for the animation to change the image.

We also provide the ability to show the application of the model to an image with the same selection criterion as the model currently on display.

Since we encode time point with row position in this display, we do not need to make use of the timeline slider in this particular display and the visualization is static. Nevertheless, this display clearly emphasizes the dynamic nature of the model over time and allows the user to understand that nature better.

In its current implementation, the parameters of this view are hardcoded. We expect to add interaction controls shortly.

We can already see some utility and insights with this view. For example, layers that are initialized with high-variance activations remain layers with high variance activations in the final model. The filters in the fifth row and last row of the first convolutional layer all start with high variance activation and remain high variance at the end. Further investigation may allow us to conclude that these filters play a dominant role in later neuron activations. We can use this information to inform our model initialization. For example, we may decide that we should initialize our filters with higher variance inputs across the board, or to apply regularization to our initial layers during training.

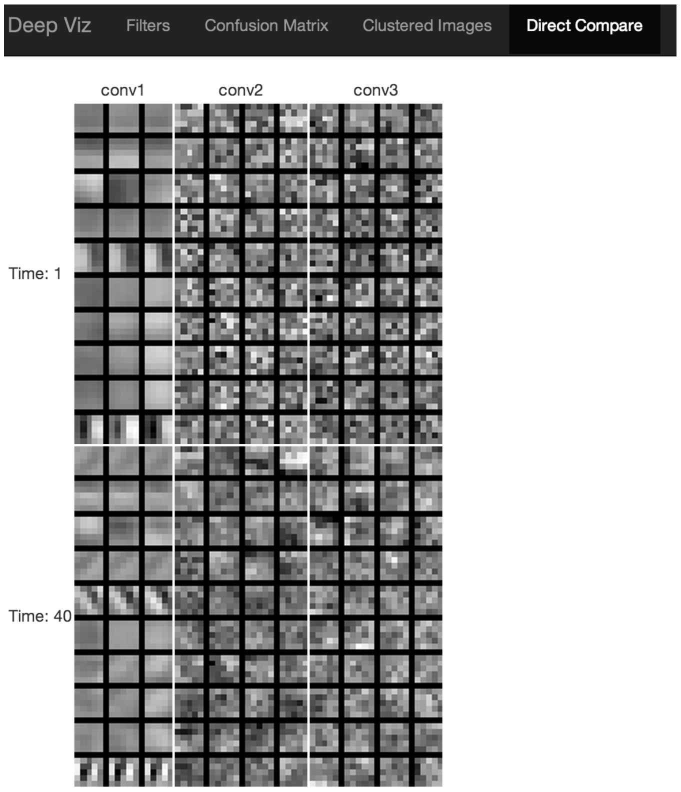


Figure 6: *Direct Compare*

6 Future Work

There are several other visualization techniques that we would like to explore in future work.

Our current interface allows users to view activations for individual images, but it could also be useful to view the aggregated activation of several images. For example, it might be helpful to compare the activation patterns of images airplanes and images of cars in order to identify features that could contribute to confusion of these classes. In addition to side-by-side comparison, we could display a single combined image of multiple aggregate activations by using hue to encode the most activating class and saturation to encode the strength of the correlation between a filter and its most activating class. With this style of visualization, we might expect the filter to be desaturated during the first few checkpoints and more saturated in deeper layers and at layer checkpoints as neurons become specialized.

To help to understand the behavior of individual filters, we plan to extend our statistics database to track the top activating images for each filter at every checkpoint. When mousing over filters, we will display these images to give a qualitative summary of the types of images for which they are most responsive.

We also plan to enable side-by-side views of models with different structures or tuning parameters in order to compare their performance and convergence rates. We hope that this will help model builders to keep track of their explorations through the model design space.

7 Conclusion

We have presented `deepViz`, a visualization tool aimed at helping experts understand and diagnose issues with their convolutional neural networks for visual classification. The tool consists of a processing backend and a web front-end that allows the user to explore various aspects of their models - from understanding the development of convolutional structure, to better understanding common types of misclassification.

Daniel Bruckner worked on instrumenting `cudaconvnet` with model checkpoint information, as well as several frontend features. Josh Rosen worked on the Flask application, the caching layer, and complete integration with `decaf`. He was also responsible for the front end and the first implementation of the statistics database. Evan Sparks worked on the initial `decaf` integration, the flexible query backend, the document clustering and direct compare view. All group members contributed to writing, poster, and presentation.

References

BOOTSRAP. http://getbootstrap.com/.

BRUNA, J., AND MALLAT, S. 2012. Invariant scattering convolution networks. *arXiv preprint arXiv:1203.1513*.

D3. http://d3js.org/.

DEAN, J., CORRADO, G. S., MONGA, R., CHEN, K., DEVIN, M., LE, Q. V., MAO, M. Z., RANZATO, M., SENIOR, A., TUCKER, P., YANG, K., AND NG, A. Y. 2012. Large scale distributed deep networks. In *NIPS*.

DONAHUE, J., JIA, Y., VINYALS, O., HOFFMAN, J., ZHANG, N., TZENG, E., AND DARRELL, T. 2013. Decaf: A deep convolutional activation feature for generic visual recognition. *arXiv preprint arXiv:1310.1531*.

DUCHI, J., HAZAN, E., AND SINGER, Y. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res. 12* (July), 2121–2159.

HEALEY, C. G. 2007. Perception in visualization. *Retrieved February 10*, 2008.

IMAGENET, 2013. http://www.image-net.org/.

JARRETT, K., KAVUKCUOGLU, K., RANZATO, M., AND LECUN, Y. 2009. What is the best multi-stage architecture for object recognition? In *Computer Vision, 2009 IEEE 12th International Conference on*, IEEE, 2146–2153.

JQUERY. http://jquery.com/.

KRIZHEVSKY, A., NAIR, V., AND HINTON, G. Cifar-10 dataset. http://www.cs.toronto.edu/~kriz/cifar.html.

KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. 2012. Imagenet classification with deep convolutional neural networks. 1106–1114.

KRIZHEVSKY, A., 2012. cuda-convnet. https://code.google.com/p/cuda-convnet/, July.

LE, Q. V., RANZATO, M., MONGA, R., DEVIN, M., CHEN, K., CORRADO, G. S., DEAN, J., AND NG, A. Y. 2011. Building high-level features using large scale unsupervised learning. *arXiv preprint arXiv:1113.6209*.

LECUN, Y., BOTTOU, L., BENGIO, Y., AND HAFFNER, P. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE 86*, 11, 2278–2324.

MOVELLAN, J. R. Tutorial on gabor filters.

NUMPY. http://www.numpy.org/.

SCIKIT.LEARN. http://scikit-learn.org/stable/.

TUFTE, E. R. 1991. Envisioning information. *Optometry & Vision Science 68*, 4, 322–324.

VEGA. http://trifacta.github.io/vega/.

VONDRICK, C., KHOSLA, A., AND MALISIEWICZ, T. 2013. HOGgles: Visualizing Object Detection Features. ... *Vision (ICCV)*.

WIKIPEDIA, 2013. Kernel (image processing) — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Kernel_(image_processing). [Online; accessed 11-Dec-2013].

YAMINS, D., TAX, D., AND BERGSTRA, J. S. 2013. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, 115–123.

ZEILER, M. D., AND FERGUS, R. 2013. Visualizing and Understanding Convolutional Networks. *arXiv.org* (Nov.).

ZEILER, M. D., TAYLOR, G. W., AND FERGUS, R. 2011. Adaptive deconvolutional networks for mid and high level feature learning. 2018–2025.