tf-lenet Documentation

Release a1

Ragav Venkatesan

Sep 27, 2017

Contents

Welcome to the Lenet tutorial using TensorFlow. From being a long time user of Theano, migrating to TensorFlow is not that easy. Recently, tensorflow is showing strong performance leading to many defecting from theano to tensorflow. I am one such defector. This repository contains an implementation Lenet, the hello world of deep CNNs and is my first exploratory experimentation with TensorFlow. It is a typical Lenet-5 network trained to classify MNIST dataset. This is a simple implementation similar to that, which can be found in the tutorial that comes with TensorFlow and most other public service tutorials. This is however modularized, so that it is easy to understand and reuse. This documentation website that comes along with this repository might help users migrating from theano to tensorflow, just as I did while implementing this repository. In this regard, whenever possible, I make explicit comparisons to help along. Tensorflow has many `contrib` packages that are a level of abstraction higher than theano. I have avoided using those whenever possible and stuck with the fundamental tensorflow modules for this tutorial. While this is most useful for theano to tensorflow migrants, this will also be useful for those who are new to CNNs. There are small notes and materials explaining the theory and math behind the working of CNNs and layers. While these are in no way comprehensive, these might help those that are unfamiliar with CNNs but want to simply learn tensorflow and would rather not spend time on a semester long course.

Note: The theoretical material in this tutorial are adapted from a forthcoming book chapter on *Feature Learning for Images*

To begin with, it might be helpful to run the entire code in its default setting. This will enable you to ensure that the installations were proper and that your machine was setup.

Obviously, you'd need tensorflow and numpy installed. There might be other tools that you'd require for advanced uses which you can find in the `requirements.txt` file that ships along with this code. Firstly, clone the repository down into some directory as follows,

```
git clone http://github.com/ragavvenkatesan/tf-lenet
cd tf-lenet
```

You can then run the entire code in one of two ways. Either run the `main.py` file like so:

```
python main.py
```

or type in the contents of that file, line-by-line in a python shell:

```
from lenet.trainer import trainer
from lenet.network import lenet5
from lenet.dataset import mnist

dataset = mnist()
net = lenet5(images = dataset.images)
net.cook(labels = dataset.labels)
bp = trainer (net, dataset.feed)
bp.train()
```

Once the code is running, setup tensorboard to observe results and outputs.

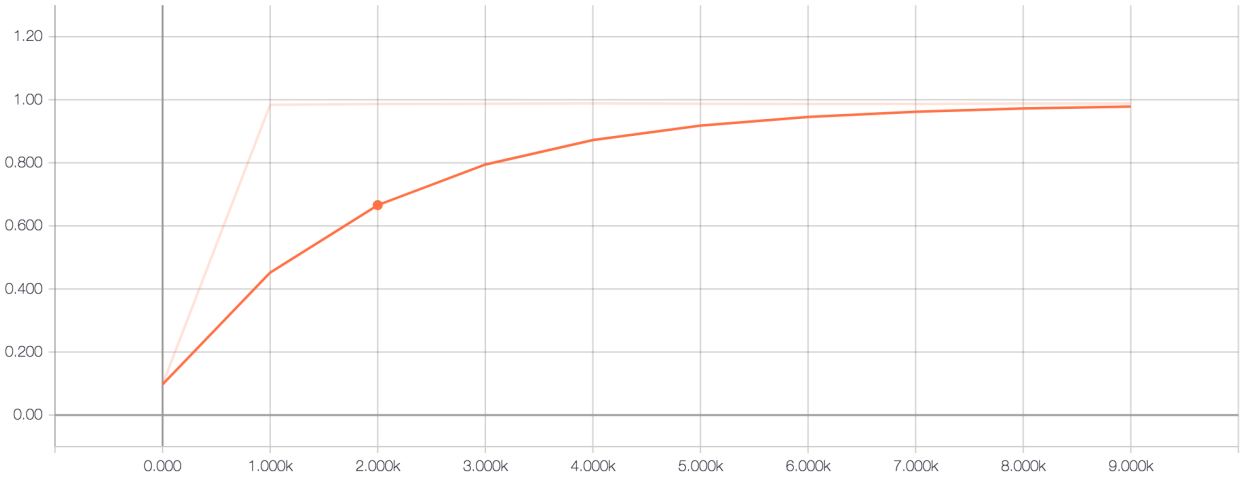
```
tensorboard --logdir=tensorboard
```

If everything went well, the tensorboard should have content populated in it. Open a browser and enter the address `0.0.0.0:6006`, this will open up tensorboard. The accuracy graph in the scalars tab under the test column will look like the following:

This implies that the network trained fully and has achieved about 99% accuracy and everything is normal. From the next section onwards, I will go in detail, how I built this network.

If you are interested please check out my Yann Toolbox written in theano completely. Have fun!

test/accuracy_1



CHAPTER 1

Tutorial

This is a CNN tutorial. This tutorial contains some minimal explanations of the types of what CNNs are and some discussions on the types of layers and so on. This is not detailed enough to be a complete guide, but just, so that it could be used as a recap. The tutorial is more implementation centric. Whenever possible code snippets are provided and comparisons made between theano and tensorflow to help migrants.

The tutorial is organized in such a way that the reader should be able to go article-by-article by clicking the *next* button at the end of each article. Although there is some ordering in the tutorial, one may skip sections that they are aware of and simple navigate to the section of interest as each one is written independently of the others. Have fun!

Introduction

The features that were classic to computer vision were designed by hand. Some, like PCA were designed with a particular objective, such as creating representations on basis where the variance was large. These features were, for the most part, general enough that they were used for a variety of tasks. Features like HOG *[DT05]* and SIFT *[Low99]* for instance, have been used for tasks including video activity recognition *[WOC+07]*, *[SMYC10]*, *[OL13]*, vehicle detection *[KHD11]*, *[SBM06]*, object tracking *[ZYS09]*, *[ZvdM13]*, pedestrian detection *[DWSP12]* and face detection *[ZYCA06]*, *[LMT+07]*, *[KS04]*, just to list a few. While this school of thought continue to be quite popular and some of these features have standardized implementations that are available for most researchers to plug and play for their tasks, they were not task-specific. These were designed to be useful feature representations of images that are capable of providing cues about certain aspects of images. HOG and SIFT for instance, provided shape-related information and were therefore used in tasks involving shape and structure. Features like color correlogram *[VCLL12]* provided cues on color transitions and were therefore used in medical images and other problems where, shape was not necessarily an informative feature. In this tutorial we will study a popular technique used in *learning to create good features* and task-specific feature extractors. These machine learn to extract useful feature representations of images using the data for a particular task.

Multi-layer neural networks have long since been viewed as a means of extracting hierarchical task-specific features. Ever since the early works of Rumelhart et al., *[RHW85]* it was recognized that representations learnt using back-propagation had the potential to learn fine-tuned features that were task-specific. Until the onset of this decade, these methods were severely handicapped by a dearth of large-scale data and large-scale parallel compute hardware to be leveraged sufficiently. This, in part, directed the creativity of computer vision scientists to develop the aforementioned

general-purpose feature representations. We now have access to datasets that are large enough and GPUs that are capable of large-scale parallel computations. This has allowed an explosion in neural image features and their usage. In the next sections we will study some of these techniques.

An artificial neural network is a network of computational neurons that are connected in a directed acyclic graph. There are several types of neural networks. While dealing with images, we are mostly concerned with the use the convolutional neural network (CNN). Each neuron accepts a number of inputs and produces one output, which can further be supplied to many other neurons. A typical function of a computational neuron is to weight all the inputs, sum all the weighted inputs and generate an output depending on the strength of the summed weighted inputs. Neurons are organized in groups, where each group typically receives input from the same sources. These groups are called as layers. Layers come in three varieties, each characterized by its own type of a neuron. They are, the dot-product or the fully-connected layer, the convolutional layer and the pooling layer.

Dot-Product Layers

Consider a $1D$ vector of inputs $\mathbf{x} \in [x_0, x_1, \ldots x_d]$ of d dimensions. This may be a vectorized version of the image or may be the output of a preceding layer. Consider a dot-product layer containing n neurons. The j^{th} neuron in this layer will simply perform the following operation,

$$z_j = \alpha\left(\sum_{i=0}^{d-1} x_i \times w_i^j\right),$$

where, α is typically an element-wise monotonically-increasing function that scales the output of the dot-product. α is commonly referred to as the activation function. The activation function is used typically as a threshold, that either turns ON (or has a level going out) or OFF, the neuron. The neuron output that has been processed by an activation layer is also referred to as an *activity*. Inputs can be processed in batches or mini-batches through the layer. In these cases $\mathbf{x}$ is a matrix in $\mathbb{R}^{b\times d}$, where b is the batch size. Together, the vectorized output of the layer is the dot-product operation between the weight-matrix of the layer and the input signal batch,

$$\mathbf{z} = \alpha(\mathbf{x} \cdot \mathbf{w}),$$

where, $\mathbf{z} \in \mathbb{R}^{b\times n}$, $\mathbf{w} \in \mathbb{R}^{d\times n}$ and the $(i, j)^{\text{th}}$ element of $\mathbf{z}$ represents the output of the j^{th} neuron for the i^{th} sample of input.

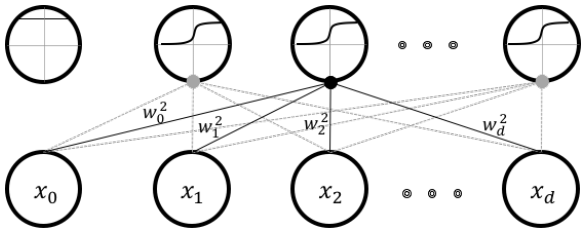


Fig. 1.1: A typical dot-product layer

The above figure shows such a connectivity of a typical dot-product layer. This layer takes in an input of d dimensions and produces an output of n dimensions. From the figure, it should be clear as to why dot-product layers are also referred to as fully-connected layers. These weights are typically *learnt* using back-propagation and gradient descent *[RHW85]*.

Implementation

Dot-product layers are implemented in tensorflow by using the `tf.matmul()` operation, which is a dot product operation. Consider the following piece of code:

```
# Open a new scope
with tf.variable_scope('fc_layer') as scope:
    # Initialize new weights
    weights = tf.Variable(initializer([input.shape[1].value,neurons], name = 'xavier_
↪weights'),\
                                        name = 'weights')
    # Initialize new bias
    bias = tf.Variable(initializer([neurons], name = 'xavier_bias'), name = 'bias')
    # Perform the dot product operation
    dot = tf.nn.bias_add(tf.matmul(input, weights, name = 'dot'), bias, name = 'pre-
↪activation')
    activity = tf.nn.relu(dot, name = 'activity' )  # relu is our alpha and activity␣
↪is z

    # Add things to summary
    tf.summary.histogram('weights', weights)
    tf.summary.histogram('bias', bias)
    tf.summary.histogram('activity', activity)
```

This code block is very similar to how a dot product would be implemented in theano. For instance, in yann I implemented a dot product layer like so:

```
w_values = numpy.asarray(0.01 * rng.standard_normal(
            size=(input_shape[1], num_neurons)), dtype=theano.config.floatX)
weights = theano.shared(value=w_values, name='weights')
b_values = numpy.zeros((num_neurons,), dtype=theano.config.floatX)
bias = theano.shared(value=b_values, name='bias')
dot = T.dot(input, w) + b
activity = theano.nnet.relu(dot)
```

We can already see that the `theano.shared()` equivalent in tensorflow is the `tf.Variable()`. They work in similar fashion as well. `tf.Variable()` is a node in a computational graph, just like `theano.shared()` variable. Operationally, the rest is easy to infer from the code itself.

There are some newer elements in the tensorflow code. Tensorflow graph components (variables and ops) could be enclosed using `tf.variable_scope()` declarations. I like to think of them as *boxes* to put things in literally. Once we go through tensorboard, it can be noticed that sometimes they literally are boxes. For instance, the following is a tensorboard visualization of this scope.

The initialization is also nearly the same. The API for the Xavier initializer can be found in the *lenet.support.initializer()* module. Tensorflow summaries is an entirely new option that is not available clearly in theano. Summaries are hooks that can write down or export information presently stored in graph components that can be used later by tensorboard to read and present in a nice informative manner. They can be pretty much anything of a few popular hooks that tensorflow allows. the `summary.histogram` allows us to track the histogram of particular variables as they change during iterations. We will go into more detail about summaries as we study the *lenet.trainer.trainer.summaries()* method, but at this moment you can think of them as *hooks* that export data.

The entire layer class description can be found in the *lenet.layers.dot_product_layer()* method.

Convolutional Layers

Fully-connected layers require a huge amount of memory to store all their weights. They involve a lot of computation as well. Vitally, they are not ideal for use as feature extractors for images. This is because, a dot product layer has an extreme receptive field. A receptive field of a neuron is the range of input flowing into the neuron. It is the view of the input that the neuron has access to to. In our definition of the dot-product layer, the receptive field of a neuron is

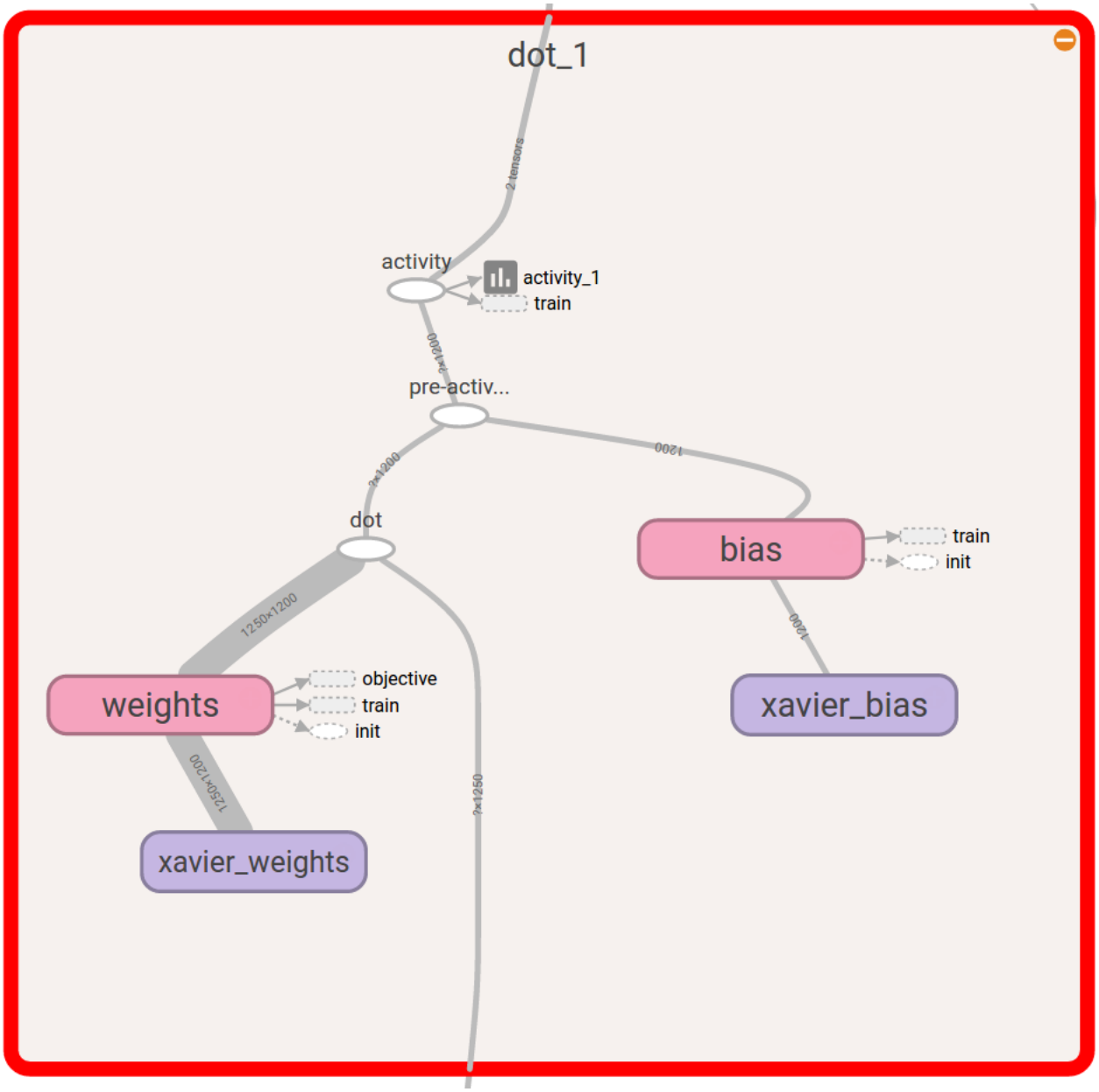


Fig. 1.2: A dot-product layer scope visualized in tensorboard

the length of the signal d. Most image features such as SIFT or HOG have small receptive fields that are typically a few tens of pixels such as 16×16. A convolution layer also has a small receptive field. This idea of having a small receptive field is also often referred to as *local-connectivity*.

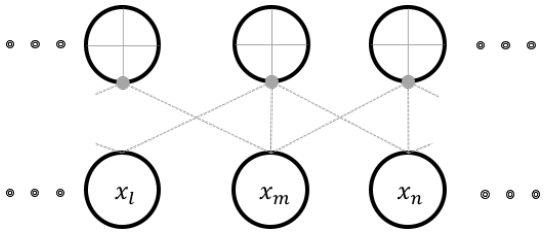


Fig. 1.3: Locally-connected neurons with a receptive field of $r = 3$

Locality arises in this connection because the input dimensions are organized in some order of spatial significance of the input itself. This implies that adjacent dimensions of inputs are locally-related. Consider a neuron that has a local-receptive field of $r = 3$. The figure above illustrates this connectivity. Each neuron in this arrangement, is only capable of detecting a pattern in an image that is local and small. While each location in an image might have spatially independent features, most often in images, we find that spatial independence doesn't hold. This implies that one feature learnt from one location of the image can be reasonably assumed to be useful at all locations.

Although the above figure shows the neurons being independent, typically several neurons share weights. In the representation shown in the figure, all the neurons share the same set of weights. Even though we produce $n - r + 1$ outputs, we only use r unique weights. The convolutional layer shown here takes a $1D$ input and is therefore a $1D$ convolutional layer. The figure below illustrates a $2D$ convolutional layer, which is what in reality, we are interested in. This figure does not show independent neurons and their connectivities but instead describe the weight shared neurons as *sliding* convolutional filters. In cases where the input has more than one channel, convolution happens along all channels independently and the outputs are summed location-wise.

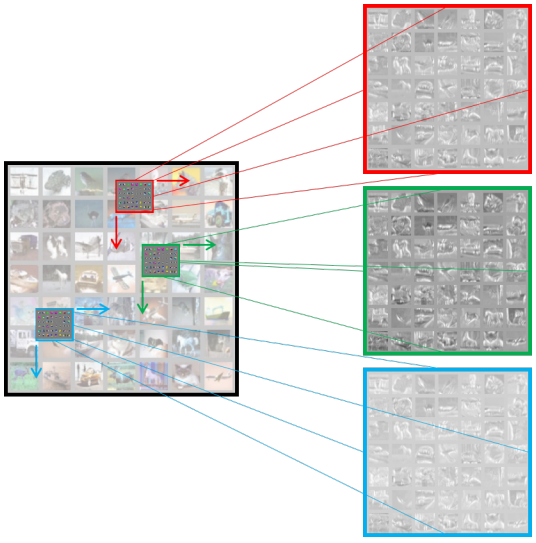


Fig. 1.4: A typical convolution layer.

The $2D$ convolution layer typically performs the following operation:

$$z(j, d_1, d_2) = \alpha \left[\sum_{c=0}^{C-1} \sum_{u=0}^{r-1} \sum_{v=0}^{r-1} x_{c,d_1+u,d_2+u} \times w_{u,v}^{j} \right], \forall j \in [0, 1, \ldots n] \text{ and } \forall d_1, d_2 \in [0, 1, \ldots d],$$

where, the weights $\mathbf{w} \in \mathbb{R}^{j,r,r}$ are j sets of weights, each set being shared by several neurons, each with a receptive field of r working on an input $x \in \mathbb{R}^{d_1 \times d_2}$. Since we have j sets of weights shared by several neurons, we will produce j activation *images* each (due to boundary conditions) of size $\mathbb{R}^{d-r+1 \times d-r+1}$.

In the context of convolution layers, the activations are also referred to as feature maps. The figure above shows three feature maps being generated at the end of the layer. The convolutional layer's filters are also *learnt* by back-propagation and gradient descent. Once learnt, these filters typically work as pattern detectors. Each filter produces one feature map. The feature map is a spatial map of confidence values for the existence of the pattern, the filter has adapted to detect.

Implementation

Similar to the dot-product layer, the conv layer implementation is also very similar to a theano structure. It can be implemented as follows:

```
# Create graph variables using the same xavier initialization
weights = tf.Variable(initializer(  f_shp,
                                    name = 'xavier_weights'),\
                                    name = 'weights')
bias = tf.Variable(initializer([neurons], name = 'xavier_bias'), name = 'bias')
# Produce the convolution output
c_out = tf.nn.conv2d(   input = input,
                        filter = weights,
                        strides = stride,
                        padding = padding,
                        name = scope.name  )
c_out_bias = tf.nn.bias_add(c_out, bias, name = 'pre-activation')
# activation through relu
activity = tf.nn.relu(c_out_bias, name = 'activity')

# Prepare summaries. Notice how even 4D tensors can be summarized using histogram
tf.summary.histogram('weights', weights)
tf.summary.histogram('bias', bias)
tf.summary.histogram('activity', activity)

# Visualize the first layer weights
visualize_filters(weights, name = 'filters_' + name)
```

While most of the code is easily understandable and migrated analogously from theano, the visualization needs to be adapted for tensorboard. The *`lenet.support.visualize_filters()`* method is a wrapper to a nice function written by kukurza . The code rastarizes the filters similar to what we are used to from pylearn2. The original code is hosted on their gist . My modified version is in *`lenet.third_party.put_kernels_on_grid()`*.

Some arguments to `tf.nn.conv2d()` are different from theano's `conv2d` structure. For instance, the arguments supplied here are:

```
filter_size = (5,5),
stride = (1,1,1,1),
padding = 'VALID',
```

Also the filter and image shapes is a little different as well. Images are 4D tensors in NHWC format. NHWC stands for number of images, height, width and channels, which for theano users is the `b01c`. This format difference is what that put me off while trying to implement this myself and is a useful reminders for migrants to keep in the back of their minds. The filters created in the *`lenet.support.initializer()`* method take `f_shp` shape where,

```
f_shp = [filter_size[0], filter_size[1], input.shape[3].value, neurons]
```

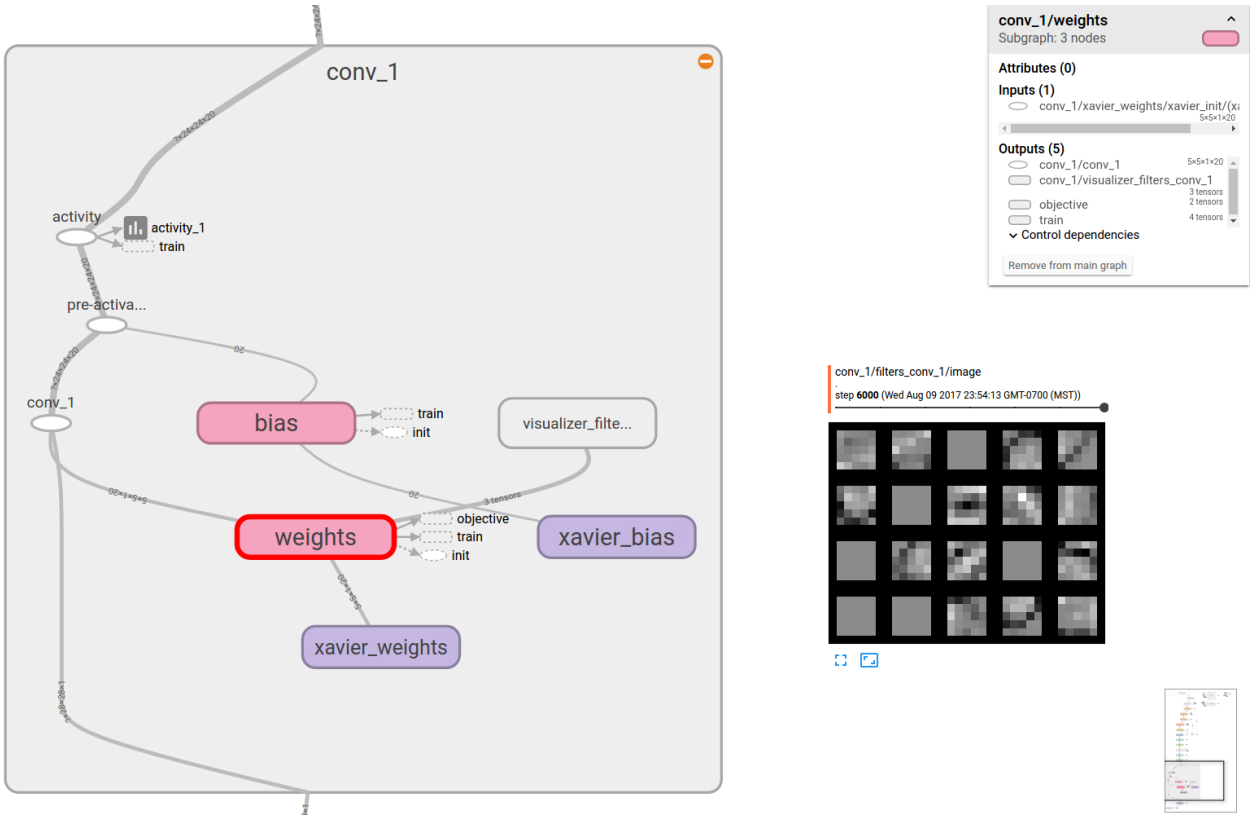


Fig. 1.5: A convolution layer scope visualized in tensorboard. The filters that it learnt are also shown.

That is, filter height, filter width, input channels, number of kernels. This is also a little strange for theano users and might take some getting used to. The entire layer class description can be found in the *lenet.layers.conv_2d_layer()* method.

Pooling Layers

The convolution layer creates activations that are $d - r + 1$ long on each axis. Adjacent activities in each of these feature maps are often related to each other. This is because, in imaging contexts, most patterns spread across a few pixels. We want to avoid storing (and processing) these redundancies and preferably only use the most prominent of these features.

This is typically accomplished by using a pooling or a sub-sampling operation. Pooling is done typically using non-overlapping sliding windows, where each window will sample one activation. In the context of images, pooling by maximum (max-pooling) is typically preferred. Pooling by p (widow size of p) reduces the sizes of activations by p fold. A pooling layer has no learnable components.

Implementation

A maxpooling layer for 4D tensors can be implemented as follows:

```
# The pooling size and strides are 4 dimensions also.
pool_size = (1,2,2,1)
stride = (1,2,2,1)
padding = 'VALID'
output = tf.nn.max_pool (   value = input,
                            ksize = pool_size,
                            strides = stride,
                            padding = padding,
                            name = name )
```

The only difference is between theano and tensorflow syntactically is that the arguments are different from `theano.pool2d()`. The arguments for pooling size (`ksize`) and `strides` are 4 dimensions as well. The shapes of inputs remain consistent with the `conv2d` module as discussed before. The entire layer class description can be found in the *lenet.layers.max_pool_2d_layer()* method.

Softmax Layer

The filter weights that were initialized with random numbers become task specific as we *learn*. Learning is a process of changing the filter weights so that we can expect a particular output mapped for each data samples. Consider the task of handwritten digit recognition *[LBD+90]*. Here, we attempt to map images (28×28 pixels) to an integer $y \in [0, 1 \ldots 9]$. MNIST is the perfect dataset for this example, which was desigend for this purpose. MNIST images typically look as follows:

Learning in a neural network is typically achieved using the back-prop learning strategy. At the top end of the neural network with as many layers is a logistic regressor that feeds off of the last layer of activations, be it from a fully-connected layer as is conventional or a convolutional layer such as in some recent network implementations used in image segmentation *[LSD15]*. When the layer feeding into a softmax layer is a dot-product layer with an identity activation $\alpha(x) = x$, we refer to the inputs often as *logits*. In modern neural networks, the logits are not limited in operation with any activations. Often, this regressor is also implemented using a dot-product layer, for logistic regression is simply a dot-product layer with a softmax activation.

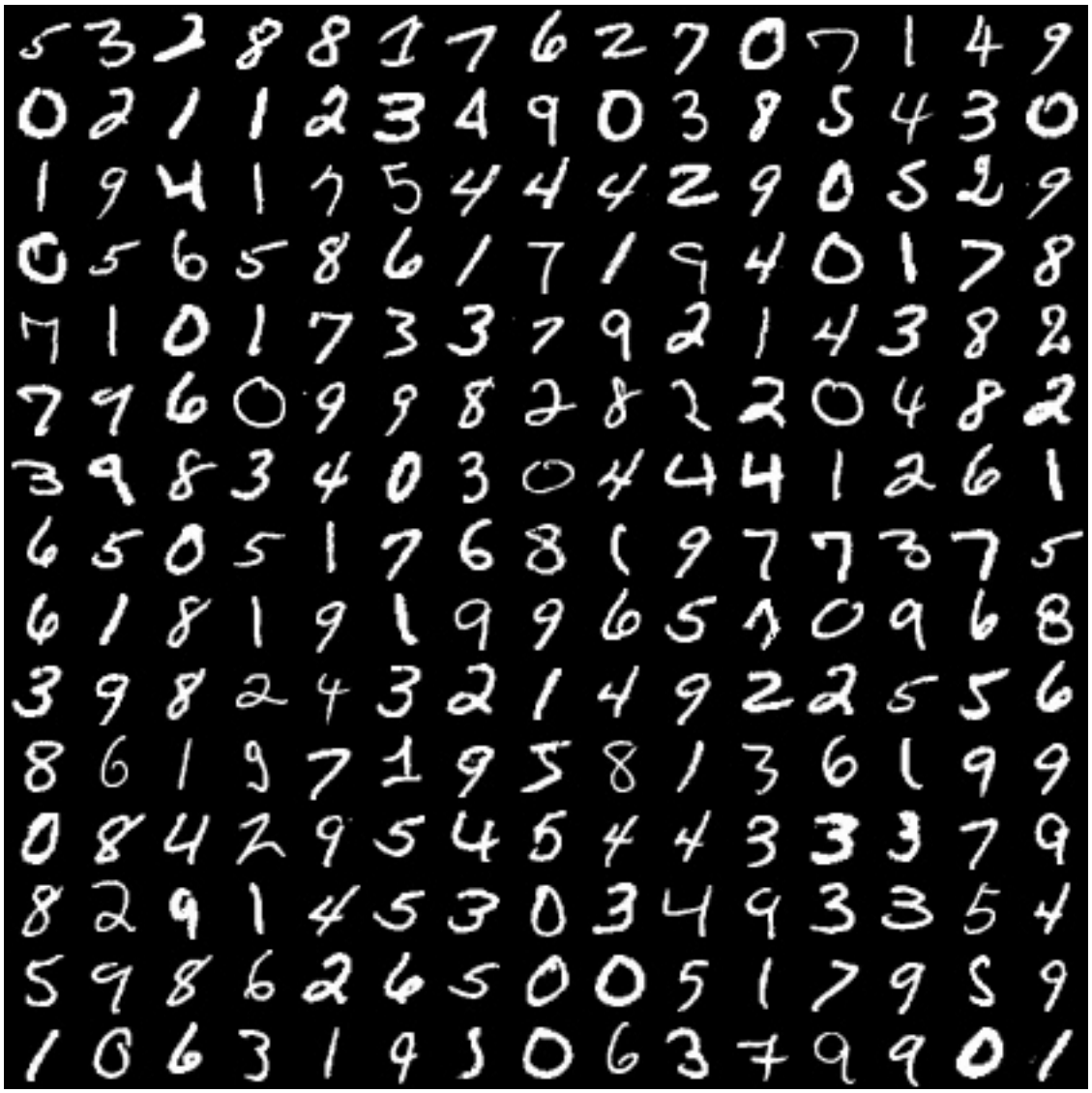


Fig. 1.6: MNIST dataset of handwritten character recognition.

A typical softmax layer is capable of producing the probability distribution over the labels $y \in [0, 1, \ldots c]$ that we want to predict. Given an input image $\mathbf{x}$, $P(y|\mathbf{x})$ is estimated as follows,

$$\begin{bmatrix} P(y=1|\mathbf{x}) \\ \vdots \\ P(y=c|\mathbf{x}) \end{bmatrix} = \frac{1}{\sum_{p=1}^{c} e^{w^p N'(\mathbf{x})}} \begin{bmatrix} e^{w^1 N'(\mathbf{x})} \\ \vdots \\ e^{w^c N'(\mathbf{x})} \end{bmatrix},$$

where, $\mathbf{w}$ is the weight matrix of the dot-product layer preceding the softmax layer with w^p representing the weight vector that produces the output of the class p and $N'(\mathbf{x})$ is the output of the layer in the network N, immediately preceding this dot-product-softmax layer. The label that the network predicts $\hat{y}$, is the maximum of these probabilities,

$$\hat{y} = \arg\max_{y} P(y|\mathbf{x}).$$

Implementation

I implemented the softmax layer as follows:

- Use a `lenet.layers.dot_product_layer()` with 10 neurons and `identity` activation.
- Use a `lenet.layers.softmax_layer()` to produce the softmax.

In the softmax layer, we can return computational graph nodes to predictions, logits and softmax. The reason for using logits will become clear in the next section when we discuss errors and back prop. Essentially, we will create a layer that will look like the following image in its tensorboard visualization:

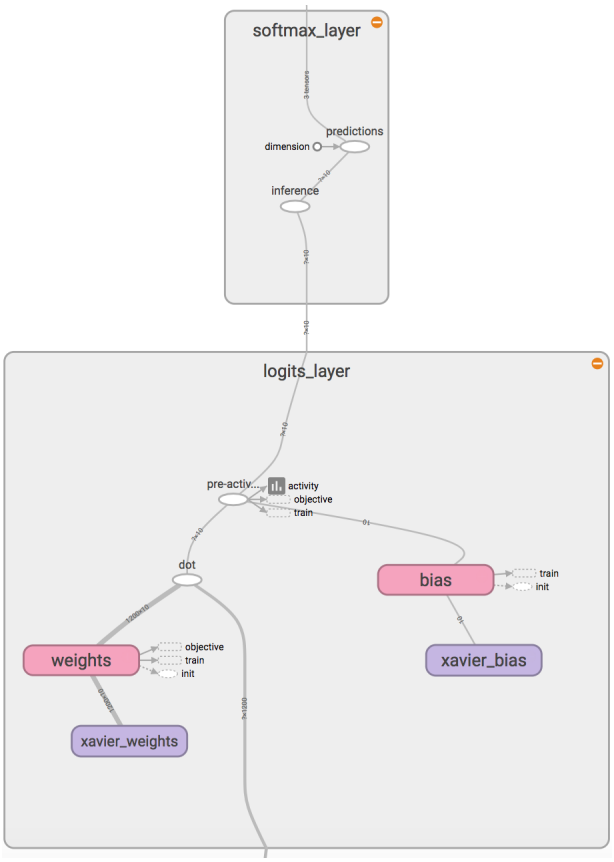


Fig. 1.7: Softmax Layer implementation.

The logits layer is a *lenet.layers.dot_product_layer()* with identity activation (no activation). The `inference` node will produce the softmax and the `prediction` node will produce the label predicted. Th softmax layer is implemented as:

```
inference = tf.nn.softmax(input, name = 'inference')
predictions = tf.argmax(inference, 1, name = 'predictions')
```

Where `tf.nn.softmax()` and `tf.nn.argmax()` have similar syntax as the theano counterparts. To have the entire layer, in the *lenet.network.lenet5* which is where these layer methods are called, I use the following strategy:

```
# logits layer returns logits node and params = [weights, bias]
logits, params = lenet.layers.dot_product_layer  (
                                        input = fc2_out_dropout,
                                        neurons = C,
                                        activation = 'identity',
                                        name = 'logits_layer')
# Softmax layer returns inference and predictions
inference, predictions = lenet.layers.softmax_layer (
                                        input = logits,
                                        name = 'softmax_layer' )
```

Where `C` is a globally defined variable with `C=10` defined in `lenet.gloabl_definitions` file. The layer definitions can be seen in full in the documentation of the *lenet.layers.softmax_layer()* method.

Reshaping layers

Most often datasets are of the form where $\mathbf{x} \in [x_0, x_1, \ldots x_d]$ of d dimensions. We want $\mathbf{x} \in \mathbb{R}^{\sqrt{d} \times \sqrt{d}}$ for the convolutional layers. For this and for other purposes, we might be helped by having some reshaping layers. Here are a few in tensorflow.

For flattening, which is converting an image into a vector, used for instance, before feeding into the first fully-connected layers:

```
in_shp = input.get_shape().as_list()
output = tf.reshape(input, [-1, in_shp[1]*in_shp[2]*in_shp[3]])
```

The reshape command is quite simlar to theano. The nice thing here is the `-1` option, which implies that any dimension that have `-1` immediately accommodates the rest. This means that we don't have to care about knowing the value of that dimension and could assign it during runtime. I use this for the mini batch size being unknown. One network can now run in batch, stochastic or online gradient descent and during test time, I can supply how many ever samples I want.

Similarly, we can also implement an unflatten layer:

```
dim = int( np.sqrt( input.shape[1].value / channels ) )
output = tf.reshape(input, [-1, dim, dim, channels])
```

These are also found in the *Layers* module in *lenet.layers.flatten_layer()* and *lenet.layers.unflatten_layer()*.

CNN Architecture Philosophies

Analogous to model design in most of machine learning and to the practice of hand-crafting features, CNNs also involve some degree of skilled hand-crafting. Most of hand-crafting involves the design of the architecture of the network. This involves choosing the types and number of layers and types of activations and number of neurons in each layer. One important design choice that arises particularly in image data and CNNs, is the design of the receptive fields.

The receptive field is typically guided by the size of filters (weights) in each layer and by the use of pooling layers. The receptive field grows after each layer as the range of the signal received from the input layer grows progressively. There are typically two philosophies relating to the choice of filter sizes and therefore to the receptive fields. The first was designed by Yann LeCun et al., *[LBBH98]*, *[LBD+90]* and was later re-introduced and widely preferred in modern day object categorization by Alex Krizhevsky et al., *[KSH12]*. They employ a *relatively* large receptive field at the earlier layers and continue growing with the rate of growth reducing by a magnitude. Consider AlexNet *[KSH12]*. This network won the ImageNet VOC challenge *[DDS+09]* in 2012 which involves recognizing objects belonging to 1000 categories with each image being 224×224 in size. This network has a first layer with 11×11 convolutional filters (which are strided by 4 pixels), followed by a 3×3 pooling (strided by 2). The next layer is 5×5, followed by 3×3, each with their own respective pooling layers.

The second of these philosophies is increasing the receptive field as minimally as possible. These were pioneered by the VGG group *[SZ14]* and one particular implementation won the 2014 ImageNet competition *[DDS+09]*. These networks have a fixed filter size, typically of 3×3 and have fixed pooling size of 2×2 at some checkpoint layers. These philosophies aim to hierarchically learn better filters over various growth of small receptive fields. The classic LeNet from Yann LeCun is a trade-off between these two case studies *[LBBH98]*.

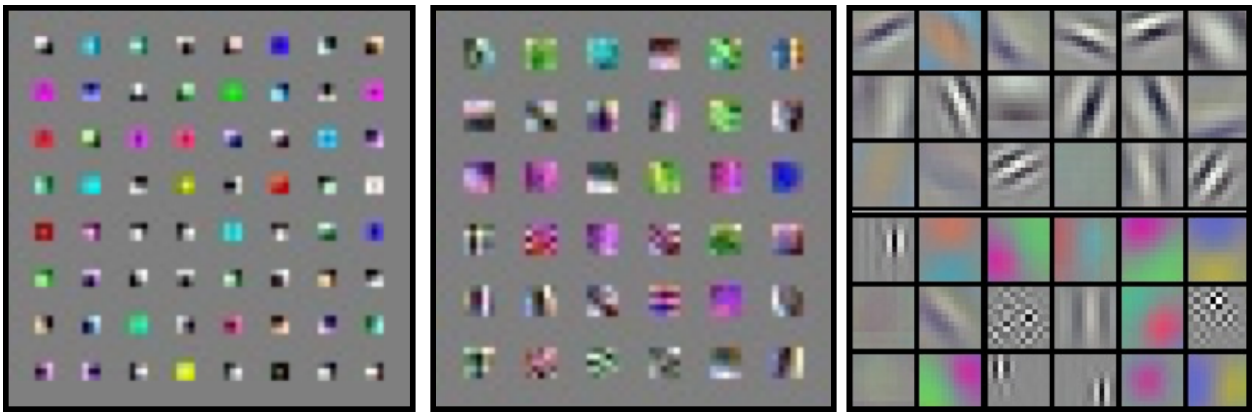


Fig. 1.8: Filters learnt by CNNs.

The figure above show various filters that were learnt by each of these philosophies at the first layer that is closest to the image. From left to right are filters learnt by VGG (3×3), a typical Yann LeCun style LeNet (5×5) and a AlexNet (11×11). It can be noticed that although most first layer filters adapt themselves to simple edge filters and Gabor filters, the 3×3 filters are simpler lower-dimensional corner detectors while the larger receptive field filters are complex high-dimensional pattern detectors.

Although we studied some popular network architecture design and philosophies, several other styles of networks also exists. In this section we have only studied those that feed forward from the input layer to the task layer (whatever that task might be) and there is only one path for gradients to flow during back-prop. Recently, several networks such as the GoogleNet *[SLJ+15]* and the newer implementations of the inception layer *[SVI+16]*, *[SIVA17]*, Residual Net *[HZRS16]* and Highway Nets *[SGS15]* have been proposed that involve creating DAG-style networks that allow for more than one path to the target. One of these paths typically involve directly feeding forward the input signal. This therefore allows for the gradient to not attenuate and help in solving the vanishing gradient problems *[BSF94]*.

Normalization Layer

Implementing normalization was much simpler than using theano. All I had to do was

```
output = tf.nn.lrn(input)
```

`tf.nn.lrn()` is an implementation of the local response normalization *[LS08]*. This layer definition could also be found in the *`lenet.layers.local_response_normalization_layer()`* method.

Dropout Layers

Dropout layers are an indirect means of regularization and ensemble learning for neural networks *[SHK+14]*. Consider that we have a layer with n activations. Consider now, we randomly zero-out neurons independently with Bernoulli probability 0.5 everytime we provide the network a training sample. We update the weights for only those weights that were not zeroed-out during backprop. This in essence is the working of dropouts.

Dropouts have several different interpretations. The first and most often used is that dropout is a form of ensembling. In this form of dropping out some neurons, we are in-effect sampling from a pool of $2n$ architectures, every time we feed-forward the sample is going through a new architecture (network). Which means each *network* in this pool of network architectures will only see one if it sees any sample at all. We train each of these networks with one update. Since all the networks in this system share weights, they all learn simultaneously.

During test time, we halve the weights (if we used 0.5 as the Bernoulli probability) of weights. This halving is needed because during test time we do not dropout and the output signals of each layer are therefore in expectation doubled. Theoretically, this halving (at least with just one layer) is the same as the output of the geometric ensemble of all the networks in the pool.

A second way of looking at dropout is from a regularization perspective. Dropouts are a strong form of regularizer, because in the pool of all th networks, one can think of dropouts as penalizing the network in learning weights that are regularized by the weights on the other networks. Yet another perspective of dropout is that dropout avoid neurons to co-adapt with each other. They learn functionalities that are unique to itself and are independent. Therefore dropout neurons are more powerful than co-adapted neurons.

Implementation

In theano, dropouts are typically implemented as follows:

```
from theano.sandbox.rng_mrg import MRG_RandomStreams as RandomStreams
# from theano.tensor.shared_randomstreams import RandomStreams
# The above import is an experimental code. Not sure if it works perfectly, but I␣
↪have seen no problems
# yet.
srng = RandomStreams(rng.randint(1,2147462468), use_cuda=None)
# I have raised this issue with the theano guys, use_cuda = True is creating a␣
↪duplicate
# process in the GPU.
mask = srng.binomial(n=1, p=1-dropout_rate, size=params.shape, dtype = theano.config.
↪floatX )
output = input * mask
```

Ignoring the comments about the `RandomStreams` itself, the implementation is a little quirky. We create a `mask` which is the same shape as the input, and multiply the input by that mask to get the dropout output. This output is used during training time. We create another *parallel copy of the network* which uses the already initialized weights but are *halved* during inference time.

```
# If dropout_rate is 0, this is just a wasted multiplication by 1, but who cares.
w = dropout_layer.w * (1 - dropout_rate)
b = dropout_layer.b

inference_layer = dot_product_layers(params = [w,b] )
```

Assuming of course that the layers are defined as classes.

In tensorflow, we will make use a *placeholder* node in the graph to implement this system (which also could be done using theano). A placeholder is a node in tensorflow similar to `theano.tensor()` or `theano.scalar()`. Let us create the dropout probability as a placeholder node.

```
dropout_prob = tf.placeholder(tf.float32, name = 'dropout_probability')
```

In tensorflow, we have a `dropout` method written for us internally, which can use a placeholder probability node. Let us supply the placeholder we created into the dropout layer method and create a dropout layer.

```
output = tf.nn.dropout (input, dropout_prob)
```

During test time, we can feed into the placeholder 1.0, which implies that none of the nodes are dropped out. During training time, we can feed whatever value we want into the dropout variable. The internal tensorflow implementation of dropout will scale the input accordingly (note that it does not scale the weights, so this has problems when implementing on non-dot-product layers some times).

When there are multiple layers, we can still use the same placeholder and therefore control the action of dropout globally. The tensorboard will look like the following when we use these ideas to create dropout layers.

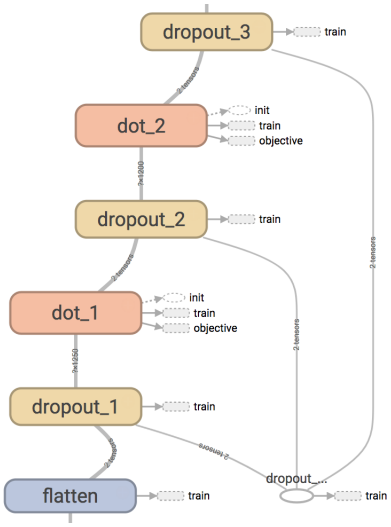


Fig. 1.9: Dropout visualized in tensorflow. One placeholder controls three dropout layers. Note: we also dropout the *input* signal even before the first dot-product layer. This idea was taken from the de-noising auto-encoder paper *[VLBM08]*.

The layer definition can be seen in the *`lenet.layers.dropout_layer()`* module.

Network

Now that we have seen all the layers, let us assemble our network together. Assembling the network together takes several steps and tricks and there isn't one way to do that. To make things nice, clean and modular, let us use python class to structure the network class. Before we even begin the network, we need to setup the dataset. We can then setup our network. We are going to setup the popular Lenet5 *[LBD+90]*. This network has many incarnations, but we are going to setup the latest one. The MNIST images that are input are 28×28. The input is fed into two convolution layers with filter sizes 5×5 and 3×3 with 20 and 50 filters, respectively. This is followed by two fully-connected layers of 800 neurons each. The last softmax layer will have 10 nodes, one for each class. In between, we add some dropout layers and normalization layers, just to make things a little better.

Let us also fix this by using global definitions (refer to them all in `lenet.gloabl_definitions` module).

```
# Some Global Defaults for Network
C1 = 20      # Number of filters in first conv layer
C2 = 50      # Number of filters in second conv layer
D1 = 1200     # Number of neurons in first dot-product layer
D2 = 1200     # Number of neurons in second dot-product layer
C = 10       # Number of classes in the dataset to predict
F1 = (5,5)   # Size of the filters in the first conv layer
F2 = (3,3)   # Size of the filters in the second conv layer
DROPOUT_PROBABILITY = 0.5 # Probability to dropout with.

# Some Global Defaults for Optimizer
LR = 0.01    # Learning rate
WEIGHT_DECAY_COEFF = 0.0001 # Co-Efficient for weight decay
L1_COEFF = 0.0001 # Co-Efficient for L1 Norm
MOMENTUM = 0.7 # Momentum rate
OPTIMIZER = 'adam' # Optimizer (options include 'adam', 'rmsprop') Easy to upgrade if␣
↪needed.
```

Dataset

Tensorflow examples provides the MNIST dataset in a nice feeder-worthy form, which as a theano user, I find very helpful. The example itself is at `tf.examples.tutorials.mnist.input_data()` for those who want to check it out. You can quite simply import this feeder as follows:

```
from tensorflow.examples.tutorials.mnist import input_data as mnist_feeder
```

Using this, let us create a class that will not only host this feeder, but will also have some placeholders for labels and images.

```
def __init__ (self, dir = 'data'):
    self.feed = mnist_feeder.read_data_sets (dir, one_hot = True)

    #Placeholders
    with tf.variable_scope('dataset_inputs') as scope:
        self.images = tf.placeholder(tf.float32, shape=[None, 784], name = 'images')
        self.labels = tf.placeholder(tf.float32, shape = [None, 10], name = 'labels')
```

This now creates the following section of the graph:

Fashion-MNIST

Fashion-MNIST is a new dataset that appears to take the place of MNIST as a good CV baseline dataset. It has the

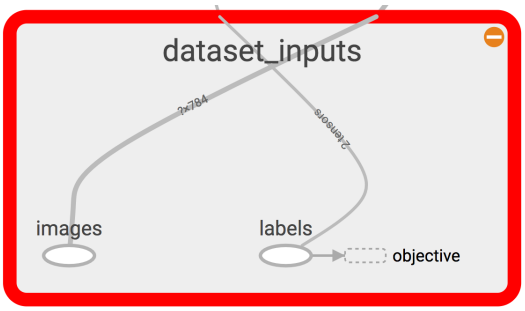


Fig. 1.10: Dataset visualized in tensorboard.

same characteristics as MNIST itself and could be a good drop-in dataset in this tutorial. If you prefer using this dataset instead of the classic MNIST, simply download the dataset from here into the `data/fashion` directory and use the *`lenet.dataset.fashion_mnist()`* instead of the old *`lenet.dataset.mnist()`* method. This uses the data in the new directory.

Network Architecture

With all this initialized, we can now create a network class (*`lenet.network.lenet5`*), whose constructor will take this image placeholder.

```
def __init__ (  self,
                images ):
    """
    Class constructor for the network class.
    Creates the model and all the connections.
    """
    self.images = images
```

As can be seen in the documentation of *`lenet.network.lenet5`*, I have a habit of assigning some variables with `self` so that I can have access to them via the objects. This will be made clear when we study further *`lenet.trainer.trainer`* module and others. For now, let us proceed with the rest of the network architecure.

The first thing we need is to *unflatten* the images placeholder into square images. We need to do this because the images placeholder contains images in shape $\mathbf{x} \in [x_0, x_1, \ldots x_d]$ of d dimensions. To have the input feed into a convolution layer, we want, 4D tensors in NHWC format as we discussed in the convolution layer *Implementation* section. Let us continue building our network constructor with this unflatten added.

```
images_square = unflatten_layer ( self.images )
visualize_images(images_square)
```

The method *`lenet.support.visualize_images()`* will simply add these images to tensorboard summaries so that we can see them in the tensorboard. Now that we have a unflattened image node in the computational graph, let us construct a couple of convolutional layers, pooling layers and normalization layers.

```
# Conv Layer 1
conv1_out, params =  conv_2d_layer (    input = images_square,
                                        neurons = C1,
                                        filter_size = F1,
                                        name = 'conv_1',
                                        visualize = True )
process_params(params)
pool1_out = max_pool_2d_layer ( input = conv1_out, name = 'pool_1')
```

```
lrn1_out = local_response_normalization_layer (pool1_out, name = 'lrn_1' )

# Conv Layer 2
conv2_out, params =  conv_2d_layer (    input = lrn1_out,
                                        neurons = C2,
                                        filter_size = F2,
                                        name = 'conv_2' )
process_params(params)

pool2_out = max_pool_2d_layer ( input = conv2_out, name = 'pool_2')
lrn2_out = local_response_normalization_layer (pool2_out, name = 'lrn_2' )
```

lenet.layers.conv_2d_layer() returns one output tensor node in the computation graph and also returns the parameters list `[w, b]`. The parameters are sent to the *lenet.network.process_params()*. This method is a simple method which will add the parameters to various *collections*.

```
tf.add_to_collection('trainable_params', params[0])
tf.add_to_collection('trainable_params', params[1])
tf.add_to_collection('regularizer_worthy_params', params[0])
```

These tensorflow collections span throughout the implementation session, therefore these collections can be used at a later time to apply gradients to the `trainable_params` collections or to add regularization to `regularizer_worthy_params`. I typically do not regularize biases.

If this method was not called after a layer was added, you can think of it as being used for frozen or obstinate layers as is typically used in mentoring networks purposes *[VL16]*. We now move on to the fully-connected layers. Before adding them, we need to *flatten* the outputs we have so far. We can use the *lenet.layers.flatten_layer()* to reshape the outputs.

```
flattened = flatten_layer(lrn2_out)
```

In case we are implementing a dropout layer, we need a dropout probability placeholder that we can feed in during train and test time.

```
self.dropout_prob = tf.placeholder(tf.float32, name = 'dropout_probability')
```

Let us now go ahead and add some fully-connected layers along with some dropout layers.

```
# Dropout Layer 1
flattened_dropout = dropout_layer ( input = flattened, prob = self.dropout_prob, name
↪= 'dropout_1')

# Dot Product Layer 1
fc1_out, params = dot_product_layer  ( input = flattened_dropout, neurons = D1, name
↪= 'dot_1')
process_params(params)

# Dropout Layer 2
fc1_out_dropout = dropout_layer ( input = fc1_out, prob = self.dropout_prob, name =
↪'dropout_2')
# Dot Product Layer 2
fc2_out, params = dot_product_layer  (  input = fc1_out_dropout, neurons = D2, name =
↪'dot_2')
process_params(params)

# Dropout Layer 3
fc2_out_dropout = dropout_layer ( input = fc2_out, prob = self.dropout_prob, name =
↪'dropout_3')
```

Again we supply the parameters through to a regularizer. Finally, we add a *lenet.layers.softmax_layer()*.

```
# Logits layer
self.logits, params = dot_product_layer  (  input = fc2_out_dropout, neurons = C,
                                            activation = 'identity', name = 'logits_
↪layer')
process_params(params)

# Softmax layer
self.inference, self.predictions = softmax_layer ( input = self.logits, name =
↪'softmax_layer')
```

We use the *lenet.layers.dot_product_layer()* to add a `self.logits` node that we can pass through to the softmax layer that will provide us with a node for `self.inference` and `self.predictions`.

Putting all this together, the network will look like the image above in tesorboard. The complete definition of this network class could be found in the class constructor of *lenet.network.lenet5*.

Cooking the network

Before we begin training though, the network needs several things added to it. The first one of which is a set of cost and objectives. Firstly we begin with adding a `self.labels` property to the network class. This placeholder comes from the *lenet.dataset.mnist* class.

For a loss we can start with a categorical cross entropy loss.

```
self.cost = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits ( labels = self.
↪labels,
                                                                      logits = self.
↪logits) )
tf.add_to_collection( 'objectives', self.cost )
tf.summary.scalar( 'cost', self.cost )
```

The method `tf.nn.softmax_cross_entropy_with_logits()` is another unique feature of tensorflow. This method will take in `logits` which are the outputs of the identity dot-product layer before the softmax, apply softmax to it and estimate its cross-entropy loss with a one-hot vector version of labels provided to the `labels` argument, all doing so efficiently.

We can add this to the `objectives` collection. Collections are in essence, kind of like lists that span globally as long as we are in the same tensorflow shell. There are much more to it, but for a migrant, at this stage, this is simple. We can add up everything in the `objectives` collection which ends up in a node that we want to minimize. For instance, we can add regularizers to the `objectives` collection also, so that they all can be added to the minimizing node. Since *lenet.network.process_params()* method was called after all params were created and we added parameters to collections, we can apply regularizers to all parameters in the collection.

```
var_list = tf.get_collection( 'regularizer_worthy_params')
apply_regularizer (var_list)
```

where, the *lenet.network.apply_regularizer()* adds $L1$ and $L2$ regularizers.

```
for param in var_list:
    norm = L1_COEFF * tf.reduce_sum(tf.abs(param, name = 'abs'), name = 'l1')
    tf.summary.scalar('l1_' + param.name, norm)
    tf.add_to_collection( 'objectives', norm)
```

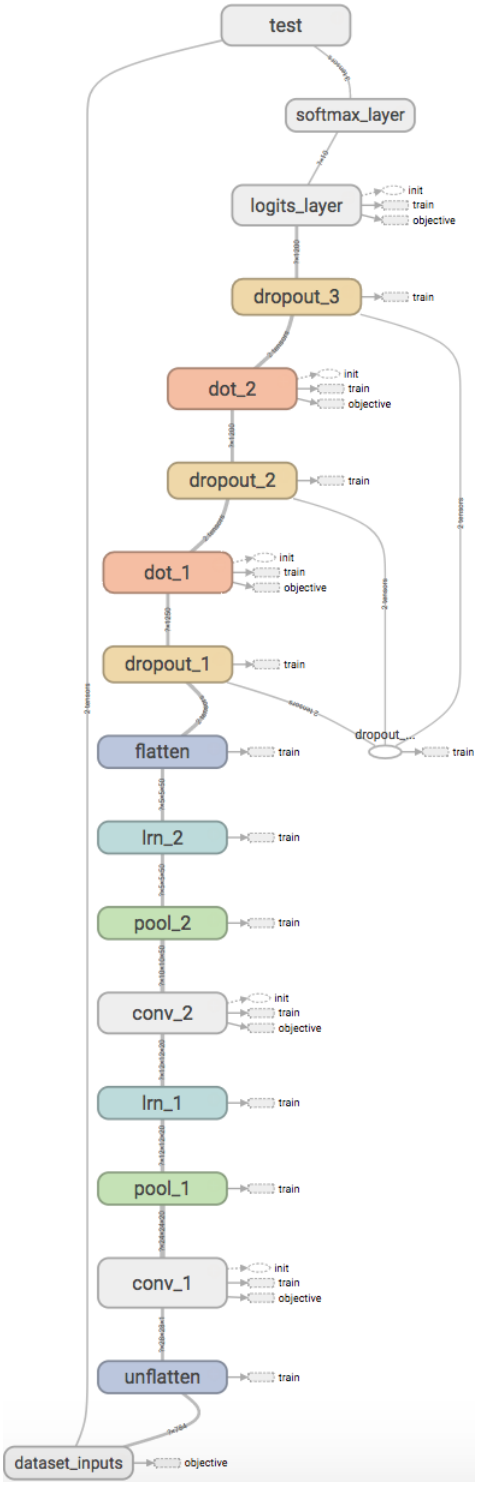


Fig. 1.11: Network visualized in tensorboard.

```
for param in var_list:
    norm = WEIGHT_DECAY_COEFF * tf.nn.l2_loss(param)
    tf.summary.scalar('l2_' + param.name, norm)
    tf.add_to_collection('objectives', norm)
```

Most of the methods used above are reminiscent of theano except for `tf.nn.l2_loss()`, which should also be obvious to understand.

The Overall objective of the network is,

$$o = \frac{1}{b}\sum_{i=1}^{n}\sum_{j=1}^{m} y_{ij}\log(l_{ij}) + l_1\sum|w| + l_2\sum||w||, \forall w \in \mathcal{N}.$$

This is essentially, the cross-entropy loss added with the weighted sum of $L1$ and $L2$ norms of all the weights in the network. Cumulatively the objective o can be calculated as follows:

```
self.obj = tf.add_n(tf.get_collection('objectives'), name='objective')
tf.summary.scalar('obj', self.obj)
```

Also, since we have an `self.obj`, we can then add an ADAM optimizer that minimizes the node.

```
back_prop = tf.train.AdamOptimizer( learning_rate = LR, name = 'adam' ).minimize(
                                                    loss = self.obj, var_list =␣
↪var_list)
```

In tensorflow, adding optimizer is as simple as that. In theano, we would have had to use `theano.tensor.grad()` method to extract gradients for each parameter and then write codes for weight updates and use `theano.function()` to create update rules. In tensorflow, we can create a `tf.train.Optimizer.minimize()` node that can be run in a `tf.Session()`, session, which will be covered in *lenet.trainer.trainer*. Similarly, we can do different optimizers.

With the optimizer is done, we are done with the training part of the network class. We can now move on to other nodes in the graph that could be used at inference time. We can create one node, which will create a flag for every correct predictions that the network is making using `tf.equal()`.

```
correct_predictions = tf.equal(self.predictions, tf.argmax(self.labels, 1), \
                                        name = 'correct_predictions')
```

We can then create one node, which will estimate accuracy and add it to summaries so we can actively monitor it.

```
self.accuracy = tf.reduce_mean(tf.cast(correct_predictions, tf.float32) , name =
↪'accuracy')
tf.summary.scalar('accuracy', self.accuracy)
```

Tensorflow provides a method for estimating confusion matrix, give labels. We can estimate labels from our one-hot labels, using the `tf.argmax()` method and create a `confusion` node. If we also reshape this into an image, we can then add this as an image to the tensorflow summary. This implies that we will be able to monitor it as an image visualization.

```
confusion = tf.confusion_matrix(tf.argmax(self.labels,1), self.predictions,
                                num_classes=C,
                                name='confusion')
confusion_image = tf.reshape( tf.cast( confusion, tf.float32),[1, C, C, 1])
tf.summary.image('confusion',confusion_image)
```

This concludes the network part of the computational graph. The cook method is described in *lenet.network.lenet5.cook()* and the entire class in *lenet.network.lenet5*.

Trainer

Todo

This part of the tutorial is currently being done.

The trainer is perhaps the module that is most unique to tensorflow and is most different from theano. Tensroflow uses `tf.Session()` to parse computational graphs unlike in theano where we'd use the `theano.function()` methods. For a detailed tutorial on how tensorflow processes and runs graphs, refer this page.

The *lenet.trainer.trainer* class takes as input an object of *lenet.network.lenet5* and *lenet.dataset.mnist*. After adding them as attributes, it then initializes a new tensorflow session to run the computational graph and initializes all the variables in the graph.

```
self.network = network
self.dataset = dataset
self.session = tf.InteractiveSession()
tf.global_variables_initializer().run()
```

The initializer class also calls the *lenet.trainer.trainer.summaries()* method that initializes the summary writer (`tf.summary.FileWriter()`) so that any processing on this computational graph could be monitored at tensorboard.

```
self.summary = tf.summary.merge_all()
self.tensorboard = tf.summary.FileWriter("tensorboard")
self.tensorboard.add_graph(self.session.graph)
```

The mnist example from `tf.examples.tutorials.mnist.input_data()` that we use here as `self.dataset` is written in such a way that given a `mini_batch_size`, we can easily query and retrieve the next batch as follows:

```
x, y = self.dataset.train.next_batch(mini_batch_size)
```

While in theano, we would use the `theano.function()` method to produce the function to run back prop updates, here we can use the minimizer that we created in *lenet.network.lenet5* (`self.network.back_prop` in *lenet.trainer.trainer*) to run one update step. We also want to collect (*fetch* is the tensorflow terminology) `self.network.obj` and `self.network.cost` (see definitions at *lenet.network.lenet5*) to be able to monitor the network training. All this can be done using the following code:

```
_, obj, cost  = self.session.run(
                    fetches = [self.network.back_prop, self.network.obj, self.network.
↪cost], \
                    feed_dict = {self.network.images:x, self.network.labels:y, \
                                 self.network.dropout_prob: DROPOUT_PROBABILITY})
```

This is similar to how we'd run a `theano.function()`. The `givens` operation which is used in theano to feed values to placeholders is now supplied here using `feed_dict` which takes in a dictionary, whose key, value pair is a node and its initialization value. Here we assign to `self.network.images` the images we just retrieved, to `self.network.labels` the `y` we just queried and to `self.network.dropout_prob` which was the node controlling the dropout Bernoulli probability, the gloabl defined dropout. We use this value of dropout, since this does back prop. If we were just feeding forward without updating the weights (such as during inference or test) we would not use this probability, instead we would use,

```
acc = self.session.run(self.network.accuracy,\
                       feed_dict = { self.network.images: x,
```

```
                                        self.network.labels: y,
                                        self.network.dropout_prob: 1.0} )
```

as was used in the *lenet.trainer.trainer.accuracy()*. The same *lenet.trainer.trainer.accuracy()* method with different placeholders could be used for testing and training accuracies.

```
# Testing
x = self.dataset.test.images
y = self.dataset.test.labels
acc = self.accuracy (images =x, labels = y)

# Training
x, y = self.dataset.train.next_batch(mini_batch_size)
acc = self.accuracy (images =x, labels = y)
```

After a desired number of iterations, we might want to update the tensorboard summaries or print out a cost to use for reference on how well we are training. We can use `self.session`, which is the same session previously used, to write out all summaries. This run call to session will write out everything we have added to the summaries along the way of building the network itself using our `self.tensorboard` writer.

```
x = self.dataset.test.images
y = self.dataset.test.labels
s = self.session.run(self.summary, feed_dict = {self.network.images: x,
                                                 self.network.labels: y,
                                                 self.network.dropout_prob: 1.0})
self.tensorboard.add_summary(s, iter)
```

The last thing we have to define is the the *lenet.trainer.trainer.train()* method. This method will run the training loops for the network that we have definied, taking in input arguments `iter= 10000`, `mini_batch_size = 500`, `update_after_iter = 1000`, `summarize = True`, with obviously named variables.

The trainer loop can be coded as:

```
# Iterate over iter
for it in range(iter):
    obj, cost = self.bp_step(mini_batch_size)  # Run a step of back prop minimizer
    if it % update_after_iter == 0:            # Check if it is time to flush out␣
↪summaries.
        train_acc = self.training_accuracy(mini_batch_size = 50000)
        acc = self.test()                      # Measure training and testing␣
↪accuracies.
        print(  " Iter " + str(it) +           # Print them on terminal.
                " Objective " + str(obj) +
                " Cost " + str(cost) +
                " Test Accuracy " + str(acc) +
                " Training Accuracy " + str(train_acc)
                )
        if summarize is True:                  # Write summaries to tensorflow
            self.write_summary(iter = it, mini_batch_size = mini_batch_size)
```

The above code essentially iterates over `iter` supplied to the method and runs one step of `self.network.back_prop` method, which we cooked in *lenet.network.lenet5.cook()*. If it was time to flush out summaries it does so. Finally, once the training is complete, we can call the *lenet.trainer.trainer.test()* method to produce testing accuracies.

```
acc = self.test()
print ("Final Test Accuracy: " + str(acc))
```

Since everything else, including the first layer filters and confusion matrices were all stored in summaries, they would have been adequately flushed out.

The trainer class documentation can be found in *Trainer*.

Run and Outputs

`main.py` essentially guides how to run the code. Firstly import all the modules we have created.

```
from lenet.trainer import trainer
from lenet.network import lenet5
from lenet.dataset import mnist
```

Begin by creating the dataset and its placeholders,

```
dataset = mnist()
```

Create the network object and cook it.

```
net = lenet5(images = dataset.images)
net.cook(labels = dataset.labels)
```

Create a trainer module that takes as input, the cooked network and the datafeed and then train it.

```
bp = trainer (net, dataset.feed)
bp.train()
```

If everything went correctly, the following output would have been produced:

```
Extracting data/train-images-idx3-ubyte.gz
Extracting data/train-labels-idx1-ubyte.gz
Extracting data/t10k-images-idx3-ubyte.gz
Extracting data/t10k-labels-idx1-ubyte.gz
.
.
.
.
name: GeForce GTX 1080
major: 6 minor: 1 memoryClockRate (GHz) 1.7335
pciBusID 0000:02:00.0
Total memory: 7.91GiB
Free memory: 7.80GiB
2017-08-15 19:58:04.465420: W tensorflow/stream_executor/cuda/cuda_driver.cc:485]␣
↪creating context when one is currently active; existing: 0x3eec930
2017-08-15 19:58:04.513975: I tensorflow/core/common_runtime/gpu/gpu_device.cc:887]␣
↪Found device 1 with properties:
name: NVS 310
major: 2 minor: 1 memoryClockRate (GHz) 1.046
pciBusID 0000:01:00.0
Total memory: 444.50MiB
Free memory: 90.00MiB
2017-08-15 19:58:04.514091: W tensorflow/stream_executor/cuda/cuda_driver.cc:485]␣
↪creating context when one is currently active; existing: 0x3ee8b70
```

```
2017-08-15 19:58:04.770913: I tensorflow/core/common_runtime/gpu/gpu_device.cc:887]␣
↪Found device 2 with properties:
name: GeForce GTX 1080
major: 6 minor: 1 memoryClockRate (GHz) 1.7335
pciBusID 0000:03:00.0
Total memory: 7.91GiB
Free memory: 7.80GiB
.
.
.
.
Iter 0 Objective 6.67137 Cost 2.39042 Test Accuracy 0.0892 Training Accuracy 0.09062
Iter 1000 Objective 0.784603 Cost 0.205023 Test Accuracy 0.9865 Training Accuracy 0.
↪98592
Iter 2000 Objective 0.707837 Cost 0.158198 Test Accuracy 0.9877 Training Accuracy 0.
↪98638
Iter 3000 Objective 0.658972 Cost 0.0991117 Test Accuracy 0.9877 Training Accuracy 0.
↪98734
Iter 4000 Objective 0.709337 Cost 0.138037 Test Accuracy 0.9882 Training Accuracy 0.
↪9889
Iter 5000 Objective 0.687822 Cost 0.115233 Test Accuracy 0.9862 Training Accuracy 0.
↪98782
Iter 6000 Objective 0.767473 Cost 0.192869 Test Accuracy 0.9863 Training Accuracy 0.
↪98504
Iter 7000 Objective 0.717531 Cost 0.138536 Test Accuracy 0.9875 Training Accuracy 0.
↪98738
Iter 8000 Objective 0.730901 Cost 0.161923 Test Accuracy 0.987 Training Accuracy 0.
↪98738
Iter 9000 Objective 0.692139 Cost 0.127491 Test Accuracy 0.9889 Training Accuracy 0.
↪98832
Final Test Accuracy: 0.9852
```

A few other prints that were unique to the systems are all skipped.

Tensorboard

The tensorboard that is created can be setup by running,

```
tensorboard --logdir=tensorboard
```

Open a browser and enter the address `0.0.0.0:6006`, this will open up tensorboard. The tensorboard will have the following sections that are populated:

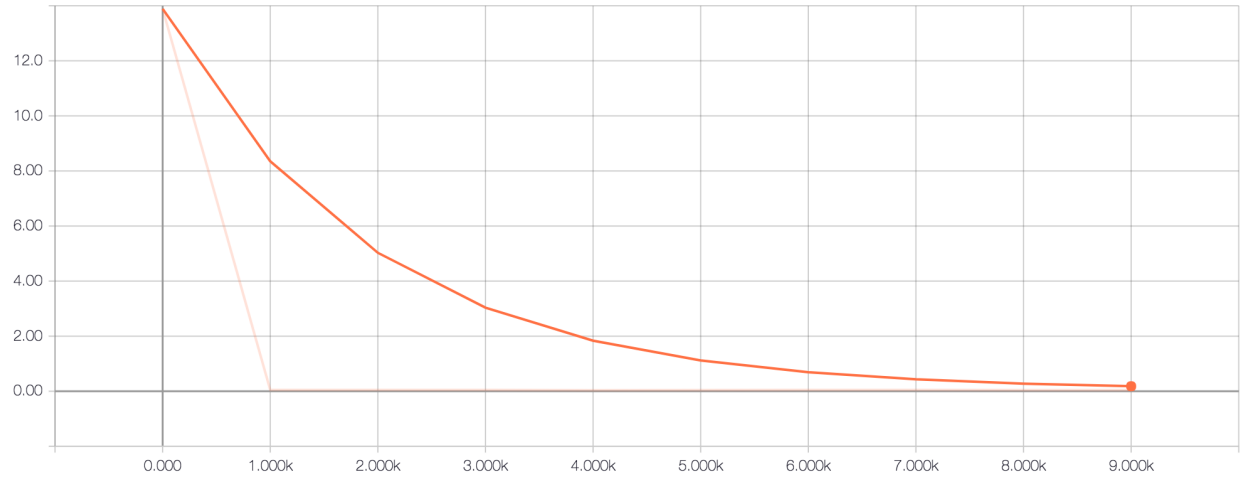
- Scalars
- Images
- Graphs
- Distributions
- Histograms

Let us go over a few of these sections, while leaving the others to the reader to interpret.

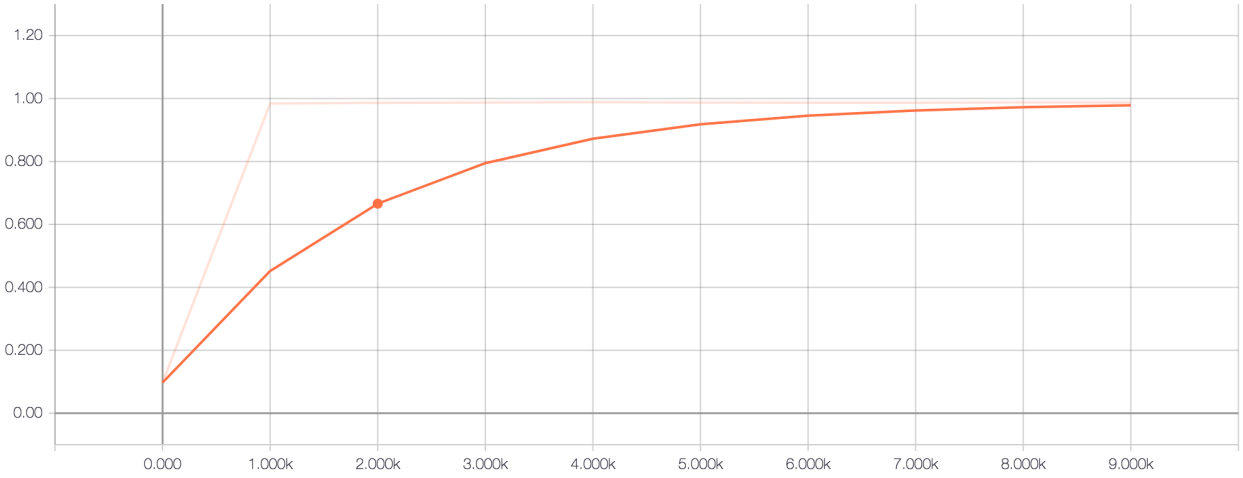
Scalars

In scalars we see all of the scalars that we were tracking in summary. Two of these that are important for us to observe are the costs going down and the test accuracies going up with iterations.

objective/cross-entropy/cost

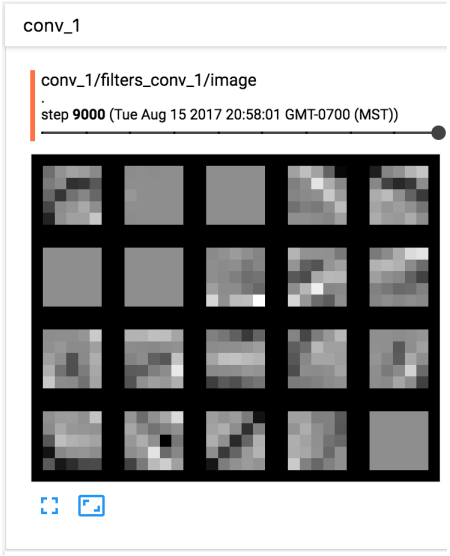


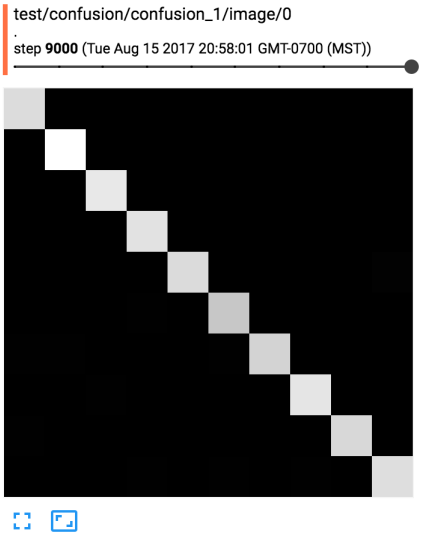
test/accuracy_1



Images

In the images section, we see the filters learnt by the first layer, the confusion matrix and the images that are used in training. Note that there are sliders in all these which could be used to scroll through various levels of information.





Graphs

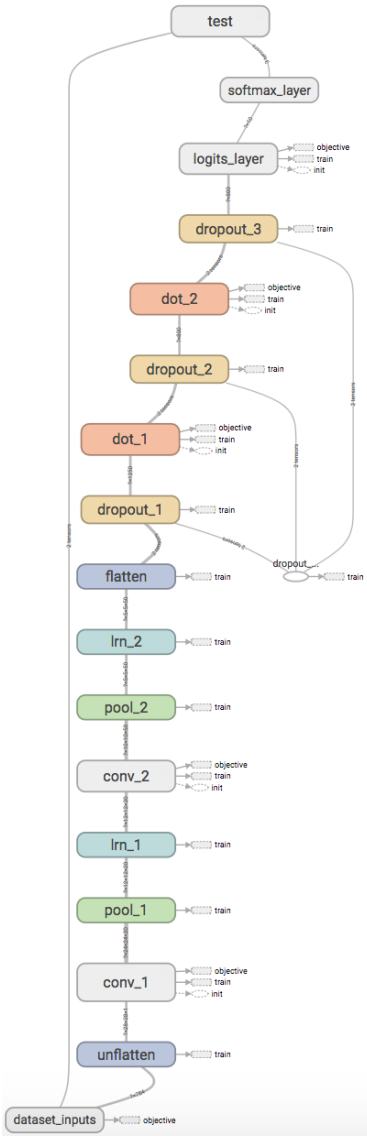
In the graphs section, we see the graph now with the training and other auxillary nodes present.

We can see here how the training modules are setup with gradients and how the objectives are derived from all the regularizers. Tensorboard is a nice utility and for a theano user it is a miracle come true. In fact, tensorboard is the primary and pretty much the only reason I am migrating to tensorflow.

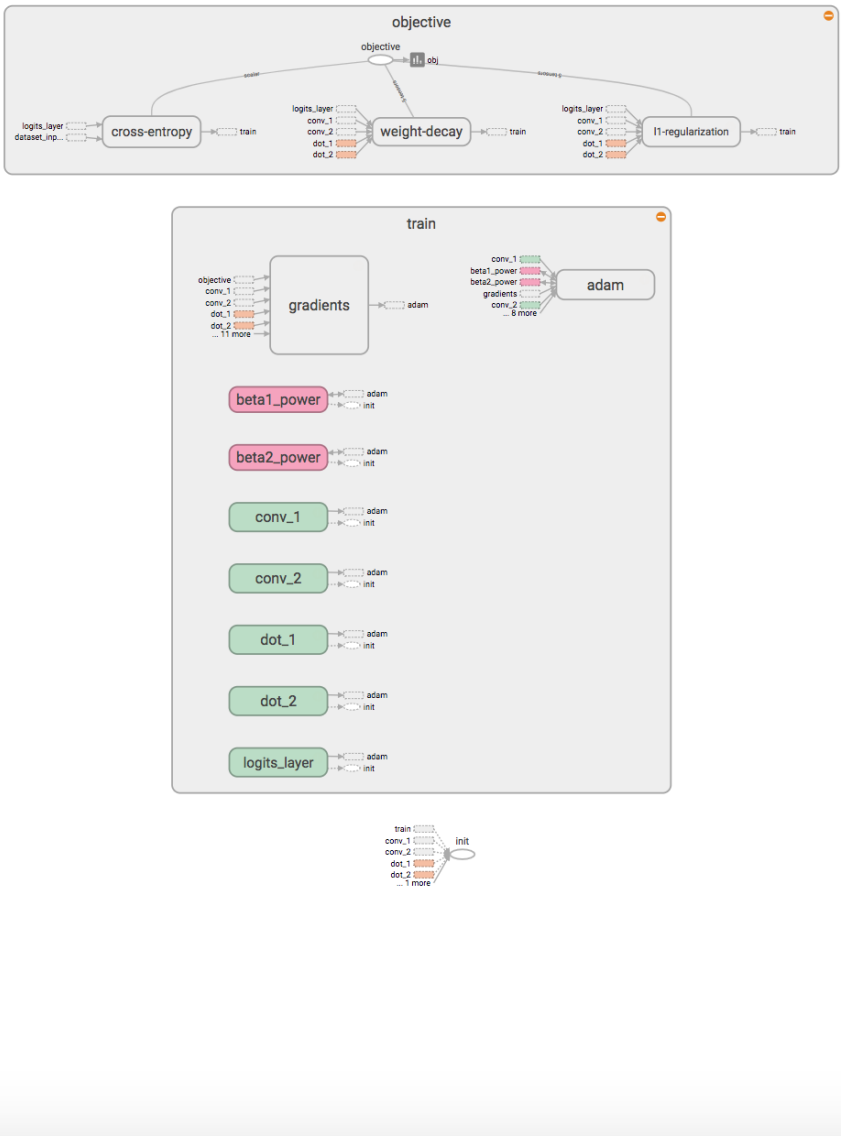
Additional Notes

Thanks for checking out my tutorial, I hope it was useful.

Main Graph



Auxiliary Nodes



Stochastic Gradient Descent

Since in theano we use the `theano.tensor.grad()` to estimate gradients from errors and back propagate ourselves, it was easier to understand and learn about various minimization algorithms. In yann for instance, I had to write all the optimizers I used, by hand as seen here. In tensorflow everything is already provided to us in the Optimizer module. I could have written out all the operations myself, but I copped out by using the in-built module. Therefore, here is some notes on how to minimize an error.

Consider the prediction $\hat{y}$. Consider we come up with some error e, that measure how different is $\hat{y}$ with y. In our tutorial, we used the categorical cross-entropy error. If e were a measure of error in this prediction, in order to learn any weight w in the network, we can acquire its gradient $\frac{\partial e}{\partial w}$, for every weight w in the network using the chain rule of differentiation. Once we have this error gradient, we can iteratively learn the weights using the following gradient descent update rule:

$$w^{\tau+1} = w^{\tau} - \eta \frac{\partial e}{\partial w},$$

where, η is some predefined rate of learning and τ is the iteration number. It can be clearly noticed in the back-prop strategy outlined above that the features of a CNN are learnt with only one objective - to minimize the prediction error. It is therefore to be expected that the features learnt thusly, are specific only to those particular tasks. Paradoxically, in deep CNNs trained using large datasets, this is often not the typical observation.

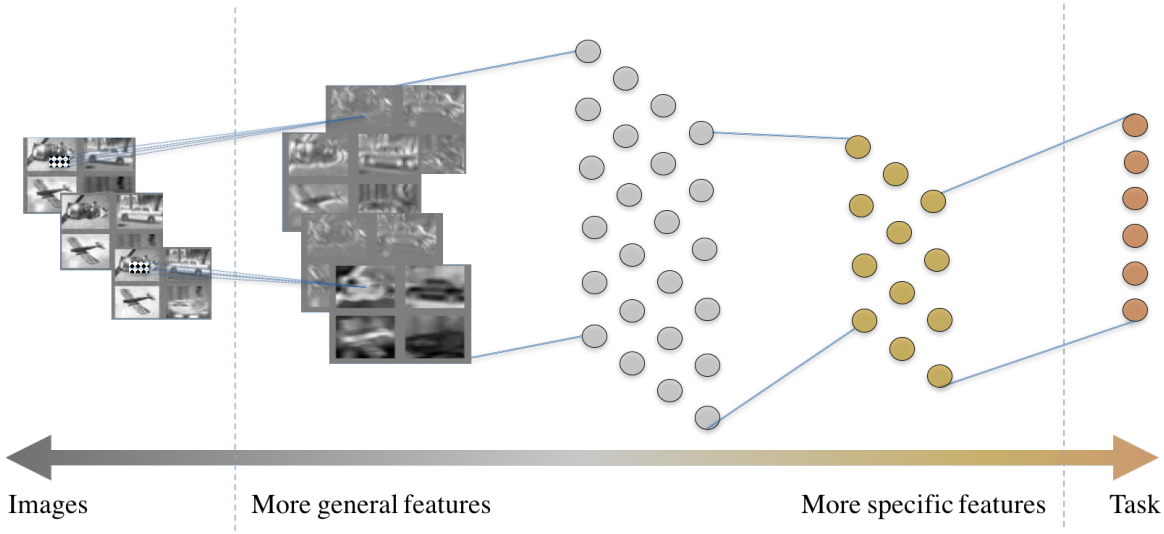


Fig. 1.12: The anatomy of a typical convolutional neural network.

In most CNNs, we observe as illustrated in the figure above, that the anatomy of the CNN and the ambition of each layer is contrived meticulously. The features that are close to the image layer are more general (such as edge detectors or Gabor filters) and those that are closer to the task layer are more task-specific.

Off-the-shelf Downloadable networks

Given this observation among most popular CNNs, modern day computer vision engineers prefer to simply *download* off-the-shelf neural networks and fine-tune it for their task. This process involves the following steps. Consider that a stable network N is well-trained on some task T that has a large dataset and compute available at site. Consider that the target for an engineer is build a network n that can make inferences on task t. Also assume that these two tasks are somehow related, perhaps T was visual object categorization on ImageNet and t on the COCO dataset *[LMB+14]* or the Caltech-101/256 datasets *[FFFP06]*. To learn the task on t, one could simply *download* N, randomly reinitialize

the last few layers (at the extreme case reinitialize only the softmax layer) and continue back-prop of the network to learn the task t with a smaller *eta*. It is expected that N is very well-trained already that they could serve as good initialization to begin *fine-tuning* the network for this new task. In some cases where enough compute is not available to update all the weights, we may simply choose to update only a few of the layers close to the task and leave the others as-is. The first few layers that are not updated are now treated as feature extractors much the same way as HOG or SIFT.

Some networks capitalized on this idea and created off-the-shelf downloadable networks that are designed specifically to work as feed-forward deterministic feature extractors. Popular off-the-shelf networks were the decaf *[DJV+14]* and the overfeat *[SEZ+13]*. Overfeat in particular was used for a wide-variety of tasks from pulmonary nodule detection in CT scans *[vGSJC15]* to detecting people in crowded scenes *[SAN16]*. While this has been shown to work to some degrees and has been used in practice constantly, it is not a perfect solution and some problems have known to exist. This problem is particularly striking when using a network trained on visual object categorization and fine-tuned for tasks in medical image.

Distillation from downloaded networks

Another concern with this philosophy of using off-the-shelf network as feature extractors is that the network n is also expected to be of the same size as N. In some cases, we might desire a network of a different architecture. One strategy to learn a different network using a pre-trained network is by using the idea of distillation *[HVD15]*, *[BRMW15]*. The idea of distillation works around the use of a temperature-raised softmax, defined as follows:

$$\begin{bmatrix} P(y=1|\mathbf{x},\Gamma) \\ \vdots \\ P(y=c|\mathbf{x},\Gamma) \end{bmatrix} = \frac{1}{\sum_{p=1}^{c} e^{\frac{w^p N'(\mathbf{x})}{\Gamma}}} \begin{bmatrix} e^{\frac{w^1 N'(\mathbf{x})}{\Gamma}} \\ \vdots \\ e^{\frac{w^c N'(\mathbf{x})}{\Gamma}} \end{bmatrix}$$

This temperature-raised softmax for $\Gamma > 1$ ($\Gamma = 1$ is simply the original softmax) provides a softer target which is smoother across the labels. It reduces the probability of the most probable label and provides rewards for the second and third most probable labels also, by equalizing the distribution. Using this *dark-knowledge* to create the errors e (in addition to the error over predictions as discussed above), knowledge can be transferred from N, during the training of n. This idea can be used to learn different types of networks. One can learn shallower *[VL16]* or deeper *[RBK+14]* networks through this kind of mentoring.

Bibiliography

CHAPTER 2

Code Documentation

This is an API detailing every method and class in this project. You can refer to all the tools using this API.

Dataset

This module contains all the dataset related classes.

class `lenet.dataset.`**`fashion_mnist`**(*dir='data/fashion'*)
Class for the fashion mnist objects. Ensure that data is downloaded from here

Parameters **`dir`** – Directory to cache at

`images`
This is the placeholder for images. This needs to be fed in using `feed_dict`.

`labels`
This is the placeholder for images. This needs to be fed in using `feed_dict`.

`feed`
This is a feeder from mnist tutorials of tensorflow. Use this for feeding in data.

class `lenet.dataset.`**`mnist`**(*dir='data'*)
Class for the mnist objects

Parameters **`dir`** – Directory to cache at

`images`
This is the placeholder for images. This needs to be fed in using `feed_dict`.

`labels`
This is the placeholder for images. This needs to be fed in using `feed_dict`.

`feed`
This is a feeder from mnist tutorials of tensorflow. Use this for feeding in data.

Layers

This module contains classes definitions for different types of layers.

lenet.layers.**conv_2d_layer**(*input*, *neurons=20*, *filter_size=(5, 5)*, *stride=(1, 1, 1, 1)*, *padding='VALID'*, *name='conv'*, *activation='relu'*, *visualize=False*)

Creates a convolution layer

Parameters

- **input** – (NHWC) Where is the input of the layer coming from
- **neurons** – Number of neurons in the layer.
- **name** – name scope of the layer
- **filter_size** – A tuple of filter size `(5,5)` is default.
- **stride** – A tuple of x and y axis strides. `(1,1,1,1)` is default.
- **name** – A name for the scope of tensorflow
- **visualize** – If True, will add to summary. Only for first layer at the moment.
- **activation** – Activation for the outputs.
- **padding** – Padding to be used in convolution. "VALID" is default.

Returns The output node and A list of parameters that are learnable

Return type tuple

lenet.layers.**dot_product_layer**(*input*, *params=None*, *neurons=1200*, *name='fc'*, *activation='relu'*)

Creates a fully connected layer

Parameters

- **input** – Where is the input of the layer coming from
- **neurons** – Number of neurons in the layer.
- **params** – List of tensors, if supplied will use those params.
- **name** – name scope of the layer
- **activation** – What kind of activation to use.

Returns The output node and A list of parameters that are learnable

Return type tuple

lenet.layers.**dropout_layer**(*input*, *prob*, *name='dropout'*)

This layer drops out nodes with the probability of 0.5 During training time, run a probability of 0.5. During test time run a probability of 1.0. To do this, ensure that the `prob` is a `tf.placeholder`. You can supply this probability with `feed_dict` in trainer.

Parameters

- **input** – a 2D node.
- **prob** – Probability feeder.
- **name** – name scope of the layer.

Returns An output node

Return type tensorflow tensor

lenet.layers.**flatten_layer**(*input*, *name='flatten'*)
This layer returns the flattened output :param input: a 4D node. :param name: name scope of the layer.

Returns a 2D node.

Return type tensorflow tensor

lenet.layers.**local_response_normalization_layer**(*input*, *name='lrn'*)
This layer returns the flattened output

Parameters

- **input** – a 4D node.
- **name** – name scope of the layer.

Returns a 2D node.

Return type tensorflow tensor

lenet.layers.**max_pool_2d_layer**(*input*, *pool_size=(1, 2, 2, 1)*, *stride=(1, 2, 2, 1)*, *padding='VALID'*, *name='pool'*)
Creates a max pooling layer

Parameters

- **input** – (NHWC) Where is the input of the layer coming from
- **name** – name scope of the layer
- **pool_size** – A tuple of filter size `(5,5)` is default.
- **stride** – A tuple of x and y axis strides. `(1,1,1,1)` is default.
- **name** – A name for the scope of tensorflow
- **padding** – Padding to be used in convolution. "VALID" is default.

Returns The output node

Return type tensorflow tensor

lenet.layers.**softmax_layer**(*input*, *name='softmax'*)
Creates the softmax normalization

Parameters

- **input** – Where is the input of the layer coming from
- **name** – Name scope of the layer

Returns `(softmax, prediction)`, A softmax output node and prediction output node

Return type tuple

lenet.layers.**unflatten_layer**(*input*, *channels=1*, *name='unflatten'*)
This layer returns the unflattened output :param input: a 2D node. :param chanels: How many channels are there in the image. (Default = `1`) :param name: name scope of the layer.

Returns a 4D node in (NHWC) format that is square in shape.

Return type tensorflow tensor

Network

This module contains the class for lenet. This contains all the architecture design.

lenet.network.**apply_adam**(*var_list*, *obj*, *learning_rate=0.0001*)
Sets up the ADAM optimizer

Parameters

- **var_list** – List of variables to optimizer over.
- **obj** – Node of the objective to minimize

Notes

learning_rate: What learning rate to run with. (Default = `0.01`) Set with `LR`

lenet.network.**apply_gradient_descent**(*var_list*, *obj*)
Sets up the gradient descent optimizer

Parameters

- **var_list** – List of variables to optimizer over.
- **obj** – Node of the objective to minimize

Notes

learning_rate: What learning rate to run with. (Default = `0.01`) Set with `LR`

lenet.network.**apply_l1**(*var_list*, *name='l1'*)
This method applies L1 Regularization to all weights and adds it to the `objectives` collection.

Parameters

- **var_list** – List of variables to apply l1
- **name** – For the tensorflow scope.

Notes

What is the co-efficient of the L1 weight? Set `L1_COEFF`.(Default = 0.0001)

lenet.network.**apply_regularizer**(*var_list*)
This method applyies Regularization to all weights and adds it to the `objectives` collection.

Parameters **var_list** – List of variables to apply l1

Notes

What is the co-efficient of the L1 weight? Set `L1_COEFF`.(Default = 0.0001)

lenet.network.**apply_rmsprop**(*var_list*, *obj*)
Sets up the RMS Prop optimizer

Parameters

- **var_list** – List of variables to optimizer over.

- **obj** – Node of the objective to minimize

Notes

•learning_rate: What learning rate to run with. (Default = `0.001`). Set `LR`

•momentum: What is the weight for momentum to run with. (Default = `0.7`). Set `MOMENTUM`

•decay: What rate should learning rate decay. (Default = `0.95`). Set `DECAY`

`lenet.network.`**apply_weight_decay**(*var_list*, *name='weight_decay'*)
This method applies L2 Regularization to all weights and adds it to the `objectives` collection.

Parameters

- **name** – For the tensorflow scope.
- **var_list** – List of variables to apply.

Notes

What is the co-efficient of the L2 weight? Set `WEIGHT_DECAY_COEFF`.(Default = 0.0001)

class `lenet.network.`**lenet5**(*images*)
Definition of the lenet class of networks.

Notes

•Produces the lenet model and returns the weights. A typical lenet has two convolutional layers with filters sizes `5X5` and `3X3`. These are followed by two fully-connected layers and a softmax layer. This network model, reproduces this network to be trained on MNIST images of size `28X28`.

•Most of the important parameters are stored in `global_definitions` in the file `global_definitions.py`.

Parameters **images** – Placeholder for images

images
This is the placeholder for images. This needs to be fed in from `lenet.dataset.mnist`.

dropout_prob
This is also a placeholder for dropout probability. This needs to be fed in.

logits
Output node of the softmax layer, before softmax. This is an output from a *lenet.layers.dot_product_layer()*.

inference
Output node of the softmax layer that produces inference.

predictions
Its a predictions node which is `tf.nn.argmax()` of `inference`.

back_prop
Backprop is an optimizer. This is a node that will be used by a *lenet.trainer.trainer* later.

obj
Is a cumulative objective tensor. This produces the total summer objective in a node.

cost
Cost of the back prop error alone.

labels
Placeholder for labels, needs to be fed in. This is added fed in from the dataset class.

accuracy
Tensor for accuracy. This is a node that measures the accuracy for the mini batch.

cook (*labels*)
Prepares the network for training

Parameters **labels** – placeholder for labels

Notes

- Each optimizer has a lot parameters that, if you want to change, modify in the code directly. Most do not take in inputs and runs. Some parameters such as learning rates play a significant role in learning and are good choices to experiment with.
- what optimizer to run with. (Default = `sgd`), other options include 'rmsprop' and 'adam'. Set `OPTIMIZER`

`lenet.network.`**process_params** (*params*)
This method adds the params to two collections. The first element is added to `regularizer_worthy_params`. The first and second elements are is added to `trainable_parmas`.

Parameters **params** – List of two.

Support

This module contains additional code that I used from outside in support of this. These are typically outside of the neural network, but used in filter visualization and similar.

`lenet.support.`**initializer** (*shape*, *name='xavier'*)
A method that returns random numbers for Xavier initialization.

Parameters

- **shape** – shape of the initializer.
- **name** – Name for the scope of the initializer

Returns random numbers from tensorflow random_normal

Return type float

`lenet.support.`**nhwc2hwcn** (*nhwc*, *name='nhwc2hwcn'*)
This method reshapes (NHWC) 4D bock to (HWCN) 4D block

Parameters **nhwc** – 4D block in (NHWC) format

Returns 4D block in (HWCN) format

Return type tensorflow tensor

lenet.support.**nhwc2hwnc**(*nhwc*, *name='nhwc2hwnc'*)
This method reshapes (NHWC) 4D bock to (HWNC) 4D block

Parameters **nhwc** – 4D block in (NHWC) format

Returns 4D block in (HWNC) format

Return type tensorflow tensor

lenet.support.**visualize_filters**(*filters*, *name='conv_filters'*)
This method is a wrapper to put_kernels_on_grid. This adds the grid to image summaries.

Parameters **tensor** (*tensorflow*) – A 4D block in (HWNC) format.

lenet.support.**visualize_images**(*images*, *name='images'*, *num_images=6*)
This method sets up summaries for images.

Parameters

- **images** – a 4D block in (NHWC) format.
- **num_images** – Number of images to display

Todo

I want this to display images in a grid rather than just display using tensorboard's ugly system. This method should be a wrapper that converts images in (NHWC) format to (HWNC) format and makes a grid of the images.

Perhaps a code like this:

‘‘‘ images = images [0:num_images-1] images = nhwc2hwcn(images, name = ‘nhwc2hwcn’ + name) visualize_filters(images, name)

Third Party

This module contains third party code that I used from outside in support of this. This file contains third party support code that I have used from elsewhere. I have cited them appropriately where ever needed

lenet.third_party.**put_kernels_on_grid**(*kernel*, *pad=1*, *name='visualizer'*)
Visualize convolutional filters as an image (mostly for the 1st layer). Arranges filters into a grid, with some paddings between adjacent filters.

Parameters

- **kernel** – tensor of shape [Y, X, NumChannels, NumKernels] (HWCN)
- **pad** – number of black pixels around each filter (between them)
- **name** – name for tensorflow scope

Returns Tensor of shape [1, (Y+2*pad)*grid_Y, (X+2*pad)*grid_X, NumChannels].

Notes

This is not my method. This was written by kukurza and was hosted at: https://gist.github.com/kukuruza/03731dc494603ceab0c5

Trainer

This module contains the trainer code and other codes regarding training.

class lenet.trainer.**trainer**(*network*, *dataset*)
Trainer for networks

Parameters

- **network** – A network class object
- **dataset** – A tensorflow dataset object

network
This is the network we initialized with. We pass this as an argument and we add it to the current trainer class.

dataset
This is also the initializer. It comes from the *lenet.dataset.mnist* module.

session
This is a session created with trainer. This session is used for training.

tensorboard
Is a summary writer tool. This writes things into the tensorboard that is then setup on the tensorboard server. At the end of the trainer, it closes this tensorboard.

accuracy(*images*, *labels*)
Return accuracy

Parameters

- **images** – images
- **labels** – labels

Returns accuracy

Return type float

bp_step(*mini_batch_size*)
Sample a minibatch of data and run one step of BP.

Parameters **mini_batch_size** – Integer

Returns total objective and cost of that step

Return type tuple of tensors

summaries(*name='tensorboard'*)
Just creates a summary merge bufer

Parameters **name** – a name for the tensorboard directory

test()
Run validation of the model

Returns accuracy

Return type float

train(*iter=10000*, *mini_batch_size=500*, *update_after_iter=1000*, *training_accuracy=False*, *summarize=True*)
Run backprop for iter iterations

Parameters

- **iter** – number of iterations to run
- **mini_batch_size** – Size of the mini batch to process with
- **training_accuracy** – if `True`, will calculate accuracy on training data also.
- **update_after_iter** – This is the iteration for validation
- **summarize** – Tensorboard operation

training_accuracy(*mini_batch_size=500*)
Run validation of the model on training set

Parameters **mini_batch_size** – Number of samples in a mini batch

Returns accuracy

Return type float

write_summary(*iter=0*, *mini_batch_size=500*)
This method updates the summaries

Parameters

- **iter** – iteration number to index values with.
- **mini_batch_size** – Mini batch to evaluate on.

CHAPTER 3

License

The MIT License

TF-Lenet Copyright (c) [2017] Ragav Venkatesan

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice, the credits below and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Credits and copyright attributions to other sources:

- To kukurza for the rasterizing code that hosted on their gist

Bibliography

[BRMW15] Anoop Korattikara Balan, Vivek Rathod, Kevin P Murphy, and Max Welling. Bayesian dark knowledge. In *Advances in Neural Information Processing Systems*, 3438–3446. 2015.

[BSF94] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[DT05] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, 886–893. IEEE, 2005.

[DDS+09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: a large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, 248–255. IEEE, 2009.

[DWSP12] Piotr Dollar, Christian Wojek, Bernt Schiele, and Pietro Perona. Pedestrian detection: an evaluation of the state of the art. *IEEE transactions on pattern analysis and machine intelligence*, 34(4):743–761, 2012.

[DJV+14] Jeff Donahue, Yangqing Jia, Oriol Vinyals, Judy Hoffman, Ning Zhang, Eric Tzeng, and Trevor Darrell. Decaf: a deep convolutional activation feature for generic visual recognition. In *International conference on machine learning*, 647–655. 2014.

[FFFP06] Li Fei-Fei, Rob Fergus, and Pietro Perona. One-shot learning of object categories. *IEEE transactions on pattern analysis and machine intelligence*, 28(4):594–611, 2006.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778. 2016.

[HVD15] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[KS04] Yan Ke and Rahul Sukthankar. Pca-sift: a more distinctive representation for local image descriptors. In *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, volume 2, II–II. IEEE, 2004.

[KHD11] Aniruddha Kembhavi, David Harwood, and Larry S Davis. Vehicle detection using partial least squares. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(6):1250–1265, 2011.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 1097–1105. 2012.

[LBD+90] Yann LeCun, Bernhard E Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne E Hubbard, and Lawrence D Jackel. Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems*, 396–404. 1990.

[LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[LMB+14] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: common objects in context. In *European conference on computer vision*, 740–755. Springer, 2014.

[LSD15] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 3431–3440. 2015.

[Low99] David G Lowe. Object recognition from local scale-invariant features. In *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, volume 2, 1150–1157. Ieee, 1999.

[LMT+07] Jun Luo, Yong Ma, Erina Takikawa, Shihong Lao, Masato Kawade, and Bao-Liang Lu. Person-specific sift features for face recognition. In *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*, volume 2, II–593. IEEE, 2007.

[LS08] Siwei Lyu and Eero P Simoncelli. Nonlinear image representation using divisive normalization. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, 1–8. IEEE, 2008.

[OL13] Omar Oreifej and Zicheng Liu. Hon4d: histogram of oriented 4d normals for activity recognition from depth sequences. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 716–723. 2013.

[RBK+14] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. Fitnets: hints for thin deep nets. *arXiv preprint arXiv:1412.6550*, 2014.

[RHW85] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical Report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[SEZ+13] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229*, 2013.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[SHK+14] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, 15(1):1929–1958, 2014.

[SGS15] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.

[SAN16] Russell Stewart, Mykhaylo Andriluka, and Andrew Y. Ng. End-to-end people detection in crowded scenes. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016.

[SMYC10] Ju Sun, Yadong Mu, Shuicheng Yan, and Loong-Fah Cheong. Activity recognition using dense long-duration trajectories. In *Multimedia and Expo (ICME), 2010 IEEE International Conference on*, 322–327. IEEE, 2010.

[SBM06] Zehang Sun, George Bebis, and Ronald Miller. On-road vehicle detection: a review. *IEEE transactions on pattern analysis and machine intelligence*, 28(5):694–711, 2006.

[SIVA17] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, 4278–4284. 2017.

[SLJ+15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 1–9. 2015.

[SVI+16] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2818–2826. 2016.

[vGSJC15] Bram van Ginneken, Arnaud AA Setio, Colin Jacobs, and Francesco Ciompi. Off-the-shelf convolutional neural network features for pulmonary nodule detection in computed tomography scans. In *Biomedical Imaging (ISBI), 2015 IEEE 12th International Symposium on*, 286–289. IEEE, 2015.

[VCLL12] Ragav Venkatesan, Parag Chandakkar, Baoxin Li, and Helen K Li. Classification of diabetic retinopathy images using multi-class multiple-instance learning based on color correlogram features. In *Engineering in Medicine and Biology Society (EMBC), 2012 Annual International Conference of the IEEE*, 1462–1465. IEEE, 2012.

[VL16] Ragav Venkatesan and Baoxin Li. Diving deeper into mentee networks. 2016.

[VLBM08] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, 1096–1103. ACM, 2008.

[WOC+07] Jianxin Wu, Adebola Osuntogun, Tanzeem Choudhury, Matthai Philipose, and James M Rehg. A scalable approach to activity recognition based on object use. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, 1–8. IEEE, 2007.

[ZvdM13] Lu Zhang and Laurens van der Maaten. Structure preserving object tracking. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 1838–1845. 2013.

[ZYS09] Huiyu Zhou, Yuan Yuan, and Chunmei Shi. Object tracking using sift features and mean shift. *Computer vision and image understanding*, 113(3):345–352, 2009.

[ZYCA06] Qiang Zhu, Mei-Chen Yeh, Kwang-Ting Cheng, and Shai Avidan. Fast human detection using a cascade of histograms of oriented gradients. In *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, volume 2, 1491–1498. IEEE, 2006.

Python Module Index

l

Index

U

V

W