Image Classification with Convolutional Networks & TensorFlow

Josephine Sullivan

August 9, 2017

1 Preliminaries

1.1 Which development environment?

There are multiple software packages available for performing deep learning research. The most prominent, and active are:

- TensorFlow
- Theano
- Torch
- Caffe

Your choice of which one to use really depends on what type of research (novel deep learning algorithms or exploit existing techniques/networks) you want to perform, which networks you will use (convolutional networks or RNNs), your type of input data and which programming languages you have experience with. The lecture Deep Learning Software has a good summary of the pros and cons of the packages mentioned. Remember though all the packages use NVIDIA's GPU library *cuDNN* for the GPU implementation of the basic deep learning operations. Thus the speed of the GPU enabled versions for training and tesing of all packages are roughly similar.
One good option if you want to build networks quickly is to use the high-level library Keras: The Python Deep Learning library. With this package you also have the flexibility to use either TensorFlow or Theano. Of course, the ease of use comes at the cost of the loss of some control, flexibility and transparency.

1.2 What GPU hardware?

If you do decide to seriously persue deep learning within your research, then the most important factor in how far you get, will not be the software package you use, but the GPU card you have access to for training and testing.

The faster the turn around time between developing code, algorithms and completing experiments, then the faster your progress will be. At the moment the most cost effective solution is to buy a relatively high end GPU card, see the webpage Which GPU(s) to Get for Deep Learning: My Experience and Advice for Using GPUs in Deep Learning for a review and advice about getting the best bang for your buck, as opposed to paying for GPU cycles via cloud services. Also you should be aware that Nvidia has a scheme to donate GPU cards to academic researchers see the the website GPU Grant Program for details.

1.3 Preliminaries for this tutorial

In this practical you will investigate multi-class image classification with fully connected and convolutional networks. We will use the software package TensorFlow and the cloud service FloydHub and the CPU on your labtop as our development environment. We are using FloydHub mainly because it's free (for the first 100 hours of GPU usage or at least it was until the start of this week, now it's down to the first 2 hours) and relatively simple to use! It will give you a feel of how much quicker training becomes (especially for convolutional neural networks) when you use a decent(ish) GPU card (K80) as opposed to the CPU for training.

1.3.1 Install TensorFlow

First you should install TensorFlow on your machine. I would highly recommend the virtualenv installation as it is relatively painless and fast. For the details on how to do this follow the instructions on the official Installing TensorFlow webpage.

1.3.2 Set up FloydHub

To use FloydHub you first have to create an account for yourself. You can do this by visiting the webpage FloydHub and clicking on the "Start Free Trial" button and following the instructions. When I signed up the confirmation e-mail sent to my e-mail address went to my spam folder.
Once you have your FloydHub account you should install Floyd CLI a *python* based command line tool to interact with FloydHub from your terminal. Once again I recommend using `virtualenv` for installing and using `floyd-cli`. The instructions of how to do this are available at the official Installation webpage.

1.3.3 Set up your environment

Create a new directory to contain the *python* files and dataset you will write for this practical:

```
$ mkdir DirName
$ cd DirName
$ mkdir Datasets
```

1.3.4 Download the Image Database: CIFAR-10

Download the CIFAR-10 dataset stored in its *python* format from this link. Move the `cifar-10-python.tar.gz` file to the `Datasets` directory you have just created, `untar` the file and then move up to the parent directory. Also download the file `read_cifar10.py` from the tutorial website and move it to `DirName`.

```
$ mv read_cifar10.py DirName/
$ mv cifar-10-python.tar.gz DirName/Datasets
$ cd DirName/Datasets
$ tar xvfz cifar-10-python.tar.gz
```

The `CIFAR-10` dataset is now in the directory `DirName/Datasets/cifar-10-batches-py/`. The dataset has 5 batches for training and validation and a test batch. Each batch contains 10,000 labelled images. Each image has a single label and there are 10 different labels. The file `read_cifar10.py` contains functions to read in the dataset into `numpy` arrays. But more about this later...

2 TensorFlow the basics

TensorFlow separates the definition of your computations from their execution. Therefore, there are two distinct phases to any TensorFlow programme you write. The first phase is where you **define your computation graph** and the second is where you define a **session** to execute operations based on the specified computation graph.
For this practical I will assume that you have a virtualenv installation of TensorFlow. The *python* commands I specify will be the bare bones needed. I'm sure most of you are much more proficient *python* coders than me, so please do follow your normal best coding practices when we begin to write more involved code. For a much detailed to guide to TensorFlow you should check out the Stanford course TensorFlow for Deep Learning Research. Most of the following section is a summary of the necessary highlights from the notes from this course.

2.1 My first TensorFlow programme

We will now write our first TensorFlow programme which adds two numbers together. In your favourite text editor create the file `example0.py` with the content.

```
import tensorflow as tf
x = 3
y = 5
a = tf.add(x, y)
print(a)
```

```
with tf.Session() as sess:
   print(sess.run(a))
```

Next start the virtual environment for TensorFlow:

```
$ source tensorflowDir/tensorflow/bin/activate
```

Within the virtual environment you can run your new function with:

```
(tensorflow) $ python example0.py
```

You may get a few warning messages that you haven't compiled TensorFlow with all the possible optimizations for your machine. After these messages your programme will print out

```
Tensor("Add:0", shape=(), dtype=int32)
8
```

The first line is information regarding the object `a` and the second line is the actual value of `a`. In `example0.py` the data graph you define is displayed in figure 2.1.1 and a session is then created to evaluate (the portion of) the

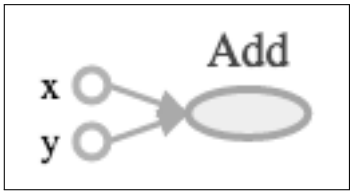

Figure 1: Graph associated with the programme `example0.py`

graph computations needed to calculate `a`.

2.1.1 Visualize your graph with TensorBoard

TensorFlow has a great tool for visualizing your assembled graph plus other values that you calculate during your session. We can extend our simple example to use TensorBoard to allow us to display the assembled graph. You have to create a *summary writer* after the graph definition and before running the session. We can do this by updating the code in `example0.py` with the lines in red.

```
import tensorflow as tf
x = 3
y = 5
a = tf.add(x, y)
print(a)
with tf.Session() as sess:
   writer = tf.summary.FileWriter('./graphs', sess.graph)
   print(sess.run(a))
writer.close()
```

Then go to your terminal and run

```
(tensorflow) $ python network1.py
(tensorflow) $ tensorboard --logdir="./graphs" --port 6006
```

Open your browser to the page `http://localhost:6006/` and you should then click on the "Graphs" option to see the graph shown in figure . You should learn to use TensorBoard well as it will help a lot when you build complicated models. However, to make complicated graphs viewable you'll have to give your Variables and constants names.

Exercise 1: *Your turn - Have you grasped the simple stuff?*

Here is another simple TensorFlow programme (note we changed the definition of `x` and `y`so that they are now constants and have explicitly named them):

```
import tensorflow as tf
x = tf.constant(2, name='x')
y = tf.constant(3, name='y')
op1 = tf.add(x, y)
op2 = tf.mul(x, y)
op3 = tf.pow(op2, op1)
with tf.Session() as sess:
   print(sess.run(op3))
```

Before running the code (and potentially using TensorBoard to visualize the specified graph) you should answer the following questions:

- What value will this programme print out?
- Draw the computational graph defined by this programme.

2.2 TensorFlow ops

In TensorFlow all constants, variables, and operators are referred to as ops. Please refer to the document Lecture note 2: TensorFlow Ops from the course *CS 20SI: TensorFlow for Deep Learning Research* for a more comprhensive overview of the possible ops. There is, of course, explicit documentation at the TensorFlow homepage. In this document I will just give a very brief overview (summarizing the important aspects of *Lecture note 2: TensorFlow Ops*) of the ops you need to start specifying and building simple neural networks.

2.2.1 Declaring constant types

There are multiple ways to create constant scalar or tensor values:

- The basic command

```
tf.constant(value, dtype=None, shape=None, name='Const',
verify_shape=False)
# Examples
a = tf.constant([2, 2], name="a")
b = tf.constant([[0, 1], [2, 3]], name="b")
```

- Create tensors whose elements are of a specific value similar to the commands `numpy.zeros`, `numpy.zeros_like`, `numpy.ones`, `numpy.ones_like`

```
a = tf.zeros([2, 3], tf.int32)  ⟹  a = [[0,0,0], [0,0,0]]
a = tf.fill([2, 3], 8)  ⟹  a = [[8, 8, 8], [8, 8, 8]]
```

- Create constants that are sequences

```
a = tf.linspace(10.0, 13.0, 4)  ⟹  a = [10.0 11.0 12.0 13.0]
a = tf.range(3, 18, 3)  ⟹  a = [3, 6, 9, 12, 15]
```

- Create random constants from certain distributions

```
tf.random_normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32,
seed=None, name=None)
```

2.2.2 Maths Operations

TensorFlow's mathematics ops are pretty standard and similar to NumPy. Here's a quick example:

```
a = tf.constant([3, 6])
b = tf.constant([2, 2])
c = tf.add(a, b)  ⟹  c = [5 8]
```

Visit TensorFlow Math operations for the specific details of the arthimetic, basic math functions and matrix operations available and how they are implemented and defined (applied elementwise etc.).

2.2.3 Data Types

TensorFlows data types are based on those of NumPy. The page Tensor types is the official documentation for the defined types of TensorFlow tensors.

2.2.4 Variables

You use Variables to hold and update parameters. Variables are in-memory buffers containing tensors. They must be explicitly initialized before using them, They can be updated during training and can be saved to disk during and after training. You can later restore saved values to use or analyze the network. When the variable is created, 3 ops are added to a graph: variable op, initializer op, and ops for the initial value. (`tf.Variable` is a class hence the uppercase).

Declaring variables here are a couple examples of how to declare variables

```
# create a variable "a" with a scalar value
a = tf.Variable(2)

# create a variable "b" as a 1d tensor
b = tf.Variable([2,3])

# create a variable "c" as a 2d tensor
c = tf.Variable([[0, 1], [2, 3]])

# create a variable "W" as a 784 × 10 tensor.  filled with zeros
W = tf.Variable(tf.zeros([784, 10]))
```

How to initialize variables before using them. The easiest way to initialize all variables in your graph at once is:

```
init = tf.global_variables_initializer()
with tf.Session() as sess:
   tf.run(init)
```

Evaluate values of variables To get the value of a variable, you need to evaluate it using `eval()`, if you just call `print(W)` then you will only see the tensor object.

```
W = tf.Variable(tf.truncated_normal([700, 10]))
with tf.Session() as sess:
  # just initialize the one Variable in the graph
  sess.run (W.initializer)
  print(W)
  print(W.eval())
```

2.2.5 Control dependencies

Sometimes you will have multiple indepedent operations and you would like to specify which op should be run first then you use `tf.control_dependencies(control_inputs)`. Example:

```
# your default graph has 4 ops:  a, b, c, d
a = ...
b = ...
tf.control_dependencies([a,b]):
  # operations c and d will only run after a and b have executed
  c = ...
  d = ...
```

You need this type of dependency when implementing batch normalization.

2.2.6 Placeholders and `feed_dict`

In TensorFlow you can specify a graph without knowing all the values of the variables or constants needed for its computation. In the definition of graph you can declare these constant and variables as placeholders and you pass in their actual values at run time. This `tf.placeholder` is a function. Typically, the variables corresponding to the input data and its ground truth labels are declared as placeholders. To define a placeholder use

```
tf.placeholder(dtype, shape=None, name=None)
```

In summary the basic properties of placeholders in TensorFlow are that you can't update a placeholder. They should not be initialized, but because they are a promise to have a tensor, you need to feed a value into them at run time i.e. `sess.run(<op>, a:  <some_val>)`. In comparison to a Variable, a placeholder might not know its shape beforehand. You can either provide parts of the dimensions or provide nothing at all. The latter is obviously dangerous as you may feed it a vale that is not compatible with the Variables it interacts with.
Here is an example of how a placeholder is used

```
# create placeholder of type float 32-bit, shape:  vector of 3 elements
a = tf.placeholder(tf.float32, shape=[3])
# create constant of type float 32-bit, shape:  vector of 3 elements
b = tf.constant([5, 5, 5], tf.float32)
# use placeholder as you would a constant or a variable
c = a + b
with tf.Session() as sess:
  # feed [1, 2, 3] to ‘‘a" via {a:[1, 2, 3]} & compute value of c
  print(sess.run(c, {a: [1, 2, 3]}))
```

Note that you can also feed values into Variables and constants:

```
# create placeholder of type float 32-bit, shape:  vector of 3 elements
a = tf.placeholder(tf.float32, shape=[3])
# create constant of type float 32-bit, shape:  vector of 3 elements
b = tf.constant([5, 5, 5], tf.float32)
# use placeholder as you would a constant or a variable
c = a + b
with tf.Session() as sess:
  # feed [1, 2, 3] to ‘‘a" and [8, 4, 5] to ‘‘b" & compute ‘‘c"
  print(sess.run(c, feed_dict = {a: [1, 2, 3], b:  [8, 4, 5]}))
```

This is especially useful if you want to debug a part of your code and bypass earlier expensive calculations.

2.2.7 Neural Network operations

TensorFlow has also defined multiple operations specific to neural networks. These include activation, classification and loss functions. For example to apply the Relu activation function to the tensor `x` is specified by `a = tf.nn.relu(x)` Visit TensorFlow Neural Network operations for the listing and the details of the available operations.

3 Training & testing fully connected networks

You are almost ready now to get your hands dirty with TensorFlow and use the library to define simple feed-forward networks and train them. Because our time is limited I have provided the *python* code for the first network. This code explicitly shows the basic structure of a TensorFlow programme performing network learning. In the first set of exercises you will play around with different parameter settings and see how it affects training. In the subsequent exercises you will build on the provided code to build more complicated networks and use more sophisticated optimization algorithms for training.

3.1 1-layer network trained with cross-entropy loss

To begin you will train and test a one layer network with multiple outputs to classify images from the CIFAR-10 dataset. You will train the network using mini-batch gradient descent applied to a cost function that computes the cross-entropy loss of the classifier applied to the labelled training data. For transparency we now give the mathematical details of the network and training in the next subsection.

Mathematical details of the network and training

Given an input vector, $\mathbf{x}$, of size $d \times 1$ our classifier outputs a vector of probabilities, $\mathbf{p}$ $(K \times 1)$, for each possible output label:

$$\mathbf{s} = W\mathbf{x} + \mathbf{b} \tag{1}$$

$$\mathbf{p} = \text{SOFTMAX}(\mathbf{s}) \tag{2}$$

where the matrix W has size $K \times d$, the vector $\mathbf{b}$ is $K \times 1$ and SOFTMAX is defined as

$$\text{SOFTMAX}(\mathbf{s}) = \frac{\exp(\mathbf{s})}{\mathbf{1}^T \exp(\mathbf{s})} \tag{3}$$

The predicted class corresponds to the label with the highest probability:

$$k^* = \arg \max_{1 \leq k \leq K} \{p_1, \ldots, p_K\} \tag{4}$$

(For the CIFAR-10 dataset $K = 10$ and $d = 32 \times 32 \times 3 = 3072$.)
The parameters W and $\mathbf{b}$ of our classifier are what we have to learn by exploiting the labelled training data. Let $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$, with each $y_i \in \{1, \ldots, K\}$ and $\mathbf{x}_i \in \mathbb{R}^d$, represent our labelled training data. In the morning lecture we described how to set the parameters by minimizing the cross-entropy loss. Mathematically this cost function is

$$J(\mathcal{D}, W, b) = \frac{1}{|\mathcal{D}|} \sum_{(\mathbf{x},y) \in \mathcal{D}} l_{\text{cross}}(\mathbf{x}_i, y_i, W, \mathbf{b}) \tag{5}$$

where

$$l_{\text{cross}}(\mathbf{x}, y, W, \mathbf{b}) = -\log(p_y) \tag{6}$$

and $\mathbf{p}$ has been calculated using equations (1, 2). (Note if the label is encoded by a one-hot representation then the cross-entropy loss is defined as $-\log(\mathbf{y}^T\mathbf{p})$.) The optimization problem we have to solve is

$$W^*, \mathbf{b}^* = \arg \min_{W,\mathbf{b}} J(\mathcal{D}, W, b) \tag{7}$$

In this assignment (as described in the lectures) we will solve this optimization problem via mini-batch gradient descent.
For mini-batch gradient descent we begin with a sensible random initialization of the parameters $W, \mathbf{b}$ and we then update our estimate for the parameters with

$$W^{(t+1)} = W^{(t)} - \eta \left. \frac{\partial J(\mathcal{B}^{(t+1)}, W, \mathbf{b})}{\partial W} \right|_{W=W^{(t)}, \mathbf{b}^{(t)}} \tag{8}$$

$$\mathbf{b}^{(t+1)} = \mathbf{b}^{(t)} - \eta \left. \frac{\partial J(\mathcal{B}^{(t+1)}, W, \mathbf{b})}{\partial \mathbf{b}} \right|_{W=W^{(t)}, \mathbf{b}^{(t)}} \tag{9}$$

where η is the learning rate and $\mathcal{B}^{(t+1)}$ represents the mini-batch at time $t+1$ and is a random subset of the training data $\mathcal{D}$ and

$$\frac{\partial J(\mathcal{B}^{(t+1)}, W, \mathbf{b})}{\partial W} = \frac{1}{|\mathcal{B}^{(t+1)}|} \sum_{(\mathbf{x},y)\in\mathcal{B}^{(t+1)}} \frac{\partial l_{\text{cross}}(\mathbf{x}, y, W, \mathbf{b})}{\partial W} \tag{10}$$

$$\frac{\partial J(\mathcal{B}^{(t+1)}, W, \mathbf{b})}{\partial \mathbf{b}} = \frac{1}{|\mathcal{B}^{(t+1)}|} \sum_{(\mathbf{x},y)\in\mathcal{B}^{(t+1)}} \frac{\partial l_{\text{cross}}(\mathbf{x}, y, W, \mathbf{b})}{\partial \mathbf{b}} \tag{11}$$

To compute the relevant gradients for the mini-batch, we then have to compute the gradient of the loss w.r.t. each training example in the mini-batch. Luckily for you, TensorFlow uses automatic differentiation to compute the necessary gradients. Thus you do not need to calculate the gradients by hand and code up the expresssions.

Implementing the network in TensorFlow

Remember the overall structure of a network in TensorFlow has two phases. In the phase 1 you assemble the graph. Phase 1 usually has, at the bare minimum, these sub-phases:

1. Define the placeholders for the input and their ground truth labels.
2. Define the parameters of the model as variables.
3. Define the operations in the graph that define the network function.
4. Define the loss function.
5. Define the optimizer.

In phase 2 you start a session and make computations based on the graph. For network training these computations involve iterating: evaluate the network function on some input data, compute the loss function, compute the necessary gradients and update the values of the network's parameters accordingly. PUT IN CITATION TO THE Stanford COURSE
You should now download the file `network1.py` from the tutorial website. You can run the code with the command (The programme relies on `read_cifar10.py` so it should be in the same directory as `network1.py`.):

(tensorflow) $ `python network1.py`

The code prints out the training and validation loss and accuracy after every 50th update step of the mini-batch gradient descent algorithm and then prints out the final test accuracy of the network after the final update step. 1000 update iterations are performed in total. Before playing around with or adapting the code you should examine the code and read the comments. This will allow you to see the syntax of TensorFlow and its structure in practice. It will also show how to call the functions defined in `read_cifar10.py` to

read in the CIFAR-10 dataset. Examining `read_cifar10.py` will also allow you to see how the input images are normalized before being entered into the network.

Exercise 2: *Playing around with training your first network*

To display the effects of different parameter settings on training and the final test accuarcy you can either use TensorFlow or you can add code to write the accuracy and loss values to a text file and then display the training and validation loss curves using `matplotlib` or `matlab`. I recommend you perform the following experiments:

- Change the learning rate to first to a low (.0001) and then to a high value (.1) and see what effect it has on training by looking at the resulting training and validation curves.
- Change the batch size low to high and once again see the effects on the speed and stability of training.
- Train for longer with a sensible batch size and learning rate. Does the model overfit or does the validation loss/accuracy just saturate?

3.2 A 1-hidden layer network

The network that you have just played around with is very simple with a limited capacity. Thus its performance staturates at $< 42\%$. The easiest way we can increase the network's capacity is to add a fully connected layer and with an attached activation function. Mathematically our updated network will represent this function:

$$\mathbf{s}_1 = W_1\mathbf{x} + \mathbf{b}_1 \tag{12}$$
$$\mathbf{x}_1 = \max(0, \mathbf{s}_1) \tag{13}$$
$$\mathbf{s} = W_2\mathbf{x}_1 + \mathbf{b}_2 \tag{14}$$
$$\mathbf{p} = \text{SOFTMAX}(\mathbf{s}) \tag{15}$$

where the matrix W_1 and W_2 have size $m \times d$ and $K \times m$ respectively and the vectors $\mathbf{b}_1$ and $\mathbf{b}_2$ have sizes $m \times 1$ and $K \times 1$.

3.2.1 Speeding up training: Add momentum to training

The vanilla version of mini-batch gradient descent with a sensible learning-rate is painfully slow for the size of the network and data we use in this exercise. To speed up training, we must add a momentum term in the update step. This is achieved as follows. You initialize a momentum vector (matrix) $\mathbf{v}_0$ for each parameter of the network ($\mathbf{v}_0$ has the same dimension as

the parameter vector/matrix and is initialized to have zero in all its entries) and then at each time step t:

$$\mathbf{v}_t = \rho \mathbf{v}_{t-1} + \eta \frac{\partial J}{\partial \boldsymbol{\theta}}$$
$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \mathbf{v}_t$$

where η is the learning rate as in standard mini-batch gradient descent, $\rho \in [0, 1]$ and $\boldsymbol{\theta}$ is a generic placeholder to represent one of the parameters of the model. Typically ρ is set to .9 or .99.

Exercise 3: *Implement a 2-layer network with momentum training*

Your tasks now are to update the code in `network1.py` to produce a new programme `network2.py` that

- implements the equations (12) to (15) (remember the operation `tf.nn.relu` applies the ReLu activation function elementwise to a tensor),
- performs the optimization using mini-batch gradient descent + momentum. Check out the webpage (TensorFlow) Training and the operation `tf.train.MomentumOptimizer` to read how to do this.

Once you have written and debugged the code then you should run it and explore what level of performance you can get with reasonable parameter settings and number of update steps. Is it possible to overfit this model? (After 5000 update steps you should get a test performance of $\sim$ ..%.)

3.3 An n-layer fully connected network

There is a definite performance bump with adding a fully connected layer. Do you get more performance gains by adding more fully connected layers? Your next task will be to generalize your code so that you can build a network with an arbitrary number of fully connected layers. Mathematically this corresponds to

$$\text{for } i = 1 \text{ to } (n-1)$$
$$\mathbf{s}_i = W_i \mathbf{x}_{i-1} + \mathbf{b}_i \tag{16}$$
$$\mathbf{x}_i = \max(0, \mathbf{s}_i) \tag{17}$$

and then finally

$$\mathbf{s} = W_n \mathbf{x}_{n-1} + \mathbf{b}_n \tag{18}$$
$$\mathbf{p} = \text{SOFTMAX}(\mathbf{s}) \tag{19}$$

where $\mathbf{x}_0$ denotes $\mathbf{x}$, $\mathbf{x}_0$ has dimensionality $d \times 1$ and each subsequent $\mathbf{x}_i$ has dimensionality $m_i \times 1$.

3.3.1 Tip to improve training: Weight initialization

You will probably find that training a fully connected network, from a random initialization, with many layers is very slow especially at the beginning. One practice that can make training possible/faster is to use Xavier initialization [1] (the aim is to keep the mean and variance of the histogram of the output responses similar to the input ones at each layer). Here you still perform a random initialization by drawing numbers from a Gaussian distribution, but you set the standard deviation of the Gaussian distribution at layer i with

$$\sigma_i^2 = \frac{1}{m_{i-1}} \tag{20}$$

There are other variation of this initialization known as He [2] initialization.

Exercise 4: *TensorFlow implementation of a n-layer fc network*

Your tasks now are to generalize the code you have written so that

- You can build a network with an arbitrary number of fully connected layers as described by equations (17) to (19).
- Train a network with 2 hidden layers with/without Xavier initializations with reasonable parameter settings for the optimizer and number of hidden nodes at each layer (~ 100).
- Train a network with 3 hidden layers with/without Xavier initializations with reasonable parameter settings for the optimizer and number of hidden nodes at each layer (~ 100).
- Check if you can learn and or if overfitting is possible.

4 Convolution layers

From the last exercise you should see that there are only small performance gains (if any) to be made by adding additional extra layers (after the first two) to your fully connected network. It is now time to see the benefit of adding convolutional layers instead. The first network with a convolutional layer we will investigate is one whose first layer is a convolutional one (where n_F 3D convolutions are applied) which is then followed by to fully connected

layers. Mathematically:

$$S_i = X * F_i + b_i \quad \text{for } i = 1, \ldots, n_F \tag{21}$$
$$S = \{S_1, \ldots, S_{n_F}\} \tag{22}$$
$$X_1 = \max(0, S) \tag{23}$$
$$H = \text{MaxPool}(X_1, k_x, k_y, s_x, s_y) \tag{24}$$
$$\mathbf{s}_1 = W_1 \text{vec}(H) + \mathbf{b}_1 \tag{25}$$
$$\mathbf{x}_1 = \max(0, \mathbf{s}_1) \tag{26}$$
$$\mathbf{s} = W_2 \mathbf{x}_1 + \mathbf{b}_2 \tag{27}$$
$$\mathbf{p} = \text{SOFTMAX}(\mathbf{s}) \tag{28}$$

where $*$ deontes a 2D convolution operation, the dimensions of the network's input, outputs and intermediary outputs are

- X the input image has size $32 \times 32 \times 3$,
- each S_i has size 32×32 (assuming zero-padding is used in the 2D convolution to maintain the spatial dimensions of the output to be the same as the input),
- S the volume of output responses has size $32 \times 32 \times n_F$,
- X_1 the volume of output responses, post activation function, has size $32 \times 32 \times n_F$,
- H the volume of output responses, after a max-pooling operation, has size $16 \times 16 \times n_F$ if we assume the spatial strides are $s_x = s_y = 2$ (k_x and k_y represent the width of the pooling regions),
- $\text{vec}(H)$, the flattened version of H, has size $(16 * 16 * n_F) \times 1$,
- $\mathbf{s}_1$ has size $m \times 1$,
- $\mathbf{s}$ has size 10×1

and the dimensions of the network's parameters are:

- each F_i has size $f \times f \times 3$,
- W_1, the weight matrix for the first fully connected layer, has size $m \times (n_F * 16 * 16)$,
- W_2, the weight matrix for the second fully connected layer has size $10 \times m$.

(We include the `max-pooling` operation into the network mainly to reduce the computational complexity of the network and the make it somewhat feasible to train the network on the CPU.)
For the next task you will need to create in TensorFlow:

1. the parmeters for n_F convolution filters of size $f \times f \times 3$ and
2. link a convolutional operation to the input batch of images.

The following code creates a variable containing the parameters for 64 convolutional filters of size `5×5×3` and the accompanying bias vector:

```
F = tf.Variable(tf.truncated_normal([5, 5, 3, 64], stddev=sig))
b = tf.Variable(tf.constant(.1, shape=[64]))
```

Assuming that `X_input` is the tensor containing the input images, you can create a convolutional layer applied to `X_input` with the code:

```
S = tf.nn.conv2d(X_input, F, strides=[1, 1, 1, 1], padding='SAME') + b
X1 = tf.nn.relu(S)
```

The webpage tf.nn.conv2d gives the explicit details of the parameters for the function. While the `max-pooling` operation is performed with

```
H = tf.nn.max_pool(X1, ksize=[1, 3, 3, 1], strides=[1, 2, 2, 1],
padding='SAME')
```

please read the documentation tf.nn.max_pool for detailed explanation of the inputs to this function.

Exercise 5: *Implementation of a convolutional network*

In this exercise you should

- Write code to implement the network defined by equations (22) to (28). For the initial version of the network use parameter settings $n_F = 50, f = 3, k_x = k_y = 3, r_x = r_y = 2, m = 100$. The code you write should be very similar to what you have written previously. The main difference, besides the addition of the convolutional layer, is that you do not need to reshape the input images into a 1D tensor when you read them. Thus when you read in the data you should call and the declaration of the `tf.placeholder` for the input images should have shape

  ```
  x_input = tf.placeholder(tf.float32, shape = [None, 32, 32, 3])
  ```

  as `tf.nn.conv2d` expects the images to be in their original shape of `32×32×3`. You can also recreate the operation vec(H) (in equation (25)) with the TensorFlow command:

  ```
  tf.reshape(H, [-1, int(32*32*50)])
  ```

- Change the optimizer to *Adam* so that you don't have to play around too much with the basic learning rate and have an adaptive learning rate for each parameter. Check out the webpage (TensorFlow) Training and the operation `tf.train.AdamOptimizer` for details.

- Train the model for 1000 update steps with the learning rate of the *Adam* optimizer set to .001. If you follow these directions you should get a test accuracy of $\sim$ 53%. Quick tip: Because networks with convolutions layers are slow to run, when you monitor the progress of training by computing the training loss and accuracy you should use a smaller random subset of the training data, just to keep the computational costs manageable.

At this stage you will have noticed that the training of this network with just one convolutional layer is quite slow. The idea of training on a powerful GPU card at this point may seem very appealing.

4.0.1 Time to try out FloydHub

You have working code to define and train a convolutional network. It is quite likely the code runs quite slowly and to perform many update steps on your laptop's CPU in a reasonable time. You can now use FloydHub if using your laptop's GPU is not an option! I will assume that you have created an account on FloydHub, have confirmed your e-mail address and installed the FloydHub command. I'm also assuming you are using `python2` though you can, of course, use `python3`.
The sequence of commands you should run for your first set of experiments are

$ `source floydDir/bin/activate` ← starts the virtual environment for floyd
(floyd) $ `floyd login` ← login in to your FloydHub account
(floyd) $ `cd DirName` ← move to the directory where your code exists
(floyd) $ `floyd init cifar10-experiments` ← Initialize the current directory to an existing or new project called cifar10-experiments
(floyd) $ `floyd run --env tensorflow:py2 --gpu --data sullivan/cifar10-data/1:cifar-10 "python network4.py"` ← run the command "python network4.py" on a FloydHub gpu

The flag `--data` and subsequent parameters are a link to my copy of the CIFAR-10 dataset I uploaded to FloydHub and makes it available at the `/cifar-10/` path. Thus before running your code on FloydHub you should change your code to read your dataset to point to the correct directory

```
dataset = input_data.read_data_sets('/cifar-10/', one_hot=True,
reshape=False)
```

Note you can upload your own dataset and the dataset will be available at the path `/input`.
You can monitor the progress of your job with the command

(floyd) $ `floyd output job_ID`

where the `job_ID` will have been printed out after the `run` command. With this command you will also see what has been printed to the terminal so far. You also have the option of logging in to your FloydHub account with your browser (doesn't work with Safari so use FireFox or Chrome instead) and checking progress within the "Projects" option.
If your *python* programme explicitly writes data to a file then you should set the directory to which you write as `/output/`. You are able to download all the files you write there to examine and use on your local machine. It is possible to use Tensorboard with http://docs.floydhub.com/guides/jobs/tensorboard/

Exercise 6: *Train your first convolutional network on a GPU*

For this exercise you should:

- Train your first convoutional network using a FloydHub GPU !
- Increase the number of update steps and see if the performance saturates and if you begin to overfit at a certain point.
- **Optional**: increase the size of f, n_f to 5 and 64 repectively and see if you get a bump in test performance.

4.1 Avoid overfitting - Data augmentation

In the lectures I mentioned that you can use data-augmentation, artifically increasing the size of your dataset by applying small geometric and photometric transformations, to bump the performance of your trained network. It also helps prevent overfitting. In the code `cifar10_read.py` I provide the ability to do a simple version of this. You can set a flag

```
dataset = cifar10_read.read_data_sets(data_dir, one_hot=True,
distort_train=True, reshape=False)
```

so that a random geometric transformation (a random crop and/or a left-right flip), are applied to each training image when it is put into a mini-batch. I use `scipy` to apply the geometric transformations.

Exercise 7: *Take advantage of data-augmentation*

For this exercise you should:

- Re-train your convoutional network using my basic data-augmentation with a decent number of update steps. Record how much of a performance bump you get.
- **Optional**: TensorFlow has the functionality to distort images. Update your code so that the transformation of the mini-batch image

data is performed by TensorFlow. See Images for a list of the possible transformations. I suggest you limit yourself to: left-right flips, small scale changes, small translational shifts and small photometric adjustments. You can use TensorBoard to save and visualize the transformed images and check you have no bugs when you apply the transformations. When you apply the different augmentations you can check which have the most impact of the final performance of the network.

Exercise 8: *Add a second convolutional layer to your network*

For this exercise you should:

- Add another layer of "convolutionals + relu + max-pooling" operations to you network. You can use the same parameters for these operations as for the first convolutional layer. Train the network and see what performance gains you get. You should definitely use data-augmentation. If you train for $> 10,000$ you should get test performance pushing $\sim$80%

For this network you can train for a long time hundred of thousands of update steps and still continue to get marginal improvements. Though the magnitude of the improvements becomes smaller as training continues.

4.2 Batch normalization

If you have made it this far well done! Batch normalization [3] is a modern development to the training process that has improved the speed and stability of convergence of training. It also helps with regularization and it has been observed by many (but not always) to improve the test performance of a network when it is used during training. Thus if you do run a serious research project involving training a ConvNet or a fully connected network you should train using batch normalization.
Unfortunately, it is slightly complicated to implement batch normalization because you have to implement a slightly different algorithm depending on whether you are training or testing. During training you compute the mean and variance of the batch data and use these to normalize the batch data. While during testing you use the population mean and variance that was calculated during training. One solution to this problem is that you create a placeholder flag that indicates whether you are classifying input data for training or testing purposes. Assuming that `x` is the Variable corresponding to the responses after an affine or a convolutional operation then the following code snippet should implement batch normalization on the batch of reponses stored in `x`.

```
# create placeholder for the flag of whether we are training or testing
is_training = tf.placeholder(tf.float32, shape = [1])

# create Variables learnt by the network used by batch normalization
gamma = tf.Variable(tf.ones(shape[-1]))
beta = tf.Variable(tf.zeros(shape[-1]))

# create Variables not trained by the optimizer class to keep track
of the population mean and variance
pop_mean = tf.Variable(tf.zeros(shape[-1]), trainable=False)
pop_var = tf.Variable(tf.ones(shape[-1]), trainable=False)

# compute the mean and variance of the responses in the batch
shape = x.get_shape().as_list()
batch_mean, batch_var = tf.nn.moments(x, range(len(shape)-1))

# if in training mode update the population mean and variance
decay = .9 * is_training[0] + 1 * (1 - is_training[0])
train_mean = tf.assign(pop_mean, pop_mean * decay + batch_mean * (1 -
decay))
train_var = tf.assign(pop_var, pop_var * decay + batch_var * (1 - decay))

# assign the values to pop_mean and pop_var before the following ops
with tf.control_dependencies([train_mean, train_var]):

  # Compute the mean and var to use in bn op, depends on is_training[0]
  u_mean = batch_mean * is_training[0] + pop_mean * (1 - is_training[0])
  u_var = batch_var * is_training[0] + pop_var * (1 - is_training[0])

  # apply batch normalization to x
  next_x = tf.nn.batch_normalization(x, u_mean, u_var, beta, gamma,
10e-8)
```

Exercise 9: *Apply batch normalization*

For this optional exercise you should

- Explain what the lines

  ```
  shape = x.get_shape().as_list()
  batch_mean, batch_var = tf.nn.moments(x, range(len(shape)-1))
  ```

  calculate and how these calculations differ depending on whether you `x` has size `[batch_sz, w, h, d]` or `[batch_sz, m]`.

- Add batch normalization after each layer (except the last layer before the softmax operation) in one of your networks. Debug the code and see if the number of updates needed to reach a certain level of performance decreases from when you don't use batch normalization.

References

[1] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, 2010.

[2] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV*, 2015.

[3] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *arXiv:1502.03167 [cs]*, 2015.